DOI 10.12694/scpe.v20i3.1572
Scalable Computing: Practice and Experience ISSN 1895-1767
Volume 20, Number 3, pp. 563–576. http://www.scpe.org © 2019 SCPE

SIMD IMPLEMENTATION OF THE AHO-CORASICK ALGORITHM USING INTEL AVX2

OURLIS LAZHAR [*]AND BELLALA DJAMEL [†]

Abstract. The Aho-Corasick (AC) algorithm is a multiple pattern exact string-matching algorithm proposed by Alfred V. Aho and Margaret J. Corasick. It is used to locate all occurrences of a finite set of patterns within an input text simultaneously. The AC algorithm is in the heart of many applications including digital forensics such as digital signatures demonstrating the authenticity of a digital message or document, full text search (utility programs such as *grept*, *awk* and *sed* of Unix systems), information retrieval (biological sequence analysis and gene identification), intrusion detection systems (IDS) in computer networks like SNORT, web filtering, spam filters, and anti-malware solutions (virus scanner). In this paper we present a vectorized version of the AC algorithm designed with the use of packed instructions based on the Intel®streaming SIMD (Single Instruction Multiple Data) extensions AVX2 (Advanced Vector Extensions 2.0). This paper shows that the vectorized AC algorithm reduces significantly the time matching process comparing to the implementation of the original AC algorithm.

Key words: Pattern-matching, Aho-Corasick algorithm, Vectorization, Intel®Streaming SIMD Extensions 2.0 (AVX2).

AMS subject classifications. 68W32, 68W10

1. Introduction. String-matching is the concept of finding a sequence of characters, often called patterns, inside a provided text. The matching process includes the location of these characters inside the text. Formally, pattern-matching is the problem of locating all occurrences of a pattern P of length m in a text T of length n. Both the pattern and the searched text are vectors of elements of a finite set called an alphabet Σ. We say that pattern P occurs with valid shift s in text T if $0 <= s <= n - m$ and $T[s + 1.s + m] = P[1..m]$.The pattern-matching is therefore the problem of finding all valid shifts with which a pattern P occurs inside a text T [1].

Pattern-matching algorithmic complexity is usually analysed by the running time of the algorithm and the memory space required by the computations. To this, we can add a setup time, or a substantial amount of computation before the algorithm can begin searching (time spent during the pre-processing phase), and the need for backtracking or not (since moving back and forth through the search text can entail some form of buffering if the search text doesnt exist in memory, but sent to the program as a stream of data) [2]. Considering the running time feature, all pattern-matching algorithms perform on the order of $a + bn$, where a is the pre-processing time, b is a constant indicating the number of comparisons made for each character, and n is the number of characters in the text being searched. The effectiveness of the algorithms is measured according to their ability to lower the value of b. Effective ones perform in less than one comparison per character searched; $(b < 1)$ termed as *sub-linear* pattern-matching algorithms [2].

Many algorithms to solve pattern-matching problem exist. They are classified either as single or multiple pattern algorithms based on the number of patterns to look for. Applications that relay on this class of algorithms may require exact or approximate pattern-matching [1]. Many solutions for exact string-matching of multiple patterns have been developed. However, most of them have been designed for moderately sized pattern sets, and therefore they do not fit well with larger sets of patterns. The most commonly used solutions are Aho-Corasick (AC), Wu-Manber (WM) and Commentz-Walter (CW) algorithms.

The Aho-Corasick algorithm [3] uses a set of patterns to construct a finite automaton during the pre-processing phase. The matching involves the automaton scanning the text string, stepping through the input characters one at a time and changing the state of the automaton. At every state transition for every text character, we check if there is a match by observing whether the current state is an output state (which indicates that a pattern has been found) or not. The time complexity of the AC algorithm is proportional to the total length of the patterns during the pre-processing phase and only proportional to the size of the text being processed during the searching phase [2]. The AC algorithm has the advantage of examining each text character only once, and locating all occurrences of patterns within a given text in one pass. Its major drawback is the space memory required to store the automaton states, which increases with their number.

[*]Department of Industrial Engineering, Faculty of Technology, University of Batna2, Batna, Algeria. (ourlisl@yahoo.fr)
[†]Department of Computer Science, University of Batna2, Batna, Algeria. (bellala_djamel@yahoo.co.uk).

The Wu-Manber algorithm [4] uses the bad-character shift (indicating the characters to skip during the scanning phase) from the Boyer-Moore algorithm, but looks at blocks of text instead of single characters during the scanning phase to improve the matching performance: both the pattern and the text are treated as blocks. The algorithm also builds three tables during the pre-processing phase: the hashing or SHIFT table to determine the number of characters that can be skipped when scanning the text, the HASH and PREFIX tables to determine which pattern is a candidate for the match (and eventually to verify the match) when the shift value equals to 0. In practice, the algorithm is more suitable when dealing with patterns of similar length, and its running time does not increase in proportion to the size of the pattern set. But, in case of short patterns, the scanning time increases due to the decrease in characters shifting [5].

The Commentz-Walter algorithm [6] is a suffix-based, exact multiple string matching solution that combines the concepts of Boyer-Moore and AC algorithms. In the pre-processing phase, the CW algorithm constructs a finite automaton similar to that of the AC algorithm but from the reversed set of patterns to use the shifting method of the Boyer-Moore algorithm, which usually handles mismatches in a way to obtain a sub-linear time complexity. The matching involves the automaton scanning through the text string in a backward manner: characters of the patterns are scanned from right-to-left beginning with the rightmost one, exactly as the Boyer-Moore algorithm does. The length of the matching window is the minimum pattern length. In case of a mismatch or a complete match of the pattern, the CW algorithm uses a recomputed shift table to identify portions of text to be skipped and shifts the window to the right accordingly. With small numbers of patterns to look for, Boyer-Moore aspects of the CW algorithm can make it faster than the AC algorithm, but with larger numbers of patterns, the AC algorithm has a slight advantage [7, 8].

To improve the AC algorithm, many approaches are proposed. Nishimura et al. [9]described the speed-up of string pattern-matching by collecting states used frequently for CPU (Central Processing Unit) cache efficiency using data compression. Their approach reduced the elapsed time in case of a compressed English text to about 55%. In order to accelerate their Network Intrusion Detection System (NIDS) engine, used to identify network attacks by inspecting packet content against a collection of many thousands of predefined patterns, Lin et al. [10] proposed a Graphic Processor Unit (GPU). Their algorithm on GPU achieved up to 4,000 times speed-up compared to the AC algorithm on CPU. Arudchutha et al. [11] improved the speed of the AC algorithm using a multicore CPU based software implementation through parallel manipulation of pattern-sets using POSIX (Portable Operating System Interface) thread utility to handle a large amount of data in the form of strings (Bio-computing applications). They arrived at the conclusion that their implementation gives better results compared to that of a parallel implementation of the same algorithm. In this paper, we present another technique to speed up the AC algorithm, which benefits from Intels AVX 2.0 introduced in the Haswell microarchitecture in 2013.

2. Vectorization: Data Parallelism. Vectorization, or SIMD processing, is the process in which an algorithm is converted from a sequential implementation, that performs an operation one pair of operands at a time, to a vector process where multiple data operands are simultaneously performed in only one instruction (data-level parallelism), provided that this scalar algorithm is suitable for being parallelized using vectorization techniques [12]. In SIMD processing, operands (adjacent data items of the same type and size refer to as vectors) are treated not as individual integers or float-point numbers but rather as whole vectors. Moreover, not only a single instruction operates on whole vectors but also reading from and writing to memory (load/store instructions) operate on whole vectors too.

A SIMD operation like addition, logical OR, data movement, conversion or comparison is a simple operation that is performed on vector items in parallel (all operations in SIMD are vector operations). Only one instruction is needed to process these vector items (refer to as packed data), whereas in sequential processing, without SIMD, multiple instructions would be used instead [13]. There are different approaches to achieving vectorization:

1. The first form of vectorization is to explicitly call the processors SIMD instructions using low-level assembly code and available registers. This way of utilising vector instructions offers more control over the program and yields the maximum performance of the system but cross-platform porting is quite difficult;
2. Use of the compiler intrinsic functions (assembly-coded C style functions that provide access to many Intel®instructions, including SIMD instructions, without the need to explicitly write assembly code).

In this approach, high level languages such as C/C++ or FORTRAN can be used to write the code. It provides almost the same benefits as using inline assembly and saves us to deal with register allocations, instructions scheduling and stack calls;

3. Compiler auto-vectorization: the easiest form of vectorization where no changes to source code are required. As a result, the portability of the code is preserved;
4. Use of a high-level library, such as Intel®MKL (Math Kernel library) that contains implementations which use vector instructions of common mathematical operations [14];
5. Use of Intel®Cilk™Plus array notation and elemental function syntax. These notations can be used separately from each other to help the compiler perform parallelisation. More details to using these language extensions can be found in [15].

Compiler auto-vectorization is the easiest form of vectorization, where specific portions of the sequential code are converted, by the compiler, into equivalent parallel ones intended for use on vector processors. This is the most convenient way to do vectorization because cross-platform porting is preserved in contrast to embedding SIMD assembly code into a source program, which is rather complicated because it requires careful handling of data transfers between available vector registers and memory. This latter method may deliver better performance than the compiler auto-vectorization, but cross-platform porting is not guaranteed [14]. It is chosen here because it seems to be the most effective one for speeding up the AC algorithm.

In SIMD processing, the same operation is applied to each element of the source operands. However, this is a constraint because programs often do not map well onto pure vector operations and even if it is the case, they might not reach the best resource utilisation [16]. Programs can benefit from SIMD processing if they have highly repetitive loops, and use intensively instructions that operate on independent values in parallel. The practical speed-up with vectorization comes from the efficient data movement and the identification of vectorization possibilities in the program itself [17]. Vectorization fails if the structure of the program is not suitable for parallelisation and its data types are either mixed, dependent or not aligned. Intel processors implement architectures that include data parallelism in the form of a vector instruction set. Common examples include MMX™, SSE (Streaming SIMD Extensions) and AVX instructions. In the next section, we introduce Intel®AVX used to vectorize the AC algorithm.

3. Advanced Vector Extensions (AVX). For Intel®processors, the vector instructions have been gradually introduced in different processor generations. Started with the MMX™in 1996 (instructions in this extension operate only on packed integer values and rarely used in modern processors), followed by several SSE versions from 1999 to 2008 (available in almost any today processor), and continued to this day with AVX (supported in new processors).

AVX is a 256-bit SIMD operations extension of Intel®SSE, designed to deal with applications which make a more intense use of floating-point operations (scientific applications, visual processing, data mining, cryptography, 3D modeling and gaming). In this extension, the support for integer operands is lacking. It was released in the early 2011s as part of the second-generation Intel®Core™processor family (supported first by the Intel®Sandy Bridge processor released in Q1, 2011). AVX extends the previous SIMD instructions by expanding the 128-bit SIMD registers to 256 bits, adds a three-operand nondestructive operation where the destination register is different from the two source operands, and introduces a new prefix coding scheme VEX in instruction encoding format [18]. Processors based on the Sandy Bridge microarchitecture and supporting AVX include Core i7, Core i5, and Core i3 second- and third-generation processors along with Xeon series E3, E5, and E7 processors [19].

AVX2 also known as Haswell New Instructions was released in 2013 with the fourth generation Intel®Core™ processor family (supported first by the Intel®Haswell processor released in Q2, 2013), and is designed to support integer computation demanding algorithms by extending SSE and AVX with 256-bit integer instructions. AVX2 includes the Fused Multiply-Add (FMA) extension which allows numbers to be multiplied and added in one operation, enhances vectorization with *Gather* operation that loads vector items from non-contiguous memory locations, and introduces vector Shift operations. The specification of the AVX2 instruction set and information needed to create applications using this extension are available under the Intel Architecture Instruction Set Extensions Programming Reference [20]. Processors based on the Haswell microarchitecture and supporting AVX2 include Core i7, Core i5, and Core i3 fourth generation processors and Xeon E3 (v3) series processors (based on the Haswell microarchitecture) [19].

AVX-512 consists of multiple extensions of the AVX and AVX2 family of SIMD instructions, announced by Intel®in July of 2013. It uses a new EVEX prefix encoding with support for 512-bit vector registers, and offers a high level of compatibility with AVX.

In the vectorization process of the AC algorithm proposed next, we use AVX2 extension to benefit from SIMD instructions operating on integer data type since states generated in the pattern-matching machine by running the AC algorithm are coded as integer values.

4. Vectorization process of the Aho-Corasick algorithm.

In this section, we first describe the two versions of the AC algorithm, the standard version and the version that uses the *next-move* function as proposed by the authors in [3]. Then, we present an optimized version of the AC algorithm, which is obtained through vectorizing the AC algorithm that uses the *next-move* function.

4.1. Review of the AC algorithm.

The AC algorithm, first described by Alfred V. Aho and Margeret J. Corasick at Bell laboratories in 1975 [3], is a direct extension of the Knuth-Morris-Pratt (KMP) algorithm [21], which uses automata approach to solve multiple pattern-matching problems. Initially designed to accelerate the library bibliographic search program. The performance gained using this algorithm was between 5x and 10x faster compared to the original program [3].

The AC algorithm builds a finite state machine (FSM) from patterns, then uses the pattern-matching machine to locate all occurrences of patterns (that may overlap) within a given text in one pass. Each state of the machine is identified by a number (an integer value). When a character of an input text is processed, one or more finite automaton state transitions are made. State transitions (or machine behaviour) are dictated by three functions, namely: (i) the *goto* function g indicates state transitions by mapping a pair (current state, input character) into either a state or a fail state. In the latter case, the fail state is indicated by calling the failure function. Some of these states are designed as output states indicating that a set of patterns has been found, (ii) the *failure* function f defines which state transition to move to in case of a mismatched character, (iii) the *output* function indicates the patterns found and their locations in the text when the machine reaches an output state.

We can better describe how the algorithm works by considering an example. Suppose we want to search a text for the set of patterns {these, this, the, set}. An appropriate state machine for these four (04) patterns is described in the Goto, Failure and Output tables as shown in Fig 4.1. We start with an empty *goto* function ($all\ characters \rightarrow S0$), and then we add all patterns, one by one. In the same time, we construct the Output table. In the second step, we compute the *failure* function and update the Output table as described next.

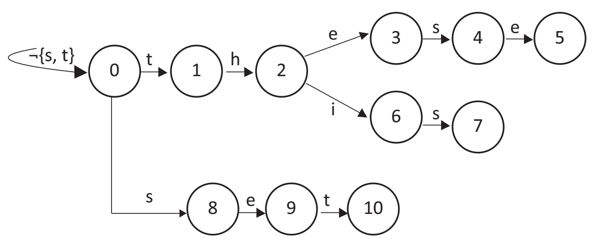
Once the automaton is built, the matching process is straightforward using the above-mentioned functions. The machine inspects all characters from the beginning of the text successively, one character at a time, by changing the state of the automaton (*goto* function is operating) and occasionally emitting output (*output* function is operating). If there is no valid state transition for the character under inspection, then the machine detects a transition failure and tries to investigate a match from other states (*failure* function is operating). We start in state 0, and then we examine each character of the text. For example, if a character t is seen in state 0, then we move to state 1. An h character would then take us to state 2. However, if the next character is not an e or i, we consult Fail [2] and change to state 0. Note that from state 2, Goto[2] ['e'] = 3 and Goto[2] ['i'] = 6, but Goto [2][anything else] = FAIL_STATE. If we arrive at a state with a non-empty Output function, then a matching string is found. The corresponding c++ code is given in listing 1.

Listing 1
C++ implementation of algorithm 1 - Pattern-matching maching according to [3]

```
//str is a pointer to the text to search
//N is the number of the characters to be examined (size of the text)

void AC_Search(unsigned char *str, int N)
{
   int state = 0;
   for (int i = 0; i<N; i++)
   {
```

a) a Goto function



All transitions from state 0 are defined.
All undefined transitions are to a state known as FAIL_STATE.

b) a Failure function

state	1	2	3	4	5	6	7	8	9	10
Fail [state]	0	0	0	8	9	0	8	0	0	1

c) an Output function

state	Output [state]
3	the
5	these, the
7	this
10	set

Fig. 4.1. *Pattern-matching machine for the set of patterns* {*these, this, the, set*}

```
        unsigned char c = str[i];
        while(Goto[state][c] == FAIL_STATE) {state = Fail[state];}
        state = Goto[state][c];
        //output
         if (Output[state] != NULL) {//Here we report a match}
    }
}
```

In the following, we construct the various tables of the AC algorithm. First, we construct the Goto and Output tables. We start with an empty Goto table and step by step we process each word from the pattern set by keeping track of the state transition made in the Goto table when examining each character. Where possible, we overlay patterns that begin with the same characters. We also construct the Output table at the same time: each pattern is identified within the Output table by using its output state as an index to this table. For example, to build the Goto table in Fig 4.1, we start with the pattern *these*$(S0 — t \rightarrow S1 — h \rightarrow S2 — e \rightarrow S3 — s \rightarrow S4 — e \rightarrow S5)$. Next, we add the pattern *this*$(S2 — i \rightarrow S6 — s \rightarrow S7)$, and finally the pattern *set*$(S0 — s \rightarrow S8 — e \rightarrow S9 — t \rightarrow S10)$. The pattern *the*$(S0 — t \rightarrow S1 — h \rightarrow S2 — e \rightarrow S3)$ overlays with the pattern *these*, which is already processed. We just consider its output state in the Output table. Then, we construct the Fail table by taking into account the overlapping search patterns, and the possibility that a given state might find multiple search patterns. We proceed by examining all states of the Goto table by defining "depth" to be the distance of state x from state 0. Then, states 1 and 8 are at depth 1, while states 2 and 9 are at depth 2. Intuitively, Fail[depth 1 states] should bring us to state 0. But Fail [depth 2 states] depends on both Fail and Goto functions of the depth 0 and 1 states. In general terms, Fail[depth n states] depends on the Fail and Goto functions of all states of lesser depth. Consider, for example, the computation of Fail [10] shown in Fig 4.1. The character that brought us from state 3 to state 4 was *s*. Now, we search all other transitions from state 0 to state 3 that make use of *s*. We find just one *s* that takes us from state 0 to state 8. Therefore, Fail [10] is set to 8. Note how the calculation of Fail has turned up the overlap of the patterns *these* and *the*. Output is updated at the same time. When an overlap is discovered, we need only to update (by concatenating) the content indexed by the two overlapping states.

Finally, we must deal with issues that may affect the performance of the algorithm such as its searching speed and the memory requirements. Indeed, we must define the number of possible states, stored as integers, that our machine might have *(#define MAXSTATES xx)*. This number depends on the number and the length of the patterns to look for. For example, in case of searching an English text for a set of 10 short patterns (words) having length less or equal 18 characters, MAXSTATES is set to about 80. While searching for a set of 10 long patterns (sentences) having length between 19 and 70 characters, MAXSTATES will be set to about 600 (see the experience results reported in Tables 5.2 and 5.3). The next issue is how to store the state transitions of the Goto function. This would involve a jump table for each state, which depends on the type of the search being performed: if we are searching an English text for some words (patterns are subset of 256 characters), then the table would contain a next state for each ASCII character (among 256 possibilities), and the Goto table will be defined as follows *(int Goto[MAXSTATES][256];)*. While searching DNA sequences for DNA keywords (patterns are subset of 4 characters A, C, G and T), the Goto table will be declared as follows *(int Goto[MAXSTATES][4];)*. Finally, when dealing with virus signatures based on byte stream (patterns are subset of hexadecimal characters and wildcards), defining the Goto table as *(int Goto[MAXSTATES][25];)* is more than enough. Notice that, the memory requirements of the AC algorithm can be taken directly from the different tables used in constructing the automaton during the pre-processing phase since it is the only structure used in the matching process. Unfortunately, the space complexity can be quite large depending on the alphabet and the patterns set. In the worst case it would be O(mc) where c is the size of the alphabet Σ and m is the total length of the patterns.

4.2. Review of the AC algorithm using the *next-move* function. The failure function f as implemented in the previous AC algorithm is not optimal. Consider the pattern-matching machine of Fig 4.1. We see Goto[10] ['e'] = 8. If the machine is in state 4 and the current input character is not an *e*, then the machine would enter state Fail[10] = 8. Since the machine has already determined that an input character is not an *e*, it does not need to consider the value of the goto function of state 8 on *e*. In fact, if the pattern *set* was not present, then the machine could change directly from state 4 to state 0. Thus, skipping an intermediate transition to state 0. To eliminate unnecessary failure transitions in the previous version of the AC algorithm (the standard version), the *next-move* function of a deterministic finite automaton (DFA) can be used in place of the goto and failure functions. Converting the algorithm to a DFA allows to indicate for each pair (given state, given character) the next state to move to, which is always a valid state.

The *next-move* function is computed from the goto and failure functions using algorithm 4 in [3]. The purpose of this function is to build the three (03) following tables: len_movef, movef_char and movef_nextstate (indicating respectively for each state the number of characters to process, the character being processed and the next state in case of a match) used by the algorithm. The *next-move* function is coded below as follows. For example, in state 0, we have a transition on *s* to state 8, a transition on *t* to state 1, and all other transitions on any other characters are to state 0 (not shown in Fig 4.2). The c++ code for the *next-move* function is given in listing 2.

Listing 2
C++ implementation of the AC algorithm using the next-move function according to [3]

```
//len_movef:table indicating the number of characters to process for each state
//movef_char: table indicating the character being processed for a given state
//movef_nextstate: table indicating the next state if a match occurs

void AC_Nextmove_search(unsigned char *str, int N)
{
   int state = 0;
   for (int j = 0; j<N; j++)
   {
      unsigned char c = str[j];
      for (int i = 0; i < len_movef[state]; i++)
   {
      if (c == movef_char[state][i])
      {
         state = movef_nextstate[state][i];
         goto nextOK;
      }
   }
   state = 0;
   nextOK:

   //Statement used in [3] and can be skipped if we want to do output later
   if (Output[state] != NULL) { //Here we report a match }

   //Here we use an array to do output after processing the whole text
   OutCount[state] += OutPutf[state]
   }
}
```

Theoretically, the DFA can reduce up to 50% of state transitions, which would reduce extra comparisons and accelerate significantly the matching process. In practice such accelerations cannot be reached since the pattern-matching machine spends a lot of its time in state 0, from which there are no failure transitions [3] as shown in our experience results reported in Tables 5.1, 5.2 and 5.3.

From the above pseudo-code and algorithms 1, 2 and 3 given in [3], it appears clearly that building the different tables is the difficult task of the AC algorithm. The searching process is easy once these tables are available. This method of encoding state transitions is more economical than storing them as a two-dimensional array. However, it requires extra memory space larger than the corresponding representation for the goto function. Using the *next-move* function with the set of patterns {these, this, the, set} would make the sequence of state transitions shown in Fig 4.2.

4.3. Simultaneous searching of multiple patterns using AVX2. One effective way to benefit from different levels of parallelism included in modern processors is the use of vectorization. The program to optimise has to be written using available vector registers (XMM, YMM and ZMM in case of Intel®processors) and SIMD instructions (MMX^{TM}, Streaming SIMD Extensions SSE and AVX instructions in case of Intel®processors).

In this section, we try to identify vectorization opportunities from the structure of the AC program (its code structure), we particularly look to parallelise some data processing. The second version of the AC algorithm, using the *next-move* function, seems vectorizable. Considering the sequence of state transitions in Fig 4.2. Each time the machine enters a new state, a text character being inspected, has to be matched against all characters of that state transitions one at a time. One way to speed up the searching process is by comparing simultaneously the text character to all characters of that state transitions. For example, if the machine enters state S2 then the character being examined is concurrently compared to e, i, t and s characters (data-level parallelism). This can be achieved using vector processing techniques.

To implement the vectorized algorithm, we use VPBROADCASTB and VPCMPEQB Intel®AVX2 instructions. The VPBROADCASTB instruction loads a byte integer (a text character being processed) from memory via ECX and EDX registers and broadcasts it to thirty-two (32) locations in YMM1 register (here we are coding in 32-bit mode). Next, we load all characters of state transitions of a current state in YMM2 register using VMOVUPS instruction. Finally, a SIMD comparison is performed between these two registers YMM1 and YMM2 using VPCMPEQB instruction, and the result is given in YMM3 register. The next state is indicated by using the value of YMM3 register as an index to another table, namely movef_nextstate table. This parallel matching process, as shown in Fig 4.3, is repeated until all characters of an input text are processed.

Note that when the number of patterns of a given state (max = 32 patterns/state) is not important, only few bytes of YMM2 register will be used. In this case, as shown in the example of Fig 4.3, we can write code to avoid comparing YMM1 register bytes ranging from byte 4 to byte 31 (having "ch$_i$" value) to their counterparts in YMM2 register (having zero value), but this is useless since the speed of comparison will not be affected

```
state 0: ( length = 2 )
        s - next state: 8
        t - next state: 1

state 1: state 10: ( length = 3 )
        h - next state: 2
        t - next state: 1
        s - next state: 8

state 2: ( length = 4 )
        e - next state: 3
        i - next state: 6
        t - next state: 1
        s - next state: 8

state 3: ( length = 2 )
        s - next state: 4
        t - next state: 1

state 4: ( length = 3 )
        e - next state: 5
        t - next state: 1
        s - next state: 8

state 5: state 9: ( length = 2 )
        t - next state: 10
        s - next state: 8

state 6: ( length = 2 )
        s - next state: 7
        t - next state: 1

state 7: state 8: ( length = 3 )
        e - next state: 9
        t - next state: 1
        s - next state: 8
```

Fig. 4.2. *State transitions using the next-move function*

(remains the same) because we are doing SIMD comparison. The vectorized version of the AC program using the *next-move* function is given in listing 3.

Listing 3
AVX2 implementation of the AC algorithm using the next-move function

```
// Here we convert the scalar AC algorithm in listing 2 to to a vector process

_declspec(naked) void __fastcall AC_NextMove_search_avx2(unsigned char *strp,int N)
{
    // fastcall ==>     edx == N       ecx == strp     eax    ==  last param
   _asm push ebp
   _asm mov ebp, esp
   _asm push esi
   _asm push ebx
   _asm push edi

   _asm  add edx, ecx  // edx == N+strp
   _asm  xor esi, esi  // esi == state   == 0

 loopj :
   _asm VPBROADCASTB ymm1, [ecx]      // ymm1 size = 32 bytes(characters)
   // first we start by coding the loopi code fragment in listing 2:
   // (1) for (int i = 0; i < len_movef[state]; i++) { ......}
   // (2) if (c == movef_char[state][i]) { ..(3)....}
   _asm lea ebx, movef_char
   _asm mov eax, esi
   _asm shl eax, 8   //edx == state*256 (256 bytes-line size in "movef_char" array)
   _asm vmovups ymm2, [eax + ebx]
   _asm vpcmpeqb ymm3, ymm2, ymm1
             //comparison result in ymm3   0xff 0x00 0x00 0xff .....  0xff 0x00
   _asm vpmovmskb eax, ymm3
             //comparison result in eax      1    0    0    1  .....    1    0
```

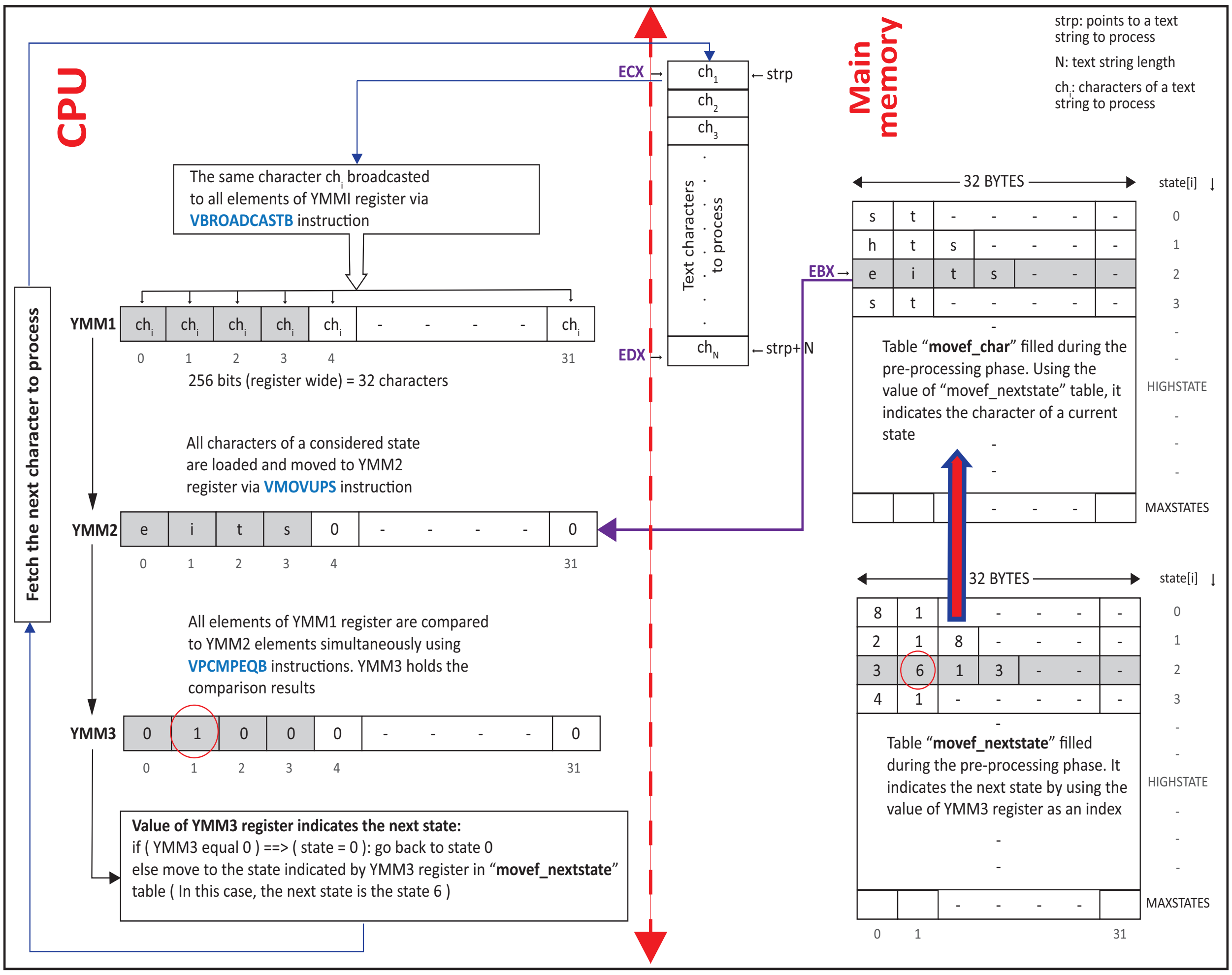

CPU
Main memory
strp: points to a text string to process
N: text string length
ch$_i$: characters of a text string to process
ECX
EDX
EBX
ch$_1$
ch$_2$
ch$_3$
Text characters to process
ch$_N$
strp
strp+N
The same character ch$_i$ broadcasted to all elements of YMMI register via VBROADCASTB instruction
YMM1
256 bits (register wide) = 32 characters
All characters of a considered state are loaded and moved to YMM2 register via VMOVUPS instruction
YMM2
All elements of YMM1 register are compared to YMM2 elements simultaneously using VPCMPEQB instructions. YMM3 holds the comparison results
YMM3
Value of YMM3 register indicates the next state:
if (YMM3 equal 0) ==> (state = 0): go back to state 0
else move to the state indicated by YMM3 register in "movef_nextstate" table (In this case, the next state is the state 6)
Fetch the next character to process
32 BYTES
state[i]
Table "movef_char" filled during the pre-processing phase. Using the value of "movef_nextstate" table, it indicates the character of a current state
HIGHSTATE
MAXSTATES
Table "movef_nextstate" filled during the pre-processing phase. It indicates the next state by using the value of YMM3 register as an index

FIG. 4.3. *Vectorization process of the AC algorithm using AVX2 instructions*

```
  _asm bsf eax, eax //i == eax  ==                                     2

  //for a given state, the number of state transitions <=32 (len_movef[state]<=32)
  _asm jz  nonzero              // bsf instruction (if flagzero == 1)  then jump
      // (3) {state = movef_nextstate[state][i]; goto nextOK;}
      // state  = state*256, 256 bytes - size of a line in "movef_nextstate" array
  _asm shl esi, 8
  _asm add esi, eax              //state  = state + i
   _asm lea edi, movef_nextstate //state = movef_nextstate[state][i];
  _asm mov esi, [edi + esi*4]    //4 bytes-"movef_nextstate" array element size
  goto nextOK;

nonzero:
  _asm xor esi,esi

nextOK :  // OutCount[state] += OutPutf[state];
  _asm mov eax, OutPutf[  esi * 4]    //OutPutf must be static
  _asm add OutCount[esi * 4],  eax    //OutCount must be static

  _asm inc ecx
  _asm cmp ecx,edx
  _asm jle loopj

  _asm pop edi
  _asm pop ebx
```

```
   _asm pop esi
   _asm pop ebp
   _asm ret
}
```

5. Experimental results. In this section we evaluate the performance of the vectorized AC algorithm by comparing it to the AC algorithm using the *next-move* function as proposed by the authors in [3]. In particular, we compare the running time of the AC_NextMove_search_AVX2 program to the AC_NextMove_search _ASM program (the AC algorithm using the *next-move* function before vectorizing it coded in assembly, not listed in this paper). Comparison is done by including the time of the pre-processing phase.

The programs have been compiled with Microsoft Visual Studio Ultimate 2013 (version 12.0.21005.1 REL) and experiments were executed on LENOVO laptop, a 1.9 GHz CPU Intel I3-4030U with 4 GB RAM running Microsoft Windows 8.1 Pro 64 bits.

For the evaluation purpose, we first use different files of plain English text, collected from the Gutenberg website (`http://www.gutenberg.org`), their total size is ranging from 1 MB to 1 GB, and different pattern size categorised as follows: (i) *sp* short patterns designing English words of length less or equal 18 bytes and, (ii) *lp* long patterns designing English phrases of length greater than 18 bytes. Patterns were arbitrarily chosen, some of them may exist or not in text files to be processed (especially when matching an important number of patterns).

Each input file is processed by running respectively AC_NextMove_search_ASM and AC_NextMove _search_AVX2 programs. During the matching phase, the text characters of each file is matched against either *sp* and *lp* patterns of different size. All functions of the pre-processing phase are coded in C++ language. Only the search function, which behaves as a finite state string pattern-matching machine is coded in assembly (in case of AC_NextMove_search_ASM program) and codded using AVX2 instructions (in AC_NextMove_search_AVX2 program). The pre-processing phase consists of the following functions: the *goto*, the *failure* and the *next-move* functions that implement respectively algorithms 2, 3 and 4 in [3].

Table 5.1 summarises the running times of the algorithms. Here we would point out that the vectorized AC algorithm, as implemented here, use only one YMM register (32 bytes length), which limits the number of state transitions to 32 (that can be reached once more than 100 patterns are used especially in case of long ones).

To deal with an important number of patterns, the AC_NextMove_AVX2 program has to be adjusted in a way that it uses a variable to store YMM value when the number of next states (state transitions) exceeds 32, or uses more than one YMM register instead. Another alternative is the use of ZMM registers (AVX- 512) where the width of the register is increased to 512 bits allowing to have up to 64 state transitions/register. These proposals will be investigated in depth in a future work.

The experimental results reported in Table 5.1 show that the AC_NextMove_AVX2 program performs better than its counterpart AC_NextMove_ASM in all cases. The speed up gained depends on the patterns length and the number of patterns to search for.

It is important to highlight the fact that in practice the AC algorithm, as stated by the authors in [3] and confirmed by our experiments, spends most of its time in state 0 (more than 90% of its processing time in most cases). From Table 5.1, we can also notice that the speed-up gained is independent on the file size.

In the second phase of our experiment, we use 10 different patterns (*sp* first then *lp* next having different lengths) to search for, through different file sizes, and we measure the speed-up gained in terms of time processing (in %) when executing the vectorized AC algorithm (see Tables 5.2 and 5.3). Here, we give a theoretical value (TheoV) of the speed-up when comparing the running time of the A_NextMove_AVX2 algorithm against the AC_NextMove_ASM algorithm. This value is calculated by incrementing a variable inside an inner loop in AC_NextMove_search function and dividing its value by the number of the processed characters at the end of the outer loop (which is the size of the input file). When considering these values, it is clearly noticeable from Tables 5.2 and 5.3 that the speed-up gained is independent on the file size.

Table 5.4 shows the differences, from the results of experiments, between our implementation of the AC algorithm and some of the implementations discussed here.

6. The vectorized AC code portability. As mentioned earlier, the vectorized AC code consists up of the following functions:

Table 5.1
Running times obtained while searching different text file size for sets of different patterns

Text file size (KB)	Number of patterns	Total patterns length (Bytes)	T1(sec) AC NextMove (ASM)	T2(sec) AC NextMove (AVX2)	Speed-up(%) [AVX2/ ASM]	Time spent in state 0 (%)
1125 (1MB)	1 sp	8	0,005340	0,004020	32,84	99,43
	10 sp	53	0,014807	0,005675	160,92	96,90
	100 sp	632	0,043139	0,016071	168,43	76,96
	1 lp	55	0,004349	0,002532	71,76	99,80
	10 lp	578	0,013965	0,004723	195,68	98,95
	100 lp	3914	0,043059	0,018160	137,11	74,27
9690 (10MB)	1 sp	8	0,036198	0,022953	57,70	99,46
	10 sp	53	0,130331	0,054303	140,01	97,01
	100 sp	632	0,360853	0,142437	153,34	76,94
	1 lp	55	0,041455	0,024482	69,33	99,77
	10 lp	578	0,125042	0,044071	183,73	98,76
	100 lp	3914	0,303146	0,151035	100,71	76,07
102240 (100MB)	1 sp	8	0,479668	0,368961	30,01	99,49
	10 sp	53	1,444595	0,592323	143,89	97,25
	100 sp	632	4,546556	1,694503	168,31	77,31
	1 lp	55	0,385994	0,193441	99,54	99,75
	10 lp	578	1,399632	0,509368	174,78	98,99
	100 lp	3914	3,659735	1,775760	106,09	76,03
511955 (500MB)	1 sp	8	2,262958	1,691504	33,78	99,49
	10 sp	53	5,409291	2,541107	112,87	97,26
	100 sp	632	22,958459	8,336736	175,39	77,35
	1 lp	55	2,090559	0,968195	115,92	99,77
	10 lp	578	7,656948	2,818840	171,63	99,05
	100 lp	3914	15,797668	8,034843	96,61	76,22
1048906 (1GB)	1 sp	8	4,163488	2,823843	47,44	99,49
	10 sp	53	12,267914	5,024411	144,17	97,25
	100 sp	632	40,054606	15,947976	151,16	77,34
	1 lp	55	3,991538	1,955866	104,08	99,76
	10 lp	578	12,254003	4,495377	172,59	99,05
	100 lp	3914	32,940271	16,499932	99,64	76,22

(i) The *goto*, the *failure* and the *next-move* functions (executed during the pre-processing phase of the AC algorithm), which are all coded in C++ language;
(ii) The search function *AC_NextMove_search_AVX2* (the pattern-matching machine), which is vectorized by explicitly calling the processors SIMD instructions (AVX2) using low-level assembly code and available registers.

The three functions of the pre-processing phase should not pose portability issues since they are coded and compiled from a high-level language. The search function *AC_NextMove_search_AVX2*, as implemented, is supported by Intel®processors (such as Haswell, Broadwell, Skylake, Kaby Lake, Skylake-X, Coffee Lake, and Cannon Lake processors), AMD Excavator, Zen, and Zen+ processors, and later manufacturers' processor x86 microarchitectures (implementations of the x86 ISA) such as AMD Zen2, and Intel®Cascade Lake processors expected in 2019.

To overcome the portability issues, vector operations can be implemented by the compiler using automatic vectorization, after taking care of organizing data and loops in a convenient way. This is the most recommended method to employ vector instruction support, because cross platform porting is provided by the compiler.

7. Conclusion. In this paper, we have presented the vectorized Aho-Corasick algorithm in attempting to speed up the string-matching process. Experimental results clearly show that the vectorized version of the Aho-Corasick algorithm, using the *next-move* function, yields better performance in terms of speed whose running time was up to five ($\times 5$) of its original counterpart (not vectorized). The speed-up gained depends on the number of the patterns to search for and their length but not on the file size.

Table 5.2
Running times obtained while searching different file size for sets of 10 different short patterns

Text file size (KB)	Total patterns length (Bytes)	Number of States (MAXSTATES)	Time spent in state 0 (%)	T1(sec) AC NextMove (ASM)	T2(sec) AC NextMove (AVX2)	Speed-up(%) [AVX2/ ASM]	Speed-up(%) [AVX2/ ASM] TheoV
10 MB	48	39	98,23	0,073203	0,044321	65,17	484,49
	50	38	98,42	0,046869	0,029719	57,71	302,42
	53	45	98,33	0,082621	0,048733	69,54	410,67
	53	50	97,02	0,098311	0,047394	107,43	570,69
	65	51	96,44	0,081480	0,048452	68,17	472,95
	82	73	93,83	0,118258	0,085747	37,92	454,41
100 MB	48	39	98,29	0,085187	0,046276	84,08	484,79
	50	38	98,55	0,631662	0,412058	53,29	302,62
	53	45	98,46	0,836997	0,475502	76,02	411,45
	53	50	97,25	1,362945	0,580876	134,64	572,15
	65	51	96,41	0,961730	0,558828	72,10	472,60
	82	73	93,44	1,574822	1,024999	53,64	452,15
1 GB	48	39	98,30	11,628367	4,678585	148,54	484,93
	50	38	98,56	5,675574	3,606079	57,39	302,65
	53	45	98,46	9,870739	5,026897	96,36	411,53
	53	50	97,26	12,450564	5,123501	143,01	572,14
	65	51	96,47	11,870859	5,670317	109,35	472,86
	82	73	93,43	14,372971	9,443510	52,20	452,22

Table 5.3
Running times obtained while searching different file size for sets of 10 different long patterns

Text file size (KB)	Total patterns length (Bytes)	Number of States (MAXSTATES)	Time spent in state 0 (%)	T1(sec) AC NextMove (ASM)	T2(sec) AC NextMove (AVX2)	Speed-up(%) [AVX2/ ASM]	Speed-up(%) [AVX2/ ASM] TheoV
10 MB	560	558	98,72	0,204947	0,061213	234,81	884,82
	466	465	98,90	0,186187	0,048387	284,79	886,51
	409	402	95,22	0,134060	0,052976	153,06	767,76
	565	557	99,32	0,160171	0,028548	461,06	693,44
	373	362	94,65	0,078206	0,044593	75,38	577,11
	327	324	91,73	0,197331	0,099463	98,40	694,54
100 MB	560	558	98,97	1,965827	0,623113	215,48	887,10
	466	465	99,20	2,178835	0,612167	255,92	889,77
	409	402	94,77	2,002379	0,649050	208,51	765,40
	565	557	99,54	1,389077	0,344041	303,75	695,20
	373	362	94,24	1,107682	0,562100	97,06	575,62
	327	324	91,17	1,847791	1,027445	79,84	689,78
1 GB	560	558	99,02	18,291795	5,667745	222,73	887,74
	466	465	99,26	20,000147	5,505017	263,31	890,33
	409	402	94,79	17,471766	6,047175	188,92	765,57
	565	557	99,59	14,13522	2,825973	400,19	695,48
	373	362	94,25	13,016356	5,959448	118,42	575,69
	327	324	91,26	16,198667	9,344094	73,36	690,75

The use of vectorization results on a higher return in performance and efficiency that depend on the code structure. But, once again, the programmer must examine at first the structure of the code to vectorize in depth to identify parts (code fragments) that offer vectorization opportunities, and make the data structures in the code nearly fit the structure built into the hardware.

Table 5.4
Comparison of related work

Optimized implementations of the AC algorithm	Speed-up of Aho-Corasick Pattern Matching Machines by Rearranging States [6]	String matching with multicore CPUs: Performing better with the Aho-Corasick algorithm [11]	SIMD implementation of the Aho-Corasick algorithm using Intel AVX2 (Our implementation)
Experiment results	The elapsed time of string pattern-matching is reduced to 71(%) (Speedup 142(%)) in case of a random text and 55(%) (Speedup 110(%)) in case of a compressed text	Throughput ratio for 10MB pattern set (DNA sequences – short patterns) for 70 KB input size is 4.62 and for 30MB pattern set is 12.25. The throughput ratio is increasing with the increasing pattern size *(Throughput = Number of bytes in the input/ Total time for processing).*	Speedup between 30(%) and 195(%) while searching different text file size for sets of different patterns. Speedup between 37(%) and 148(%) while searching different text file size for sets of different short patterns. Speedup between 73(%) and 461(%) while searching different text file size for sets of different long patterns.

REFERENCES

[1] Corman, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2002). Introduction to Algorithms-String matching. *EEE Edition, 2nd Edition, Page,* (906-907).

[2] Binstock, A., & Rex, J. (1995). Searching. *Practical Algorithms for Programmers* (pp. 95-171). Addison-Wesley Longman Publishing Co., Inc..

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

[4] Wu, S., & Manber, U.(1994). *A fast algorithm for multi-pattern searching.* (pp. 1-11). University of Arizona. Department of Computer Science.

[5] Salmela, L., Tarhio, J., & Kytjoki, J. (2006). Multi pattern string-matching with q-grams. *In the proc. of Journal of Experimental Algorithmic (JEA)*, 11, 1-19.

[6] Commentz-Walter, B. (1979, July). A string matching algorithm fast on the average. In *Proceeding of the 6th International Colloquium on Automata, Languages, and Programming* (pp. 118-132). Springer, Berlin, Heidelberg.

[7] Aho, A. V. (1990). Algorithms for finding patterns in strngs, Handbook of Theoretical Computer Science Vol A. *A, ed. J. van Leeuwen, ElsevierSciencePublishersB, 1990,* 257-297.

[8] Jony, A. I. (2014). Analysis of multiple string pattern matching algorithms. *International Journal of Advanced Computer Science and Information Technology (IJACSIT)*, 3(4), 344-353.

[9] Nishimura, T., Fukamachi, S., & Shinohara, T. (2001, November). Speed-up of Aho-Corasick pattern matching machines by rearranging states. In *Proceedings Eighth Symposium on String Processing and Information Retrieval* (pp. 175-185). IEEE.

[10] Lin, C. H., Tsai, S. Y., Liu, C. H., Chang, S. C., & Shyu, J. M. (2010, December). Accelerating string-matching using multi-threaded algorithm on GPU. In *Proceedings of IEEE Global Telecommunications Conference GLOBECOM 2010* (pp. 1-5). IEEE.

[11] Arudchutha, S., Nishanthy, T., & Ragel, R. G. (2013, December). String matching with multicore CPUs: Performing better with the Aho-Corasick algorithm. In *2013 IEEE 8th International Conference on Industrial and Information Systems* (pp. 231-236). IEEE.

[12] Intel (2016). An introduction to Vectorization with the Intel C++ Compiler. `https://software.intel.com/sites/default/files/An\_Introduction\_to\_Vectorization\_with\_Intel\_C++\_Compiler\_021712.pdf`. [Online; accessed 02 February 2016].

[13] Gtzfried, J. (2012). Advanced Vector Extensions to accelerate crypto primitives. *Friedrich-Alexander-Universitt, Erlangen-Nrnberg. Bachelor Thesis.*

[14] Vladimirov, A., Asai, R., & Karpusenko, V. (2015). ExpressingParallelism. *Parallel Programming and Optimization with Intel®XEON PHI^{TM}Coprocessors* (2nd ed., pp. 157-158). Colfax International.

[15] Oxford e-Research Centre (2016). AVX Vectorisation and Cilk Plus. `http://www.oerc.ox.ac.uk/projects/asearch/software/avx`. [Online; accessed 26 April 2016].

[16] Page, D. (2009). Advanced Processor Design. *A Practical Introduction to Computer Architecture* (pp. 389-390). Springer Science & Business Media.

[17] Jeffers, J., Reinders, J., & Sodani, A. (2016). Introduction. *Intel®Xeon Phit Coprocessor High Performance Programming. Knights Landing Edition.* (pp. 11-12). Morgan Kaufmann.

[18] Intel (2016). Intel®Advanced Vector Extensions (Intel®AVX).`https://software.intel.com/en-us/isa-extensions/intel-avx`. [Online; accessed 07 January 2016].

[19] Kusswurm, D. (2014). X86-32 Core Architecture. *Modern X86 Assembly Language Programming: 32-bit, 64-bit, SSE, and AVX.* (pp. 01-03). Apress.

[20] Intel (2016). Intel®Architecture Instruction Set Extensions Programming Reference. https://www.naic.edu/\~\phil/software/intel/319433-014.pdf. [Online; accessed 06 January 2016].

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

Edited by: Dana Petcu
Received: June 6, 2019
Accepted: September 1, 2019