ISBN 978-981-11-3671-9
Proceedings of 2017 the 7th International Workshop on Computer Science and Engineering (WCSE 2017)
Beijing, 25-27 June, 2017, pp.73-80
doi: 10.18178/wcse.2017.06.012

The Optimization of Aho-Corasick Automaton with Large-scale Pattern Sets

Bo Sun[1], Yifu Li [1], Xiangzhan Yu [2] and Shan Yao[1 +]

[1] National Computer Network Emergency Response Technical Team/Coordination Center of China, Beijing, China 100029

[2] School of Computer Science and Technology Harbin Institute of Technology Harbin, China 150001

Abstract. Multi-pattern matching is widely used in text retrieval, intrusion detection, information retrieval and many other areas. Aho-Corasick algorithm is a main method for multi-pattern matching for its high efficiency and stability. However, with the continuous expansion of the scale of the pattern set, Aho-Corasick algorithm faces the double test of memory consumption and the processing time. This paper summarizes several frequently used optimized methods of Aho–Corasick automaton, and proposes a new algorithm called BIT-AVL-AC, which is fused with the AVL tree and the bit vector. The experiments show that this algorithm provides a good trade-off between memory usage and processing speed with large-scale pattern sets.

Keywords: multi-pattern matching, Aho-Corasick algorithm, large-scale pattern set.

1. Introduction

Multi-pattern matching is a classical problem in computer science which widely used in text retrieval, intrusion detection, information retrieval and many other areas. An efficient algorithm to solve this problem is Aho-Corasick(abbr. as AC) algorithm[1],which is an extension of the single-keyword matching Knuth–Morris–Pratt algorithm [2] and based on an automata approach. Wu–Manber [2] algorithm is another famous algorithm for matching multiple patterns. It is an extension of the Boyer–Moore [3] single-keyword matching algorithm, and employs Boyer-Moore [3] techniques like SHIFT table and HASH table techniques. From the perspective of universality, Wu–Manber algorithm has high requirements for pattern sets, and its time complexity relates to the minimum length of the pattern string. The shortcoming of Wu-Manber algorithm makes itunsuitable for multi-pattern matching applications with large-scale pattern sets. However, the time complexity of AC algorithm is irrelevant with the pattern sets while can be affected by the size of the target text. Therefore, AC algorithm is much more commonly applied in multi-pattern matching. Many optimized methods based on AC automata have been proposed in recent years [4,5]. However, as the scale of the pattern sets increases, memory consumption and processing efficiency, including the preprocessing and searching speed, have become the main bottleneck.

Our aim is to find an optimized method of AC automaton which can provide the best trade-off between memory usage and processing speed with large-scale pattern sets under different conditions .In this paper we propose a new algorithm called BIT-AVL-AC based on AC automaton which is fused with the AVL tree and the bit vector. We study different efficient optimized methods of AC algorithm and compare them with BIT-AVL-AC algorithm. Our method shows great reduction in memory space, as well as fast preprocessing and searching speed. Experiments are also conducted to verify the method’s efficiency.

The rest of this paper is organized as follows: Section 2 reviews the classic optimized methods of AC automaton. In section 3 we present our optimized method for optimized AC automaton. Next, we compare

[+] Corresponding author. Tel.: +0861082991535.
E-mail address: yaoshan@cert.org.cn.

our method with other classic optimized methods and demonstrate the experimental results in section 4. Finally, Section 5 concludes the paper.

2. Related work

Optimized methods based on AC automaton to reduce the memory consumption and accelerate the processing speed have been greatly sought by many researchers [6,7]. A few methods are based on the hardware [8, 9], but most optimized algorithms are improved from existing ones by software implementation. There are two main optimized methods for the AC automaton. One is to optimize the data structure used in implementing the transition function, and possible data structures to use are array, linked list, balanced tree and so on. Let n be the number of transitions leaving from a state of the automaton, a linked list has $O(1)$ insertion time complexity and $O(n)$ access time complexity, an array has $O(1)$ insertion time complexity and $O(1)$ access time complexity. Another optimization is the table compression method for extended AC automaton, which is based on the matrix implementation of the automaton [10]. The key idea of the matrix method is very simple: a matrix T[S][Σ] is construed to store the automaton, the number of the automaton states is S and the size of the character set is Σ. We assume that the current state is q and the input character is s, $q \in$ S and $s \in \Sigma$, T[q, s] represents the next state. There are some typical algorithms based on the matrix of AC automaton, such as compressed-row [10], banded-row [11], triple-array [12] and bitmap[13]. In order to achieve a significant spatial compression effect, only the transitions are stored in the matrix. The failure nodes will be stored in another array.

In this section we will introduce 4 classic and efficient optimized methods for the AC automaton, such as AVL tree, compressed-row, bitmap and double-array [14].

2.1. Compressed-row

This method is often used to compress the sparse matrix. Here is the key idea of the compressed-row: each row of the matrix only stores the non-empty elements and will be organized like (r, s_1, s_2,..., s_r, q_1, q_2, ..., q_r), r is the number of the non-empty elements, s_i ($1 \leqslant i \leqslant r$) represents the non-empty elements and q_i ($1 \leqslant i \leqslant r$) represents the state reached after reading the input symbol s_i. Let n be the number of transitions leaving from a state of the automaton, and this algorithm has $O(i)$ insertion time complexity and $O(n)$ access time complexity. The memory space can be greatly compressed when the matrix is sparse, however, the compression efficiency will be low when the matrix is dense.

2.2. Bitmap

Bitmap compression applied to AC automaton uses a bit matrix B[S][Σ] to indicate the storage condition of the matrix T[S][Σ] which holds the AC automaton. Each row of B maintains a |Σ| bit bitmap indicating whether a traversal with a given character is valid or requires traversing along the failure pointer path. The non-empty elements of each row in T are stored in a linked list. Let n be the number of transitions leaving from a state of the automaton, and bitmap method has $O(1)$ insertion time complexity and $O(n)$ average access time complexity. The compression efficiency of bitmap will be low when the matrix is dense.

2.3. Double-array

As explained in [12], a DFA compression could be done using three linear arrays, namely **BASE**, **NEXT** and **CHECK**.

The triple-array structure is composed of:

1. BASE. Each element in the **BASE** corresponds to a node of the trie. For a trie nodes, base[s] is the starting index within the next and check pool (to be explained later) for the row of the node s in the transition table.

2. NEXT. This array, in coordination with **CHECK**, provides a pool for the allocation of the sparse vectors for the rows in the trie transition table. The vector data, that is, the vector of transitions from every node, would be stored in this array.

3. CHECK. This array works in parallel to the **NEXT**. It marks the owner of every cell in the **NEXT**. This allows the cells next to one another to be allocated to different trie nodes, which means the sparse vectors of transitions from more than one node are allowed to be overlapped.

In the double-array structure, the **BASE** and **NEXT** are merged, resulting in only two parallel arrays, namely, the **BASE** and **CHECK**. The removal of the **NEXT** array is achieved by renumbering the states of the automaton, so that the information on the **NEXT** array is stored in the state numbers. The defect of double-array is that it will take lots of time to modify the pattern set, therefore, it is not suitable for multi-pattern matching with large-scale pattern sets.

2.4. AVL tree

An AVL tree [15] is a self-balancing binary search tree, which is named after its two Soviet inventors, G.M. Adelson-Velskii and E.M.Landis. In an AVL tree, the heights of the two child subtrees of any node differ by at most one; if at any time they differ by more than one, rebalancing is done to restore this property. Lookup, insertion, and deletion all take *O (logn)* time complexity in both the average and worst cases, where *n* is the number of nodes in the tree prior to the operation. Insertions and deletions may require the tree to be rebalanced by one or more tree rotations.

Optimizing AC automaton with AVL tree, the transitions of the automaton are implemented with AVL tree. Compared with the array method, the AVL tree can greatly reduce memory consumption, although it is slower in searching. It has *O (logn)* complexity insertion time and *O (logn)* average access time complexity, where *n* is the number of transitions leaving from a state of the automaton.

3. The Bit-avl-ac Algorithm

In order to optimiz the implementation of the transitions of the automaton, we propose a new algorithm called BIT-AVL-AC, which is based on AC automaton and fused with the AVL tree and the bit vector.

3.1. The idea

The BIT-AVL-AC algorithm uses an AVL tree to store the transitions of each state of the AC automaton. From the section of AVL tree we know that it has *O (logn)* insertion time complexity and *O (logn)* access time complexity. Clearly, it is faster than the implementation of linked list in examining the current state and has a success pointer labeled by the current character. However, whether the next state can be reached by the current character, the whole AVL tree will be searched, which leads to low search efficiency, especially when using the big alphabet, for example the Unicode character set that has the size of 65536 symbols. Because of the big alphabet and large-scale pattern sets, there will be quantities of transitions of a state, and the depth of the AVL tree will be so large that takes lots of time to search the whole tree.

Therefore, the BIT-AVL-AC algorithm creates a bit vector for each state as a fast lookup table. The mapping relations in the character set and the bit vector can be adopted for the two methods.

Method 1. Each bit of the bit vector represents one input character of the character set. For example: assume that the alphabet size is 256 (e.g., ASCII characters), then the length of the bit vector is 256, which means that each bit represents only one character. However, if the character is too big (e.g., Unicode characters), then the second method will be more suitable.

Method 2. Each bit of the bit vector represents two or more input characters of the character set. Also, we can adjust the bit vector by the frequency distribution of the character set to make the method more efficient.

Although the BIT-AVL-AC algorithm occupies more memory space by using the bit vector for each state, the preprocessing speed and searching speed are accelerated sufficiently. And compared with the AVL method for AC automaton, it is more efficient, especially with the big character set. The length of the bit vector can be adjusted flexibly to provide a good trade-off between memory usage and processing time.

3.2. Preprocessing phase of the algorithm

The construction of the automaton and the transition function are the main differences between BIT-AVL-AC algorithm and AC algorithm, other aspects like the failure function and the output function are the same.

The data structure of BIT-AVL-AC:

```
struct BIT-AVL-AC
{
   struct BIT-AVL-AC *avlRoot;  // root of AVL tree
   struct BIT-AVL-AC *lChild;  //left child node
   struct BIT-AVL-AC *rChild;  //right child node
   int bf;  // balance factor
   int data;  // node data
   int output;  // output status
   struct BIT-AVL-AC *fail;  // fail pointer
   bit bitVec: vecLength;  // length of the bit vector
};
```

The transition function of BIT-AVL-AC algorithm is different from that of AC algorithm. Now assume the current state is *q*, and the current input character is *s*. There are two conditions:

Condition 1. In accordance with method 1 in section A: Firstly, we check if q.bitVec [*s*]=1. If true, the next state can be reached by *s*, we should search the AVL tree of current state, in which the transitions are stored, and then reach the next state. Otherwise, the automaton changes to state f(*q*) use the failure function.

In this condition, the transition function *goto* is described in algorithm 1. The function of getFromAvl(*q*.avlRoot, *s*) is that we search the AVL tree of the current state's to match the input character *s*, and return the next state which can be reached by *s*. This function is also used in algorithm 1.

Algorithm 1. the transition function *goto* of BIT-AVL-AC in condition 1

```
procedure goto(q, s)
  while(q.bitVec[s]=0) do
      q := fail(q);
  q:=getFromAvl(q.avlRoot,s);
  end while
  return q;
end procedure
```

Condition 2. In accordance with method 2 in section A: After reading *s*, we check if *q*.bitVec[*s*]=1. If true, the next state may be reached by *s*. The reason is that, for example, the bit represents the character *s* and character *t* or other more characters, so we can not make sure that the next state can be reached by *s*. Only after searching the whole AVL tree can we know that. But if *q*.bitVec[*s*]=0, it can be sure that the next state can not be reached by *s*.

In this condition, the transition function *goto* is described in algorithm 2. And in the algorithm, the function of inAvl(*q*.avlRoot, *s*) is that: search the AVL tree of the current state. If the state of the input character *s* can be found in it, then return true , or else return false.

Algorithm 2. the transition function *goto* of BIT-AVL-AC in condition 2

```
procedure goto(q, s)
  while (1) do
    if(q.bitVec[s]=1)
      if(inAvl(q.avlRoot,s))
        break;
```

```
            end if
            else $q$ := $fail$ ($q$);
          end if
          else $q$ := $fail$ ($q$);
        end while
        $q$:=getFromAvl($q$.avlRoot , $s$);
        return $q$;
      end procedure
```

The preprocessing phase of BIT-AVL-AC algorithm and AC algorithm are the same in essence. However, in the preprocessing phase of BIT-AVL-AC, if the current state q has a success pointer labeled by the current character s, then set q.bitVec[s]=1. We mainly discuss the phase of the construction of the AC automaton for patterns K = {k_1,k_2...,k_n}, where the transition function goto and part of the output function output are computed.

We assume that the initial state is *start* and Σ is the character set. The construction process is divided into the following 4 steps:

Step 1: Do the ENTER function for every pattern string. The function of ENTER is to build the trie by reading one pattern string.

Step 2: In the ENTER function: start with the initial state *start*, and read the pattern string k_i, which is divided sequentially into the character queue {a_1,a_2...,a_n}, one character at a time.

Step 3: Check if $goto(state, a_j) \neq$ fail, where s is the current state of the automaton. If true, the automaton changes to the state $goto(state, a_j)$ until $goto(state, a_j) = fail$.

Step 4: Read the rest of the character queue {a_j,a_{j+1}...,a_n} sequentially and add new states to the trie: assume the current state is *state* and the input character is a_p, $j \leq p \leq n$. Set state.bitVec[a_p]=1 and then set a new state *newState*. After that, add the *newState* into the current state's AVL tree. Then, the automaton changes to the state $goto(state, a_p)$.

The phase of the AC automaton construction is described in Algorithm 3.

Algorithm 3. Construction of the automaton for patterns K = {k_1,k_2...,k_n}, where the transition function *goto* and part of the output function output are computed.

```
for i := 1 to $n$ do ENTER ($k_i$);
end for
Set $goto(start, a)$ := start for each $a \in \Sigma$ such that $goto(start, a) = fail$;
procedure ENTER($a_1$,$a_2$...,$a_n$)
  $state$ := $start$;
  $j$ := 1;
  while $goto$(state,$a_j$) $\neq$ $fail$ do
    $state$ := $goto(state, a_j)$;
    $j$ := $j$ + 1;
  end while
  for $p$ := $j$ to $m$ do
    $state$.bitVec[$a_p$ ]=1;
    $newState$.bitVec=0;
    $newState$.data= $a_p$ ;
    insertIntoAvl($state$.avlRoot , $newState$);
    $state$ := $goto(state, a_p)$;
  end for
```

```
    $output(state) := \{ a_1,a_2...,a_m \}$;
  end procedure
```

3.3. Searching phase

The searching process is divided into the following 4 steps:

Step 1: Check if the current state *q* has a success pointer labeled by the current character *s*. If true, go to the step 2; otherwise turn to the step 3. The transition function *goto* used in checking is described in section 3.

Step 2: The automaton Changes to the state $goto(q,s)$, and search the next character of the target text.

Step 3: The automaton changes to state $f(q)$ using the failure function, until $goto(q,s) \neq fail$.

Step 4: Repeat the first three steps until the all characters of the target text have been processed.

4. Experimental Results

In the experiments, a computer with an AMD Opteron (TM) 64 Processor 6136 (16 × 750 MHz) CPU, 32GB memory is used. The host’s operating system is Linux Red Hat Enterprise Linux Server release 5.4 (Tikanga). All the algorithms are implemented with C++ and compiled with g++.

The alphabet is the ASCII character set which has 256 symbols. The target length of the randomly built text is 1GB. The length of the patterns ranges from 1 and 100 and the average length is 50. All patterns are randomly built. The numbers of the patterns are 200000, 400000, 600000, 800000 and1000000.

We compared 6 methods based on AC automaton, which include array, AVL tree, compressed-row, bitmap, double-array and BIT-AVL-AC described in section 3.

4.1. Memory consumption

The memory consumption results are shown in Figure 1. When the number of patterns is 40000, the memory consumption of the array implementation is over 32GB, which exceeds the memory limitation of the computer used for the experiment. Obviously the array method is not suitable for large-scale pattern sets. Therefore, this method is not measured in the follow experiments.

Comparing with array, the other 5 methods can reduce the memory consumption effectively. The memory consumption of double-array is the lowest while the BIT-AVL-AC algorithm takes the maximum memory space.

4.2. Preprocessing speed

The preprocessing speed results are shown in Figure 2. When the number of patterns is 100000, it takes more than 24 hours for preprocessing with the method of double-array. And plenty of time will be taken during every update or delete of the pattern set. Therefore, double-array is also unsuitable for large-scale pattern sets obviously and it is not measured in the follow experiments.

It is clearly seen from Figure 2 that the BIT-AVL-AC method has the fastest preprocessing speed. There is little difference between the other three methods at preprocessing speed, among which the bitmap has the lowest speed.

4.3. Searching speed

The results of searching speed are shown in Figure 3. Here, the searching speed of the BIT-AVL-AC method is faster than other methods. However, the searching speed of the bitmap is significantly slower than the others.

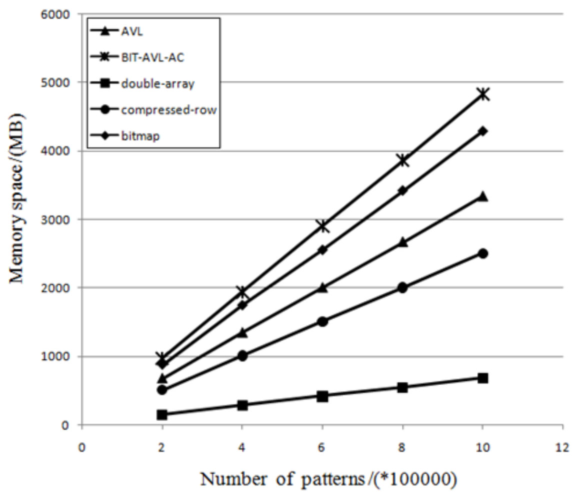

Fig. 1: Memory consumption with different number of patterns

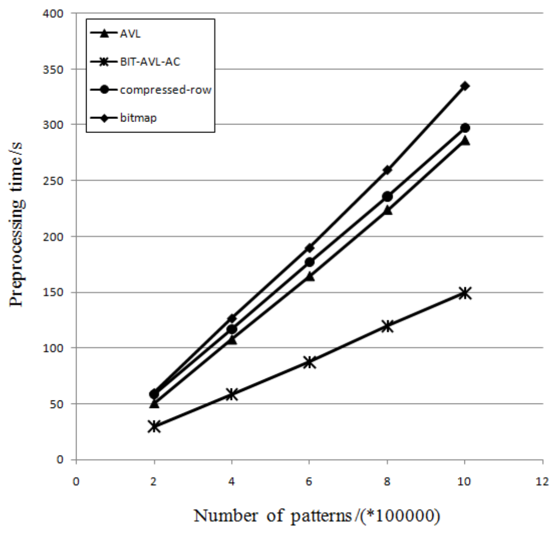

Fig.2: Preprocessing speed with different number of patterns

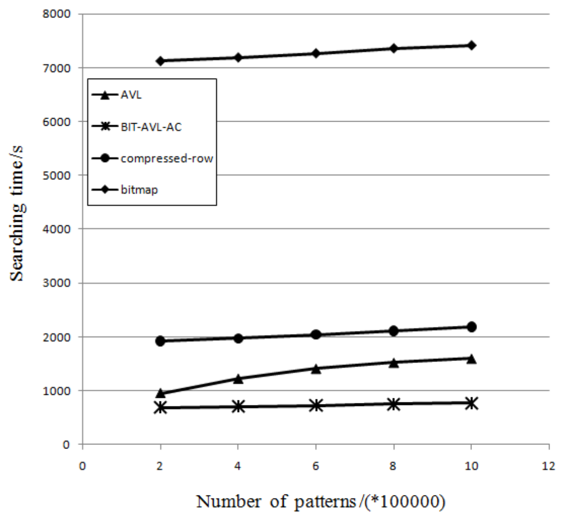

Fig.3: Searching speed with different number of patterns.

5. Conclusions

In this paper, we compare several classic optimized methods of AC automaton. The AVL tree, compressed-row, bitmap and BIT-AVL-AC methods are all suitable methods for the implementation of AC automaton which can greatly reduce the memory consumption. Among them, firstly, the compressed-row method has the best efficiency in reducing memory consumption. However, both the preprocessing speed and the searching speed are not so fast, thus being suitable especially when the memory space is limited. Secondly, the bitmap method is slow at preprocessing and searching speed, and it does not possess obvious superiority in terms of compression efficiency compared with other methods. Thirdly, although the BIT-AVL-AC method requires a little more memory usage than the other methods, it still meet the request of the memory limitation in most cases. Moreover, more importantly, it is faster at preprocessing and searching than the other methods. The obvious advantage of processing speed makes the BIT-AVL-AC method more efficient for multi-pattern matching with large-scale pattern sets. In conclusion, the BIT-AVL-AC method provides a good trade-off between memory consumption and processing speed, which is an efficient method to optimize AC automaton with large-scale pattern sets.

6. Acknowledgment

This work is supported by the National Natural Science Foundation of China under Grant No. 61173144, and the National Key R&D Program of China under Grant No.2016Y******501.

7. References

[1] A. Aho and M. Corasick, "Efficient String Matching: An Aid to Bibliographic Search ", Communications of the ACM,1975,18(6): 333-340.

[2] Wu S, Manber U. A fast algorithm for multi-pattern searching. Technical Report TR-94-17, Department of Computer Science, University of Arizona, 1994.

[3] BOYER R S, MOORE J S. A fast string searching algorithm[J]. Communications ofthe ACM, 1977, 20:762-772.

[4] Cantone, D., Faro, S., Giaquinta, E.: On the bit-parallel simulation of the nondeterministic aho-corasick and suffix automata for a set of patterns. J. Discrete Algorithms 11, 2012, 25–36.

[5] D. Pao, W. Lin, and B. Liu. A Memory Efficient Pipelined Implementation of the Aho-Corasick String Matching Algorithm. ACM Trans. Architecture and Code Optimization, 2010,vol. 7, no. 2, pp. 1-27.

[6] J. Nieminen and P. Kilpel. Efficient Implementation of Aho-Corasick Pattern Matching Automata Using Unicode. Software-Practice and Experience, 2007,37(6): 669–690.

[7] Yang Y, Liu Y, Liu P, et al. Automaton Compact Representation Technology in String Matching Algorithm. Computer Engineering, 2009, 21: 016.

[8] Wang Q, Prasanna V K. Multi-core architecture on fpga for large dictionary string matching. Field Programmable Custom Computing Machines. FCCM'09, 2009,96-103.

[9] Tumeo, A. Villa, O. Chavarrı´a-Miranda, D.G. AhoCorasick String Matching on Shared and Distributed Memory Parallel Architectures. IEEE Transactions on Parallel and Distributed Systems, 2012, 436-443.

[10] Dencker P, Dorre K. Optimization of Parser Tables for Portable Compilers. ACM Transactions on Programming Languages and Systems, 1984, 6(4): 546-572.

[11] Norton M. Optimizing Pattern Matching for Intrusion Detection[Z].[2008-12-05]. http://www.idsresearch.org.

[12] Aho V, Sethi R, Ullman J D. Compilers: Principles, Techniques, and Tools[M]. [S. l.]: Addison-Wesley, 1985.

[13] Tuck N, Sherwood T, Calder B, et al. Deterministic Memoryefficient String Matching Algorithms for Intrusion Detection. Proc. of IEEE INFOCOM'04. Hong Kong, China, 2004:2628-2639.

[14] Aoe J. A practical method for implementing string pattern matching machines. Information Sciences 1992; 64(1–2):95-114.

[15] G. M. Adelson-Velskii and Y. M. Landis. An algorithm for the organization of information. Soviet Math. Dokl., 3:1259-1262, 1962.