Fast Multiple Pattern Cartesian Tree Matching

Geonmo Gu[1], Siwoo Song[1], Simone Faro[2], Thierry Lecroq[3], and Kunsoo Park[1](✉)

[1] Seoul National University, Seoul, Korea
{gmgu,swsong,kpark}@theory.snu.ac.kr
[2] University of Catania, Catania, Italy
faro@dmi.unict.it
[3] Normandie University, Rouen, France
thierry.lecroq@univ-rouen.fr

Abstract. Cartesian tree matching is the problem of finding all substrings in a given text which have the same Cartesian trees as that of a given pattern. In this paper, we deal with Cartesian tree matching for the case of multiple patterns. We present two fingerprinting methods, i.e., the parent-distance encoding and the binary encoding. By combining an efficient fingerprinting method and a conventional multiple string matching algorithm, we can efficiently solve multiple pattern Cartesian tree matching. We propose three practical algorithms for multiple pattern Cartesian tree matching based on the Wu-Manber algorithm, the Rabin-Karp algorithm, and the Alpha Skip Search algorithm, respectively. In the experiments we compare our solutions against the previous algorithm [18]. Our solutions run faster than the previous algorithm as the pattern lengths increase. Especially, our algorithm based on Wu-Manber runs up to 33 times faster.

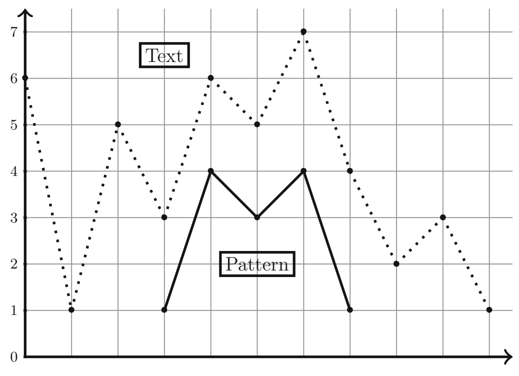
Keywords: Multiple pattern Cartesian tree matching · Parent-distance encoding · Binary encoding · Fingerprinting methods

1 Introduction

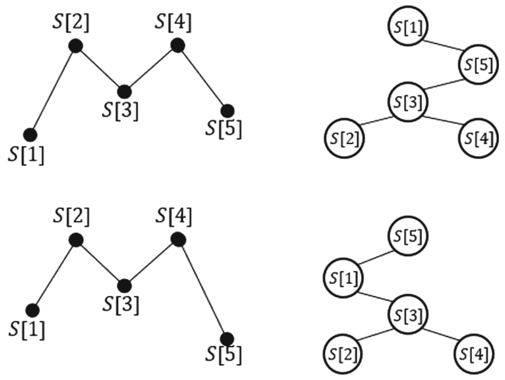
Cartesian tree matching is the problem of finding all substrings in a given text which have the same Cartesian trees as that of a given pattern. For instance, given text $T = (6, 1, 5, 3, 6, 5, 7, 4, 2, 3, 1)$ and pattern $P = (1, 4, 3, 4, 1)$ in Fig. 1a, P has the same Cartesian tree as the substring $(3, 6, 5, 7, 4)$ of T. Among many generalized matchings, Cartesian tree matching is analogous to order-preserving

A full version of this paper is available at https://arxiv.org/abs/1911.01644. Gu, Song, and Park were supported by Collaborative Genome Program for Fostering New Post-Genome industry through the National Research Foundation of Korea (NRF) funded by the Ministry of Science ICT and Future Planning (No. NRF-2014M3C9A3063541).

© Springer Nature Switzerland AG 2020



(a) Cartesian tree matching



(b) Left: double-top patterns. Right: corresponding Cartesian trees.

Fig. 1. Cartesian tree matching: multiple Cartesian trees are required for the double-top pattern.

matching [5,9,13,15] in the sense that they deal with relative order between numbers. Accordingly, both of them can be applied to time series data such as stock price analysis, but Cartesian tree matching can be sometimes more appropriate than order-preserving matching in finding patterns [18].

In this paper, we deal with Cartesian tree matching for the case of multiple patterns. Although finding multiple different patterns is interesting by itself, multiple pattern Cartesian tree matching can be applied in finding one meaningful pattern when the meaningful pattern is represented by multiple Cartesian trees: Suppose we are looking for the double-top pattern [17]. Two Cartesian trees in Fig. 1b are required to identify the pattern, where the relative order between $S[1]$ and $S[5]$ causes the difference. In general, the more complex the pattern is, the more Cartesian trees having the same lengths are required. (e.g., the head-and-shoulder pattern [17] requires four Cartesian trees.)

Recently, Park et al. [18] introduced (single pattern) Cartesian tree matching, multiple pattern Cartesian tree matching, and Cartesian tree indexing with their respective algorithms. They proposed the parent-distance representation that has a one-to-one mapping with Cartesian trees, and gave linear-time solutions for the problems, utilizing the representation and existing string algorithms, i.e., KMP algorithm, Aho-Corasick algorithm, and suffix tree construction algorithm. Song et al. [19] proposed new representations about Cartesian trees, and proposed practically fast algorithms for Cartesian tree matching based on the framework of filtering and verification.

Extensive works have been done to develop algorithms for multiple pattern matching, which is one of the fundamental problems in computer science [11,16,20]. Aho and Corasick [1] presented a linear-time algorithm based on an automaton. Commentz-Walter [6] presented an algorithm that combines the Aho-Corasick algorithm and the Boyer-Moore technique [3]. Crochemore et al. [8] proposed an algorithm that combines the Aho-Corasick automaton and a Directed Acyclic Word Graph, which runs linear in the worst case and runs in

$O((n/m)\log m)$ time in the average case, where m is the length of the shortest pattern. Rabin and Karp [12] proposed an algorithm that runs linear on average and $O(nM)$ in the worst case, where M is the sum of lengths of all patterns. Charras et al. [4] proposed an algorithm called Alpha Skip Search, which can efficiently handle both single pattern and multiple patterns. Wu and Manber [22] presented an algorithm that uses an extension of the Boyer-Moore-Horspool technique.

In this paper we present practically fast algorithms for multiple pattern Cartesian tree matching. We present three algorithms based on Wu-Manber, Rabin-Karp, and Alpha Skip Search. All of them use the filtering and verification approach, where filtering relies on efficient fingerprinting methods of a string. Two fingerprinting methods are presented, i.e., the parent-distance encoding and the binary encoding. By combining an efficient fingerprinting method and a conventional multiple string matching algorithm, we can efficiently solve multiple pattern Cartesian tree matching. In the experiments we compare our solutions against the previous algorithm [18] which is based on the Aho-Corasick algorithm. Our solutions run faster than the previous algorithm. Especially, our algorithm based on Wu-Manber runs up to 33 times faster.

2 Problem Definition

2.1 Notation

A *string* is a sequence of characters drawn from an alphabet Σ, which is a set of integers. We assume that a comparison between any two characters can be done in constant time. For a string S, $S[i]$ represents the i-th character of S, and $S[i..j]$ represents the substring of S starting from i and ending at j.

A *Cartesian tree* [21] is a binary tree derived from a string. Specifically, the Cartesian tree $CT(S)$ for a string S can be uniquely defined as follows:

– If S is an empty string, $CT(S)$ is an empty tree.
– If S is not empty and $S[i]$ is the minimum value in $S[1..n]$, $CT(S)$ is the tree with $S[i]$ as the root, $CT(S[1..i-1])$ as the left subtree, and $CT(S[i+1..n])$ as the right subtree. If there is more than one minimum value, we choose the leftmost one as the root.

Given two strings $T[1..n]$ and $P[1..m]$, where $m \leq n$, we say that P *matches* T at position i if $CT(T[i-m+1..i]) = CT(P[1..m])$. For example, given $T = (6,1,5,3,6,5,7,4,2,3,1)$ and $P = (1,4,3,4,1)$ in Fig. 1a, P matches T at position 8. We also say that $T[4..8]$ is *a match* of P in T.

Cartesian tree matching is the problem of finding all the matches in the text which have the same Cartesian trees as a given pattern.

Definition 1. *(Cartesian tree matching [18]) Given two strings text $T[1..n]$ and pattern $P[1..m]$, find every $m \leq i \leq n$ such that $CT(T[i-m+1..i]) = CT(P[1..m])$.*

2.2 Multiple Pattern Cartesian Tree Matching

Cartesian tree matching can be extended to the case of multiple patterns. *Multiple pattern Cartesian tree matching* is the problem of finding all the matches in the text which have the same Cartesian trees as at least one of the given patterns.

Definition 2. *(Multiple pattern Cartesian tree matching [18]) Given a text* $T[1..n]$ *and patterns* $P_1[1..m_1], P_2[1..m_2], ..., P_k[1..m_k]$*, find every position in the text which matches at least one pattern, i.e., it has the same Cartesian tree as that of at least one pattern.*

3 Fingerprinting Methods

Fingerprinting is a technique that maps a string to a much shorter form of data, such as a bit string or an integer. In Cartesian tree matching, we can use fingerprints to filter out unpromising matching positions with low computational cost.

In this section we introduce two fingerprinting methods, i.e., the parent-distance encoding and the binary encoding, for the purpose of representing information about Cartesian tree as an integer. The two encodings make use of the parent-distance representation and the binary representation, respectively, both of which are strings that represent Cartesian trees.

3.1 Parent-Distance Encoding

In order to represent Cartesian trees efficiently, Park et al. proposed the *parent-distance representation* [18], which is another form of the all nearest smaller values [2].

Definition 3. *(Parent-distance representation) Given a string S[1..n], the parent-distance representation of S is an integer string PD(S)[1..n], which is defined as follows:*

$$PD(S)[i] = \begin{cases} i - \max_{1 \le j < i}\{j : S[j] \le S[i]\} & \textit{if such } j \textit{ exists} \\ 0 & \textit{otherwise} \end{cases} \tag{1}$$

Intuitively, $PD(S)[i]$ stores the distance between $S[i]$ and the parent of $S[i]$ in $CT(S[1..i])$. For example, the parent-distance representation of string $S = (11, 14, 13, 15, 12)$ is $PD(S) = (0, 1, 2, 1, 4)$, where $PD(S)[3] = 3 - 1 = 2$ stores the distance between $S[3]$ and $S[1]$ ($S[1]$ is the parent of $S[3]$ in $CT(S[1..3])$). The parent-distance representation has a one-to-one mapping to the Cartesian tree [18], and so if two strings have the same parent-distance representations, the two strings also have the same Cartesian trees. The parent-distance representation of a string can be computed in linear time [18]. Note that $PD(S)[i]$ holds a value between 0 to $i - 1$ by definition, and $PD(S)[1] = 0$ at all times.

With the parent-distance representation, we can define a fingerprint encoding function that maps a string to an integer, using the factorial number system [14].

Definition 4. *(Parent-distance Encoding) Given a string* $S[1..n]$*, the encoding function* $f(S)$*, which maps* S *into an integer within the range* $[0..n! - 1]$*, is defined as follows:*

$$f(S) = \sum_{i=2}^{n} (PD(S)[i]) \cdot (i-1)!. \tag{2}$$

The parent-distance encoding maps a string into a unique integer according to its parent-distance representation. That is, given two strings S_1 and S_2, $CT(S_1) = CT(S_2)$ if and only if $f(S_1) = f(S_2)$. This is because if $PD(S_1) \neq PD(S_2)$ then $f(S_1) \neq f(S_2)$ due to the fact that $PD(S)[i] < i$. The encoding function $f(S[1..n])$ can be computed in $O(n)$ time, since $PD(S)$ can be computed in linear time. For a long string, the fingerprint may not fit in a word size, so we select a prime number by which we divide the fingerprint, and use the residue instead of the actual fingerprint. A similar encoding function was used to solve the multiple pattern order-preserving matching problem [10].

3.2 Binary Encoding

For order-preserving matching, the representation of a string as a binary string is first presented by Chhabra and Tarhio [5]. Recently, Song et al. make use of the *binary representation* for Cartesian tree matching as follows [19].

Definition 5. *(Binary representation) Given an* n*-length string* S*, binary representation* $\beta(S)$ *of length* $n-1$ *is defined as follows: for* $1 \leq i \leq n-1$*,*

$$\beta(S)[i] = \begin{cases} 1 & \text{if } S[i] \leq S[i+1] \\ 0 & \text{otherwise.} \end{cases} \tag{3}$$

Given two strings $S_1[1..n]$ and $S_2[1..n]$, the binary representations $\beta(S_1)$ and $\beta(S_2)$ are the same if the Cartesian trees $CT(S_1)$ and $CT(S_2)$ are the same [19]. Obviously, the Cartesian tree has a many-to-one mapping to the binary representation. Thus, two strings whose binary representations are the same may not have the same Cartesian trees, but two strings whose Cartesian trees are the same have the same binary representations.

A fingerprint encoding function $f(S)$ can be defined using the binary representation.

Definition 6. *(Binary Encoding) Given a string* $S[1..n]$*, encoding function* $f(S)$*, which maps* S *into an integer within the range* $[0..2^{n-1} - 1]$*, is defined as follows:*

$$f(S) = \sum_{i=1}^{n-1} (\beta(S)[i] \cdot 2^{n-1-i}). \tag{4}$$

Since $f(S)$ is a polynomial, it can be efficiently computed in linear time using Horner's rule [7]. Moreover, a fingerprint computed by the binary encoding can be reused when two strings overlap, which is discussed in the full version of this paper. Like the parent-distance encoding, in case the fingerprint does not fit in a word size, we select a prime number by which we divide the fingerprint, and use the residue instead of the actual fingerprint.

4 Fast Multiple Pattern Cartesian Tree Matching Algorithms

In this section we introduce three algorithms for multiple pattern Cartesian tree matching. Each of them consists of preprocessing and search. In the preprocessing step, hash tables are built using fingerprints of patterns. In the search step, the filtering and verification approach is adopted. To filter out unpromising matching positions, a fingerprinting method is applied to either length-m substrings of the text, where m is the length of the shortest pattern, or much shorter length-b substrings of the text (we will discuss how to set b in Sect. 4.4). Then each candidate pattern is verified by an efficient comparison method (which is described in the full version of this paper).

Algorithm 1. Algorithm based on Wu-Manber

```
 1: input: text $T[1..n]$ and patterns $P_1[1..m_1], P_2[1..m_2], ..., P_k[1..m_k]$
 2: output: every position in $T$ that matches at least one of the patterns
 3: procedure PREPROCESSING
 4:     $m \leftarrow \min(m_1, m_2, ..., m_k)$
 5:     $b \leftarrow \log_2(km)$
 6:     Initialize each entry of SHIFT to $m - b + 1$
 7:     for $i \leftarrow 1$ to $k$ do
 8:         for $j \leftarrow b$ to $m - 1$ do
 9:             $fp \leftarrow f(P_i[j - b + 1..j])$
10:             if SHIFT[$fp$] $> m - j$ then
11:                 SHIFT[$fp$] $\leftarrow m - j$
12:         $fp \leftarrow f(P_i[m - b + 1..m])$
13:         HASH[$fp$].$add(i)$
14: procedure SEARCH
15:     $index \leftarrow m$
16:     while $index \leq n$ do
17:         $fp \leftarrow f(T[index - b + 1..index])$
18:         for $i \in$ HASH[$fp$] do
19:             if $P_i$ matches $T[index - m + 1..index - m + m_i]$ then
20:                 output $index - m + m_i$
21:         $index \leftarrow index +$ SHIFT[$fp$]
```

4.1 Algorithm Based on Wu-Manber

Algorithm 1 shows the pseudo-code of an algorithm for multiple pattern Cartesian tree matching based on the Wu-Manber algorithm [22]. The algorithm uses two hash tables, HASH and SHIFT. Both tables use a fingerprint of length-b string, called a *block*. Either the parent-distance encoding or the binary encoding is used to compute the fingerprint. Given patterns $P_1, P_2, ..., P_k$, let m be the length of the shortest pattern. HASH maps a fingerprint *fp* of a block to the list of patterns P_i such that the fingerprint of the last block in P_i's length-m prefix is the same as *fp*. For a block $B[1..b]$ and a fingerprint encoding function f, HASH is defined as follows:

$$\text{HASH}[f(B)] = \{i : f(P_i[m-b+1..m]) = f(B), 1 \leq i \leq k\} \tag{5}$$

SHIFT maps a fingerprint *fp* of a block to the amount of a valid shift when the block appears in the text. The shift value is determined by the rightmost occurrence of a block in terms of the fingerprint among length-$(m-1)$ prefixes of the patterns. For a block $B[1..b]$ and a fingerprint encoding function f, we define the rightmost occurrence r_B as follows:

$$r_B = \begin{cases} \max_{b \leq j \leq m-1}\{j : f(P_i[j-b+1..j]) = f(B), 1 \leq i \leq k\} & \text{if such } j \text{ exists} \\ 0 & \text{otherwise} \end{cases} \tag{6}$$

Then SHIFT is defined as follows:

$$\text{SHIFT}[f(B)] = m - r_B \tag{7}$$

In the preprocessing step, we build HASH and SHIFT (as described in Algorithm 1). In the search step, we scan the text from left to right, computing the fingerprint of a length-b substring of the text to get a list of patterns from HASH. Let *index* be the current scanning position of the text. We compute fingerprint *fp* of $T[index-b+1..index]$, and get a list of patterns in the entry HASH[*fp*]. If the list is not empty, each pattern is verified by an efficient comparison method (see the full version). Consider $P_i[1..m_i]$ in the list. The comparison method verifies whether P_i matches $T[index-m+1..index-m+m_i]$. After verifying all patterns in the list, the text is shifted by SHIFT[*fp*].

The worst case time complexity of Algorithm 1 is $O((M+b)n)$, where M is the total pattern length, b is the block size, and n is the length of the text (consider $T = 1^n$ and the patterns of which prefixes are 1^m). On the other hand, the best case time complexity of Algorithm 1 is $O(\frac{bn}{m-b})$.

4.2 Algorithm Based on Rabin-Karp

The second algorithm for multiple pattern Cartesian tree matching is based on the Rabin-Karp algorithm [12]. The algorithm uses one hash table, namely HASH. HASH is similarly defined as in Algorithm 1 except that we consider length-m prefixes instead of blocks and we use only binary encoding for fingerprinting. For a string $S[1..m]$ and the binary encoding function f, HASH is defined as follows:

$$\mathrm{HASH}[f(S)] = \{i : f(P_i[1..m]) = f(S), 1 \leq i \leq k\} \tag{8}$$

In the preprocessing step, we build HASH. In the search step, we shift one by one, and compute the fingerprint of a length-m substring of the text to get candidate patterns by using HASH. Again, each candidate pattern is verified by an efficient comparison method.

Given a fingerprint at position i of the text, the next fingerprint at position $i+1$ can be computed in constant time if we use the binary encoding as a fingerprinting method. Let the former fingerprint be $fp_i = f(T[i-m+1..i])$ and the latter one be $fp_{i+1} = f(T[i-m+2..i+1])$. Then,

$$fp_{i+1} = 2(fp_i - 2^{m-2}\beta(T)[i-m+1]) + \beta(T)[i] \tag{9}$$

Subtracting $2^{m-2}\beta(T)[i-m+1]$ removes the leftmost bit from fp_i, multiplying the result by 2 shifts the number to the left by one position, and adding $\beta(T)[i]$ brings in the appropriate rightmost bit.

The worst case time complexity of the algorithm is $O(Mn)$ (consider $T = 1^n$ and patterns of which prefixes are 1^m). The best case time complexity is $O(n)$ since fingerprint f_i at position i, $m+1 \leq i \leq n$, can be computed in $O(1)$ time using Eq. (9).

4.3 Algorithm Based on Alpha Skip Search

The third algorithm for multiple pattern Cartesian tree matching is based on Alpha Skip Search [4]. Recall that a length-b string is called a block. The algorithm uses a hash table POS that maps the fingerprint of a block to a list of occurrences in all length-m prefixes of the patterns. Either the parent-distance encoding or the binary encoding is used for fingerprinting. For a block $B[1..b]$ and a fingerprint encoding function f, POS is defined as follows:

$$\mathrm{POS}[f(B)] = \{(i,j) : f(P_i[j-b+1..j]) = f(B), 1 \leq i \leq k, b \leq j \leq m\} \tag{10}$$

In the preprocessing step, we build POS. In the search step, we scan the text from left to right, computing the fingerprint of a length-b substring of the text to get the list of pairs (i,j), meaning that the fingerprint of $P_i[j-b+1..j]$ is the same as that of the substring of the text. Verification using an efficient comparison method is performed for each pair in the list. Note that the algorithm always shifts by $m-b+1$.

The worst case time complexity of the algorithm is $O((M+b)n)$, where M is the total pattern length, b is the block size, and n is the length of the text (consider $T = 1^n$ and patterns of which prefixes are 1^m). On the other hand, the best case time complexity of the algorithm is $O(\frac{bn}{m-b})$ since the algorithm always shifts by $m-b+1$.

4.4 Selecting the Block Size

The size of the block affects the running time of the presented algorithms based on Wu-Manber and Alpha Skip Search. A longer block size leads to a lower probability of candidate pattern occurrences, so it decreases verification time. On the other hand, a longer block size increases the overhead required for computing fingerprints. Thus, it is important to set a block size appropriate for each algorithm.

In order to set a block size, we first study the matching probability of two strings, in terms of Cartesian trees. Assume that numbers are independent and identically distributed, and there are no identical numbers within any length-n string.

Lemma 1. *Given two strings $S_1[1..n]$ and $S_2[1..n]$, the probability $p(n)$ that S_1 and S_2 have the same Cartesian tree can be defined by the recurrence formula, where $p(0) = 1$ and $p(1) = 1$, as follows:*

$$p(n) = \frac{p(0)p(n-1) + p(1)p(n-2) + \cdots + p(n-1)p(0)}{n^2} \tag{11}$$

We have the following upper bound on the matching probability.

Theorem 1. *Assume that numbers are independent and identically distributed, and there are no identical numbers within any length-n string. Given two strings $S_1[1..n]$ and $S_2[1..n]$, the probability that the two strings match, in terms of Cartesian trees, is at most $\frac{1}{2^{n-1}}$, i.e., $p(n) \leq \frac{1}{2^{n-1}}$.*

We set the block size $b = \log_2(km)$ if $\log_2(km) \leq m$; otherwise we set $b = m$, where k is the number of patterns and m is the length of the shortest pattern, in order to get a low probability of match and a relatively short block size with respect to m. By Theorem 1, if we set $b = \log_2(km)$, $p(b) \leq \frac{2}{km}$.

5 Experiments

We conduct experiments to evaluate the performances of the proposed algorithms against the previous algorithm. We compare algorithms based on Aho-Corasick (AC) [18], Wu-Manber (WM), Rabin-Karp (RM), and Alpha Skip Search (AS). By default, all our algorithms use optimization techniques described in the full version of this paper, except the min-index filtering method which is evaluated in the experiments. Particularly, in order to compare the fingerprinting methods and see the effect of min-index filtering method, we compare variants of our algorithms. The following algorithms are evaluated.

- AC: multiple Cartesian tree matching algorithm based on Aho-Corasick [18].
- WMP: algorithm based on Wu-Manber that uses the parent-distance encoding as a fingerprinting method.
- WMB: algorithm based on Wu-Manber that uses the binary encoding as a fingerprinting method. The algorithm reuses fingerprints when adjacent blocks overlap $b - 1$ characters (i.e., when the text shifts by one position), where b is the block size.

– WMBM: WMB that exploits additional min-index filtering.
– RK: algorithm based on Rabin-Karp that uses the binary encoding as a fingerprinting method.
– ASB: algorithm based on Alpha Skip Search that uses the binary encoding as a fingerprinting method. The algorithm reuses fingerprints when adjacent blocks overlap $b - 1$ characters.

All algorithms are implemented in C++. Experiments are conducted on a machine with Intel Xeon E5-2630 v4 2.20 GHz CPU and 128 GB memory running CentOS Linux.

The total time includes the preprocessing time for building data structures and the search time. To evaluate an algorithm, we run it 100 times and measure the average total time in milliseconds.

We randomly build a text of length 10,000,000 where the alphabet size is 1,000. A pattern is extracted from the text at a random position.

5.1 Evaluation on the Equal Length Patterns

We first conduct experiments with sets of patterns of the same length. Figures 2a, 2c, and 2e show the results, where k is the number of patterns and x-axis represents the length of the patterns, i.e., m. As the length of the patterns increases, WMB, WMBM, and ASB become the fastest algorithms due to a long shift length, low verification time, and light fingerprinting method. WMBM and WMB outperforms AC up to 33 times ($k = 100$ and $m = 256$). ASB outperforms AC up to 28 times ($k = 10$ and $m = 256$). RK outperforms AC up to 3 times ($k = 50, 100$ and $m = 16$). When the length of the patterns is extremely short, however, AC is the clear winner ($m = 4$). In this case, other algorithms naïvely compare the greatest part of patterns for each position of the text. WMP works visibly worse when $m = 8$ due to the extreme situation and overhead of the fingerprinting method. Since short patterns are more likely to have the same Cartesian trees, the proposed algorithms are sometimes faster when $m = 4$ than when $m = 8$ due to the grouping technique described in the full version of this paper. Comparing WMB and WMBM, the min-index filtering method is more effective when there are many short patterns ($k = 100$ and $m = 4, 8$).

5.2 Evaluation on the Different Length Patterns

We compare algorithms with sets of patterns of different lengths. Figures 2b, 2d, and 2f show the results. The length is randomly selected in an interval, i.e., [8, 32], [16, 64], [32, 128], and [64, 256]. After a length is selected, a pattern is extracted from the text at a random position. When there are many short patterns, i.e., $k = 100$ and patterns of length 8–32, AC is the fastest due to the short minimum pattern length.

When the length of the shortest pattern is sufficiently long, however, the proposed algorithms outperform AC. Specifically WMB outperforms AC up to 20 times ($k = 10, 50, 100$ and patterns of length 64–256). ASB outperforms AC

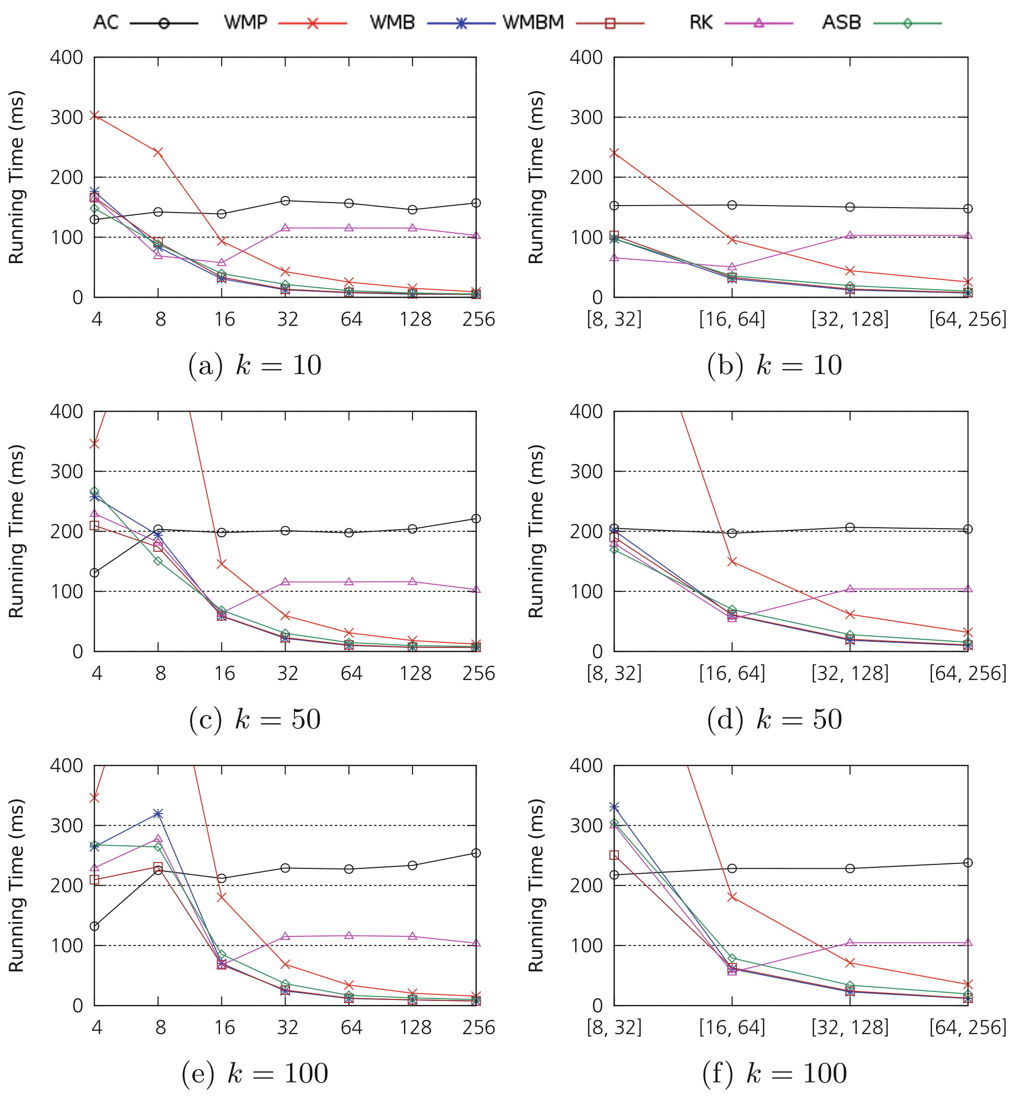


Fig. 2. Evaluation on the length of pattern. Left: patterns of equal length. Right: patterns of different lengths.

up to 14 times ($k = 10$ and patterns of length 64–256). RK outperforms AC up to 4 times ($k = 100$ and patterns of length 16–64).

5.3 Evaluation on the Real Dataset

We conduct experiment on a real dataset, which is a time series of Seoul temperatures. The Seoul temperatures dataset consists of 658,795 integers referring to the hourly temperatures in Seoul (multiplied by ten) in the years 1907–2019 [19]. In general, temperatures rise during the day and fall at night. Therefore, the Seoul temperatures dataset has more matches than random datasets when patterns are extracted from the text. Figure 3 shows the results on the Seoul temperatures dataset with sets of patterns of the same length. As the pattern length grows, the proposed algorithms run much faster than AC. For short patterns ($m = 4, 8$), AC is the fastest algorithm, and AC is up to twice times faster

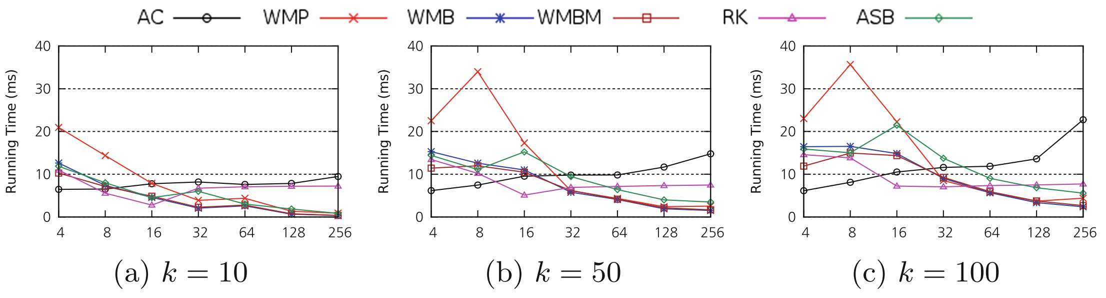


Fig. 3. Evaluation on the Seoul temperatures dataset.

than WMBM ($m = 4$ and $k = 100$) and 1.7 times faster than RK ($m = 8$ and $k = 100$). For moderate-length patterns ($m = 16, 32$), RK is up to 2.8 times faster than AC ($m = 16$ and $k = 10$), and WMB is up to 4 times faster than AC ($m = 32$ and $k = 10$). For relatively long patterns ($m = 64, 128, 256$), all the proposed algorithms outperform AC. Specifically, WMB, WMBM, ASB, and WMP outperform AC up to 28, 26, 11, and 10 times, respectively ($m = 256$ and $k = 10$), and RK outperforms AC up to 2.9 times ($m = 256$ and $k = 100$).

References

1. Aho, A.V., Corasick, M.J.: Efficient string matching: an aid to bibliographic search. Commun. ACM **18**(6), 333–340 (1975)
2. Berkman, O., Schieber, B., Vishkin, U.: Optimal doubly logarithmic parallel algorithms based on finding all nearest smaller values. J. Algorithms **14**(3), 344–370 (1993)
3. Boyer, R.S., Moore, J.S.: A fast string searching algorithm. Commun. ACM **20**(10), 762–772 (1977)
4. Charras, C., Lecroq, T., Pehoushek, J.D.: A very fast string matching algorithm for small alphabets and long patterns. In: Farach-Colton, M. (ed.) CPM 1998. LNCS, vol. 1448, pp. 55–64. Springer, Heidelberg (1998). https://doi.org/10.1007/BFb0030780
5. Chhabra, T., Tarhio, J.: Order-preserving matching with filtration. In: Gudmundsson, J., Katajainen, J. (eds.) SEA 2014. LNCS, vol. 8504, pp. 307–314. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-07959-2_26
6. Commentz-Walter, B.: A string matching algorithm fast on the average. In: Maurer, H.A. (ed.) ICALP 1979. LNCS, vol. 71, pp. 118–132. Springer, Heidelberg (1979). https://doi.org/10.1007/3-540-09510-1_10
7. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to algorithms second edition. The Knuth-Morris-Pratt Algorithm (2001)
8. Crochemore, M., Czumaj, A., Gasieniec, L., Lecroq, T., Plandowski, W., Rytter, W.: Fast practical multi-pattern matching. Inf. Process. Lett. **71**(3–4), 107–113 (1999)
9. Ganguly, A., Hon, W.K., Sadakane, K., Shah, R., Thankachan, S.V., Yang, Y.: Space-efficient dictionaries for parameterized and order-preserving pattern matching. In: 27th Annual Symposium on Combinatorial Pattern Matching (CPM), pp. 2:1–2:12. LIPIcs (2016)

10. Han, M., Kang, M., Cho, S., Gu, G., Sim, J.S., Park, K.: Fast multiple order-preserving matching algorithms. In: Lipták, Z., Smyth, W.F. (eds.) IWOCA 2015. LNCS, vol. 9538, pp. 248–259. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-29516-9_21
11. Hua, N., Song, H., Lakshman, T.: Variable-stride multi-pattern matching for scalable deep packet inspection. In: IEEE INFOCOM 2009, pp. 415–423. IEEE (2009)
12. Karp, R.M., Rabin, M.O.: Efficient randomized pattern-matching algorithms. IBM J. Res. Dev. **31**(2), 249–260 (1987)
13. Kim, J., et al.: Order-preserving matching. Theor. Comput. Sci. **525**, 68–79 (2014)
14. Knuth, D.E.: The Art of Computer Programming, volume 2: Seminumerical algorithms. Addison-Wesley Professional, Boston (2014)
15. Kubica, M., Kulczyński, T., Radoszewski, J., Rytter, W., Waleń, T.: A linear time algorithm for consecutive permutation pattern matching. Inf. Process. Let. **113**(12), 430–433 (2013)
16. Liao, H.J., Lin, C.H.R., Lin, Y.C., Tung, K.Y.: Intrusion detection system: a comprehensive review. J. Netw. Comput. Appl. **36**(1), 16–24 (2013)
17. Liu, J.N., Kwong, R.W.: Automatic extraction and identification of chart patterns towards financial forecast. Appl. Soft Comput. **7**(4), 1197–1208 (2007)
18. Park, S., Amir, A., Landau, G.M., Park, K.: Cartesian tree matching and indexing. In: 30th Annual Symposium on Combinatorial Pattern Matching (CPM), pp. 16:1–16:14. LIPIcs (2019)
19. Song, S., Ryu, C., Faro, S., Lecroq, T., Park, K.: Fast cartesian tree matching. In: Brisaboa, N.R., Puglisi, S.J. (eds.) SPIRE 2019. LNCS, vol. 11811, pp. 124–137. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-32686-9_9
20. Song, T., Zhang, W., Wang, D., Xue, Y.: A memory efficient multiple pattern matching architecture for network security. In: IEEE INFOCOM 2008-The 27th Conference on Computer Communications, pp. 166–170. IEEE (2008)
21. Vuillemin, J.: A unifying look at data structures. Commun. ACM **23**(4), 229–239 (1980)
22. Wu, S., Manber, U.: A fast algorithm for multi-pattern searching. Technical report. TR-94-17, Department of Computer Science, University of Arizona (1994)