Tarhio, Jorma; Holub, Jan; Giaquinta, Emanuele

Technology beats algorithms (in exact string matching)

Published in:
SOFTWARE-PRACTICE AND EXPERIENCE

DOI:
10.1002/spe.2511

Published: 01/01/2017

Document Version
Publisher's PDF, also known as Version of record

Please cite the original version:
Technology beats algorithms (in exact string matching)

Jorma Tarhio1 | Jan Holub2 | Emanuele Giaquinta3

1Department of Computer Science, Aalto University, Espoo, Finland
2Department of Theoretical Computer Science, Faculty of Information Technology, Czech Technical University in Prague, Prague, Czech Republic
3F-Secure Corporation, Helsinki, Finland

Summary
More than 120 algorithms have been developed for exact string matching within the last 40 years. We show by experiments that the naïve algorithm exploiting SIMD instructions of modern CPUs (with symbols compared in a special order) is the fastest one for patterns of length up to about 50 symbols and extremely good for longer patterns and small alphabets. The algorithm compares 16 or 32 characters in parallel by applying SSE2 or AVX2 instructions, respectively. Moreover, it uses loop peeling to further speed up the searching phase. We tried several orders for comparisons of pattern symbols, and the increasing order of their probabilities in the text was the best.

KEYWORDS
AVX2, exact string matching, SIMD instructions, SSE

1 | INTRODUCTION

The exact string matching is one of the oldest tasks in computer science. The need for it started when computers began processing text. At that time, the documents were short and there were not so many of them. Now, we are overwhelmed by amount of data of various kind. The string matching is a crucial task in finding information, and its speed is extremely important.

The exact string matching task is defined as counting or reporting all the locations of given pattern \( p \) of length \( m = |p| \) in given text \( t \) of length \( n = |t| \) assuming \( m \ll n \), where \( p \) and \( t \) are strings over a finite alphabet \( \Sigma \). The first solutions designed were to build and run deterministic finite automaton (running in space \( \Theta(m|\Sigma|) \) and time \( \Theta(n) \)), the Knuth-Morris-Pratt automaton (running in space \( \Theta(m) \) and time \( \Theta(n) \)), and the Boyer-Moore algorithm (running in best case time \( \Omega(n/m) \) and worst case time \( \Theta(mn) \)). There are numerous variations of the Boyer-Moore algorithm like the previous studies. In total, more than 120 exact string matching algorithms have been developed since 1970.

Modern processors allow computation on vectors of length 16 bytes in case of SSE2 and 32 bytes in case of AVX2. The instructions operate on such vectors stored in special registers XMM0 to XMM15 (SSE2) and YMM0 to YMM15 (AVX2). As one instruction is performed on all data in these long vectors, it is considered as SIMD (Single Instruction, Multiple Data) computation.

Some authors call string matching with SIMD “packed string matching.” This term is somewhat misleading because the input is not packed but packing happens in SIMD operations. We think that “packed string matching” suits better to cases where the input is explicitly packed, eg, one byte represents 4 DNA characters.

2 | ALGORITHMS

2.1 | Naïve approach

In the naïve approach (shown as Algorithm 1), the pattern \( p \) is checked against each position in the text \( t \), which leads to running time \( \Theta(mn) \) and space \( \Theta(1) \). However, it is not bad in practice for large alphabets as it performs only 1.08 comparisons on
average on each character of $t$ for English text. The variable $\text{found}$ in Algorithm 1 is not quite necessary. It is presented to have a connection to the SIMD version to be introduced.

Like in the testing environment of Hume and Sunday\textsuperscript{7} and the SMART library,\textsuperscript{9} we consider the counting version of exact string matching. It can be easily transformed into the reporting version by printing position $i$ in line 11.

Using SIMD instructions (shown in Algorithm 2), we can compare $\alpha$ bytes in parallel, where $\alpha = 16$ in case of SSE2 or $\alpha = 32$ in case of AVX2 and “AND” represents the bit-parallel “and”. This allows huge speedup of a run.

For a given position $i$ in the text $t$, the idea is to compare the pattern $p$ with the $\alpha$ substrings $t[i + k..i + k + m - 1]$, for $0 \leq k < \alpha$, in parallel, in $O(m)$ time in total. To this end, we use a primitive SIMDcompare$(t[i + j - 1], p[j], \alpha)$, which, given a position $i$ in $t$ and $j$ in $p$, compares the strings $S_1 = t[i + j - 1..i + j - 1 + \alpha]$ and $S_2 = p[j]^\alpha$ and returns an $\alpha$-bit integer such that the $k$-th bit is set iff $S_1[k] = S_2[k]$, in $O(1)$ time. In other words, the output integer encodes the result of the $j$-th symbol comparison for all the $\alpha$ substrings. For example, consider the $\alpha$ leftmost substrings of length $m$ of $t$, corresponding to $i = 1$ (See Figure 1). For $j = 1$, the function compares $t[1..\alpha]$ with $p[1]^\alpha$, i.e., the first symbol of the substrings against $p[1]$. For $j = 2$, the function compares $t[2..\alpha + 1]$ with $p[2]^\alpha$, i.e., the second symbol against $p[2]$. Let $\text{found}$ be the bitwise and of the integers SIMDcompare$(t[i + j - 1], p[j], \alpha)$, for $j = 1, \ldots, m$. Clearly, $t[i + k..i + k + m - 1] = p$ iff the $k$-th bit of $\text{found}$ is set. We compute $\text{found}$ iteratively, until we either compare the last symbol of $p$ or no substring has a partial match (i.e., the vector $\text{found}$ becomes zero). Then, the text is advanced by $\alpha$ positions, and the process is repeated starting at position $i + \alpha$. For a given $i$, the number of occurrences of $p$ is equal to the number of bits set in $\text{found}$ and is computed using a popcount instruction. Reporting all matches in line 11 would add an $O(s)$ time overhead, as $O(s)$ instructions are needed to extract the positions of the bits set in $\text{found}$, where $s$ is the number of occurrences found.

\begin{algorithm}
\caption{SIMD-NAIVE-SEARCH algorithm}
\begin{algorithmic}[1]
\State $\text{found} \leftarrow 0$
\For{$i = 1$ to $n - m + 1$}
\State $\text{found} \leftarrow 1$
\For{$j = 1$ to $m$}
\State $\text{found} \leftarrow \text{found} \land (t[i + j - 1] = p[j])$
\EndFor
\If{$\text{found} = \text{false}$}
\State $\text{count} \leftarrow \text{count} + 1$
\EndIf
\EndFor
\EndFunction
\end{algorithmic}
\end{algorithm}

\begin{figure}
\centering
\begin{tabular}{cccccccccccc}
$i = 1, j = 4$: & d & d & d & d & d & \cdots & d & \hline
0 & 1 & 0 & 0 & 0 & 0 & \cdots & 0 \\
$i = 1, j = 3$: & e & c & c & c & c & \cdots & c & \hline
0 & 1 & 0 & 0 & 0 & 0 & \cdots & 0 \\
$i = 1, j = 2$: & b & b & b & b & b & \cdots & b & \hline
0 & 1 & 0 & 0 & 0 & 0 & \cdots & 0 \\
$i = 1, j = 1$: & a & a & a & a & a & \cdots & a & \hline
1 & 1 & 0 & 0 & 0 & 0 & \cdots & 0 \\
\end{tabular}
\begin{tabular}{cccccccccccc}
$t$: & a & a & b & c & d & \cdots & a & b & c & d & \cdots & d & b & c & d & \cdots & \hline
$i = \alpha + 1, j = 1$: & a & a & a & a & a & \cdots & a & \hline
1 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 \\
$i = \alpha + 1, j = 2$: & b & b & b & b & b & \cdots & b & \hline
1 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 \\
$i = \alpha + 1, j = 3$: & c & c & c & c & c & \cdots & c & \hline
1 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 \\
$i = \alpha + 1, j = 4$: & d & d & d & d & d & \cdots & d & \hline
1 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 \\
\end{tabular}
\end{figure}

\textbf{FIGURE 1} Example of comparisons for the text $t = abcd \cdots$ and pattern $p = abcd$ using the SIMD-NAIVE-SEARCH algorithm (alignment of pattern vector and vector $\text{found}$ to text $t$)
The 16-byte version of function SIMDcompare is implemented with SSE2 intrinsic functions as follows:

\[
\text{SIMDcompare}(x, y, 16) = \_\text{mm}\_\text{loadu\_si128}(x) \quad \text{\_\text{mm}\_\text{cmpeq\_epi8}}(\_\text{mm\_movemask\_epi8}(\_\text{mm\_cmpeq\_epi8}(x, y, 16)))
\]

Here \(s(y, 16)\) is the starting address of 16 copies of \(y\). The instruction \_\text{mm\_loadu\_si128}(x) loads 16 bytes (=128 bits) starting from \(x\) to a SIMD register. The instruction \_\text{mm\_cmpeq\_epi8} compares bytewise 2 registers, and the instruction \_\text{mm\_movemask\_epi8} extracts the comparison result as a 16-bit integer. For the 32-byte version, the corresponding AVX2 intrinsic functions are used. For both versions, the SSE4 instruction \_\text{mm\_popcnt\_u32} is used for popcount.

Algorithm 2 is closely related to the EPSMa algorithm\(^1\): both algorithms apply the same comparison method. However, the inner loops of the algorithms are different besides other differences. The loop of Algorithm 2 exits after \(\text{found}\) becomes zero, while the loop of EPSMa runs always \(\min(m, \alpha/2)\) rounds making EPSMa unsuitable for long patterns. Note that the implementation of EPSMa in the SMART Library\(^*\) does not contain any inner loop because it is applied only to patterns with at most 3 characters.

### 2.2 Frequency involved

To identify nonmatching positions in the text as fast as possible, individual characters of the pattern are compared to the corresponding positions in the text in the order given by their frequency in standard text. First, the least frequent symbol is compared, then the second least frequent symbol, etc. Therefore, the text type should be considered and frequencies of symbols in the text type should be computed in advance from some relevant corpus of texts of the same type. Hume and Sunday\(^7\) use this strategy in the context of the Boyer-Moore algorithm.

Algorithm 3 shows the naïve approach enriched by frequency consideration. A function \(\pi\) gives the order in which the symbols of pattern should be compared (ie, \(p(\pi(1)), p(\pi(2)), \ldots, p(\pi(m))\)) to the corresponding symbols in text. An array for the function \(\pi\) is computed in \(O(m \log m)\) time using a standard sorting algorithm on frequencies of symbols in \(p\).

\(^*\)http://www.dmi.unict.it/~faro/smart/, loaded in June 2016

---

**Algorithm 2**

1. function SIMD-NAIVE-SEARCH\((p, m, t, n, a)\)
2. \(\text{count} \leftarrow 0\)
3. for \(i \leftarrow 1\) to \(n - m + 1\) step \(a\) do
4. \(\text{found} \leftarrow 11.1\)
5. for \(j \leftarrow 1\) to \(m\) do
6. \(\text{found} \leftarrow \text{found} \text{AND SIMDcompare}(i + j - 1, p[j], a)\)
7. if \(\text{found} = 0\) then
8. \(\text{go to } 12\)
9. \(\text{end if}\)
10. \(\text{end for}\)
11. \(\text{count} \leftarrow \text{count} + \text{popcount(found)}\)
12. \(\text{end for}\)
13. \(\text{end function}\)

---

**Algorithm 3**

1. function FREQ-SIMD-NAIVE-SEARCH\((p, m, t, n, a)\)
2. \(\text{count} \leftarrow 0\)
3. for \(i \leftarrow 1\) to \(n - m + 1\) step \(a\) do
4. \(\text{found} \leftarrow 11.1\)
5. for \(j \leftarrow 1\) to \(m\) do
6. \(\text{found} \leftarrow \text{found} \text{AND SIMDcompare}(i + \pi(j) - 1, p[\pi(j)], a)\)
7. if \(\text{found} = 0\) then
8. \(\text{go to } 12\)
9. \(\text{end if}\)
10. \(\text{end for}\)
11. \(\text{count} \leftarrow \text{count} + \text{popcount(found)}\)
12. \(\text{end for}\)
13. \(\text{end function}\)
Hume and Sunday\(^7\) call this strategy *optimal match*, although it is not necessarily optimal. For example, the pattern “qui” is tested in the order ‘q’-‘i’-‘u’, but the order ‘q’-‘u’-‘i’ is clearly better in practice because ‘q’ and ‘u’ appear often together. Külekci\(^11\) compares optimal match with more advanced strategies based on frequencies of discontinuous \(q\)-grams\(^1\) with conditional probabilities. His experiments show that the frequency is beneficial in case of texts of large alphabets like texts of natural language. Computing all possible frequencies of \(q\)-grams is rather complicated, and the possible speedup to optimal match is likely marginal. Thus, we consider only simple frequencies of individual symbols.

### 2.3 Loop peeling

Guard test\(^7,12\) is a widely used technique to speed up string matching. The idea is to test a certain pattern position before entering a checking loop. Instead of a single guard test, 2 or even 3 tests have been used\(^13,14\). Guard test is a representative of a general optimization technique called loop peeling, where a number of iterations is moved in front of the loop. As a result, the loop becomes faster because of fewer loop tests. Moreover, loop peeling makes possible to precompute certain values used in the moved iterations. For example, \(p[\pi(1)]\) is explicitly known. In some cases, loop peeling may even double the speed of a string matching algorithm applying SIMD computation as observed by Chhabra et al.\(^15\)

In the following, we call the number of the moved iterations the peeling factor \(r\). We assume that the first loop test is done after \(r\) iterations. Thus, our approach differs from multiple guard test, where checking is stopped after the first mismatch. All \(r\) iterations are performed in our approach.

#### Algorithm 4

```plaintext
1: function LP-FRQ-SIMD-NAIVE-SEARCH(p, m, t, n, a)
2:     count ← 0
3:   for i ← 1 to n − m + 1 step a do
4:     found ← SIMDcompare(t[i + \pi(1) − 1], p[\pi(1)], a)
5:     found ← found AND SIMDcompare(t[i + \pi(2) − 1], p[\pi(2)], a)
6:     if found = 0 then
7:        go to 16
8:   end if
9:   for j ← 3 to m do
10:      found ← found AND SIMDcompare(t[i + \pi(j) − 1], p[\pi(j)], a)
11:     if found = 0 then
12:        go to 16
13:   end if
14: end for
15: count ← count + popcount(found)
16: end for
17: return
```

Loop peeling for \(r = 2\) is shown in Algorithm 4. The first 2 comparisons of characters are performed regardless the result of the first comparison (line 4).

If we consider string matching in English texts, it is less probable that all the \(a\) comparisons fail at the same time than the other way round in the case of a pattern picked randomly from the text. Therefore, it is advantageous to use the value \(r > 1\) for English.

In theory, \(r = 3\) would be good for DNA. Namely, every iteration nullifies roughly 3/4 of the remaining set bits of the bitvector \(\text{found}\). However, we achieved the best running time in practice with \(r = 5\).

### 2.4 Alternative checking orders

If the computation of character frequencies is considered inappropriate, there are other possibilities to speed up checking. In natural languages, adjacent characters have positive correlation. To break correlations, one can use a fixed order that avoids adjacent characters. We applied the following heuristic order \(\pi_h\); \(p[1], p[m], p[4], p[7], \ldots, p[3], p[6], \ldots, p[2], p[5], \ldots\).

In letter-based languages, the space character is the most frequent character. We can transform \(\pi_h\) to a slightly better scheme \(\pi_{hs}\) by moving first all the spaces to the end and then processing the remaining positions as for \(\pi_h\).

---

\(^1\)Basically, if a symbol \(p[i]\) in a position \(i\) of the pattern \(p\) matches to the text, compare next the position of \(p\) that most unlikely matches.
3 | EXPERIMENTS

We have selected 4 files of different types and alphabet sizes to run experiments on: bible.txt (Figure 2, Table 1) and E.coli.txt (Figure 3, Table 2) taken from Canterbury Corpus,16 Dostoevsky-TheDouble.txt (Figure 4, Table 3), novel The Double by Dostoevsky in Czech language taken from Project Gutenberg, and protein-hs.txt (Figure 5, Table 4) taken from Protein Corpus.17 File Dostoevsky-TheDouble.txt is a concatenation of 5 copies of the original file to get file length similar to the other files. The best running times are in bold in Tables 1–4.

We have compared methods Naive16 and Naive32 having 16 and 32 bytes processed by one SIMD instruction, respectively. Naive16-freq and Naive32-freq are their variants where comparison order given by nondecreasing probability of pattern symbols (Section 2.2). Naive16-fixed and Naive32-fixed are the variants where comparison order is fixed (Section 2.4). Our methods

---

1https://www.gutenberg.org/
FIGURE 3  Search time for E.coli.txt (|Σ| = 4) [Colour figure can be viewed at wileyonlinelibrary.com]

TABLE 2  Search times (ms) for E.coli.txt (|Σ| = 4)

<table>
<thead>
<tr>
<th>m</th>
<th>N16</th>
<th>N16-freq</th>
<th>EPSM</th>
<th>SBNDM2</th>
<th>SBNDM4</th>
<th>N32</th>
<th>N32-freq</th>
<th>N16-fixed</th>
<th>N32-fixed</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>0.462</td>
<td>0.651</td>
<td>0.520</td>
<td>4.749</td>
<td>3.840</td>
<td>0.307</td>
<td>0.467</td>
<td>0.655</td>
<td>0.371</td>
</tr>
<tr>
<td>8</td>
<td>0.526</td>
<td>0.589</td>
<td>0.890</td>
<td>3.525</td>
<td>1.017</td>
<td>0.373</td>
<td>0.375</td>
<td>0.375</td>
<td>0.576</td>
</tr>
<tr>
<td>12</td>
<td>0.527</td>
<td>0.592</td>
<td>0.887</td>
<td>2.781</td>
<td>0.725</td>
<td>0.373</td>
<td>0.375</td>
<td>0.375</td>
<td>0.576</td>
</tr>
<tr>
<td>16</td>
<td>0.533</td>
<td>0.598</td>
<td>0.447</td>
<td>2.121</td>
<td>0.595</td>
<td>0.373</td>
<td>0.384</td>
<td>0.384</td>
<td>0.566</td>
</tr>
<tr>
<td>20</td>
<td>0.531</td>
<td>0.606</td>
<td>0.445</td>
<td>1.698</td>
<td>0.517</td>
<td>0.373</td>
<td>0.381</td>
<td>0.381</td>
<td>0.567</td>
</tr>
<tr>
<td>24</td>
<td>0.525</td>
<td>0.605</td>
<td><strong>0.258</strong></td>
<td>1.422</td>
<td>0.463</td>
<td>0.371</td>
<td>0.382</td>
<td>0.366</td>
<td>0.366</td>
</tr>
<tr>
<td>28</td>
<td>0.529</td>
<td>0.608</td>
<td><strong>0.257</strong></td>
<td>1.237</td>
<td>0.428</td>
<td>0.370</td>
<td>0.388</td>
<td>0.563</td>
<td>0.365</td>
</tr>
<tr>
<td>32</td>
<td>0.533</td>
<td>0.613</td>
<td><strong>0.200</strong></td>
<td>1.090</td>
<td>0.406</td>
<td>0.372</td>
<td>0.387</td>
<td>0.573</td>
<td>0.363</td>
</tr>
<tr>
<td>36</td>
<td>0.531</td>
<td>0.606</td>
<td><strong>0.198</strong></td>
<td>0.980</td>
<td>0.387</td>
<td>0.373</td>
<td>0.387</td>
<td>0.569</td>
<td>0.363</td>
</tr>
<tr>
<td>40</td>
<td>0.535</td>
<td>0.612</td>
<td><strong>0.172</strong></td>
<td>0.893</td>
<td>0.374</td>
<td>0.373</td>
<td>0.389</td>
<td>0.572</td>
<td>0.368</td>
</tr>
<tr>
<td>44</td>
<td>0.528</td>
<td>0.608</td>
<td><strong>0.171</strong></td>
<td>0.812</td>
<td>0.359</td>
<td>0.372</td>
<td>0.388</td>
<td>0.572</td>
<td>0.366</td>
</tr>
<tr>
<td>48</td>
<td>0.526</td>
<td>0.611</td>
<td><strong>0.156</strong></td>
<td>0.748</td>
<td>0.348</td>
<td>0.374</td>
<td>0.390</td>
<td>0.560</td>
<td>0.365</td>
</tr>
<tr>
<td>52</td>
<td>0.529</td>
<td>0.607</td>
<td><strong>0.156</strong></td>
<td>0.697</td>
<td>0.343</td>
<td>0.373</td>
<td>0.390</td>
<td>0.561</td>
<td>0.367</td>
</tr>
<tr>
<td>56</td>
<td>0.532</td>
<td>0.609</td>
<td><strong>0.145</strong></td>
<td>0.642</td>
<td>0.334</td>
<td>0.373</td>
<td>0.389</td>
<td>0.577</td>
<td>0.365</td>
</tr>
<tr>
<td>60</td>
<td>0.528</td>
<td>0.609</td>
<td><strong>0.146</strong></td>
<td>0.600</td>
<td>0.326</td>
<td>0.375</td>
<td>0.389</td>
<td>0.562</td>
<td>0.366</td>
</tr>
<tr>
<td>64</td>
<td>0.530</td>
<td>0.613</td>
<td><strong>0.136</strong></td>
<td>0.567</td>
<td>0.321</td>
<td>0.382</td>
<td>0.389</td>
<td>0.573</td>
<td>0.367</td>
</tr>
</tbody>
</table>

were compared with the fastest exact string matching algorithms\(^{10}\) up to now SBNDM2, SBNDM4,\(^{18}\) and EPSM\(^{§10}\) taken from the SMART Library\(^1\).

The experiments were run on GNU/Linux 3.18.12, with x86_64 Intel Core i7-4770 CPU 3.40GHz with 16 GB RAM. The computer was without any other workload, and user time was measured using POSIX function getrusage(). The average of 100 running times is reported. The accuracy of the results is about ±2%.

The experiments show for both SSE2 and AVX2 instructions that for natural text (bible.txt) with the scheme \(\pi_h\) of fixed frequency of comparisons improves the speed of SIMD-NAIVE-SEARCH but it is further improved by considering frequencies of symbols in the text. In case of natural text with larger alphabet (Dostoevsky-TheDouble.txt), the scheme \(\pi_h\) improves

---

\(^{1}\)As of July 2016, the EPSM algorithm in the SMART library has problems for pattern lengths greater than 16. We used a corrected version in our tests.

\(^{4}\)http://www.dmi.unict.it/~faro/smart/
the speed only for AVX2 instructions. The comparison based on real frequency of symbols is the best for both SSE2 and AVX2 instructions. In case of small alphabets (E. coli.txt, protein-hs.txt), the order of comparison of symbols does not play any role (except for protein-hs.txt and SSE2 instructions).

For files with large alphabet (bible.txt, Dostoevsky-TheDouble.txt), the peeling factor $r = 3$ gave the best results for all our algorithms except for Naive16-freq and Naive32-freq where $r = 2$ was the best. The smaller the alphabet is, the less selective the bigrams or trigrams are. For file protein-hs.txt, $r = 3$ was still good and but for DNA sequences of 4 symbols, $r = 5$ turned to be the best.

We also tested NAiVE-SEARCH. In every run, it was naturally considerably slower than SIMD-NAiVE-SEARCH. Frequency order and loop peeling can also be applied to NAiVE-SEARCH. However, the speedup was smaller than in case of SIMD-NAiVE-SEARCH in our experiments.
FIGURE 5  Search time for plot-protein-hs.txt ($|\Sigma| = 19$) [Colour figure can be viewed at wileyonlinelibrary.com]

| Table 4  | Search times (ms) for plot-protein-hs.txt ($|\Sigma| = 19$) |
|----------|----------------------------------------------------------|
| $m$      | N16 | N16-freq | EPSM | SBNM2 | SBNM4 | N32 | N32-freq | N16-fixed | N32-fixed |
| 4        | 0.214 | 0.254 | 0.369 | 1.219 | 2.607 | 0.156 | 0.156 | 0.254 | 0.154 |
| 8        | 0.214 | 0.256 | 0.319 | 0.657 | 0.531 | 0.156 | 0.156 | 0.258 | 0.156 |
| 12       | 0.215 | 0.259 | 0.319 | 0.469 | 0.319 | 0.157 | 0.153 | 0.258 | 0.154 |
| 16       | 0.213 | 0.256 | 0.311 | 0.372 | 0.219 | 0.156 | 0.151 | 0.255 | 0.153 |
| 20       | 0.213 | 0.253 | 0.315 | 0.319 | 0.186 | 0.157 | 0.150 | 0.255 | 0.156 |
| 24       | 0.213 | 0.255 | 0.178 | 0.286 | 0.168 | 0.156 | 0.151 | 0.256 | 0.156 |
| 28       | 0.216 | 0.254 | 0.178 | 0.261 | 0.147 | 0.157 | 0.150 | 0.261 | 0.157 |
| 32       | 0.213 | 0.258 | 0.138 | 0.242 | 0.130 | 0.156 | 0.150 | 0.257 | 0.156 |
| 36       | 0.214 | 0.254 | 0.137 | 0.231 | 0.118 | 0.156 | 0.150 | 0.258 | 0.155 |
| 40       | 0.214 | 0.256 | 0.119 | 0.222 | 0.112 | 0.156 | 0.150 | 0.260 | 0.156 |
| 44       | 0.215 | 0.257 | 0.120 | 0.211 | 0.105 | 0.156 | 0.149 | 0.258 | 0.157 |
| 48       | 0.214 | 0.256 | 0.107 | 0.204 | 0.102 | 0.157 | 0.151 | 0.258 | 0.156 |
| 52       | 0.216 | 0.256 | 0.107 | 0.197 | 0.099 | 0.157 | 0.152 | 0.260 | 0.153 |
| 56       | 0.215 | 0.257 | 0.099 | 0.192 | 0.096 | 0.156 | 0.149 | 0.258 | 0.152 |
| 60       | 0.215 | 0.254 | 0.099 | 0.187 | 0.094 | 0.156 | 0.149 | 0.258 | 0.153 |
| 64       | 0.215 | 0.256 | 0.092 | 0.182 | 0.092 | 0.158 | 0.149 | 0.259 | 0.152 |

4 | CONCLUDING REMARKS
In spite of how many algorithms were developed for exact string matching, their running times are in general outperformed by the AVX2 technology. The implementation of the naïve search algorithm (FREQ-SIMD-NAİVE-SEARCH) that uses AVX2 instructions, applies loop peeling, and compares symbols in the order of increasing frequency is the best choice in general for counting matches. However, previous algorithms EPSM and SBNM4 have an advantage for small alphabets and long patterns. Short patterns of 20 characters or less are objects of most searches in practice, and our algorithm is especially good for such patterns. For texts with expected equiprobable symbols (like in DNA or protein strings), our algorithm naturally works well without the frequency order of symbol comparisons. Our algorithm is considerably simpler than its SIMD-based competitor EPSM, which is a combination of 6 algorithms.
ACKNOWLEDGMENTS
We would like to thank an anonymous referee to point us to a related method. This work was done while Jan Holub was visiting the Aalto University under the AScI Visitor Programme (Dean’s decision 12/2016).

ORCID

Jan Holub http://orcid.org/0000-0003-3022-2694

REFERENCES

How to cite this article: Tarhio J, Holub J, Giaquinta E. Technology beats algorithms (in exact string matching). Softw Pract Exper. 2017;47:1877-1885. https://doi.org/10.1002/spe.2511