Efficient Exact Online String Matching Through Linked Weak Factors

Matthew N. Palmer
The British Computer Society, Swindon, United Kingdom

Simone Faro[1]

Department of Mathematics and Computer Science, University of Catania, Italy

Stefano Scafiti
Department of Mathematics and Computer Science, University of Catania, Italy

Abstract

Online exact string matching is a fundamental computational problem in computer science, involving the sequential search for a pattern within a large text without prior access to the entire text. Its significance is underscored by its diverse applications in data compression, data mining, text editing, and bioinformatics, just to cite a few, where efficient substring matching is crucial. While the problem has been a subject of study for years, recent decades have witnessed a heightened focus on experimental solutions, employing various techniques to achieve superior performance. Notably, approaches centered around *weak factor recognition* have emerged as leaders in experimental settings, gaining increasing attention. This paper introduces *Hash Chain*, a novel algorithm founded on a robust weak factor recognition approach that links adjacent factors through hashing. Building upon the efficacy of weak recognition techniques, the proposed algorithm incorporates innovative strategies for organizing data structures and optimizations to enhance performance. Despite its quadratic worst-case time complexity, the new proposed algorithm demonstrates sublinear behavior in practice, outperforming currently known algorithms in the literature.

2012 ACM Subject Classification Theory of computation → Bloom filters and hashing; Theory of computation → Pattern matching

Keywords and phrases String matching, text processing, weak recognition, hashing, experimental algorithms, design and analysis of algorithms

Digital Object Identifier 10.4230/LIPIcs.SEA.2024.24

Funding This work is funded by the National Centre for HPC, Big Data and Quantum Computing, Project CN00000013, affiliated to Spoke 10.

1 Introduction

The *string matching* problem consists in finding all the occurrences of a pattern x of length m in a text y of length n, both strings defined over an alphabet Σ of size σ. The relevance of this problem is due to its direct applications in many fields, such as data compression, data mining, text editing, and bioinformatics, where fast and efficient pattern matching is critical. Given its fundamental nature in computer science, marked by intriguing theoretical and practical aspects that make it particularly compelling and challenging, the string matching problem has garnered significant attention in the literature.

The problem can be addressed in *online* mode, when you do not have access to the text before carrying out the search, or in *offline* mode, in which case it is possible to preprocess the text to speed up the search phase. In this work we consider online string matching.

[1] Corresponding author

© Matthew N. Palmer, Simone Faro, and Stefano Scafiti;
licensed under Creative Commons License CC-BY 4.0

Online string matching can be solved in linear time, with respect to the size of the text, but although the first optimal solutions appeared already in the 1970s, such as the Knuth, Morris and Pratt [22] (KMP) algorithm, several solutions have been proposed over the past decades [15], many of which appeared in recent years [18, 19]. This suggests that the interest in increasingly efficient solutions has been significant since the beginning and is still high.

Boyer and Moore (BM) provided the first sub-linear solution on average [4], while the Backward-DAWG-Matching (BDM) algorithm [9] was instead the first solution to reach the optimal $\mathcal{O}(n \log_\sigma(m)/m)$ time complexity on the average, as proved by Yao [27]. Both the KMP and the BDM algorithms are based on an *exact factor recognition*: they simulate the deterministic automaton for the language $\Sigma^\star x$ and the deterministic suffix automaton for the language of the suffixes of x, respectively. Many of the papers in the literature have amply demonstrated (see for instance [25, 10, 6, 14]) how the efficiency of these solutions is strongly affected by the encoding used for simulating the underlying automaton, especially when the bit-parallelism technique [2] is used. For this reason, recent research [11, 18, 7, 10] has focused more on approaches based on a *weak recognition*.

An algorithm performs weak recognition when it is able to recognize a broader language than the set of pattern sub-strings. The Backward Oracle Matching [1] (BOM) can be considered the pioneer of this approach, making use of the Factor Oracle of the reverse pattern. In the same direction, the Weak Factor Recognition algorithm [7] (WFR) approach is based on indexing all the $O(m^2)$ subsequences of the pattern x using a *bloom filter* [3]. In [10], Q-gram Filtering (QF) ensures that q-grams read in a window all belong to the same chain of q-grams in the pattern. More recently, Faro and Scafiti introduced the Range Automaton [18], a non-standard, weak version of the non-deterministic suffix automaton.

In this paper, we introduce the Hash Chain algorithm (HC), a new efficient algorithm for online exact string matching based on weak factor recognition and hashing. The HC algorithm stands out for its refined experimental design, marked by distinctive features and optimizations that elevate its practical performance to exceptional levels. It is based on a robust improved filtering approach which *links* together hash values corresponding to adjacent factors of the input string x. Despite the $O(nm)$ worst-case time complexity, the HC algorithm exhibits a sublinear behaviour in practice, obtaining the best running times when compared against the most effective algorithms known to date in the literature. This makes our proposed solution one of the most flexible algorithms in practical cases.

The paper is organized as follows. In Section 2 we introduce the new algorithm, describing its preprocessing and searching phase in detail. In Section 3 we propose some practical optimizations to improve the performance of the algorithm. Finally, we present in Section 4 the results of an extensive experimentation. We draw our conclusions in Section 5.

2 The Hash Chain Algorithm

In this section we present an efficient algorithm for the exact string matching problem based on a weak-factor-recognition approach using hashing. Our proposed algorithm is named *Hash Chain* (HC) and consists of a preprocessing and a searching phase. The algorithm is influenced by the Weak Factor Recognition (WFR) algorithm [8], which employs indexing via a Bloom filter to store hash values derived from all pattern factors. This implies that if, during the search phase, two consecutive text factors are identified that match portions of the pattern but are not in adjacent positions, the search continues even if an exact match will be not found. The HC algorithm also incorporates information about adjacent factors into its data structure to streamline the verification phase and drastically decrease the occurrence of false positives. In addition, by identifying q-grams which are not adjacent to each other in the pattern, the HC algorithm enables a large forward shift on average.

Before delving into the details, it is essential to formalize certain concepts and introduce key definitions that will prove valuable throughout the paper. We represent a string $x \in \Sigma^m$ as an array $x[0\,..\,m-1]$ of characters of Σ and write $|x| = m$. For $m = 0$ we obtain the empty string ε. Thus, $x[i]$ is the $(i+1)$-st character of x, for $0 \leq i < m$, and $x[i\,..\,j]$ is the substring of x contained between its $(i+1)$-st and the $(j+1)$-st characters, for $0 \leq i \leq j < m$. A q-gram of x is a substring of x with a fixed length q. Given a finite alphabet Σ, we denote by Σ^m, with $m \geq 0$, the set of all strings of length m over Σ and put $\Sigma^* = \bigcup_{m \in \mathbb{N}} \Sigma^m$.

Throughout the paper we assume x is a pattern of length m and y is a text of length n. We also assume that both strings x and y are drawn from a common alphabet Σ of size σ. In addition, we make use of the following bitwise operators: the bitwise OR "|", the bitwise AND "&", and bit-shift left "$\ll$".

We are now ready to provide a detailed description of the Hash-Chain algorithm, discussing, in the following three sections, the preprocessing phase, the structure of the hash function, and the searching phase, respectively.

2.1 The preprocessing phase

The preprocessing phase consists of the computation of an extended Bloom filter data structure indexing all the q-grams of the pattern x. It is backed by a bit-vector F of 2^α words, where each word has w bits and α controls the size of F. Two hash functions are used: $h : \Sigma^q \rightarrow \{0, 1, ..., 2^\alpha - 1\}$, which produces an index into a word in F, and $\lambda : \{0, 1, ..., 2^\alpha - 1\} \rightarrow \{0, 1, ..., 2^w - 1\}$, which outputs a word with only one of its bits set.

The filter is built by linking together each pair of adjacent non-overlapping factors, $u_1 \cdot u_2$ of fixed size q using the following formula:

$$F[h(u_2)] \leftarrow F[h(u_2)] \mid \lambda(h(u_1)). \tag{1}$$

We use the bitwise OR operator "|" to retain bits already set from previous q-gram pairs, if they had hashed to the same word in F. More formally, formula 1 is iterated for each pair of distinct q-grams $\langle u_1, u_2 \rangle$ such that:

- $|u_i| = q, i = 1, 2,$
- $u_1 = x[i..j]$, where $j = i + q - 1$, for some $0 \leq i \leq m - 2 \cdot q$
- $u_2 = x[j + 1..j + q]$

Note that the first q positions of any pattern do not have a q-gram to their left. To ensure they are recognized as factors of the pattern, we index any such factor u separately using the following formula:

$$F[h(u)] \leftarrow F[h(u)] \mid 1 \tag{2}$$

To apply formula 1 efficiently to each pair of non-overlapping adjacent factors of x, Hash Chain groups sequences of non-overlapping q-grams according to the position they appear in the pattern. More specifically, each pattern position j, with $m - q \leq j < m$, defines a sequence of $\lfloor (j + 1)/q \rfloor$ non-overlapping q-grams, given by:

$$\{x[i..i + q - 1] \quad | \quad i \geq 0, i = j - q + 1, j - 2q + 1, ...\}.$$

Each of such sets is denoted as a *q-gram chain* of the pattern. By processing q-grams in chains, we can pass a hash value from one linked pair of q-grams to the computation of the next pair. This approach enables us to compute the hash only once for each q-gram in the pattern. Figure 1 shows the 3 chains of q-grams arising in a pattern of length $m = 13$ with $q = 3$.

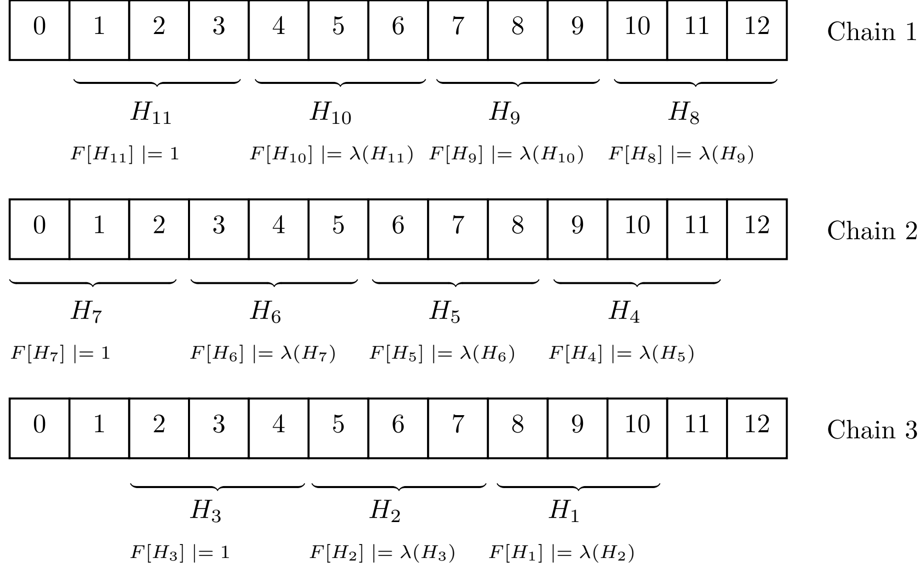


Figure 1 The three q-gram chains for a pattern x of length $m = 13$ and $q = 3$, and the operations performed for each q-gram. The $|=$ symbol represents the logical OR operator (the result is stored in the left hand operand). Each hash H_n is labelled with a number n reflecting the order in which they are calculated during pre-processing.

Obviously, a pattern where $m = q$ can only have one chain, and one q-gram, in it. More generally, when $m < 2 \cdot q - 1$, it only has $m - q + 1$ distinct chains, and when $m \geq 2 \cdot q - 1$, it has q distinct chains of q-grams.

Figure 2 shows the process of linking four adjacent q-grams of the pattern $x =$ acgtgtacgctgcaca. To complete pre-processing, we would need to process the remaining chains of q-grams in pattern x: those starting at position 1 (cgtg, tacg, ctgc), the ones at position 2 (gtgt, acgc, tgca) and those at position 3 (tgta, cgct and gcac).

2.2 Hash functions

As described above, the preprocessing phase makes use of two different hash functions, and specifically the HASH function $h : \Sigma^q \to \{0, 1, ..., 2^\alpha - 1\}$, which produces an index into a word in F, and the LINK-HASH function $\lambda : \{0, 1, ..., 2^\alpha - 1\} \to \{0, 1, ..., 2^w - 1\}$, which outputs a word with only one of its bits set. The design of h and λ strongly affect the performance of the algorithm and, for this reason, has been carefully defined.

The HASH function h has been designed to follow a shift-then-add structure, which can be implemented efficiently in modern architectures [26]. It is recursively defined as:

$$h(x) = \begin{cases} 0 & \text{if } m = 0 \\ (h(x[1..m-1]) \cdot 2^s + x[0]) & \text{otherwise.} \end{cases} \mod 2^\alpha \tag{3}$$

The hash value is multiplied by 2^s, or equivalently bit-shifted left by s, for each additional character added. To ensure we do not shift data much beyond the maximum hash value, we calculate the bit shift s by the following formula:

$$s \leftarrow \lfloor \alpha / q \rfloor \tag{4}$$

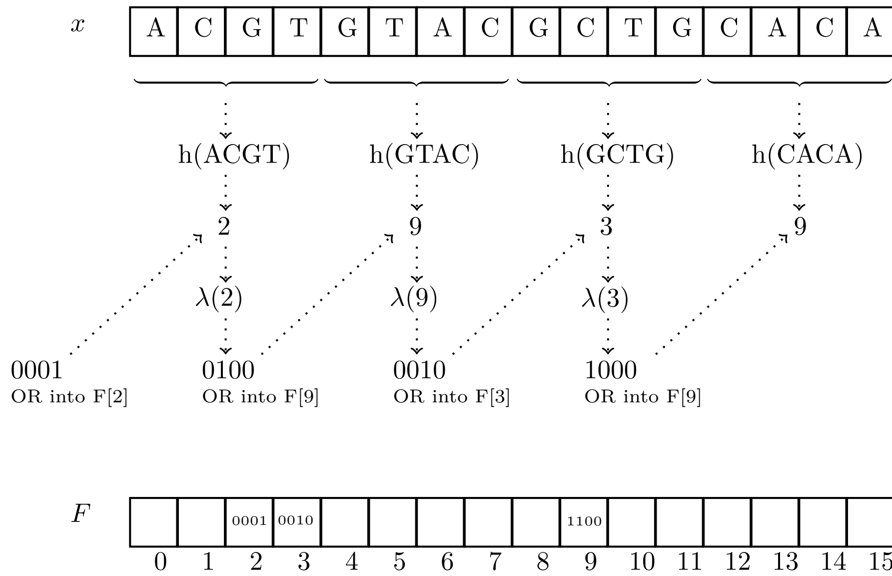


Figure 2 Linking of adjacent q-grams $q = 4$, in a pattern $m = 16$, with $w = 4$ and $\alpha = 4$. The top part of the figure shows the calculations which are performed for the first four adjacent q-grams in the pattern and where they are stored. The bottom part of the figure shows the final state of the bit vector F after all calculations have been performed. Empty cells have no entries and contain 0000.

The final value of the hash is taken as $\mod 2^\alpha$, which can be efficiently computed by bitwise ANDing it with $2^\alpha - 1$.

The Link-Hash function λ is a simple function mapping each value $0 \leq v < 2^\alpha$ to the set $\{2^0, 2^1, ..., 2^{w-1}\}$. It is meant to link together adjacent factors of the input pattern x, and that's why we refer to it as the link hash function. Its definition is given by:

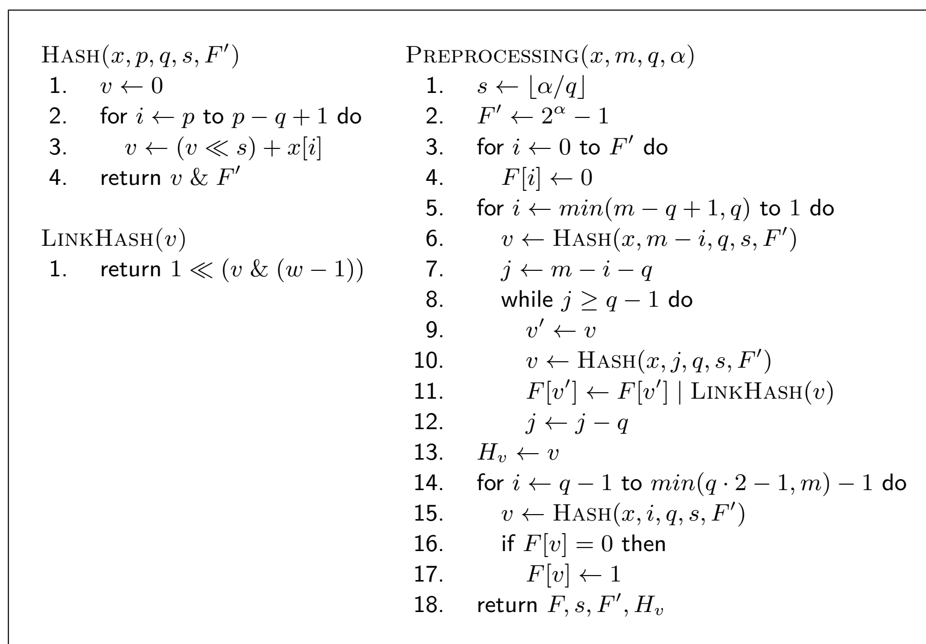
$$\lambda(v) = 2^{(v \mod w)}. \tag{5}$$

where v is the value to obtain a link hash for and w is the number of bits in a word in the bit vector F. Given that w is a power of two, the $\mod w$ operation can be efficiently computed by logically bitwise ANDing it with $w - 1$. It returns a word with a single bit set in it.

Pseudo-codes for the hash function h (Hash), the link hash function λ (Link-Hash), and the Preprocessing procedure are given in Figure 3. We don't pass w into the Link-Hash function as it is assumed to be hard-coded. Note that the Preprocessing procedure processes each chain of q-grams backwards, calculates the hashes for the first q q-grams last, and returns a hash value H_v in order to facilitate some optimisations discussed in section 3. It uses $min()$ functions on lines 5 and 14 to ensure that we only process q-gram chains that actually exist in the pattern, as short patterns where $m < 2 \cdot q - 1$ have fewer than q chains.

Obviously, a pattern where $m = q$ can only have one chain, and one q-gram, in it; more generally when $m < 2 \cdot q$, it has $m - q + 1$ distinct chains of a single q-gram each.

Regarding the time complexity of the preprocessing phase, it is proportional to the number of q-grams in the pattern. A pattern contains $m - q + 1$ distinct q-grams in it, each of which requires a hash computing for it once if the q-grams are processed in chains. It also re-computes the hash for the first q q-grams again to set a bit for them. Each q-gram requires $O(q)$ time to be read, so the time complexity is $O(m \cdot q)$.



```
HASH(x, p, q, s, F')
1.   v ← 0
2.   for i ← p to p − q + 1 do
3.       v ← (v ≪ s) + x[i]
4.   return v & F'

LINKHASH(v)
1.   return 1 ≪ (v & (w − 1))

PREPROCESSING(x, m, q, α)
1.   s ← ⌊α/q⌋
2.   F' ← 2^α − 1
3.   for i ← 0 to F' do
4.       F[i] ← 0
5.   for i ← min(m − q + 1, q) to 1 do
6.       v ← HASH(x, m − i, q, s, F')
7.       j ← m − i − q
8.       while j ≥ q − 1 do
9.           v' ← v
10.          v ← HASH(x, j, q, s, F')
11.          F[v'] ← F[v'] | LINKHASH(v)
12.          j ← j − q
13.  H_v ← v
14.  for i ← q − 1 to min(q · 2 − 1, m) − 1 do
15.      v ← HASH(x, i, q, s, F')
16.      if F[v] = 0 then
17.          F[v] ← 1
18.  return F, s, F', H_v
```

Figure 3 (On the left) The pseudo-codes of the HASH function $h : \Sigma^q \to \{0, 1, ..., 2^\alpha - 1\}$, which produces an index into a word in F, and the LINK-HASH function $\lambda : \{0, 1, ..., 2^\alpha - 1\} \to \{0, 1, ..., 2^w - 1\}$, which outputs a word with only one of its bits set. (On the right) The pseudo-code of the preprocessing phase of the Hash-Chain algorithm.

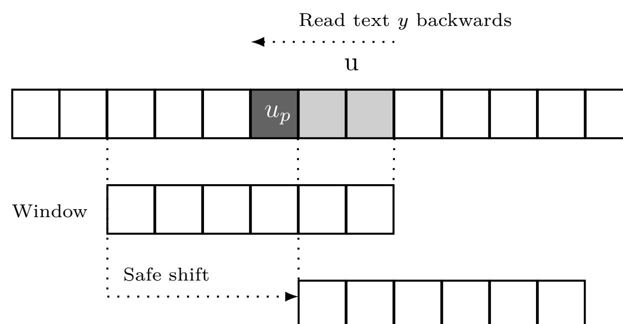


Figure 4 The general approach of factor based search algorithms. A factor u is read backwards from the end of the current window until $u_p u$ is not a factor of the pattern; it is then safe to shift the window past u_p. If the entire window is read, then a possible match must be verified when a weak recognition approach is used.

2.3 The Searching Phase

The searching phase works like any factor algorithm, such as BOM, WFR or QF. The difference between them lies in how valid factors of the pattern are determined.

A window of size m is slid along the text, starting at position 0, and shifted to the right after each attempt, until we reach the end of the text. A factor u of the pattern is read backwards in the text y, from the position aligned with the end of the window. If $u_p u$ is not a factor of the pattern, then it is safe to shift the window after u_p. This is shown in figure 4.

The pseudocode of the HC search algorithm is given in figure 5. We determine whether a q-gram is a valid factor u by first calculating its hash value v. For the first q-gram processed, aligned with the end of the current window, it looks in the word at $F[v]$ to see if it is empty at line 6. If a word in F is empty, no q-gram in the pattern hashed to v, and so u cannot be a factor. We then shift on from the end of the window by the maximum distance it is possible to shift without missing a possible occurrence, $m - q + 1$, and look at the next window.

If $F[v] \neq 0$, then we have a possible match for u and the algorithm enters the while else loop at line 8. Hash chain must now look in turn at the q-grams in the window preceding u to see if they are also possible factors of the pattern. Since pre-processing links adjacent q-grams together with the λ function, we calculate the hash value v_p of the preceding factor u_p, and check to see if the bit returned by $\lambda(v_p)$ is set in $F[v]$ at line 11. If the bit is not set, then the two factors were not linked during pre-processing, and $u_p u$ cannot be a factor of the pattern. It is then safe to shift from the position of u_p and look at the next window.

This is repeated until we reach the start of the current window, when the else branch at line 14 executes. On line 15, the position of the search j is updated such that adding $j + m - q + 1$ to it on line 18 results in the window being advanced only by one. Finally, a check for the existence of the pattern is executed from line 16 to verify the actual presence of the pattern before reporting any occurrence. Note that, according to the while else semantics., the else branch at line 14 is only taken if the execution of the loop has not been interrupted by the break statement at line 12: naive check is thus not executed unless the window has been fully scanned.

The window is eventually advanced at line 18 depending on the size of the scanned window, thus starting a new iteration of the algorithm.

Concerning the algorithm's complexity, it is evident that the search phase exhibits a time complexity of $O(nm)$ in the worst-case scenario and necessitates additional space of $O(2^\alpha)$. With the preprocessing phase consuming computational time at $O(mq)$, the overall complexity of the algorithm is delineated as $O(m(n + q))$. While its complexity may not rival other solutions that demonstrate superior worst-case efficiency, the proposed algorithm manifests commendable performance in practical scenarios, as underscored by our experimental tests elucidated in Section 4.

3 Algorithmic Optimisations

Optimizing algorithms is a crucial aspect in algorithm design, particularly in experimental contexts. The efficiency of an algorithm not only impacts its practical applicability but also influences resource consumption, execution speed, and overall performance. In experimental settings, where algorithms are tested and evaluated, optimization becomes paramount to ensure that computational resources are utilized effectively.

In this section we discuss the following optimisations for the Hash Chain algorithm, which are implemented in the bench-marked versions: (i) reduce the bits set for the first q q-grams; (ii) reduce need for full pattern verification; (iii) use a sentinel pattern at the end of the text.

The first two optimisations are included in the basic Hash Chain (HC) algorithm, while the Sentinel Hash Chain algorithm (SHC) additionally implements the last one.

3.1 Reduce the Bits Set for the First q q-grams

The preprocessing order originally presented in Figure 2 is sub-optimal in one way. The first step was to place an arbitrary 1 bit into the hash address of the first q-gram, to ensure it is recognised as a factor. This simplified the description process; we start at the beginning and progressed straightforwardly to the end.

```
HASH-CHAIN(x, m, y, n, q, α)
 1.   F, s, F′, H_v ← PREPROCESSING(x, m, q, α)
 2.   j ← m − 1
 3.   while j < n do
 4.       v ← HASH(y, j, q, s, F′)
 5.       z ← F[v]
 6.       if z ≠ 0 then
 7.           i ← j − m + 2 · q
 8.           while j ≥ i do
 9.               j ← j − q
10.               v ← HASH(y, j, q, s, F′)
11.               if z & LINKHASH(v) = 0 then
12.                   break (to line 18)
13.               z ← F[v]
14.           else
15.               j ← i − q
16.               if v = H_v and y[j − q..j − q + m − 1] = x
17.                   output j − q
18.       j ← j + m − q + 1
```

Figure 5 The pseudo-code of the Hash-Chain (HC) algorithms for the exact string matching problem. In addition to the two strings and their length, the algorithm receives as input the size of the q-grams used for hashing and the parameter α which regulates the size of the hash table.

However, the algorithm only requires that the word in F for that q-gram is not *empty*, in order that it can be identified as a valid factor of the pattern. If we process all the other pairs of q-grams first, it is possible that a collision will occur and the entry for it will *already* contain one or more bits. In that case, there is no need to set an additional 1 into the entry, as it already flags that it is a possible factor, by not being empty.

Therefore, when implementing the pre-processing phase, it is advisable to process the first q q-grams with no preceding q-gram *last*, and to only place a bit into the entry for them if it is empty. This strategy ensures we set as few bits as possible in the bit vector, which reduces the chance of a false positive match. The pre-processing pseudo-code given in Figure 3 already implements this optimisation in lines 14-17.

3.2 Reduce the Full Pattern Verifications

When the algorithm reads back to the start of the current window, it always performs a full pattern verification. However, note that if the pattern does match the current window, then the last hash value v calculated on line 10 of the pseudocode in Figure 5 *must* match the hash value of the first q-gram in the chain ending at the end of the pattern, which we will call H_v. Therefore, if we return H_v from the pattern pre-processing stage, we can compare v with it before entering the pattern verification step at line 16. If the hash does not match, there is no need to perform full pattern verification.

This is the reason for the ordering of pre-processing in the pseudo-code in Figure 3. Each chain of the pattern is processed backwards from the end of the pattern. The last chain to be processed ends at the end of the pattern. We compute that chain back from the end of the pattern, so the last hash value computed in it is the first hash in that chain, which is H_v. This can then be returned by the pre-processing stage without having to re-compute it. The pseudo code for HC search in figure 5 shows this optimisation on line 16, where we test that $v = H_v$ before attempting to verify that the text matches the pattern. In Figure 1, the hash value H_{11} as the first q-gram in the first chain would be returned as H_v.

```
SENTINEL-HASH-CHAIN(x, m, y, n, q, α)
 1.   F, s, F', H_v ← PREPROCESSING(x, m, q, α)
 2.   for i ← 0 to m − 1 do
 3.       y[n + i] ← x[i]
 4.   j ← m − 1
 5.   while j < n do
 6.       while F[HASH(y, j, q, s, F')] = 0 do
 7.           j ← j + m − q + 1
 8.       if j < n then
 9.           v ← HASH(y, j, q, s, F')
10.           z ← F[v]
11.           i ← j − m + 2 · q
12.           while j ≥ i do
13.               j ← j − q
14.               v ← HASH(y, j, q, s, F')
15.               if z & LINKHASH(v) = 0 then
16.                   break (to line 22)
17.               z ← F[v]
18.           else
19.               j ← i − q
20.               if v = H_v and y[j − q..j − q + m − 1] = x
21.                   output j − q
22.           j ← j + m − q + 1
```

Figure 6 The pseudo-code of the Sentinel-Hash-Chain (SHC) algorithms for the exact string matching problem. In addition to the two strings and their length, the algorithm receives as input the size of the q-grams used for hashing and the parameter α which regulates the size of the table.

3.3 Use a Sentinel Pattern at the End of the Text

A final optimisation technique, that can be applied to many different algorithms, is the use of a sentinel pattern at the end of the text. This technique first makes a copy of the pattern into the text, just after the end of the text to be searched, called the *sentinel pattern*. When searching, it uses a fast search loop that does not have to perform a position check. This is because the sentinel pattern at the end of the text *guarantees* we will find a copy of the pattern if we go past the end of the text, so we can safely loop without checking our position. Once the fast loop exits, we have to check that we have not run past the end of the text, but if not, we have a possible match to consider.

This technique, while powerful, has some serious constraints for real-world use. It requires control over the memory allocation of the text buffer to be searched, and the ability to write data into it. Many real-world applications will not offer that control to a search algorithm, but in cases where it is possible, it can have a performance advantage. It has been implemented and bench-marked separately as the Sentinel Hash Chain algorithm (SHC).

Pseudo code for the SHC algorithm is given in figure 6. The pattern x is copied to the end of y at n in lines 2 and 3. The fast loop without a position test looking for blank words in F is at lines 6 and 7, and we test to see if we have run past the end of the text at n in line 8. If not, we proceed to validate the rest of the chain and the pattern as normal.

4 Experimental Results

We report in this section the results of an extensive experimental comparison of the HC algorithm against the most efficient solutions known in the literature for the online exact string matching problem. Specifically, the following 21 algorithms (implemented in 99 variants, depending on the values of their parameters) have been compared:

- AOSO_q: Average-Optimal variant [21] of Shift-Or [2] with $2 \leq q \leq 6$;
- BNDM_q: Backward-Nondeterministic-DAWG-Matching [24] with $1 \leq q \leq 6$;
- BRAM_q: Backwards Range Automaton [18], with $3 \leq q \leq 7$;
- BSDM_q: Backward-SNR-DAWG-Matching [14], with $2 \leq q \leq 8$;
- BSX_q: Backward-Nondeterministic-DAWG [10], with $1 \leq q \leq 8$;
- EBOM: Extended version [13] of BOM;
- FJS algorithm [20];
- LBNDM: Long BNDM algorithm [25];
- KBNDM: Factorized BNDM algorithm [6];
- $\text{FSB}_{q,s}$: Forward Simplified [6] BNDM [24], with $2 \leq q \leq 8$ and $1 \leq s \leq 6$;
- HASH_q: Hashing algorithm [23], with $3 \leq q \leq 8$;
- $\text{HC}_{q,\alpha}$: Hash Chain, and its variant $\text{SHC}_{q,\alpha}$, with $1 \leq q \leq 8$ and $8 \leq \alpha \leq 12$.
- IOM and WOM: Improved Occurrence and Worst Occurrence Matching [5];
- $\text{QF}_{q,s}$: Qgram-Filtering algorithm [10], with $2 \leq q \leq 16$ and $1 \leq s \leq 6$;
- SBNDM_q: Simplified BNDM [28] with $2 \leq q \leq 8$;
- WFR_q: Weak Factor Recognition [8], with $1 \leq q \leq 8$ and its variant TWFR_q;
- UFM_q: Unique Factor Matcher [17], with $1 \leq q \leq 10$.

For completeness, we also included the Exact Packed String Matching (EPSM) algorithm [12], which makes use of SSE or AVX instructions and which can only report counts but not the positions of occurrences. Although we report its timings, we do not compare it with the other algorithms.

All algorithms have been implemented in the C programming language and have been tested using the SMART tool [16]. All experiments have been executed locally on a computer running Linux Ubuntu 22.04.1 with an Intel Xeon E3-1226 v3 CPU @ 3.30GHz and 24GB ECC RAM.[2]

Our tests have been run on a genome sequence, a protein sequence, and an English text (each of size 100MB) extracted from the well known *Pizza&Chilli Corpus*[3]. In the experimental evaluation, patterns of length m were randomly extracted from the sequences, with m ranging over the set of values $\{2^i \mid 3 \leq i \leq 9\}$. In all cases, the mean over the search speed plus the pre-processing time (expressed in milliseconds) of 500 runs for each pattern length has been reported.

Tables 1, 2 and 3 summarise our evaluations. Each table is divided into five blocks. The first block contains algorithms based on automata. The second contains algorithms based on character comparison. The third block contains algorithms which use weak factor recognition, which includes the Hash Chain algorithm. The fourth block contains algorithms that modify the text buffer to use a "sentinel" optimisation technique; all of these are also weak factor algorithms. The final block contains algorithms which are limited to only reporting a count of occurrences, but not their positions. Results within 105% of the best time are underlined, and best results have been boldfaced (without considering EPSM in the final block). For algorithms with variant parameters, such as the q-gram length, only the fastest variant is presented in brackets in a subscript next to the result.

Time measurements are denoted in milliseconds (ms). Nonetheless, in certain instances, certain solutions exhibited notable delays. Consequently, entries marked with the symbol "> 200" signify that the algorithm exceeded 200ms in search time and experienced a timeout.

---

[2] The source code for the new algorithm and the SMART tool are available for download respectively at `https://github.com/nishihatapalmer/HashChain` and `https://github.com/smart-tool/smart`.

[3] The corpus is available at `http://pizzachili.dcc.uchile.cl/index.html`.

Table 1 Experimental results obtained for searching on a genome sequence. Entries that state >200 indicate that the algorithm took longer than 200ms to search and timed out.

m	8	16	32	64	128	256	512
AOSO$_q$	$63.17_{(2)}$	$38.65_{(4)}$	$19.66_{(6)}$	$19.75_{(6)}$	$19.71_{(6)}$	$19.74_{(6)}$	$19.66_{(6)}$
BNDM$_q$	$37.48_{(4)}$	$19.63_{(4)}$	$10.35_{(6)}$	$10.24_{(6)}$	$10.28_{(6)}$	$10.26_{(6)}$	$10.19_{(6)}$
BSDM$_q$	$\mathbf{29.0}_{(4)}$	$15.59_{(6)}$	$9.3_{(7)}$	$7.45_{(8)}$	$7.19_{(8)}$	$7.27_{(7)}$	$7.34_{(7)}$
BXS$_q$	$37.06_{(4)}$	$19.04_{(4)}$	$9.72_{(6)}$	$9.74_{(6)}$	$9.71_{(6)}$	$9.73_{(6)}$	$9.71_{(6)}$
EBOM	95.02	64.79	40.09	24.51	14.58	9.63	7.75
FSB$_{q,s}$	$35.1_{(4,1)}$	$19.13_{(6,2)}$	$10.4_{(6,1)}$	$10.38_{(6,1)}$	$10.47_{(6,1)}$	$10.49_{(6,1)}$	$10.35_{(6,1)}$
KBNDM	107.38	68.98	39.31	23.83	23.2	23.22	23.19
LBNDM	140.1	80.12	45.44	36.13	30.05	>200	>200
SBNDM$_q$	$36.8_{(4)}$	$19.03_{(4)}$	$10.5_{(6)}$	$10.52_{(6)}$	$10.49_{(6)}$	$10.49_{(6)}$	$10.4_{(6)}$
FJS	>200	>200	>200	>200	>200	>200	>200
HASH$_q$	$88.0_{(3)}$	$42.34_{(3)}$	$21.85_{(5)}$	$12.86_{(5)}$	$11.12_{(5)}$	$11.68_{(5)}$	$13.73_{(5)}$
IOM	>200	>200	>200	>200	>200	>200	>200
WOM	>200	>200	>200	97.49	84.66	72.05	64.32
BRAM$_q$	$58.56_{(5)}$	$25.13_{(5)}$	$12.67_{(7)}$	$8.76_{(7)}$	$7.83_{(7)}$	$6.92_{(7)}$	$4.03_{(7)}$
HC$_{q,\alpha}$	$30.3_{(4,12)}$	$14.2_{(6,12)}$	$\mathbf{8.5}_{(6,12)}$	$\mathbf{7.14}_{(6,12)}$	$7.06_{(6,12)}$	$5.75_{(6,12)}$	$3.35_{(6,12)}$
QF$_{q,s}$	$33.4_{(4,3)}$	$14.69_{(4,3)}$	$8.66_{(6,2)}$	$7.39_{(6,2)}$	$7.08_{(6,2)}$	$5.77_{(6,2)}$	$3.43_{(6,2)}$
UFM$_q$	$42.0_{(5)}$	$18.38_{(6)}$	$9.96_{(7)}$	$7.71_{(8)}$	$7.68_{(8)}$	$6.46_{(8)}$	$3.66_{(8)}$
WFR$_q$	$35.6_{(4)}$	$16.63_{(5)}$	$9.93_{(5)}$	$7.72_{(7)}$	$7.17_{(6)}$	$5.79_{(6)}$	$3.25_{(7)}$
SHC$_{q,\alpha}$	$29.6_{(4,12)}$	$\mathbf{12.8}_{(5,12)}$	$8.7_{(6,12)}$	$7.39_{(6,12)}$	$\mathbf{7.03}_{(6,12)}$	$\mathbf{5.63}_{(6,12)}$	$3.29_{(6,12)}$
TWFR$_q$	$31.04_{(4)}$	$15.68_{(5)}$	$9.33_{(6)}$	$7.62_{(6)}$	$7.08_{(6)}$	$5.66_{(6)}$	$\mathbf{3.17}_{(7)}$
EPSM	22.93	9.96	6.82	6.63	5.57	3.5	1.98

Table 2 Experimental results obtained for searching on a protein sequence. Entries that state >200 indicate that the algorithm took longer than 200ms to search and timed out.

m	8	16	32	64	128	256	512
AOSO$_q$	$33.7_{(4)}$	$24.18_{(4)}$	$16.35_{(6)}$	$16.17_{(6)}$	$16.16_{(6)}$	$16.16_{(6)}$	$16.2_{(6)}$
BNDM$_q$	$19.18_{(2)}$	$11.96_{(2)}$	$8.36_{(4)}$	$8.17_{(4)}$	$8.16_{(4)}$	$8.18_{(4)}$	$8.19_{(4)}$
BSDM$_q$	$17.63_{(3)}$	$10.06_{(4)}$	$7.58_{(4)}$	$\mathbf{6.8}_{(4)}$	$6.69_{(4)}$	$6.66_{(4)}$	$6.62_{(4)}$
BXS$_q$	$\mathbf{15.51}_{(2)}$	$9.95_{(3)}$	$7.77_{(4)}$	$7.76_{(4)}$	$7.76_{(4)}$	$7.77_{(3)}$	$7.79_{(4)}$
EBOM	15.98	10.97	8.97	8.11	7.06	5.52	3.11
FSB$_{q,s}$	$15.87_{(2,0)}$	$9.96_{(3,1)}$	$7.96_{(3,1)}$	$7.98_{(3,1)}$	$7.97_{(3,1)}$	$7.98_{(3,1)}$	$7.95_{(3,1)}$
KBNDM	45.47	25.63	14.9	11.95	10.95	11.42	11.41
LBNDM	68.43	42.74	20.28	14.2	11.8	9.89	9.21
SBNDM$_q$	$15.83_{(2)}$	$10.85_{(2)}$	$8.36_{(4)}$	$8.32_{(4)}$	$8.33_{(4)}$	$8.33_{(4)}$	$8.33_{(4)}$
FJS	69.82	46.86	35.53	31.05	28.51	27.39	27.22
HASH$_q$	$80.9_{(3)}$	$37.23_{(3)}$	$19.5_{(3)}$	$12.55_{(5)}$	$10.79_{(5)}$	$11.48_{(3)}$	$12.9_{(3)}$
IOM	62.5	41.85	31.74	27.41	25.23	24.26	24.22
WOM	67.28	43.7	31.41	25.62	21.99	19.92	18.49
BRAM$_q$	$31.48_{(3)}$	$16.36_{(3)}$	$11.15_{(3)}$	$8.58_{(7)}$	$7.79_{(7)}$	$6.81_{(7)}$	$3.71_{(7)}$
HC$_{q,\alpha}$	$16.17_{(3,11)}$	$9.38_{(3,11)}$	$7.58_{(3,11)}$	$6.84_{(6,12)}$	$6.23_{(3,11)}$	$4.12_{(3,11)}$	$2.37_{(4,12)}$
QF$_{q,s}$	$16.03_{(2,6)}$	$9.93_{(3,4)}$	$7.63_{(3,4)}$	$6.84_{(4,3)}$	$6.27_{(3,4)}$	$4.16_{(3,4)}$	$2.33_{(4,3)}$
UFM$_q$	$23.15_{(3)}$	$13.66_{(3)}$	$9.69_{(7)}$	$7.72_{(8)}$	$7.68_{(8)}$	$6.37_{(7)}$	$3.54_{(8)}$
WFR$_q$	$26.03_{(2)}$	$12.36_{(4)}$	$8.37_{(4)}$	$7.2_{(4)}$	$6.61_{(4)}$	$4.75_{(4)}$	$2.52_{(5)}$
SHC$_{q,\alpha}$	$15.85_{(3,11)}$	$\mathbf{9.23}_{(3,11)}$	$\mathbf{7.49}_{(3,11)}$	$6.82_{(4,12)}$	$\mathbf{6.19}_{(3,11)}$	$\mathbf{4.06}_{(3,11)}$	$\mathbf{2.29}_{(4,12)}$
TWFR$_q$	$23.37_{(4)}$	$10.73_{(4)}$	$8.09_{(4)}$	$7.1_{(4)}$	$6.52_{(4)}$	$4.62_{(4)}$	$2.49_{(5)}$
EPSM	11.44	10.06	6.87	6.69	5.62	3.52	1.95

Table 3 Experimental results obtained for searching on an English text. Entries that state > 200 indicate that the algorithm took longer than 200ms to search and timed out.

m	8	16	32	64	128	256	512
AOSO$_q$	$36.67_{(4)}$	$24.19_{(4)}$	$16.3_{(6)}$	$16.11_{(6)}$	$16.11_{(6)}$	$16.08_{(6)}$	$16.09_{(6)}$
BNDM$_q$	$24.55_{(2)}$	$13.37_{(4)}$	$8.9_{(4)}$	$8.83_{(4)}$	$8.79_{(4)}$	$8.83_{(4)}$	$8.76_{(4)}$
BSDM$_q$	$18.92_{(3)}$	$10.8_{(4)}$	$8.07_{(4)}$	$7.07_{(6)}$	$6.85_{(6)}$	$6.87_{(4)}$	$6.81_{(4)}$
BXS$_q$	$21.18_{(2)}$	$11.92_{(3)}$	$8.33_{(4)}$	$8.31_{(4)}$	$8.31_{(4)}$	$8.36_{(4)}$	$8.29_{(4)}$
EBOM	21.43	16.36	13.71	11.78	9.41	7.13	4.62
FSB$_{q,s}$	$20.44_{(3,1)}$	$11.97_{(3,1)}$	$8.71_{(4,2)}$	$8.71_{(4,2)}$	$8.7_{(4,2)}$	$8.75_{(4,2)}$	$8.68_{(4,2)}$
KBNDM	49.08	30.41	19.88	15.23	11.89	11.82	11.81
LBNDM	81.15	49.51	26.67	16.95	13.08	10.47	9.13
SBNDM$_q$	$20.75_{(2)}$	$13.41_{(4)}$	$8.89_{(4)}$	$8.84_{(4)}$	$8.83_{(4)}$	$8.9_{(4)}$	$8.83_{(4)}$
FJS	75.05	49.48	36.19	29.83	24.97	21.69	20.46
HASH$_q$	$80.36_{(3)}$	$36.76_{(3)}$	$18.67_{(3)}$	$12.33_{(3)}$	$10.71_{(3)}$	$11.49_{(3)}$	$13.87_{(3)}$
IOM	67.84	45.18	33.21	26.8	22.51	19.83	19.18
WOM	72.35	45.86	31.92	25.4	20.48	17.87	16.75
BRAM$_q$	$31.99_{(3)}$	$17.98_{(3)}$	$10.85_{(5)}$	$8.35_{(5)}$	$7.7_{(5)}$	$6.14_{(5)}$	$3.23_{(7)}$
HC$_{q,\alpha}$	$17.54_{(3,11)}$	$10.67_{(3,11)}$	$8.08_{(6,12)}$	**6.84**$_{(6,12)}$	$6.49_{(3,11)}$	$4.57_{(3,11)}$	$2.58_{(4,12)}$
QF$_{q,s}$	$20.28_{(3,4)}$	$10.47_{(4,3)}$	$8.06_{(4,3)}$	$6.99_{(4,3)}$	$6.5_{(4,3)}$	$4.64_{(4,3)}$	$2.61_{(4,3)}$
UFM$_q$	$24.71_{(3)}$	$13.94_{(5)}$	$9.02_{(6)}$	$7.47_{(6)}$	$7.4_{(6)}$	$5.69_{(6)}$	$3.21_{(8)}$
WFR$_q$	$26.47_{(2)}$	$12.75_{(4)}$	$8.67_{(4)}$	$7.3_{(4)}$	$6.63_{(4)}$	$4.77_{(5)}$	$2.54_{(5)}$
SHC$_{q,\alpha}$	**17.14**$_{(3,11)}$	**10.41**$_{(3,11)}$	**8.0**$_{(5,12)}$	$7.0_{(5,12)}$	**6.44**$_{(3,11)}$	**4.52**$_{(3,11)}$	**2.48**$_{(4,12)}$
TWFR$_q$	$23.52_{(4)}$	$11.14_{(4)}$	$8.31_{(4)}$	$7.13_{(4)}$	$6.53_{(4)}$	$4.67_{(4)}$	$2.51_{(5)}$
EPSM	12.83	9.95	6.81	6.62	5.56	3.47	1.94

For all alphabets and pattern lengths where $m > 8$, HC$q, \alpha$ and its variant SHCq, α consistently demonstrate superior speed. Specifically, on protein sequences alone, BSDM$_q$ outperforms others in terms of speed when $m = 64$. On genome sequences, TWFR$_q$ exhibits the fastest performance when $m \geq 512$, and for BSDM$_q$ when $m = 8$. Notably, among the fastest algorithms for cases where $m \geq 16$, the predominant approach is based on weak factor recognition, whereas for instances with $m < 16$, automata-based algorithms typically emerge as the fastest.

Upon comparing SHC and HC, it becomes evident that SHC outpaces its counterpart, consistently securing the highest number of top-ranking results across all alphabets. With the exception of genome sequences, SHC stands out as the fastest algorithm for various pattern lengths. In instances where SHC is not the fastest, HC frequently claims the top position. Notably, when neither algorithm secures the fastest time, their respective timings typically fall within a range of 105% of the fastest recorded time.

Regarding the performances of the different variants of the HC algorithm, the optimal parameter value for α has consistently demonstrated to be 12, with $\alpha = 11$ yielding the best running time only in occasional instances. Concerning the size of the q-grams utilized in the algorithm implementation, as intuitively anticipated, larger q values yield enhanced performance as the pattern size increases. Nevertheless, the parameter q should be consistently constrained within the range of 3 to 6.

If your primary concern is the count of pattern occurrences within the text rather than their specific positions, the EPSM algorithm stands out as a top performer. Leveraging the latest SSE (or AVX) instructions inherent in contemporary processors, it consistently delivers superior speed with markedly reduced execution times. Exploring the feasibility of implementing a weak factor recognition algorithm using similar SSE technologies could be an intriguing avenue for further investigation.

5 Conclusions

In this paper, we introduced the Hash Chain algorithm (HC) and its variant SHC, representing innovative approaches to the online exact string matching problem. These algorithms leverage weak factor recognition and a hashing function that connects adjacent hash values of the pattern x.

Our comprehensive experimental evaluation reveals the remarkable competitiveness of our proposed solutions when benchmarked against state-of-the-art algorithms documented in the literature. The impressive performance exhibited by HC and SHC, along with other analogous weak factor recognizers, underscores the promise of weak factor recognition as a compelling approach in the realm of pattern recognition. This success encourages and paves the way for further exploration and research in the same direction.

Furthermore, contemplating a linear version of HC and SHC, mirroring the conceptual framework of the Linear Weak Factor Recognition algorithm[8], appears to be a feasible and promising avenue for future investigations.

References

1. Cyril Allauzen, Maxime Crochemore, and Mathieu Raffinot. Factor oracle: A new structure for pattern matching. In Jan Pavelka, Gerard Tel, and Miroslav Bartosek, editors, *SOFSEM '99, Theory and Practice of Informatics, 26th Conference on Current Trends in Theory and Practice of Informatics, Milovy, Czech Republic, November 27 - December 4, 1999, Proceedings*, volume 1725 of *Lecture Notes in Computer Science*, pages 295–310. Springer, 1999. doi:10.1007/3-540-47849-3_18.
2. Ricardo A. Baeza-Yates and Gaston H. Gonnet. A new approach to text searching. *Commun. ACM*, 35(10):74–82, 1992. doi:10.1145/135239.135243.
3. Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970. doi:10.1145/362686.362692.
4. Robert S. Boyer and J. Strother Moore. A fast string searching algorithm. *Commun. ACM*, 20(10), 1977. doi:10.1145/359842.359859.
5. Domenico Cantone and Simone Faro. Improved and self-tuned occurrence heuristics. *J. Discrete Algorithms*, 28:73–84, 2014.
6. Domenico Cantone, Simone Faro, and Emanuele Giaquinta. A compact representation of nondeterministic (suffix) automata for the bit-parallel approach. *Inf. Comput.*, 213:3–12, 2012. doi:10.1016/j.ic.2011.03.006.
7. Domenico Cantone, Simone Faro, and Arianna Pavone. Speeding up string matching by weak factor recognition. In *Proc. of the Prague Stringology Conference 2017*, pages 42–50, 2017. URL: http://www.stringology.org/event/2017/p05.html.
8. Domenico Cantone, Simone Faro, and Arianna Pavone. Linear and efficient string matching algorithms based on weak factor recognition. *ACM J. Exp. Algorithmics*, 24(1):1.8:1–1.8:20, 2019. doi:10.1145/3301295.
9. Maxime Crochemore and Wojciech Rytter. *Text Algorithms*. Oxford University Press, 1994. URL: http://www-igm.univ-mlv.fr/%7Emac/REC/B1.html.
10. Branislav Ďurian, Hannu Peltola, Leena Salmela, and Jorma Tarhio. Bit-parallel search algorithms for long patterns. In Paola Festa, editor, *Experimental Algorithms*, pages 129–140, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.
11. Simone Faro. A very fast string matching algorithm based on condensed alphabets. In Riccardo Dondi, Guillaume Fertin, and Giancarlo Mauri, editors, *Algorithmic Aspects in Information and Management - 11th International Conference, AAIM 2016, Bergamo, Italy, July 18-20, 2016, Proceedings*, volume 9778 of *Lecture Notes in Computer Science*, pages 65–76. Springer, 2016. doi:10.1007/978-3-319-41168-2_6.

12 Simone Faro and M. Oguzhan Külekci. Fast and flexible packed string matching. *J. Discrete Algorithms*, 28:61–72, 2014. doi:10.1016/j.jda.2014.07.003.

13 Simone Faro and Thierry Lecroq. Efficient variants of the backward-oracle-matching algorithm. *Int. J. Found. Comput. Sci.*, 20:967–984, 2009.

14 Simone Faro and Thierry Lecroq. A fast suffix automata based algorithm for exact online string matching. In *Implementation and Application of Automata - 17th International Conference, CIAA 2012*, volume 7381 of *LNCS*, pages 149–158. Springer, 2012. doi:10.1007/978-3-642-31606-7_13.

15 Simone Faro and Thierry Lecroq. The exact online string matching problem: A review of the most recent results. *ACM Comput. Surv.*, 45(2):13:1–13:42, 2013. doi:10.1145/2431211.2431212.

16 Simone Faro, Thierry Lecroq, Stefano Borzi, Simone Di Mauro, and Alessandro Maggio. The string matching algorithms research tool. In *Proc. of the Prague Stringology Conference 2016*, pages 99–111, 2016. URL: http://www.stringology.org/event/2016/p09.html.

17 Simone Faro and Stefano Scafiti. Efficient string matching based on a two-step simulation of the suffix automaton. In Sebastian Maneth, editor, *Implementation and Application of Automata*, pages 165–177, Cham, 2021. Springer International Publishing.

18 Simone Faro and Stefano Scafiti. A weak approach to suffix automata simulation for exact and approximate string matching. *Theor. Comput. Sci.*, 933:88–103, 2022. doi:10.1016/j.tcs.2022.08.028.

19 Simone Faro and Stefano Scafiti. Compact suffix automata representations for searching long patterns. *Theor. Comput. Sci.*, 940(Part):254–268, 2023. doi:10.1016/j.tcs.2022.11.005.

20 Frantisek Franek, Christopher G. Jennings, and William F. Smyth. A simple fast hybrid pattern-matching algorithm. *J. Discrete Algorithms*, 5:682–695, 2005.

21 Kimmo Fredriksson and Szymon Grabowski. Practical and optimal string matching. In *SPIRE*, volume 3772, pages 376–387, November 2005. doi:10.1007/11575832_42.

22 Donald E. Knuth, James H. Morris Jr., and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(2):323–350, 1977. doi:10.1137/0206024.

23 Thierry Lecroq. Fast exact string matching algorithms. *Inf. Process. Lett.*, 102(6):229–235, May 2007. doi:10.1016/j.ipl.2007.01.002.

24 Gonzalo Navarro and Mathieu Raffinot. A bit-parallel approach to suffix automata: Fast extended string matching. In *Combinatorial Pattern Matching, 9th Annual Symposium, CPM 98*, volume 1448 of *LNCS*, pages 14–33. Springer, 1998. doi:10.1007/BFb0030778.

25 Hannu Peltola and Jorma Tarhio. Alternative algorithms for bit-parallel string matching. In *String Processing and Information Retrieval, 10th International Symposium, SPIRE 2003*, volume 2857 of *LNCS*, pages 80–94. Springer, 2003. doi:10.1007/978-3-540-39984-1_7.

26 Noriyoshi Uratani and Masayuki Takeda. A fast string-searching algorithm for multiple patterns. *Inf. Process. Manag.*, 29(6):775–792, 1993. doi:10.1016/0306-4573(93)90106-N.

27 Andrew C Yao. The complexity of pattern matching for a random string. Technical report, Stanford University, Stanford, CA, USA, 1977.

28 Branislav Ďurian, Jan Holub, Hannu Peltola, and Jorma Tarhio. *Tuning BNDM with q-Grams*, pages 29–37. Society for Industrial and Applied Mathematics, 2009. doi:10.1137/1.9781611972894.3.