Negative Factor: Improving Regular-Expression Matching in Strings

XIAOCHUN YANG, TAO QIU, and BIN WANG, Northeastern University, China
BAIHUA ZHENG, Singapore Management University, Singapore
YAOSHU WANG, Northeastern University, China
CHEN LI, University of California, Irvine

The problem of finding matches of a regular expression (RE) on a string exists in many applications, such as text editing, biosequence search, and shell commands. Existing techniques first identify candidates using substrings in the RE, then verify each of them using an automaton. These techniques become inefficient when there are many candidate occurrences that need to be verified. In this article, we propose a novel technique that prunes false negatives by utilizing *negative factors*, which are substrings that *cannot* appear in an answer. A main advantage of the technique is that it can be integrated with many existing algorithms to improve their efficiency significantly. We present a detailed description of this technique. We develop an efficient algorithm that utilizes negative factors to prune candidates, then improve it by using bit operations to process negative factors in parallel. We show that negative factors, when used with necessary factors (substrings that must appear in each answer), can achieve much better pruning power. We analyze the large number of negative factors, and develop an algorithm for finding a small number of high-quality negative factors. We conducted a thorough experimental study of this technique on real datasets, including DNA sequences, proteins, and text documents, and show significant performance improvement of the state-of-the-art tools by an order of magnitude.

CCS Concepts: • **Theory of computation** → **Regular languages**; • **Information systems** → *Structured text search;*

Additional Key Words and Phrases: Regular expression, long sequence

ACM Reference Format:
Xiaochun Yang, Tao Qiu, Bin Wang, Baihua Zheng, Yaoshu Wang, and Chen Li. 2016. Negative factor: Improving regular-expression matching in strings. ACM Trans. Database Syst. 40, 4, Article 25 (January 2016), 46 pages.
DOI: http://dx.doi.org/10.1145/2847525

1. INTRODUCTION

Regular expression, often referred to as Regexes or REs, are widely supported in programming languages (e.g., Perl, PHP, R, Ruby) and RE support is part of the standard library of many programming languages, including .NET, Java, and Python. In addition, REs are also widely used in our everyday applications, although we might not know it. For example, RE can help validate email and/or password formats on the

The work by X. Yang and B. Wang was partially supported by the NSF of China for Outstanding Young Scholars under grant no. 61322208, the National Basic Research Program of China (973 Program) under grant no. 2012CB316201, the NSF of China for Key Program under grant no. 61572122, the NSF of China under grant nos. 61272178, 61572122, and 61173031, and the work by C. Li was partially supported by the Joint Research Fund for Overseas Natural Science of China under grant no. 61129002.
Authors' addresses: X. Yang (corresponding author), T. Qiu, B. Wang, and Y. Wang, School of Computer Science and Engineering, Northeastern University, China; emails: yangxc@mail.neu.edu.cn, qiutao@stumail.neu.edu.cn, bwang@mail.neu.edu.cn, wangyaoshu@stumail.neu.edu.cn; B. Zheng, School of Information Systems, Singapore Management University; email: bhzheng@smu.edu.sg; C. Li, School of Information and Computer Sciences, University of California, Irvine; email: chenli@ics.uci.edu.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
© 2016 Copyright held by the owner/author(s).
ACM 0362-5915/2016/01-ART25 $15.00
DOI: http://dx.doi.org/10.1145/2847525

server side. To lower overall risk of a security breach, many servers set complexity requirements that passwords must meet when they are created or changed, for example, passwords must contain characters from three of the four given categories and their minimum length must be 10. Servers rely on REs to perform the validation.

In this article, we study the problem of efficiently finding matchings of an RE in a long string. This is a fundamental problem that exists in many application domains. In the domain of bioinformatics, users specify a string such as `TAC(T|G)AGA` to find its matchings in proteins or genomes [Kolakowski et al. 1992; Staden 1991]. Modern text editors—such as *vi*, *emacs*, and *eclipse*—provide the functionality of searching a given pattern in a text document being edited. The shell command *grep* is widely used to search plain-text files for lines matching an RE. For instance, the command "`grep ^a.ple fruits.txt`" finds lines in the file called `fruits.txt` that begin with the letter `a`, followed by one character, followed by the letter sequence `ple`.

A simple method to perform RE searches on a string is to construct an automaton for the RE. For each position in the string, we run the automaton to verify if a substring starting from that position can be accepted by the automaton. Note that the verification step can be computationally expensive. The main limitation of this approach is that we have to repeat the expensive verification step many times. Various algorithms have been developed to speed up the matching process by first identifying candidate occurrences in the string, then verifying them one by one [Crochemore et al. 1994; Watson 2003]. These algorithms identify candidate places based on certain substrings derived from the RE that have to appear in matching answers, such as a prefix and/or a suffix. For instance, each substring matching the RE `xy(a|b)*zw*` should start with `xy` (a prefix condition) and end with `zw` or `z` (suffix conditions). Such substrings, called *positive factors* throughout this article, can be used to locate candidate occurrences. Each will be further verified using one or multiple automata. Although these algorithms can eliminate many starting positions in the string, their performance highly depends on the pruning power of the positive factors. When the positive factors generate too many candidate occurrences, especially when the input string is long, their efficiency can still be low. Note that high performance for RE matching is important for time-critical applications, for example, Web services with many concurrent users. In these cases, a lower runtime can not only make the system more interactive to users, but also reduce the hardware requirements when we need to support a certain query throughput.

Contributions: In this article, we study how to improve the efficiency of existing algorithms for searching REs in strings. We propose a novel technique that provides significant pruning power by utilizing the substrings derived from the RE that *cannot* appear in an answer. Such substrings are called *negative factors*, formally introduced in Section 3 and initially presented in Yang et al. [2013]. We also study how to use negative factors to speed up the matching process of existing algorithms and we find that negative factors can be easily integrated with all the positive factors to offer much stronger pruning power.

This article extends the initial study, via (i) proposing a novel index structure, BitIndex, in Section 4.1, which adopts a bit vector to capture the occurrence of any single character in a given text; (ii) designing two new bit vector–based algorithms on top of BitIndex, in Section 4.4, to support RE matching; (iii) identifying the occurrence of redundant core negative factors and presenting a scheme to remove redundant core negative factors to further improve the quality of selected negative factors in Section 5.4; (iv) analyzing the impact of forward and reverse matching, and determining matching direction in Section 6 to further improve the matching process; (v) performing a more comprehensive experimental study on real datasets, including DNA sequences, protein sequences, and text documents, and demonstrating the space and time

Table I. Symbol Definition

Symbol	Definition
T	a text
n	the length of a given text T (i.e., $n = \|T\|$)
Q	a query regular expression
m	the length of a query regular expression Q (i.e., $m = \|Q\|$)
$R(Q)$	the set of strings that can be accepted by the automaton of Q
l_{min} w.r.t. Q	the length of a shortest string in $R(Q)$
$T[a, b]$	the substring ranging from the ath character to bth character
P, S, M, N	a prefix, a suffix, a necessary factor, and an N-factor with regard to a given Q
$\mathcal{M}(P, \pi_p)$, $\mathcal{M}(S, \pi_s)$, $\mathcal{M}(M, \pi_m)$, $\mathcal{M}(N, \pi_n)$	refer to the matching prefix, matching suffix, matching necessary factor, and matching N-factor starting at positions π_p, π_s, π_m, and π_n, respectively, in a text T
w_s	average word size in memory

efficiency of the proposed technique in Section 7; and (vi) improving the organization and presentation of the article by a major revision and careful proofreading.

2. PRELIMINARIES

In this section, we first define the problem of RE matching, then present positive factors used in the literature to improve the performance of RE matching, including prefix, suffix, and necessary factors. The formal definition of positive factors will be given later, after we formulate the problem of RE matching. Table I lists the symbols used frequently in this article.

2.1. Regular Expression Matching

Let Σ be a finite alphabet. A *regular expression* (RE) is a string over $\Sigma \cup \{\epsilon, |, \cdot, *, (,)\}$, which can be defined recursively as follows:

- The symbol ϵ is an RE. It denotes an empty string (i.e., the string of length zero).
- Each string $w \in \Sigma^*$ is an RE, which denotes the string set $\{w\}$.
- If e_1 and e_2 are REs that denote sets R_1 and R_2, respectively, then
  - (e_1) is an RE that represents the same set denoted by e_1.
  - $(e_1 \cdot e_2)$ is an RE that denotes a set of strings x that can be written as $x = yz$, where e_1 matches y and e_2 matches z.
  - $(e_1|e_2)$ is an RE that denotes a set of strings x such that x matches e_1 or e_2.
  - (e_1^+) is an RE that denotes a set of strings x such that, for a positive integer k, x can be written as $x = x_1 \ldots x_k$ and e_1 matches each string x_i $(1 \leq i \leq k)$. We use $\epsilon|e^+$ to express a Kleene closure e^*. In this article, we consider the general case e^*.

Given an RE Q, we use $R(Q)$ to represent the set of strings that can be accepted by the automaton of Q. We use $|Q|$ to express the number of characters that Q contains. We use l_{min} to represent the length of the shortest string(s) in $R(Q)$. For example, for the RE $Q = (\mathtt{G|T})\mathtt{A}^*\mathtt{GA}^*\mathtt{T}^*$, we have $|Q| = 6$ since it has six characters: G, T, A, G, A, and T. The set of strings $R(Q) = \{$GG, TG, GAG, TAG, GGA, TGA, GGT, TGT, GAGT, $\ldots\}$. We have $l_{min} = 2$, since its shortest strings GG and TG have the length 2.

For a text (sequence) T of the characters in Σ, we use $|T|$ to denote its length, $T[a]$ to denote its ath character (starting from 0), and $T[a, b]$ to denote the substring ranging from its ath character to its bth character.

For simplicity, in our examples, we focus on the domain of genome sequences, where $\Sigma = \{$A, C, G, T$\}$. We run experiments in Section 7 on other domains, such as proteins and English text, where Σ has more characters.

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
T A C T A G A C G T T A A T T T A C G T A

Fig. 1. Our sample text T.

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
T A C T A G A C G T T A A T T T A C G T A

Candidate occurrences

Fig. 2. Algorithm PFILTER: checking candidate occurrences of the RE $Q = (\texttt{G|T})\texttt{A}^*\texttt{GA}^*\texttt{T}^*$ using prefixes.

Pattern Matching of an RE. Consider a text T of length n and an RE Q of length m. We say Q matches a substring $T[a, b]$ of T at position a if $T[a, b]$ belongs to $R(Q)$. The substring $T[a, b]$ is called an *occurrence* of Q in T. The problem of pattern matching for an RE is to find *all* occurrences of Q in T. Figure 1 shows an example text, which serves as our sample text T throughout this article. Suppose that $Q = (\texttt{G|T})\texttt{A}^*\texttt{GA}^*\texttt{T}^*$. Then Q matches T at position 3, and the substring $T[3, 6]$ is an occurrence and the only occurrence of Q in T.

2.2. Positive Factors

One naïve approach for finding occurrences of an RE Q in a given text T is to build an automaton for Q and run it from the beginning of T. The verification fails once a new feed character could not be accepted by the automaton; otherwise, an occurrence will be reported whenever a final state of the automaton is reached. This verification process repeats at each position in the text, using the same automaton; it could be very *inefficient*, especially when T is very long (e.g., a chromosome of a human being can contain 3 billion characters).

Recent techniques have been developed for identifying sequences that contain at least one matching of an RE among multiple sequences. They utilize certain features of the RE Q to improve the performance of automaton-based methods. Their main idea is to use *positive factors*, which are substrings of Q that can be used to identify candidate occurrences of Q in T. In the following, we present three positive factors, including *prefix*, *suffix*, and *necessary factor*, that have been explored in the literature. We will explain their pruning power, and introduce the existing algorithms that have been developed based on one or multiple positive factors.

Prefixes and Algorithm PFILTER: A prefix with regard to an RE Q is defined as a prefix with length l_{min} of a string in $R(Q)$. For example, for the RE $Q = (\texttt{G|T})\texttt{A}^*\texttt{GA}^*\texttt{T}^*$, the prefixes with regard to Q are GA, TA, GG, and TG. Watson [2003] uses prefixes of strings in $R(Q)$ to find maximal safe shift distances to avoid checking every position in T. The main idea of this approach is to locate all matching substrings of prefixes on T, each of which is called a *matching prefix*. We use $\mathcal{M}(P, \pi_p)$ to express the matching prefix of P starting at position π_p in T. For example, the matching prefixes in Figure 2 are $\mathcal{M}(\texttt{TA}, 0) = T[0, 1]$, $\mathcal{M}(\texttt{TA}, 3) = T[3, 4]$, $\mathcal{M}(\texttt{GA}, 5) = T[5, 6]$, $\mathcal{M}(\texttt{TA}, 10) = T[10, 11]$, $\mathcal{M}(\texttt{TA}, 15) = T[15, 16]$, and $\mathcal{M}(\texttt{TA}, 19) = T[19, 20]$. It then only examines candidate occurrences of the text T starting from these matching prefixes using the automaton of Q. The automaton keeps examining each candidate occurrence until it fails, and reports an occurrence whenever a final state is reached. We call this kind of approach "algorithm PFILTER," where "P" stands for "Prefix."

Suffixes and Algorithm SFILTER: Instead of using prefixes, suffixes are another type of positive factor that can also be employed to serve the same purpose as prefixes. Similar to the definition of a prefix, a suffix with regard to an RE Q is defined as a

Fig. 3. Algorithm MFILTER: checking candidate occurrences of the RE Q = (G|T)A*GA*T* using necessary factor G, shown in bold font.

suffix with length l_{min} of a string in $R(Q)$. Take the RE Q = (G|T)A*GA*T* as an example. The suffixes with regard to Q are TT, AT, AA, GA, GT, AG, GG, and TG. The *matching suffixes* are $\mathcal{M}(\texttt{AG}, 4) = T[4, 5]$, $\mathcal{M}(\texttt{GA}, 5) = T[5, 6]$, $\mathcal{M}(\texttt{GT}, 8) = T[8, 9]$, $\mathcal{M}(\texttt{TT}, 9) = T[9, 10]$, $\mathcal{M}(\texttt{AA}, 11) = T[11, 12]$, $\mathcal{M}(\texttt{AT}, 12) = T[12, 13]$, $\mathcal{M}(\texttt{TT}, 13) = T[13, 14]$, $\mathcal{M}(\texttt{TT}, 14) = T[14, 15]$, and $\mathcal{M}(\texttt{GT}, 18) = T[18, 19]$. NR-grep [Navarro 2001; Navarro and Raffinot 2001] uses a sliding window of size l_{min} on the text T and recognizes reversed matching prefixes in the sliding window using a reversed automaton. We call the suffix-based approach "algorithm SFILTER," which is also similar to the algorithm PFILTER. It runs a reversed automaton from the end position of each suffix to the beginning of the text.

Necessary Factors and Algorithm MFILTER: In addition to prefixes and suffixes, there is another type of positive factor, called a *necessary factor*. A necessary factor with regard to an RE Q is a substring that must appear in every matching substring in the text T. For instance, G is a necessary factor with regard to the RE Q = (G|T)A*GA*T*[1]. Generally, an occurrence of Q in the text T must contain all the identified necessary factors. Otherwise, the corresponding candidate can be pruned without verification.

Consequently, to verify only the candidates containing a given necessary factor, based on the given necessary factor, we can divide Q into a left part and a right part with a corresponding automaton. Figure 3 shows an example of this approach, called algorithm MFILTER. Since G is a necessary factor, the algorithm MFILTER builds an automaton A_r for the right part of the RE Q, that is, GA*T*, and another automaton A_l for the left part of Q, that is, (G|T)A*G. It then runs A_r on the suffixes of T starting at positions 5, 8, and 18, and runs A_l on the prefixes starting at these positions.

Exploring Multiple Positive Factors: All three types of positive factors just introduced have different pruning power and are independent of each other. Consequently, it is possible to explore multiple positive factors together to further improve pruning power. In the following, we first introduce algorithm PS, which combines prefixes with last matching suffix; next, introduce algorithm PM, which integrates prefixes and necessary factors; then, introduce algorithm PMS, which explores prefixes, necessary factors, and last matching suffix together.

In Figure 2, the matching prefix $\mathcal{M}(\texttt{TA}, 19) = T[19, 20]$ could not be used to produce an answer string in $R(Q)$ since none of the suffixes identified earlier is behind it. In other words, we could use the last matching suffix to do an early termination in each verification step. Figure 4(a) shows the example of improving algorithm PFILTER by exploring the last-matching suffix. As we can see, by using the last-matching suffix $\mathcal{M}(\texttt{GT}, 18) = T[18, 19]$ in the text T, a verification can terminate early at position 19. We call this approach the "algorithm PS." It verifies only those substrings starting from every matching prefix $\mathcal{M}(P_i, \pi_p)$ to the last-matching suffix $\mathcal{M}(S_j, \pi_s)$ if the starting position π_p is less than or equal to the starting position π_s. We call S_j a *valid matching suffix* and each P_i a *valid matching prefix* with regard to its valid matching suffix S_j. For example, the substring $T[18, 19]$ is a valid matching suffix and the substrings $T[0, 1]$, $T[3, 4]$, $T[5, 6]$, $T[10, 11]$, and $T[15, 16]$ are valid matching prefixes, whereas

[1]Gnu Grep 2.0 employs a different heuristic approach for finding necessary factors with regard to an RE Q. The neighborhoods of these necessary factors are then verified using a lazy deterministic automaton.

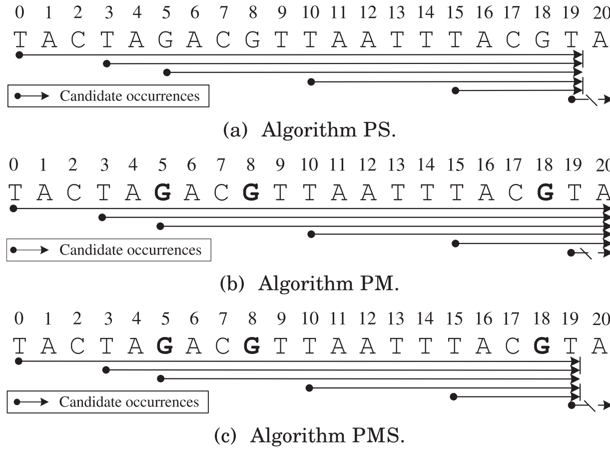


Fig. 4. Checking candidate occurrences of the RE $Q = \texttt{(G|T)A}^*\texttt{GA}^*\texttt{T}^*$ using prefixes, last matching suffix, and necessary factor G.

$T[19, 20]$ is an *invalid* matching prefix. The algorithm PS requires $O(m{\cdot}n{\cdot}n'_p)$ time to do verifications for n'_p valid matching prefixes of T, assuming that $m = |Q|$ and $n = |T|$, and the verification for each valid matching prefix requires $O(m{\cdot}n)$ time.

Figures 4(b) and 4(c) show examples of introducing necessary factors into algorithms PFilter and PS, respectively. We call the corresponding algorithms PM and PMS, respectively. In Figure 4(b), algorithm PM can prune the candidate $T[19, 20]$ since the substring does not contain a matching necessary factor. However, in Figure 4(c), algorithm PMS cannot prune any new candidate, compared with algorithm PS, because the matching necessary factor $\mathcal{M}(\texttt{G}, 18)$ appears in the last matching suffix $\mathcal{M}(\texttt{GT}, 18)$. Generally, if a necessary factor has a high probability of appearing in the late part of T, it is very likely that each candidate occurrence contains this necessary factor, hence cannot be pruned. In other words, the pruning power of a necessary factor highly depends on the position in which it appears.

3. NEGATIVE FACTORS

All the positive factors introduced in the previous section utilize certain properties that each matching substring shall satisfy. We have presented some examples to illustrate their pruning power. Alternatively, for the first time, we would like to explore a totally different concept, *negative factor*, that refers to something that all the matching substrings *should not* have. In this section, we first formulate the concept of negative factor. Next, we introduce a PNS pattern with high pruning power that can speed up the matching process, together with algorithm PNS as a new RE matching algorithm taking full advantage of PNS patterns. We demonstrate the power of combining negative factors and necessary factors via the algorithm PMNS. The details of how to construct negative factors are presented in Section 5.

3.1. Basic Concept

Definition 3.1. Negative factor (N-factor): Given a regular expression Q and a string w, a string w is called a *negative factor with respect to* Q, or simply a negative factor when Q is clear in the context, if there is no string $\Sigma^* w \Sigma^*$ in $R(Q)$.

As formally defined in Definition 3.1, an N-factor with regard to an RE Q must not appear in an answer to Q in T. For example, consider the RE $Q = \texttt{(G|T)A}^*\texttt{GA}^*\texttt{T}^*$: strings

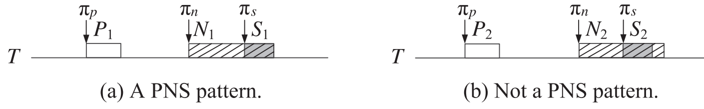


Fig. 5. A substring conforming to a PNS pattern if and only if $\pi_p \leq \pi_n < \pi_s$ and $\pi_n + |N| \leq \pi_s + l_{min}$.

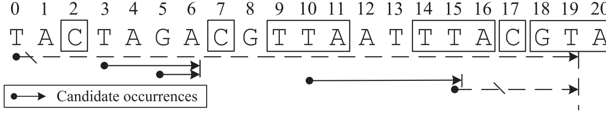


Fig. 6. Using matching N-factors $\mathcal{M}(\texttt{C}, 2)$ and $\mathcal{M}(\texttt{C}, 17)$ to prune candidates of $Q = (\texttt{G|T})\texttt{A}^*\texttt{GA}^*\texttt{T}^*$. Compared with Figure 4(a), candidates $T[0, 19]$ and $T[15, 19]$ are pruned and the verifications starting from positions 3, 5, and 10 can be terminated early by using the matching N-factors $\mathcal{M}(\texttt{C}, 7)$ and $\mathcal{M}(\texttt{TTA}, 14)$, respectively.

C, AGG, and TTA are N-factors, since they cannot appear as a substring in any answer. Therefore, an N-factor with regard to an RE Q can be a substring of neither a prefix nor a suffix with regard to Q. Given a text with length n, the number of N-factors with regard to an RE Q cannot be greater than $\sum_{i=1}^{n} |\Sigma|^i$.

3.2. A PNS Pattern

Intuitively, we say that a substring of T has a PNS pattern if it starts with a prefix of Q, has an N-factor in the middle, and ends with a suffix of Q. Formerly, let π_p, π_n, π_s be the matching (starting) positions of a prefix P, an N-factor N, and a suffix S in a text T, respectively. The substring $T[\pi_p, \pi_s + l_{min} - 1]$ conforms to a *PNS pattern* if the matching N-factor $\mathcal{M}(N, \pi_n)$ is a substring of $T[\pi_p, \pi_s + l_{min} - 1]$. Figure 5 shows that a substring conforms to a PNS pattern if and only if $\pi_p \leq \pi_n < \pi_s$ and $\pi_n + |N| \leq \pi_s + l_{min}$.

The reason for formally introducing a PNS pattern is that a substring of T conforming to a PNS pattern cannot be an occurrence of Q. Consequently, we can prune unnecessary verifications using PNS patterns. As will be demonstrated later, PNS patterns can help to either prune certain candidate occurrences from verification or help to terminate the verification early. Figure 6 shows an example of the benefit introduced by PNS patterns.

For the example in Figure 4(a), we assume that a set of N-factors is identified as {C, AGG, ATA, ATG, GGG, GTA, GTG, TAT, TGG, TTA, TTG}. Although the number of N-factors with regard to Q could be very large, we can still generate a small number of high-quality N-factors. Detailed selection criteria and selection techniques are presented in Section 5. Among all five candidate substrings identified in the example, substrings $T[0, 19]$ and $T[15, 19]$ can be pruned; the remaining three have their verifications terminated earlier, as shown in Figure 6. Take substring $T[0, 19]$ as an example. A matching N-factor $\mathcal{M}(\texttt{C}, 2)$ (i.e., $T[2]$) is located right after the matching prefix $\mathcal{M}(\texttt{TA}, 0) = T[0, 1]$, and all matching suffixes are located behind it. In other words, it is guaranteed that any candidate occurrence starting from position 0 conforms to a PNS pattern, and will not be a matching substring. Take substring $T[3, 19]$ as another example. Starting from position 3, the verification scans the substring $T[3, 19]$ character by character. When it reaches $T[6]$, it meets a matching suffix $\mathcal{M}(\texttt{GA}, 5)$, and $T[3, 6]$ is reported as a matching substring of Q. Instead of continuing the verification until the last position 19, the verification can be terminated earlier at next position 7 because of matching N-factor $\mathcal{M}(\texttt{C}, 7)$. Compared with candidate occurrences depicted in Figure 4(a), candidate occurrences shown in Figure 6 tend to be much shorter. This demonstrates the main advantage of PNS patterns, that is, effectively shortening the length of candidate occurrences.

3.3. Algorithm PNS

After introducing the pruning power of PNS patterns, we are now ready to present algorithm PNS, which *only* verifies candidate occurrences not conforming to a PNS pattern. As explained previously, a substring $T[\pi_p, \pi_s + l_{min} - 1]$ conforms to a PNS pattern if and only if $\pi_p \leq \pi_n < \pi_s$ and $\pi_n + |N| \leq \pi_s + l_{min}$. In other words, a substring might be a candidate occurrence with regard to an RE Q if and only if it does not conform to a PNS pattern, that is, at least one of the two conditions listed earlier is not satisfied. The basic idea of algorithm PNS is to verify all the candidate occurrences that do not conform to a PNS pattern. To be more specific, PNS examines the matching N-factors one by one, based on ascending order of their starting positions along T. For a given matching N-factor $\mathcal{M}(N, \pi_n)$, PNS needs to verify only the candidate occurrences $T[\pi_p, \pi_s + l_{min} - 1]$ with $\pi_p > \pi_n$ or $\pi_s < \pi_n + |N| - l_{min}$; all the occurrences $T[\pi'_p, \pi'_s + l_{min} - 1]$ with $\pi'_p \leq \pi_n$ and $\pi'_s \geq \pi_n + |N| - l_{min}$ are pruned away, as they conform to a PNS pattern.

Let list L_{P_s} and list L_{S_s} refer to the starting positions of matching prefixes and matching suffixes, respectively. Let L_{N_i} ($\forall i \in [1, \nu]$) refer to the list of starting positions of matching N-factor N_i for each given N-factor $\{N_1, \ldots, N_\nu\}$. Elements on each list are sorted based on the ascending order.

The pseudo-code of algorithm PNS is listed in Algorithm 1. To enable the evaluation of matching N-factors, we merge the lists $\{L_{N_1}, \ldots, L_{N_\nu}\}$ by maintaining the frontiers of the lists as a min-heap H_N. At each step, we pop the top from the heap, and conduct a binary search for the largest element s_{max} in L_{S_s} that satisfies the condition $s_{max} < \pi_n + |N| - l_{min}$ (lines 4–6). If such s_{max} exists, algorithm PNS performs two actions (lines 7–11). The first action is to update L_{S_s} by removing all the elements that are not greater than s_{max}. The second action is to verify all the candidate occurrences $T[\pi_p, s_{max} + l_{min} - 1]$ with $\pi_p \in L_{P_s}$ and $\pi_p \leq s_{max}$. Note that the verification of $T[\pi_p, s_{max} + l_{min} - 1]$ can locate both the occurrences ending at position $s_{max} + l_{min} - 1$ and those occurrences ending before position $s_{max} + l_{min} - 1$. This explains why we remove all the elements $\leq s_{max}$ from L_{S_s} in our first action. After that, we perform another update on set L_{P_s} by removing all the elements not greater than π_n, as all the candidate occurrences starting from those elements conform to a PNS pattern (line 12). We then remove the top element π_n from its list, and reinsert the next record position on the list (if any) to the heap H_N. This process continues until the heap H_N is empty.

After examining all the elements in H_N, algorithm PNS can terminate if L_{S_s} is empty. However, if L_{S_s} is not empty, algorithm PNS finds the suffix with the largest position s'_{max} in L_{S_s} and verifies the candidate occurrence $T[\pi_p, s'_{max} + l_{min} - 1]$ for each remaining element $\pi_p (\leq s'_{max})$ in the matching prefix list L_{P_s} (lines 14–16).

We illustrate the detailed steps of the PNS algorithm using the example shown in Figure 6. Table II lists the contents of heap H_N, and the elements in lists L_{P_s} and L_{S_s}, together with the value of s_{max} and the substrings that it verifies at each step. Note that, among all N-factors to Q, only N-factors `C` and `TTA` can be matched in text T, hence the algorithm builds up two lists L_{N_1} and L_{N_2} for these two N-factors, respectively. It inserts the frontier records of L_{N_1} and L_{N_2} to a heap H_N for each iteration. The underlined element in the H_N column refers to the head element. The only three substrings verified are consistent with those highlighted in Figure 6.

The algorithm requires $O(n_p + n_s + m_n \log l_n)$ time to generate candidates, where n_p and n_s are the number of matching prefixes and matching suffixes in T, respectively, m_n is the number of matching N-factors, and l_n is the average number of occurrences of each N-factor in T. The average length of each verification has been reduced to $\frac{n}{m_n \times l_n}$, since the candidate occurrences verified by algorithm PNS are within the range of $[\pi_n^p, \pi_n + |N| - 2]$, where π_n^p refers to the starting position of the matching negative

Table II. Illustration of Algorithm PNS for $Q = (\texttt{G|T})\texttt{A}^*\texttt{GA}^*\texttt{T}^*$

	H_N	L_{P_s}	L_{S_s}	s_{max}/s'_{max}	VERIFY
1	{<u>2</u>,9}	{0,3,5,10,15,19}	{4,5,8,9,11,12,13,14,18}	∅	-
2	{<u>7</u>,9}	{3,5,10,15,19}	{4,5,8,9,11,12,13,14,18}	5	$T[3,6]$, $T[5,6]$
3	{<u>9</u>,17}	{10,15,19}	{8,9,11,12,13,14,18}	9	-
4	{<u>14</u>,17}	{10,15,19}	{11,12,13,14,18}	14	$T[10,15]$
5	{<u>17</u>}	{15, 19}	{18}	∅	-
6	∅	{19}	{18}	18	-

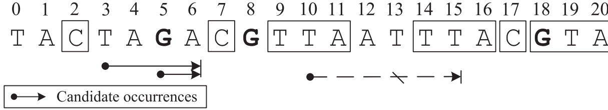


Fig. 7. Algorithm PMNS: checking candidate occurrences of the RE $Q = (\texttt{G|T})\texttt{A}^*\texttt{GA}^*\texttt{T}^*$ using PNS patterns and necessary factors, which are shown in bold font.

ALGORITHM 1: PNS

Input: A regular expression Q, a text T, a list L_{P_s} of starting positions of matching prefixes $\{P_1, \dots, P_\iota\}$ in ascending order, a list L_{S_s} of starting positions of matching suffixes $\{S_1, \dots, S_\mu\}$ in ascending order, and the set of lists $N_{set} = \{L_{N_1}, \dots, L_{N_\nu}\}$ of starting positions of matching N-factors $\{N_1, \dots, N_\nu\}$ in ascending order;

```
1  Calculate l_min for a given Q;
2  Insert the frontier records of L_N1, ..., L_Nν to a heap H_N;
3  while H_N is not empty do
4      Let π_n be the top element on H_N associated with an N-factor N;
5      Pop π_n from H_N;
6      s_max ← FINDMAX(L_Ss, π_n + |N| − l_min));
7      if s_max is found then
8          Remove all elements that are not greater than s_max in L_Ss;
9          for element π_p(≤ s_max) in the list L_Ps do
10             VERIFY(T[π_p, s_max + l_min − 1], Q);
11             Remove π_p from L_Ps;
12     Remove all elements that are not greater than π_n from L_Ps;
13     Push next record (if any) on each popped list to H_N;
14 if L_Ss is not empty then
15     s'_max ← FINDMAX(L_Ss, |T| − l_min + 1);
16     VERIFY(T[π_p, s'_max + l_min − 1], Q) for each π_p (≤ s'_max) in L_Ps;
```

factor popped out from heap H_N right before π_n. Therefore, the algorithm PNS only requires $O(\frac{n}{m_n \times l_n} n_c)$ time to do verifications, where n_c is the number of candidates.[2]

3.4. Improving Pruning Power by Combining PNS Patterns with Necessary Factors

PNS patterns actually explore prefixes, suffixes, and N-factors. However, there is another type of factor, that is, the necessary factor, that also provides certain pruning power. For example, G is a necessary factor with regard to $Q = (\texttt{G|T})\texttt{A}^*\texttt{GA}^*\texttt{T}^*$. Figure 7 shows that $T[10, 15]$ is pruned since it does not contain a necessary factor. In the following, we would like to further improve the pruning power by considering both PNS patterns and necessary factors via algorithm PMNS.

[2]According to the analysis in Section 5.5 and experimental results in Section 7, n_c is much smaller than m_p.

```
ALGORITHM 2: PMNS
   Input: ...
         M_set = {L_M1, ..., L_Mκ}: The necessary factor lists of starting positions of matching
   necessary factors {M_1, ..., M_κ} in ascending order;
 1 ...
   // Change lines 9 -- 11 in the algorithm PNS
 2 for each L_Mi in M_set do
 3 |  m^i_max ← FindMax(L_Mi, s_max + l_min − 1);
 4 |  if m^i_max is found then
 5 |   |_ Remove all elements that are not greater than π_n in L_Mi;
 6 if all m^i_max are found then
 7 |  m_max = min(m^i_max), 1 ≤ i ≤ κ;
 8 |  for element π_p(≤ m_max) in the list L_Ps do
 9 |   |  Verify(T[π_p, s_max + l_min − 1], Q);
10 |   |_ Remove π_p from L_Ps;
11 ...
```

Algorithm PMNS shares a similar idea with algorithm PNS, and integrates necessary factors with PNS patterns, based on a list of necessary factors $M = \{M_1, \ldots, M_\kappa\}$ derived from the RE Q. For each identified s_{max}, algorithm PNS needs to verify all the candidate occurrences $T[\pi_p, s_{max} + l_{min} - 1]$ with $\pi_p \leq s_{max}$. However, PMNS needs to check whether the candidate occurrences contain *all* the given necessary factors. It performs a check for each necessary factor M_i. With the help of list L_{M_i} storing the starting positions of a given necessary factor M_i, algorithm PMNS locates the position $m^i_{max}(< s_{max} + l_{min} - 1)$ that refers to the largest starting position of a *valid matching necessary factor*. Only when the m^i_{max} positions corresponding to all the necessary factors are found, the verification of $T[\pi_p, s_{max} + l_{min} - 1]$ starts. Algorithm 2 shows the pseudo-code for algorithm PMNS.

Compared with algorithm PNS, algorithm PMNS requires $O(n_p + n_s + m_n \log l_n + n_m)$ time to generate the candidates, where n_m is the total number of matching necessary factors in T. Obviously, algorithm PMNS requires extra $O(n_m)$ time cost for achieving better pruning power than PNS. It is worth mentioning that there might be cases in which no necessary factor exists; hence, the approach based on necessary factors cannot be applied. That is the reason why we present both algorithms PNS and PMNS. Although algorithm PMNS is equivalent to algorithm PNS if there is no necessary factor, we still treat them as two different algorithms to simplify our presentation.

4. BIT-PARALLEL ALGORITHMS

Algorithm PNS and algorithm PMNS, introduced in Section 3, assume that lists L_{P_s}, L_{S_s}, L_{N_i}, and L_{M_j} are available and take them as input. In this section, we discuss the issue of how to form these lists. A simple approach is to scan the text T from the beginning to the end to locate the starting positions corresponding to prefixes, suffixes, negative factors, and necessary factors, respectively. Yang et al. [2013] utilizes BWT [Lam et al. 2008] to index text, which is much better than this simple brute-force approach. BWT is a self-index structure that transforms a text T with length n into a new string T' with length $n + 1$. The new string T' has a good property that the occurrences of a substring α in T can be located in constant time (i.e., $O(|\alpha|)$) by using the suffix array SA of T', where SA is an array of indexes such that $SA[i]$ stores the starting positions of the i-th lexicographically smallest suffix in T'.

Table III. Example Bit Vectors $\vec{A}$, $\vec{C}$, $\vec{G}$, $\vec{T}$ with Regard to the Text Shown in Figure 1

	Vector																				
	0					5					10					15					20
$\vec{A}$	0	1	0	0	1	0	1	0	0	0	0	1	1	0	0	0	1	0	0	0	1
$\vec{C}$	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0
$\vec{G}$	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0
$\vec{T}$	1	0	0	1	0	0	0	0	0	1	1	0	0	1	1	1	0	0	0	1	0

Take the prefix list L_{P_s} as an example. The approach in Yang et al. [2013] builds up a list for each prefix P_i in $O(l_{min})$ time using BWT, then merges all lists into L_{P_s} in $O(l_l \log h)$ time, where h is the number of prefixes to Q and l_l is the average number of occurrences of each prefix in text T. However, merging lists to generate L_{P_s} is inefficient, especially when l_l is long. Similarly, generating L_{S_s} is also inefficient.

Can we do better than this BWT-based approach? The answer is yes. In the following, we first introduce a new index structure, BitIndex, that uses bit-vector representations to enable efficient calculation of all starting positions of any given matching substring (e.g., a matching prefix, matching suffix, matching N-factor, or matching necessary factor). Although BitIndex is not free, as it incurs small construction cost and storage overhead, it can significantly accelerate the process of forming L_{P_s}, L_{S_s}, L_{N_i}, and L_{M_j}. We then propose two bit vector–based PNS algorithms, PNS-BitC and PNS-BitG, in Section 4.2 and Section 4.3, and two bitvector–based PMNS algorithms, PMNS-BitC and PMNS-BitG, in Section 4.4, respectively. As will be demonstrated in Section 7, BitIndex and the bit vector–based algorithms can effectively speed up the matching process.

4.1. BitIndex: Representing Occurrences of Factors Using Bit Vectors

We first introduce the basic structure of BitIndex, that is, representing the occurrences of each character using a bit vector. Given an alphabet Σ, for each character $c \in \Sigma$, we use a bit vector $\vec{c}$ with length $|T|$ to represent all the occurrences of the character c in the text T. We set $\vec{c}[i] = 1 (0 \leq i < |T|)$ if c appears at the position i in T, and $\vec{c}[i] = 0$ otherwise.

Consider our running example shown in Section 2. Given the text shown in Figure 1, the first row in Table III shows an example bit vector $\vec{A}$ that records all the occurrences of character A in the text T.

A bit vector corresponding to one character in Σ with regard to T takes $|T|$ bits, and bit vectors corresponding to an alphabet Σ take $|\Sigma| \times |T|$ bits. Assuming that w_s is the average word size in memory, the overall space overhead is $O(|\Sigma| \cdot \lceil \frac{|T|}{w_s} \rceil)$.

After introducing the basic structure of BitIndex, we are ready to explain how BitIndex can facilitate the formation of lists (e.g., L_{p_s}). In the following, we first present how to calculate the starting positions with regard to a matching factor in a given text T, then explain how to calculate the starting positions with regard to multiple factors simultaneously.

Calculating Starting Positions of a Matching Factor Based on BitIndex. Given a factor α, we can easily calculate the starting and end positions of its matching factors on T based on the bit vectors maintained by BitIndex. Let bit vector $\vec{\alpha}_s$ and bit vector $\vec{\alpha}_e$ represent the starting and end positions of a matching factor α on T, respectively. Equation (1) and Equation (2) explain how to calculate $\vec{\alpha}_s$ and $\vec{\alpha}_e$ respectively.

$$\vec{\alpha}_s = \&_{i=0}^{|\alpha|-1}(\vec{\alpha[i]} << i). \tag{1}$$

$$\vec{\alpha}_e = \&_{i=0}^{|\alpha|-1}(\vec{\alpha[i]} >> (|\alpha| - 1 - i)). \tag{2}$$

Table IV. An Example of $\vec{\texttt{TTA}}_s$ and $\vec{\texttt{TTA}}_e$

	Vector																				
	0					5					10					15					20
$\vec{\texttt{TTA}}_s$	0	0	0	0	0	0	0	0	0	1	0	0	0	0	1	0	0	0	0	0	0
$\vec{\texttt{TTA}}_e$	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	1	0	0	0	0

For example, consider the N-factor TTA for the RE $Q = (\texttt{G|T})\texttt{A}^*\texttt{GA}^*\texttt{T}^*$. Table IV shows the starting positions of TTA on the text T, that is, $\vec{\texttt{TTA}}_s$, can be calculated by bit operations $\vec{\texttt{T}}\&(\vec{\texttt{T}}<<1)\&(\vec{\texttt{A}}<<2)$. Similarly, the end positions of TTA on the text T, that is, $\vec{\texttt{TTA}}_e$, can be calculated by bit operations $(\vec{\texttt{T}}>>2)\&(\vec{\texttt{T}}>>1)\&\vec{\texttt{A}}$.

Both bit vector $\vec{\alpha}_s$ and bit vector $\vec{\alpha}_e$ have length of $|T|$ and need the memory space of $O(\lceil\frac{|T|}{w_s}\rceil)$ words. It takes $O(|\alpha|\cdot\lceil\frac{|T|}{w_s}\rceil)$ time to derive bit vectors $\vec{\alpha}_s$ and $\vec{\alpha}_e$ for a given factor α.

Calculating Matching Positions of Multiple Factors. Recall that, as the input of our algorithm PNS in Algorithm 1, L_{P_s} (or L_{S_s}) is the list of starting positions of all matching prefixes (or matching suffixes). Earlier, we explained how to use BitIndex to find the starting positions of one matching prefix (or one matching suffix). We now explain how to generate the list L_{P_s} (or L_{S_s}) that records the starting positions of all matching prefixes (or all matching suffixes). Again, we use a bit vector $\vec{P}_s$ (or $\vec{S}_s$) of length $|T|$ to represent the content of L_{P_s} (or L_{S_s}). The bit $\vec{P}_s[j]$ is one if a matching prefix (or a matching suffix) starts at position j in T.

Let $P = \{P_1, P_2, \ldots, P_\iota\}$ be a set of prefixes. We could first calculate $\vec{P}_k$ for each prefix P_k in P, then merge them into one bit vector $\vec{P}_s$ by using BIT-OR operations:

$$\vec{P}_s = |_{k=1}^{|P|}(\&_{i=0}^{|P_k|-1}(\vec{P}_k[i]<<i)),\ P_k \in P. \tag{3}$$

For example, consider the prefix set $P = \{\texttt{GA}, \texttt{TA}, \texttt{GG}, \texttt{TG}\}$ with regard to the RE Q in our running example. The bit vector $\vec{P}_s$ for the identified prefixes captured by P can be calculated as: $\vec{P}_s = \vec{\texttt{GA}}\,|\,\vec{\texttt{TA}}\,|\,\vec{\texttt{GG}}\,|\,\vec{\texttt{TG}} = (\vec{\texttt{G}}\,\&\,(\vec{\texttt{A}}<<1))\,|\,(\vec{\texttt{T}}\&(\vec{\texttt{A}}<<1))\,|\,(\vec{\texttt{G}}\,\&\,(\vec{\texttt{G}}<<1))\,|\,(\vec{\texttt{T}}\,\&\,(\vec{\texttt{G}}<<1))$.

Note that, if prefixes share common substrings, this computation can be further simplified by calculating vectors for those common substrings in preference. For example, the bit vector $\vec{P}_s$ with regard to Q can be simplified as $\vec{P}_s = \vec{\texttt{GA}}\,|\,\vec{\texttt{TA}}\,|\,\vec{\texttt{GG}}\,|\,\vec{\texttt{TG}} = ((\vec{\texttt{G}}\,|\,\vec{\texttt{T}})\,\&\,(\vec{\texttt{A}}<<1))\,|\,((\vec{\texttt{G}}\,|\,\vec{\texttt{T}})\,\&\,(\vec{\texttt{G}}<<1)) = (\vec{\texttt{G}}\,|\,\vec{\texttt{T}})\,\&\,((\vec{\texttt{A}}<<1)\,|\,(\vec{\texttt{G}}<<1))$.

Recall that both algorithms PNS and PMNS need to locate the suffix starting at the largest position s'_{max} in L_{S_s} after all the starting positions of matching N-factors in L_{N_i}s are examined (e.g., lines 15–16 in Algorithm 1). To get the candidate suffixes whose starting positions are greater than the last occurrence of N-factor using bit-parallel operations, we append an auxiliary bit to the end of bit vector $\vec{N}_s$ and set $\vec{N}_s[|T|] = 1$ to mark the end of T. Similarly, both vectors $\vec{P}_s$ and $\vec{S}_s$ are extended to $|T|+1$ bits with the last bit of both vectors set to 0, that is, $\vec{P}_s[|T|] = 0$ and $\vec{S}_s[|T|] = 0$. For example, the first three rows in Table V show examples $\vec{P}_s$, $\vec{S}_s$, and $\vec{N}_s$, respectively.

4.2. The PNS-BitC Algorithm Under the Constraint $|N| \le l_{min}$

We first consider a simple case in which the length of each N-factor is no greater than the length l_{min}. Under this constraint, it is obvious that the case shown in Figure 5(b) cannot happen, that is, if positions π_p, π_n, and π_s satisfy the condition $\pi_p < \pi_n < \pi_s$, there is certainly a PNS pattern. In other words, condition $\pi_s < \pi_n$ guarantees a valid matching suffix since a matching suffix and a matching N-factor cannot start at the same position. In the following, we present a bit vector-based algorithm

Table V. Generate Candidate Regions Using Bit Operations Under the Constraint $|N| \le l_{min}$

	Vector																					
	0					5					10					15					20	
$\vec{P}_s$	1	0	0	1	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	1	0	0
$\vec{S}_s$	0	0	0	0	1	1	0	0	1	1	0	1	1	1	1	0	0	0	1	0	0	0
$\vec{N}_s$	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0	1
$\vec{A}_0$	1	1	1	0	1	0	1	1	1	1	0	1	1	1	0	1	1	1	0	1	1	1
$\vec{S}_{sm}$	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0
$\vec{B}_0$	0	0	0	1	1	1	0	0	1	1	1	1	1	1	1	0	0	0	1	0	0	1
$\vec{P}_c$	0	0	0	1	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0

Note: $l_{min} = 2$ and C is the satisfied N-factor in Figure 6.

ALGORITHM 3: PNS-BitC

Input: Bit vectors $\vec{P}_s$, $\vec{S}_s$, $\vec{N}_s$, l_{min};
1 Calculate vectors $\vec{S}_{s_m}$ and $\vec{P}_c$ using Equation (4);
2 $t \leftarrow$ the position of last suffix using $\vec{S}_{s_m}$;
3 Change $\vec{S}_{s_m}[t]$ from 1 to 0; $\pi_s \leftarrow t$; $\pi_p \leftarrow 0$;
4 **repeat**
5     $t \leftarrow$ the position of last suffix using $\vec{S}_{s_m}$;
6   **repeat**
7         $\pi_p \leftarrow$ the position of last prefix using $\vec{P}_c$;
8   Change $\vec{P}_c[\pi_p]$ from 1 to 0;
9   Verify($T[\pi_p, \pi_s + l_{min} - 1]$);
10   **until** $\pi_p < t$;
11   Change $\vec{S}_{s_m}[t]$ from 1 to 0; $\pi_s \leftarrow t$;
12 **until** *there is no* 1 *in* $\vec{S}_{s_m}$ *or* $\vec{P}_c$;

PNS-BitC, which implements algorithm PNS based on vectors $\vec{N}_s$, $\vec{P}_s$, and $\vec{S}_s$. Different from algorithm PNS, which takes lists L_{P_s}, L_{S_s}, and $\{L_{N_1}, \ldots, L_{N_v}\}$ as input, algorithm PNS-BitC takes bit vectors $\vec{P}_s$, $\vec{S}_s$, and $\vec{N}_s$ as input.

The pseudo-code of algorithm PNS-BitC is listed in Algorithm 3. It first calculates two auxiliary vectors $\vec{S}_{s_m}$ and $\vec{P}_c$ using Equation (4) (line 1). In $\vec{S}_{s_m}$, a bit $\vec{S}_{s_m}[i] = 1$ if i is the starting position of a valid matching suffix; otherwise, $\vec{S}_{s_m}[i] = 0$. In $\vec{P}_c$, a bit $\vec{P}_c[i] = 1$ if i is the starting position of a valid matching prefix, and $\vec{P}_c[i] = 0$ otherwise.

$$\begin{aligned} Intermediate: \ \vec{A}_0 &= (\vec{N}_s | \vec{S}_s) - (\vec{N}_s << 1), \\ \vec{S}_{s_m} &= (\sim \vec{A}_0) \,\&\, \vec{S}_s, \\ Intermediate: \ \vec{B}_0 &= \vec{N}_s - \vec{S}_{s_m}, \\ \vec{P}_c &= \vec{P}_s \,\&\, \vec{B}_0. \end{aligned} \tag{4}$$

For illustration purposes, we use an intermediate vector $\vec{A}_0$ to show how to calculate $\vec{S}_{s_m}$. The bit $\vec{A}_0[i]$ is set to 0 if i is the starting position of a valid matching suffix. For example, the bits $\vec{A}_0[5]$, $\vec{A}_0[14]$, and $\vec{A}_0[18]$ in Table V are all 0, and they mark three valid matching suffixes. If there is no matching suffix between any two adjacent

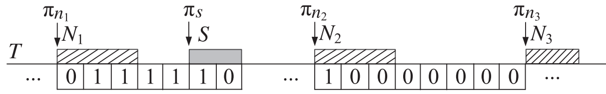


Fig. 8. Explanation of the intermediate vector $\vec{B}_0$.

matching N-factors starting at position i and position j, algorithm PNS-BitC sets $\vec{A}_0[i] = 0$ and $\vec{A}_0[l] = 1\,(i < l < j)$. Then, we calculate $\vec{S}_{s_m}$ using $(\sim\vec{A}_0)\&\vec{S}_s$.

Correspondingly, we use another intermediate vector $\vec{B}_0$ to represent the starting positions of all possible valid matching prefixes. Figure 8 shows an example of partial bits in the vector $\vec{B}_0$. The "1" bits correspond to the possible starting positions of valid matching prefixes. For the example shown in Table V, the bits $\vec{B}_0[l]\,(3 \leq l \leq 5, 8 \leq l \leq 14)$ and $\vec{B}_0[18]$ are 1, since all are the starting positions of possible valid matching prefixes.

Vector $\vec{P}_c = \vec{P}_s \,\&\, \vec{B}_0$ denotes the starting positions of valid matching prefixes. Then, algorithm PNS-BitC gets each position pair π_p and π_s using bit operations (lines 2 and 7 in Algorithm 3), which can be done in constant time.[3] It then verifies the corresponding candidate occurrence $T[\pi_p, \pi_s + l_{min} - 1]$ using $\vec{P}_c$ and $\vec{S}_{s_m}$ (line 9). For example, let $l_{min} = 2$. Table V shows the generated $\vec{P}_c$ and $\vec{S}_{s_m}$. The candidate occurrences are $T[3, 6]$, $T[5, 6]$, and $T[10, 15]$, which are consistent with the candidate occurrences shown in Figure 6.

In general, the text string T can occupy the memory space of multiple words. The algorithm proceeds in the same fashion by computing $\vec{S}_{s_m}$ and $\vec{P}_c$ for all $\lceil \frac{n}{w_s} \rceil$ words, where w_s is the word size in memory (e.g., 4B). We need to pay special attention only to the case of processing $(\vec{N}_s{<}{<}1)$ and $(\vec{N}_s - \vec{S}_{s_m})$. For the current word $\vec{N}_s^{i-1}$, when processing $(\vec{N}_s{<}{<}1)$, we need to maintain the first bit of its next word $\vec{N}_s^i$ and put it into the last bit of $\vec{N}_s^{i-1}$. Similarly, the operation $\vec{N}_s - \vec{S}_{s_m}$ of the $(i-1)$-st word is changed to $\vec{N}_s^{i-1} - \vec{S}_{s_m}^{i-1} - (\vec{N}_s^i < \vec{S}_{s_m}^i)$.

Let us analyze the space and time complexity of the algorithm. Let the word size be w_s. The algorithm requires $O(5\lceil \frac{n}{w_s} \rceil)$ to store the three input vectors and the two output vectors $\vec{S}_{s_m}$ and $\vec{P}_c$. It takes $O(\lceil \frac{n}{w_s} \rceil)$ time to calculate $\vec{S}_{s_m}$ and $\vec{P}_c$ and $O(k_1 + k_2)$ time to generate candidate occurrences, where k_1 and k_2 are number of valid matching prefixes and valid matching suffixes, respectively. It requires the same verification time as the algorithm PNS.

4.3. The PNS-BitG Algorithm without Constraints

In Figure 5(b), the position relationship $\pi_p \leq \pi_n < \pi_s$ does not guarantee a PNS pattern. It means that the comparison between the end positions of the N-factor N_2 and the suffix S_2 is also needed, that is, both conditions $\pi_p \leq \pi_n < \pi_s$ and $\pi_{n_e} \leq \pi_{s_e}$ need to be satisfied to make sure that a PNS pattern appears, where π_{n_e} and π_{s_e} are the end positions of the N-factor N_2 and the suffix S_2.

We can get an end position π_{s_e} by shifting its corresponding starting position π_s to the right for $l_{min} - 1$ bits since we know that the length of each suffix is l_{min}, that is, the vector for the end positions of the occurrences of suffixes $\vec{S}_e = \vec{S}_s{>}{>}(l_{min} - 1)$. However, the length of N-factors could be different; thus, we have to use another vector $\vec{N}_e$ to store the end positions of the occurrences of N-factors. Equation (5) shows the bit

[3] http://graphics.stanford.edu/~seander/bithacks.html.

Table VI. Generated Candidate Regions Using Bit Operations without any Constraint

	Vector																					
	0					5					10					15					20	
$\vec{P}_s$	1	0	0	1	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	1	0	0
$\vec{S}_e$	0	0	0	0	0	1	1	0	0	1	1	0	1	1	1	1	0	0	0	1	0	0
$\vec{N}_s$	0	0	1	0	0	0	0	1	0	1	0	0	0	0	1	0	0	1	1	0	0	1
$\vec{N}_e$	0	0	1	0	0	0	0	1	0	0	0	1	0	0	0	0	1	1	0	0	1	1
$\vec{A}_1$	1	1	1	0	0	1	0	1	0	1	0	1	1	1	1	0	0	1	0	0	0	1
$\vec{S}_{e_m}$	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0	1	0	0	0	1	0	0
$\vec{B}_1$	0	0	0	1	1	1	1	0	1	1	1	0	1	1	1	1	0	0	1	1	0	0
$\vec{B}_2$	1	1	1	1	1	1	0	1	1	1	0	1	1	1	1	0	1	1	1	0	1	1
$\vec{B}_3$	0	0	1	0	0	0	0	1	0	0	0	1	0	0	0	0	0	1	0	0	0	1
$\vec{B}_4$	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0
$\vec{B}_5$	0	0	0	1	1	1	0	0	1	1	1	1	1	1	1	0	0	0	1	0	0	0
$\vec{P}_c$	0	0	0	1	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0

Note: $l_{min} = 2$, `C` and `TTA` are the satisfied N-factors in Figure 6.

operations to calculate candidate occurrences.

$$
\begin{aligned}
Intermediate:\ \vec{A}_1 &= (\vec{N}_e | \vec{S}_e) - (\vec{N}_e << 1), \\
\vec{S}_{e_m} &= (\sim\vec{A}_1)\ \&\ \vec{S}_e\ \&\ (\sim\vec{N}_e), \\
Intermediate:\ \vec{B}_1 &= (\vec{N}_e - \vec{S}_{e_m})\ \&\ (\sim\vec{N}_e), \\
Intermediate:\ \vec{B}_2 &= \sim((\vec{S}_{e_m} << (l_{min} - 1)) - \vec{S}_{e_m}), \\
Intermediate:\ \vec{B}_3 &= (\vec{N}_e\ \&\ \vec{A}_1)\ |\ (\vec{N}_e\ \&\ \vec{S}_e), \\
Intermediate:\ \vec{B}_4 &= (\vec{N}_s - \vec{B}_3)\ \&\ (\sim\vec{N}_s), \\
Intermediate:\ \vec{B}_5 &= (\vec{B}_1\ \&\ \vec{B}_2)\ |\ \vec{B}_4, \\
\vec{P}_c &= \vec{P}_s\ \&\ \vec{B}_5.
\end{aligned}
\tag{5}
$$

We consider the same example that matches the RE $Q = $ `(G|T)A*GA*T*` on the text in Figure 1. Table VI shows the input vectors and intermediate results of the bit operations in Equation (5).

Similar to the bit vector $\vec{A}_0$ in Equation (4), in the intermediate bit vector $\vec{A}_1$ in Equation (5), $\vec{A}_1[i] = 0$ if i is the end position of a valid matching suffix. Then, we can get the vector $\vec{S}_{e_m}$, in which each bit 1 means an end position of a valid matching suffix. Note that, in contrast to $\vec{S}_{s_m}$ in Equation (4), the vector $\vec{S}_{e_m}$ needs to do an AND operation between $(\sim\vec{A}_1)\&\vec{S}_e$ and $(\sim\vec{N}_e)$. Before we explain why, let us consider the case in which a matching suffix $\mathcal{M}(S, t - l_{min} + 1)$ has the same end position t with a matching N-factor $\mathcal{M}(N_1, \pi_{n_1})$. According to the analysis in Section 3.3, this matching suffix $\mathcal{M}(S, t - l_{min} + 1)$ should not be used to generate candidate occurrences (see Figure 5(a)). In addition, if there is no matching suffix between the matching N-factor $\mathcal{M}(N_1, \pi_{n_1})$ and its right neighbor matching N-factor $\mathcal{M}(N_2, \pi_{n_2})$, then bit $\vec{A}_1[t]$ is 0. Therefore, algorithm PNS-BitG may choose an invalid matching suffix setting $\vec{S}_{e_m}[t] = 1$. In order to avoid choosing such an invalid matching suffix $\mathcal{M}(S, t - l_{min} + 1)$, we do the bit AND operation between $(\sim\vec{A}_1)\&\vec{S}_e$ and $(\sim\vec{N}_e)$.

For the purpose of calculating valid matching prefixes corresponding to each valid matching suffix, we use five intermediate vectors $\vec{B}_1, \ldots, \vec{B}_5$ to explain how the calculation works. Let $\mathcal{M}(S, \pi_s)$ be the rightmost matching suffix between two neighboring

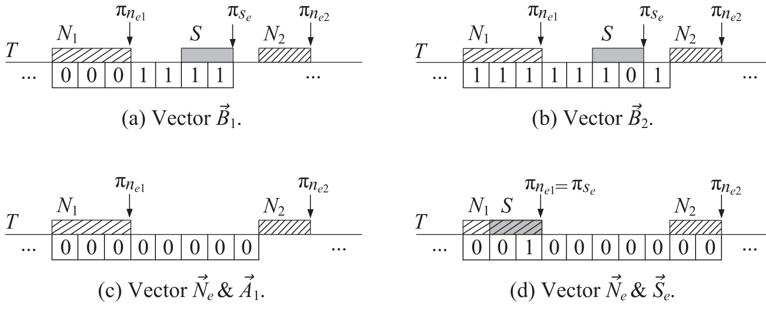


Fig. 9. Explanation of vectors.

matching N-factors $\mathcal{M}(N_1, \pi_{n_1})$ and $\mathcal{M}(N_2, \pi_{n_2})$. $\vec{B}_1[i] = 1$ if $\pi_{n_{e1}} < i \leq \pi_{s_e}$. An example vector $\vec{B}_1$ is shown in Figure 9(a). Vector $\vec{B}_2$ sets all the bits to 1 except the bit $\vec{B}_2[\pi_{s_e}]$, where π_{s_e} is the end position of a valid matching suffix. Figure 9(b) shows that $\vec{B}_2[i] = 0$ when $i = \pi_{s_e}$.

Given two matching N-factors $\mathcal{M}(N_1, \pi_{n_1})$ and $\mathcal{M}(N_2, \pi_{n_2})$, if there is no matching suffix between them, we say that $\mathcal{M}(N_1, \pi_{n_1})$ is a *useless matching N-factor*. We use $\vec{B}_3[\pi_{n_{e1}}] = 0$ to mark the useless matching N-factor $\mathcal{M}(N_1, \pi_{n_1})$ and keep $\vec{B}_3[\pi_{n_{e2}}] = 1$ at the same time. We use $\vec{N}_e \& \vec{A}_1$ to mark all the useless matching N-factors (see Figure 9(c)). However, a useless matching N-factor needs to be maintained if there is a matching suffix $\mathcal{M}(S, \pi_s)$, which happens at the same position as this matching N-factor. The reason is that a matching suffix $T[\pi_{s_s}, \pi_{s_e}]$ could be valid if there exists a prefix starting at position π_{p_s} and $\pi_{n_s} < \pi_{p_s} \leq \pi_{s_s}$, where π_{n_s} is the starting position of $\mathcal{M}(N_1, \pi_{n_1})$ (see Figure 9(d)). We use $\vec{N}_e \& \vec{S}_e$ to specify all the same end positions of matching N-factors and valid matching suffixes. Furthermore, we set each bit in $\vec{B}_4[\pi_{n_s} + 1, \pi_{s_e}]$ to 1. Then, we use vector $\vec{B}_5$ to combine all cases of possible valid matching prefixes. Finally, vector $\vec{P}_c$ marks all valid matching prefixes.

Table VI shows the generated $\vec{S}_{e_m}$ and $\vec{P}_c$, and the candidate occurrences are consistent with those shown in Figure 6.

We call the corresponding algorithm PNS-BitG. After using Equation (5) to get the two vectors $\vec{S}_{e_m}$ and $\vec{P}_c$, the algorithm gets each position pair π_s and π_p from $\vec{S}_{e_m}$ and $\vec{P}_c$, respectively. It then generates candidates $T[\pi_p, \pi_s]$ to do verification. The algorithm PNS-BitG requires the same time complexity as the algorithm PNS-BitC, but needs $O(6\lceil \frac{n}{w_s} \rceil)$ to store one more vector $\vec{N}_e$ than the algorithm PNS-BitC.

4.4. The Bit-Parallel PMNS Algorithms

As we described in Section 3.4, the necessary factors can be used to enhance the pruning power by integrating them into the PNS algorithm. We first introduce how to utilize the necessary factors by bit-parallel operations under the constraint $|N| \leq l_{min}$, with the corresponding algorithm named PMNS-BitC.

Given the set of necessary factors $M = \{M_1, \ldots, M_\kappa\}$, we can get the bit vector $\vec{M}_{s_i}$ for the starting positions of necessary factor $M_i \in M$ through Bit-Index. Since the valid matching suffix with position s_{max}, which is used to generate candidate regions, is the same as the one in PNS-BitC, we can use the same bit operations in Equation (4) to compute $\vec{S}_{s_m}$.

Table VII. Generated Candidate Regions with Necessary Factors Under the Constraint $|N| \leq l_{min}$ ($l_{min} = 2$)

	Vector																					
	0					5					10					15					20	
$\vec{P}_s$	1	0	0	1	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	1	0	0
$\vec{S}_s$	0	0	0	0	1	1	0	0	1	1	0	1	1	1	1	0	0	0	1	0	0	0
$\vec{N}_s$	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0	1
$\vec{M}_s$	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0
$\vec{A}_0$	1	1	1	0	1	0	1	1	1	1	0	1	1	1	0	1	1	1	0	1	1	1
$\vec{S}_{s_m}$	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0
$\vec{S}_{e_m}$	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0
$\vec{A}_2$	0	0	1	0	0	0	1	1	0	1	1	1	1	1	1	1	0	1	0	1	0	1
$\vec{M}_{s_m}$	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0
$\vec{B}_6$	0	0	0	1	1	1	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0	1
$\vec{P}_c$	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Consider a necessary factor $M_i \in M$. We utilize an intermediate vector $\vec{A}_2$ to calculate $\vec{M}^i_{s_m}$, which represents the starting positions m^i_{max} of the valid matching necessary factors in T. Bit $\vec{A}_2[j]$ is set to 0 if j is the starting position of a valid matching necessary factor. As shown in Table VII, bits $\vec{A}_2[5]$, $\vec{A}_2[8]$, and $\vec{A}_2[18]$ are set to 0, indicating the starting positions of three valid matching necessary factors. Note that the calculation of $\vec{A}_2$ has utilized vector $\vec{S}_{e_m}$ rather than $\vec{S}_{s_m}$. The reason is that a valid matching necessary factor can be a substring of a valid matching suffix. $\vec{S}_{s_m}[5]$ and $\vec{M}_{s_m}[5]$ can be used to illustrate it. The position 5 is s_{max} as well as m^i_{max}, since the matching suffix $\mathcal{M}(\texttt{GA}, 5)$ contains a matching necessary factor $\mathcal{M}(\texttt{G}, 5)$. Similar to the bit vector $\vec{B}_0$ in Equation (4), $\vec{B}^i_6$ can be computed using $\vec{N}_s - \vec{M}^i_{s_m}$, which represents all the possible valid matching prefixes by considering M_i. Equation (6) shows the bit operations to calculate $\vec{B}^i_6$.

$$\begin{aligned} \vec{S}_{e_m} &= \vec{S}_{s_m} >> (l_{min} - 1), \\ Intermediate: \vec{A}_2 &= (\vec{N}_s | \vec{S}_{e_m} | \vec{M}^i_s) - (\vec{S}_{e_m} << 1), \\ \vec{M}^i_{s_m} &= (\sim\vec{A}_2)\,\&\,\vec{M}^i_s, \\ Intermediate: \vec{B}^i_6 &= \vec{N}_s - \vec{M}^i_{s_m}. \end{aligned} \tag{6}$$

As we know, an occurrence of RE must contain *all* the necessary factors in M. We can use the operations $\&^{|M|}_{i=1} \vec{B}^i_6$ to get an intermediate vector $\vec{B}_6$, which represents the starting positions of all possible valid matching prefixes after considering all the necessary factors, as shown in Equation (7):

$$\begin{aligned} Intermediate: \vec{B}_6 &= \&^{|M|}_{i=1} \vec{B}^i_6, \\ \vec{P}_c &= \vec{P}_s \,\&\, \vec{B}_6. \end{aligned} \tag{7}$$

In fact, the difference between algorithm PMNS-BitC and algorithm PNS-BitC is the possible valid matching prefixes. We can see from Figure 10 that $\vec{B}_6$ has fewer valid matching prefixes than $\vec{B}_0$, which indicates that algorithm PMNS-BitC can achieve better pruning power than PNS-BitC because of necessary factors considered.

Table VII shows the generated $\vec{P}_c$ and $\vec{S}_{s_m}$. The candidate occurrences are $T[3, 6]$ and $T[5, 6]$, which are consistent with the candidate occurrences shown in Figure 7.

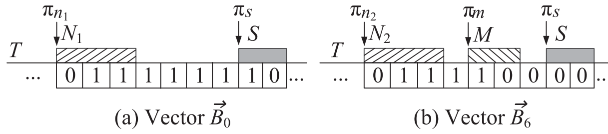


Fig. 10. Difference between vectors $\vec{B}_0$ and $\vec{B}_6$.

Algorithm PMNS-BitG is similar to the algorithm PNS-BitG, but it does not set the constraint $|N| \leq l_{min}$. In contrast to algorithm PNS-BitG, PMNS-BitG uses the bit vector $\vec{B}_7$ to represent the starting positions of possible valid matching prefixes. Finally, we can obtain the following equation, in which $\vec{B}_2$ and $\vec{B}_4$ come from Equation (5), and $\vec{B}_6$ is calculated by Equation (7).

$$\begin{aligned} Intermediate:\ \vec{B}_7 &= (\vec{B}_6 \,\&\, \vec{B}_2) \,|\, \vec{B}_4, \\ \vec{P}_c &= \vec{P}_s \,\&\, \vec{B}_7. \end{aligned} \tag{8}$$

5. CHOOSING GOOD N-FACTORS

The number of N-factors with regard to an RE could be large, and different N-factors could have different impacts on pruning non-answer substrings from candidates. For example, given an RE $Q = \texttt{C}^*\texttt{AAA}$, both `G` and `CGA` are N-factors with regard to Q. It is obvious that any occurrence of `CGA` in a text T also contains an occurrence of `G` in T. Thus, the N-factor `CGA` does not provide more filtering power than `G`. In addition, the number of chosen N-factors directly determines the number of bits set to 1 in the bit vector for N-factors (recall that we need to use Equation (2) to calculate matching positions of every N-factor). Consequently, a natural question is how to choose a small number of high-quality N-factors to improve search performance. In this section, we propose a concept called *core N-factors* to tackle this problem. Core N-factors consist of a small number of negative factors, which is a compromise between the pruning power and the number of core N-factors.

First, in Section 5.1, we explain how to define a small set of N-factors as core N-factors, and derive an upper bound on the number of core N-factors with regard to an RE Q. In Section 5.2, we describe the challenge of efficiently identifying the core N-factors, and propose an efficient algorithm. In Section 5.3, we develop a technique to speed up the generation of high-quality core N-factors by enabling early termination. In Section 5.4, we show that the number of core N-factors has a direct impact on the runtime for generating bit vector of N-factors, the smaller the better. We also observe that two core N-factors between a prefix starting at π_p and a suffix starting at π_s are redundant since each can prune the substring $T[\pi_p, \pi_s + l_{min} - 1]$ from candidates. Based on this observation, we prune those redundant core N-factors to reduce the number of core N-factors and to further improve search performance. Finally, in Section 5.5, we conduct an analysis of the pruning power of negative factors.

5.1. Core N-Factors

Definition 5.1 (Core N-factor). An N-factor with regard to an RE Q is called a *core N-factor* if each of its proper subsequences is not an N-factor with regard to Q.[4]

For example, for the RE $Q = \texttt{C}^*\texttt{AAA}$, the set of core N-factors with regard to Q is {`G`, `T`, `AC`, `AAAA`}. The substring `GA` is not a core N-factor since its subsequence `G` is an N-factor.

[4]We distinguish between substring and subsequence in this article. A substring of a string s has consecutive characters of s, while the characters in a subsequence may not be consecutive in s.

In order to compute an upper bound on the length of core N-factors with regard to an RE Q, we use a *factor automaton* [Simánek 1998] to check whether a string is an N-factor. A factor automaton is an automaton representing the set of all positive factors with regard to Q. It can accept any substrings of Q. Given an RE Q, a factor automaton can be constructed from a deterministic automaton of Q by adding epsilon transitions from the initial state to all other states and making all states final [Simánek 1998].

For a given RE Q, we first construct a nondeterministic factor automaton A_f. This automaton A_f can be further transformed to a unique minimal deterministic factor automaton A_{f_m}, which accepts exactly the same set of strings [Hopcroft and Ullman 2000]. Using A_{f_m}, we can prove an upper bound on the number of core N-factors.

THEOREM 5.2. *Let Q be an RE and A_{f_m} be its minimized deterministic factor automaton. The length of a core N-factor w.r.t. Q cannot be greater than the number of states in the longest acyclic path of A_{f_m}.*

PROOF. Let α be an N-factor with regard to Q and n_A be the number of states in the longest acyclic path of A_{f_m}. We prove that α is not a core N-factor if $|\alpha| > n_A$. According to the definition of factor automata [Simánek 1998], we know that only substrings of an RE can be accepted by A_{f_m}, that is, any N-factor could not be accepted by A_{f_m}. Let $|\alpha| = k$. For the substring $\alpha[1, k-1]$, there are two cases:

(i) $\alpha[1, k-1]$ is an N-factor. Then, according to Definition 5.1, α is not a core N-factor;
(ii) $\alpha[1, k-1]$ is not an N-factor. Then, $\alpha[1, k-1]$ must be a substring that can be accepted by A_{f_m}. Let $\alpha[1, k-1] = \alpha[1], \ldots, \alpha[k-1]$. Suppose that q_0 is the initial state in A_{f_m}. From each state q_{i-1}, A_{f_m} accepts $\alpha[i]$ and arrives at the state q_i $(1 \le i \le k-1)$, where q_{k-1} is the final state. Since $|\alpha| > n_A$, the number of states from q_0 to q_{k-1} must be greater than n_A. Therefore, there are at least two equivalent states $q_j = q_r$ $(j \le r)$, which means that at least one state in A_{f_m} accepts characters in $\alpha[1, k-1]$ more than once. Consider A_{f_m} as a finite automaton. According to the Pumping lemma for regular languages [Hopcroft and Ullman 2000], the substring $\alpha[1, k-1]$ can be represented in the form $uv^i w$, where $|uv| \le n_A$, $|v| \ge 1$, and $i \ge 1$. The automaton A_{f_m} first accepts u, arrives at a state q_j, and reaches the state q_r after seeing v. Since q_j and q_r are the same state in A_{f_m}, the fact that A_{f_m} accepts v^i means that there exists a transition from q_j to q_r for i times. When A_{f_m} accepts w, it will arrive at the final state q_{k-1}.

Therefore, we could construct a subsequence vw of $uv^i w$ such that vw can be accepted by A_{f_m}. Since $\alpha = uv^i w\alpha[k]$ could not be accepted by A_{f_m}, we know that $vw\alpha[k]$ cannot be accepted by A_{f_m} either. According to Definition 5.1, α is not a core N-factor.

Based on the analysis of these two cases, we conclude that α is not a core N-factor if $|\alpha| > n_A$. □

LEMMA 5.3. *The length of core N-factors with regard to an RE Q is upper bounded by $|Q|^2$.*

PROOF. An upper bound on the number of states in the longest acyclic path of the minimized deterministic factor automaton is the square of the number of characters that Q contains, that is, $|Q|^2$. Therefore, the length of any core N-factor should be no greater than $|Q|^2$. □

5.2. Constructing Core N-Factors Online

A naïve way to construct core N-factors is to enumerate strings with length $\le |Q|^2$ and check if they are core N-factors one by one (see Algorithm 4). The check process of a string s consists of two phases: (i) the function CHECKSUBSEQUENCE checks if a

subsequence of s is an N-factor (line 4), and (ii) otherwise, check if s itself is an N-factor, which can be determined by running the factor automaton $A_f(Q)$ (line 5).

ALGORITHM 4: NaïveCore

```
  Input: Alphabet Σ, an RE Q, a factor automaton A_f(Q);
  Output: A set of core N-factors C_N;
1 C_N ← ∅;
2 for length l ← 1; l ≤ |Q|^2; l++ do
3     for each string s ∈ Σ^l do
4         if CheckSubSequence(C_N, s) is false then
5             if s cannot be accepted by A_f(Q) then
6                 C_N ← C_N ∪ {s};
7 return C_N;
```

It can be time consuming to enumerate all the strings with a length $\leq |Q|^2$ and check each of them, especially since the core N-factors are query-dependent and we need to repeat the enumeration process for each search query. Take the RE $Q = \texttt{C}^*\texttt{AAA}$ as an example. The naïve approach needs to enumerate 340 strings within 16 iterations. In order to reduce the computational cost of constructing core N-factors, we present a more efficient method here.

Instead of enumerating all the strings (see line 3 in Algorithm 4), we can use two properties of core N-factors to generate core N-factors with a length l using a smaller set of strings with length $l-1$. In the following, we present those two important properties and explain how to utilize them for performance improvement.

Property 1. If a string x is a core N-factor, then its prefix $x[0, |x|-2]$ and suffix $x[1, |x|-1]$ can be accepted by the factor automaton $A_f(Q)$.

This property helps us improve the performance of computing N-factors by "joining" a set of strings with length $l-1$ to generate strings with length l. The formal definition of *string-join* is given here. For example, let $S = \{s_1, s_2\}$, $s_1 = \texttt{ACG}$, and $s_2 = \texttt{CGT}$, then $\widehat{S} = \{\texttt{ACGT}\}$.

Definition 5.4 (String-join). Given two strings s_1 and s_2, the *string-join* of s_1 and s_2, denoted by $\widehat{s_1 s_2}$, is computed as follows. If $|s_1| = |s_2| = l-1$ and $s_1[1, l-2] = s_2[0, l-3]$, then $\widehat{s_1 s_2}[0, l-2] = s_1$ and $\widehat{s_1 s_2}[l-1] = s_2[l-2]$; otherwise, $\widehat{s_1 s_2} = \emptyset$.

Definition 5.5 (String set self-join). Let $S = \{s_1, \ldots, s_v\}$ be a set of strings of length $l-1$. The *string set self-join* of S, denoted by $\widehat{S}$, is a set of nonempty strings $\widehat{s_i s_j}$ $(1 \leq i, j \leq v)$.

Compared with Algorithm 4, we can get the same set of core N-factors with length l by self-joining the set of strings with length $l-1$, each of which can be accepted by the factor automaton $A_f(Q)$.

Let S' be the set of strings that can be accepted by $A_f(Q)$, where each string in S' has a length $l-1$, and $S = \widehat{S'}$. We use $S_Q(\Sigma^l) \subseteq \Sigma^l$ and $S_Q(S) \subseteq S$ to represent strings that can be accepted by $A_f(Q)$, and use $S_C(\Sigma^l) \subseteq \Sigma^l$ and $S_C(S) \subseteq S$ to represent the core N-factors.

Theorem 5.6. *Sets $S_Q(\Sigma^l)$ and $S_Q(S)$ are equivalent, and sets $S_C(\Sigma^l)$ and $S_C(S)$ are also equivalent*[5].

[5]Two sets are equivalent if they have the same elements.

PROOF. We first prove that $S_Q(\Sigma^l) \equiv S_Q(S)$. Since $\Sigma^l \supseteq S$, we know that $S_Q(\Sigma^l) \supseteq S_Q(S)$. Assume that there is a string $x \in S_Q(\Sigma^l)$ and $x \notin S_Q(S)$; then, at least one substring in $x[0, l-2]$ and $x[1, l-1]$ does not belong to S'. Based on the assumption that $x \in S_Q(\Sigma^l)$, x must be accepted by $A_f(Q)$. Then, any substring of x can be accepted by $A_f(Q)$, that is, $x[0, l-2] \in S'$ and $x[1, l-1] \in S'$, which contradicts to the assumption that $x \notin S_Q(S)$. That is, for any string $x \in S_Q(\Sigma^l)$, we know that $x \in S_Q(S)$. Therefore, for any string $x \in S_Q(\Sigma^l)$ and $x \in S_Q(S)$, based on the fact that $S_Q(\Sigma^l) \supseteq S_Q(S)$, we know that $S_Q(\Sigma^l)$ and $S_Q(S)$ are equivalent.

We then prove that $S_C(\Sigma^l) \equiv S_C(S)$. Since $\Sigma^l \supseteq S$, we know that $S_C(\Sigma^l) \supseteq S_C(S)$. Assume that there is a string $x \in S_C(\Sigma^l)$ and $x \notin S_C(S)$; then, at least one substring in $x[0, l-2]$ and $x[1, l-1]$ does not belong to S', that is, $x[0, l-2]$ or $x[1, l-1]$ is an N-factor. By the definition of core N-factors, x must not be a core N-factor, which contradicts the assumption $x \in S_C(\Sigma^l)$. That is, for any string $x \in S_C(\Sigma^l)$, we know that $x \in S_C(S)$. Therefore, for any string $x \in S_C(\Sigma^l)$ and $x \in S_C(S)$, based on the fact that $S_C(\Sigma^l) \supseteq S_C(S)$, we know that $S_C(\Sigma^l)$ and $S_C(S)$ are equivalent. □

Property 2. Given a core N-factor x with a length greater than 1, let s be a string whose prefix $s[0, |s|-2]$ and suffix $s[1, |s|-1]$ can be accepted by $A_f(Q)$. If x is a subsequence of s, then $x[0] = s[0]$ and $x[|x|-1] = s[|s|-1]$.

Let $|x| = l_x$ $(1 < l_x \leq n)$. We construct string s as follows. Suppose that S' is a set of strings with length $l-1$ that can be accepted by $A_f(Q)$. Let $S = \widehat{S'}$ and $s \in S$. We know that there must exist two strings $s_1, s_2 \in S'$, such that $s_1 = s[0, l-2]$ and $s_2 = s[1, l-1]$. We first assume that $x[0] \neq s[0]$. Since x is a subsequence of a string $s \in S$, we know that x should be a subsequence of $s[1, l-1]$, that is, x is a subsequence of s_2. It contradicts the fact that any core N-factor is not a subsequence of a string in S. Therefore, $x[0]$ must be equivalent to $s[0]$. Similarly, we have that $x[l_x - 1] = s[l-1]$.

Algorithm 5 is an enhanced version of the naïve algorithm presented in Algorithm 4, by considering the two important properties of core N-factors delineated earlier. According to Property 1, the algorithm QUICKCORE in Algorithm 5 initially generates strings with length 1 and maintains those that can be accepted by the factor automaton $A_f(Q)$. By doing a self-join of the retained strings with length $l-1$, the algorithm generates the strings with one more character (lines 3–16).

Furthermore, when there is a core N-factor x that does not satisfy $s[0] = x[0]$ and $s[l-1] = x[|x|-1]$, we do not need to invoke the function CHECKSUBSEQUENCE based on Property 2 (lines 7–9). In the earlier example, for $Q =$ `C*AAA`, among the 65 strings, only 8 strings need to be further checked using the function CHECKSUBSEQUENCE.

5.3. Early Termination of Constructing Core N-Factors

We observe that the upper bound $|Q|^2$ on the length of an N-factor in Algorithm 5 is loose, which can result in many unnecessary iterations in the algorithm (see line 3). As we can see in Table VIII, all core N-factors for the sample query `C*AAA` (i.e., {`T`, `G`, `AC`, `AAAA`}) have been generated before the fifth iteration. However, the absence of incremental core N-factors generated at an iteration cannot guarantee that there will not be any more core N-factors generated in the following iterations, that is, a new core N-factor might still be generated even though nothing is produced in the previous iterations. For example, Algorithm 5 generates a new core N-factor in the fourth iteration, although it generates nothing new in the third iteration (see the second column in Table VIII) and it has to finish the total 16 iterations.

A natural question is whether we can early terminate such construction without loosing any core N-factors. The answer is yes, as shown next.

Table VIII. Early Termination for $Q = \texttt{C}^*\texttt{AAA}$, Where the Underlined Strings are Core N-factors ($k = 1$)

Iteration	Processing Strings Using Algorithm 5	Processing Strings Using Algorithm 6
1	A, C, $\underline{\texttt{T}}$, $\underline{\texttt{G}}$	A, C, $\underline{\texttt{T}}$, $\underline{\texttt{G}}$
2	AA, CA, CC, $\underline{\texttt{AC}}$	AA, CA, $\underline{\texttt{AC}}$
3	AAA, CAA, CCA, CCC	AAA, CAA
4	$\underline{\texttt{AAAA}}$, CAAA, CCAA, CCCA, CCCC	$\underline{\texttt{AAAA}}$, CAAA
5	CCAAA, CCCAA, CCCCA, CCCCC	∅
6	CCCAAA, CCCCAA, CCCCCA, CCCCCC	
...	...	
16	...	

ALGORITHM 5: QuickCore

```
   Input: Alphabet Σ, an RE Q, a factor automaton A_f(Q);
   Output: A set of core N-factors;
 1 C_N ← ∅; S ← Σ;
 2 Create a hash table H_T;
 3 for length l ← 1; l ≤ |Q|^2; l++ do
 4     for each string s ∈ S do
 5         FOUND ← false;
 6         if l > 2 then
 7             New a string y, y[0] ← s[0] and y[1] ← s[l − 1];
 8             if y is in H_T then
 9                 FOUND ← CheckSubSequence(C_N, s);
10         if FOUND is false then
11             if s cannot be accepted by A_f(Q) then
12                 C_N ← C_N ∪ {s}; S ← S − {s};
13                 Insert y into H_T, where y[0] ← s[0] and y[1] ← s[l − 1];
14         else
15             S ← S − {s};
16     S ← Ŝ;
17 return C_N;
```

Observation 1. Let C be the set of characters in e^*, and k be the summation of frequencies of characters of C in Q. For the i-th iteration in Algorithm 5, let s be any string that matches the pattern e^* with $|s| > k$, and we cannot find any string s' in the same iteration that can make $\widehat{ss'}$ an N-factor.

As we can see in the second column in Table VIII, Algorithm 5 keeps generating strings and checks if some are N-factors. In the second iteration, the string CC is generated whose length is greater than $k = 1$ and is used to generate strings CCA and CCC in the third iteration, and to generate strings CCAA, CCCA and CCCC in later iterations. As we know, CC is not an N-factor because $\texttt{C}^*$ exists in the RE Q. Using Algorithm 5, any generated string based on CC in the later iterations cannot be an N-factor, since the generated string must be accepted by the factor automaton $A_f(Q)$. Therefore, there is no need to keep CC in the second iteration.

Based on this observation, we propose the algorithm EarlyCore in Algorithm 6 to terminate the construction of core N-factors earlier without any false dismissal. The algorithm first counts the frequency of each character in each Kleene closure e^* in Q (lines 2–7), then uses it to check if a generated string needs to be reserved for self-join in the next iteration (line 12).

ALGORITHM 6: EarlyCore

```
   Input: Alphabet Σ, an RE Q, a factor automaton A_f(Q);
   Output: A set of core N-factors;
 1 C_N ← ∅; S ← Σ; C_S ← ∅; l ← 1;
 2 for each expression e* in Q do
 3 |  Let C ← the set of characters in e; Let freq(C) ← 0;
 4 |  for each character c in e do
 5 |  |  freq(C) ← freq(C) + number of c in Q;
 6 |  Insert C into C_S;
 7 while S is not empty do
 8 |  for each string s ∈ S do
 9 |  |  if ∃C ∈ C_S and s ∈ C^l then
10 |  |  |  if l = freq(C) + 1 then // bound of C
11 |  |  |  |  S ← S − {s};
12 |  |  ... // see lines 5 -- 15 in Algorithm 5
13 |  S ← Ŝ;  l ← l + 1;
14 return C_N;
```

Algorithm EarlyCore is expected to significantly reduce the number of iterations. For example, Table VIII shows the numbers of iterations for $Q = \texttt{C}^*\texttt{AAA}$, under Algorithm 5 and Algorithm 6, respectively. Algorithm 6 needs only 4 iterations compared with the 16 iterations needed in Algorithm 5.

Theorem 5.7. *Algorithm* EarlyCore *correctly finds all core N-factors.*

Proof. Given an RE Q, for each iteration in Algorithm 6, S is the current set of strings with length l. Let S_l be a subset of S that can be accepted by the factor automaton $A_f(Q)$. For ease of explanation, we consider the situation in which there are two different symbols in the Kleene closure, which can be easily extended to the cases in which more than two symbols exist in the Kleene closure. We demonstrate only the convergence of S with the Kleene closure being $(\alpha|\beta)^*$ for the reason that all the other Kleene closures are included in this case.

We prove it by demonstrating the following proposition: there would be no new core N-factors produced by a string $s \in S_l$ if s matches $(\alpha|\beta)^*$ and $|s| > freq(\{\alpha, \beta\})$ in the RE Q. Let $\ell = freq(\{\alpha, \beta\})$. We make an assumption that there is a core N-factor x produced by a string s that matches $(\alpha|\beta)^*$ and $|s| = \ell + 1$. The N-factor x can be formalized as $X_1 \cdots X_{\ell+1}Y$, where X_i $(1 \le i \le \ell + 1)$ is a symbol in α or β, and Y is a symbol other than $\{\alpha, \beta\}$. According to Definition 5.1, the subsequences of x must be accepted by $A_f(Q)$, that is, $X_1 \cdots X_\ell$ and $X_2 \cdots X_{\ell+1}Y$ can be accepted by $A_f(Q)$. We know that the total number of X_i in Q is ℓ, which means that at least one character X_i in $X_2 \cdots X_{\ell+1}Y$ matches $(\alpha|\beta)^*$. That is, the RE Q must contain $(\alpha|\beta)^*Y$. Therefore, $A_f(Q)$ can accept each string with the form $X_1 \cdots X_hY$, where h could be a positive integer. This result contradicts the assumption that $x = X_1 \cdots X_{\ell+1}Y$ is a core N-factor.

Based on this proposition, we can remove the strings matching the pattern $(\alpha|\beta)^*$ from S_l without missing any core N-factor when $l > freq(\{\alpha, \beta\})$. □

Theorem 5.8. *The set S in Algorithm 6 always converges to an empty set within $|Q|+1$ steps.*

Proof. It is easy to see that, if there is no Kleene closure in Q, the number of strings in S_l would converge to 0 within $|Q| + 1$ steps when l is big enough, since each string in

Fig. 11. An example of redundant matching core N-factors, in which N_1 is a discardable core N-factor.

$R(Q)$ has a finite length $|Q|$. Next, we concentrate on the situation in which the Kleene closure exists in Q.

Based on Theorem 5.7, we can remove the strings matching the pattern $(\alpha|\beta)^*$ from S_l without missing any core N-factor when $l > freq(\{\alpha, \beta\})$. Therefore, the size of S would converge to 0 eventually. In addition, s matches the pattern e^* with $|s| > k$, and the length of any of its substring s' that matches pattern e^* must be no greater than k, that is, $|s'| \leq |Q|$. Therefore, the length of the string generated in the last iteration will be no greater than $|Q| + 1$.

Therefore, the size of S would converge to 0 eventually. □

5.4. Removing Discardable Core N-Factors for Efficiently Generating Bit Vector of N-factors

Recall that for a given set of N-factors, we need to generate vector $\vec{N_s}$ (and vector $\vec{N_e}$) for N-factors (see Section 4). The construction of vectors $\vec{N_s}$ and $\vec{N_e}$ has two main steps: (i) calculating a single vector for each N-factor (see Equations (1) and (2)) and (ii) performing BIT-OR operations among all generated bit vectors for a single N-factor. Both steps introduce time overhead to answer the RE query.

We observed from the experiments (see Section 7) that there might be more than one matching core N-factor between a pair of a matching prefix and matching suffix. We call these matching core N-factors *redundant matching core N-factors*. A core N-factor is *discardable* if its matching substrings in the text are always redundant with another matching core N-factor, and discarding such discardable core N-factors will not affect the filtering power of our approach.

Figure 11 shows such an example, in which $\mathcal{M}(P_1, \pi_{p_1})$ and $\mathcal{M}(P_2, \pi_{p_2})$ are matching prefixes, $\mathcal{M}(S_1, \pi_{s_1})$ and $\mathcal{M}(S_2, \pi_{s_2})$ are matching suffixes, and $\mathcal{M}(N_1, \pi_{n_1})$, $\mathcal{M}(N_2, \pi_{n_2})$, and $\mathcal{M}(N_3, \pi_{n_2})$ are matching core N-factors. N-factor N_1 is discardable since, after removing it, we can still determine that neither $[\pi_{p_1}, \pi_{s_1}]$ nor $[\pi_{p_2}, \pi_{s_2}]$ could answer the RE Q.

In order to identify those discardable core N-factors, we need to check every consecutive matching core N-factor, which is impractical since all these operations need to be done online, which introduces non-negligible time. An alternative way is to sacrifice the filtering power slightly and remove the core N-factors that have a higher probability to be discardable. Consequently, we are willing to sacrifice the filtering power to gain efficiency by choosing a set of core N-factors, as follows.

Heuristic Rule for Removing Discardable Core N-Factors: Ideally, an undiscardable core N-factor N should have the following two features:

(i) Its prefix overlaps with the suffix of a prefix, so that its matching substring has a high probability of pruning matching prefixes in T. We use f_1 to represent the average overlap ratio between the core N-factor N and every prefix $P_i \in P_{set}$ $(i \in [1, \ldots, \iota])$. Let $\widetilde{P_i \cdot N}$ represent the overlapping substring of a prefix P_i and N.

$$f_1(N, P_{set}) = \frac{1}{\iota} \cdot \sum_{i=1}^{\iota} \left(\frac{|\widetilde{P_i \cdot N}|}{|N|} \right). \tag{9}$$

(ii) It does not overlap with other core N-factors, so that it has a low probability of being a redundant core N-factor with another one. We use f_2 to represent the average

overlap ratio between N and another core N-factor $N_i \in N_{set}$ ($i \in [1, \ldots, \nu]$).

$$f_2(N, N_{set}) = \frac{1}{\nu - 1} \cdot \sum_{i=1}^{\nu-1} \left(\frac{|\widetilde{N \cdot N_i}| + |\widetilde{N_i \cdot N}|}{|N|} \right). \tag{10}$$

Based on these two features, we hope that a chosen undiscardable core N-factor could make f_1 as high as possible and f_2 as low as possible. Accordingly, we develop Rule 1 to determine whether a core N-factor shall be kept.

Rule 1. Let $P_{set} = \{P_1, \ldots, P_\iota\}$ be a set of prefixes with regard to an RE Q and $N_{set} = \{N_1, \ldots, N_\nu\}$ be a set of core N-factors with regard to Q. A core N-factor N_i can be chosen if $f_1(N, P_{set}) - f_2(N, N_{set}) > 0$.

5.5. Pruning Power of N-factors

In this section, we analyze the pruning power of N-factors via a theoretical analysis. The results are experimentally evaluated on real datasets in Section 7.1.4.

Consider a substring $T[a, d]$ conforming to a PNS pattern, in which the substring $T[a, b]$ is a matching prefix and the substring $T[c, d]$ is a matching suffix ($a \leq b \leq c \leq d$). Let $p_1(n)$ denote the probability that the length of $T[b, c]$ is equal to n and $p_2(n)$ denote the probability that there is at least one N-factor matching in $T[b, c]$. Then, the probability of filtering any prefix matching in T using N-factors can be calculated as follows:

$$p_f = \sum_{n=0}^{|T|-2 \cdot l_{min}} p_1(n) \times p_2(n). \tag{11}$$

We first calculate $p_1(n)$. Let $S = \{S_1, \ldots, S_\mu\}$ be the set of suffixes with regard to the RE Q. As defined in Watson [2003], each suffix in S has the same length l_{min}. Let $B_n(\mu, l_{min})$ denote the number of substrings in $T[b, c]$ with a length n such that any suffix in S is not a substring of $T[b, c]$. Then, we could get the following recurrence function for $n \geq 0$:

$$B_n(\mu, l_{min}) = \begin{cases} |\Sigma|^n & \text{if } n < l_{min}, \\ |\Sigma| \cdot B_{n-1}(\mu, l_{min}) - \mu \cdot B_{n-l_{min}}(\mu, l_{min}) & \text{otherwise.} \end{cases}$$

Then, we have

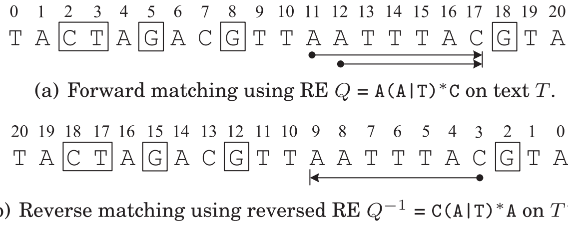
$$p_1(n) = \frac{B_n(\mu, l_{min})}{|\Sigma|^n} \times \frac{\mu}{|\Sigma|^{l_{min}}}. \tag{12}$$

Similarly, let $N = \{N_1, \ldots, N_\nu\}$ be the set of N-factors with regard to the RE Q and each N-factor in N has a length l'. Then, we have

$$p_2(n) = 1 - \frac{B_n(\nu, l')}{|\Sigma|^n}. \tag{13}$$

6. DETERMINING MATCHING DIRECTION

In previous sections, we introduced the N-factor concept to effectively reduce the number of candidates of an RE on the text T, and introduced bit-parallel algorithms to accelerate the pruning process. Thereafter, the remaining candidates have to be verified based on an automaton of Q (denoted $A(Q)$), via a matching process named *forward matching*. As we observe, the total number of valid matching prefixes in T decides the number of times that the automaton $A(Q)$ has to be invoked. In the case that the total number of valid matching prefixes is large, the automaton $A(Q)$ has to be executed many times, and the matching process could be costly. This analysis inspires us to



(a) Forward matching using RE Q = A(A|T)*C on text T.

(b) Reverse matching using reversed RE Q^{-1} = C(A|T)*A on T^{-1}.

Fig. 12. An example of the impact of matching direction on performance.

query a reversed RE (denoted Q^{-1}) on the reversed text (denoted T^{-1}), which might help to reduce the number of times that the automaton $A(Q^{-1})$ has to be performed to further improve the matching performance.

In Section 6.1, we first explain the impact of the matching direction (i.e., forward vs. reverse) on matching performance. In Section 6.2, we then propose an approach to determining a good matching direction by using BitIndex.

6.1. Impact of Matching Direction

Figure 12 shows the impact of matching direction using an RE $Q =$ A(A|T)*C on text T. By using forward matching, core N-factors are G, CA, CC, and CT, prefixes with regard to Q are AA and AT, and suffixes with regard to Q are AC and TC. According to algorithm PNS, $T[11, 12]$ and $T[12, 13]$ are valid matching prefixes, and $T[1, 2]$, $T[6, 7]$, and $T[16, 17]$ are valid matching suffixes. Therefore, $T[11, 17]$ and $T[12, 17]$ are the two candidates for Q. We need to run automaton $A(Q)$ starting from positions 11 and 12 to verify these two candidates, respectively. We find that substring $T[12, 17]$ actually will be checked twice, which means there are certain duplicated calculations using $A(Q)$ in verification steps. Furthermore, through the analysis on the experimental results reported in Section 7, we find that most candidates are the matching results of Q after pruning by N-factors, which indicates that the cases shown in Figure 12(a) are very common in verifying candidates.

Figure 12(b) shows that, by using reverse matching, our algorithm needs to invoke automaton $A(Q^{-1})$ only once, where reversed RE is $Q^{-1} =$ C(A|T)*A, core N-factors are G, AC, CC, and TC, prefixes with regard to Q^{-1} are CA and CT, and suffixes with regard to Q^{-1} are AA and TA. Note that there are still two candidates $T^{-1}[3, 8]$ and $T^{-1}[3, 9]$ for the reversed RE Q^{-1}, but only one candidate $T^{-1}[3, 9]$ needs to be verified, since the automaton $A(Q^{-1})$ will continue checking $T^{-1}[9, 9]$ when it arrives the final state at $T^{-1}[8, 8]$. Finally, we could reverse each result to Q^{-1} to get the final results to Q.

Now, we show the correctness and completeness of using reverse matching.

Theorem 6.1. *Given a text T and an RE Q, using the bit-parallel algorithms* PNS-BitC *and* PNS-BitG *to match Q^{-1} on T^{-1} will not cause any false dismissals.*

Proof. For an RE Q, let R be the set of occurrences with regard to Q on T, and R^{-1} be the set of occurrences with regard to Q^{-1} on T^{-1}. First, we prove that, for any occurrence r ($r \in R$), the reverse occurrence r^{-1} must exist in R^{-1}. Assume that there is an occurrence r_1 ($r_1 \in R$), and the reverse occurrence $r_1^{-1} \notin R^{-1}$. Since r_1 is an occurrence of Q on T, there must be a substring r_1^{-1} on T^{-1}. Moreover, because r_1 can be recognized by the automaton $A(Q)$ with regard to Q, it is certain that substring r_1^{-1} can also be accepted by the automaton $A(Q^{-1})$ with regard to Q^{-1}. Therefore, the substring r_1^{-1} on T^{-1} is an occurrence of Q^{-1}, which contradicts the assumption that $r_1^{-1} \notin R^{-1}$.

	Vector																				
	0					5					10					15					20	
$\vec{P}_s$	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0
$\vec{S}_s$	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
$\vec{N}_s$	0	0	1	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0	1
$\vec{P}_c$	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0

(a) Generate candidate regions using bit operations under the constraint $|N| \leq l_{min}$ ($l_{min} = 2$ and G and CT are the satisfied N-factor in Figure 12(a)).

c_i	$\mathcal{C}_f(c_i)$
00000001	1
00000010	1
...	...
11111111	8

(b) Mapping table.

Fig. 13. An example of counting bits by lookup table.

Similarly, we can also prove that, for any occurrence r^{-1} ($r^{-1} \in R^{-1}$), there must be an occurrence r ($r \in R$) that is the reverse of r^{-1}. Consequently, we can conclude that there is a one-to-one correspondence between the occurrences in R and R^{-1}, that is, reverse matching will not cause any false dismissals. $\square$

6.2. Determining Matching Direction Using BitIndex

As explained earlier, matching direction has a direct impact on the performance of the matching process. Ideally, we want to quickly determine the matching direction that actually generates a smaller number of candidates.

Let $\mathcal{C}_f$ be the total number of valid matching prefixes for Q using forward matching on T, and $\mathcal{C}_r$ be the number of valid matching prefixes for Q^{-1} using reverse matching on T^{-1}. Intuitively, if $\mathcal{C}_f \leq \mathcal{C}_r$, we use forward matching, otherwise, we use reverse matching. In the following, we demonstrate how to efficiently derive $\mathcal{C}_f$ and $\mathcal{C}_r$ by using BitIndex.

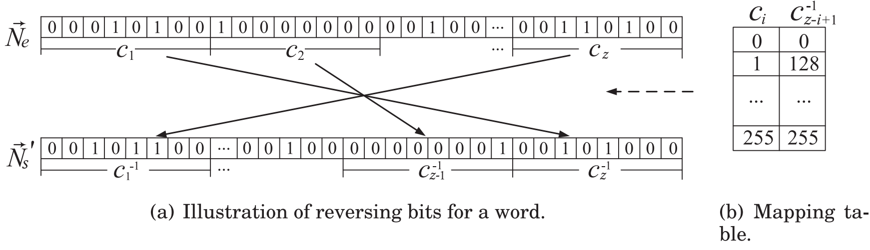
(i) *Computing* $\mathcal{C}_f$. Remember that using algorithm PNS-BitC or PNS-BitG, we can calculate vector $\vec{P}_c$ based on BitIndex. Each bit 1 in $\vec{P}_c$ represents a starting position of a valid matching prefix with regard to Q. Therefore, $\mathcal{C}_f$ can be easily calculated by counting the bits 1s of $\vec{P}_c$.

Let $\vec{P}_c$ occupy n bits and z bytes, that is, $\vec{P}_c = c_1 c_2 \ldots c_z$, where $z = \lceil \frac{n}{8} \rceil$. Then, $\mathcal{C}_f = \sum_1^z \mathcal{C}_f(c_i)$, where $\mathcal{C}_f(c_i)$ is the summation of 1 bits in a byte c_i. Each byte corresponds to an unsigned char, thus we can use a precomputed mapping table to store the mapping pairs between every unsigned char and number of 1 bits in its byte [Anderson 2005], as shown in Figure 13(b). The size of this mapping table is small (i.e., only $2^8 = 256$ bytes) and counting bits 1s of a word w can be done in constant time [Anderson 2005].

Figure 13(a) shows the calculated vector $\vec{P}_c$ with regard to RE $Q =$ A(A|T)*C on text T using algorithm PNS-BitC. Several bytes will be used to represent the vector $\vec{P}_c$, by counting the bits on the bytes (there are two 1 bits on $\vec{P}_c$ at positions 11 and 12, respectively).

(ii) *Computing* $\mathcal{C}_r$. In order to compute $\mathcal{C}_r$ for the reversed text T^{-1}, we need to compute vectors that represent the starting positions of matching prefixes, matching suffixes, and matching N-factors for Q^{-1} in T^{-1}, denoted as $\vec{P}_s{}'$, $\vec{S}_s{}'$, and $\vec{N}_s{}'$ respectively. Later, we show an efficient approach to calculate the vectors for Q^{-1} in T^{-1} using $\vec{P}_e$, $\vec{S}_e$, and $\vec{N}_e$, respectively. Recall that vectors $\vec{P}_e$, $\vec{S}_e$, and $\vec{N}_e$ represent the end positions of matching prefixes, matching suffixes, and matching N-factors for the RE Q in text T, respectively.

For ease of presentation, we first show how to calculate $\vec{N}_s{}'$ using $\vec{N}_e$. As we know, $\vec{N}_s{}'$ is a reversed vector of $\vec{N}_e$; instead of scanning the reversed text T^{-1} to get $\vec{N}_s{}'$,



(a) Illustration of reversing bits for a word.

(b) Mapping table.

Fig. 14. An example of reversing bits by lookup table.

we can easily convert $\vec{N_e}$ to $\vec{N_s}'$, as shown in Figure 14(a). Since each vector is stored in z bytes, we first convert bits in each byte, then put the last byte c_z in $\vec{N_e}$ as the first one c_1^{-1} in $\vec{N_s}'$, the second last byte c_{z-1} in $\vec{N_e}$ as the second one c_2^{-1} in $\vec{N_s}'$, and so on, until the first byte c_1 in $\vec{N_e}$ is converted as the last one c_z^{-1} in $\vec{N_s}'$. Again, each byte c_i in $\vec{N_e}$ corresponds to an unassigned char, and we can precompute another mapping table to store the mapping pairs between each byte c_i and its converted byte c_{z-i+1}^{-1}, as shown in Figure 14(b) [Anderson 2005]. Thereafter, converting a byte c_i in $\vec{N_e}$ to its corresponding byte c_{z-i+1}^{-1} in $\vec{N_s}'$ can be done in constant time by looking up the mapping table.

Similarly, vector $\vec{P_s}'$ is the reversed vector of $\vec{S_e}$, and vector $\vec{S_s}'$ is the reversed vector of $\vec{P_e}$. Using the approach just delineated, we can derive vectors $\vec{P_s}'$ and $\vec{S_s}'$ based on vectors $\vec{S_e}$ and $\vec{P_e}$. Thereafter, we can use the same approach of calculating $\mathcal{C}_f$ to calculate $\mathcal{C}_r$ by using $\vec{P_s}'$, $\vec{S_s}'$, and $\vec{N_s}'$.

In order to accelerate the calculations of $\mathcal{C}_f$ and $\mathcal{C}_r$, we can further estimate them by using a subsequence of T. Let λ be a sampling ratio. We randomly choose a subsequence with length $\lambda \cdot |T|$ at positions from 0 (beginning of the text) to $(1 - \lambda) \cdot |T|$.

7. EXPERIMENTS

In this section, we present experimental results of the N-factor technique on multiple real datasets.

Experiment Setup. We conducted the experiments on three public datasets, including Human Genome, Protein sequences, and English texts.

—*Human Genome*: The genomic sequence (GRCh37) was assembled from a collection of DNA sequences, which consisted of 24 chromosomes with a length varying from 48 million to 249 million.[6]

—*Protein sequences*: We adopted the database Pfam 26.0, which contained a large amount of protein families and is composed of Pfam-A and Pfam-B.[7] The symbol set consisted of all the capital English letters, excluding “O” and “J.” We randomly picked text with a length varying from 101 to 9,143 from Pfam-B.

—*English texts*: We used DBLP-Citation-network[8], which included 1,632,442 blocks, each of which corresponded to one paper. Each block contained several attributes of a paper, for example, title, authors, abstract, and so on. We extracted the abstract from

[6]http://hgdownload.cse.ucsc.edu/goldenPath/hg18.
[7]http://www.ncbi.nlm.gov/pmc/articles/PMC3245129.
[8]https://aminer.org.citation.

every block. The symbol set consisted of 52 English letters, 10 digits, white space, and punctuation characters.

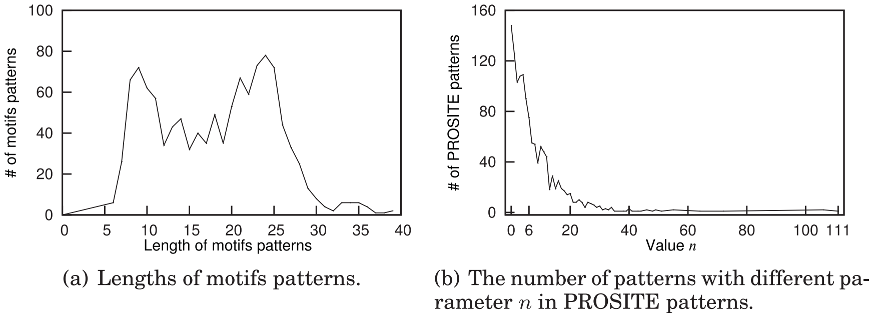
We selected the following algorithms as the representative of state-of-the-art RE matching algorithms in this set of experiments. The first three existing algorithms are developed for matching REs on a set of short sequences. When a sequence contains more than one occurrences of a query, only the first occurrence of the query is returned. For the purpose of comparability, we modified their source code so that they can find all the occurrences, as our algorithms do.

—*Agrep* is an efficient string matching tool that supports simple strings, extended strings, and REs simultaneously. For REs, it utilizes the bit-parallel algorithm BPThompson to match the occurrences [Wu and Manber 1992].
—*Gnu Grep* is a necessary-factor, filtering-based algorithm. It first matches all initial matchings by necessary factors, then verifies them by running an automaton.[9]
—*NR-grep* uses reversed prefixes to identify initial matchings by the algorithm RegularBNDM [Navarro 2001; Navarro and Raffinot 1999], then verifies them using an automaton.
—*RE2* is a Google-developed software library that supports matching for regular expressions. It also includes a regex generator. It was first developed in 2009 and has frequent updates. We downloaded the version on June 28, 2015 and used it to do the comparison.[10]

We implemented the following algorithms using core N-factors; the algorithms are based on the structure BitIndex except for the ones with superscript BWT.

—PNS is the algorithm that utilizes core N-factor to prune candidates.
—PNS-BitC is the bit-parallel PNS algorithm with the constraint that the length of core N-factors is no greater than l_{min}.
—PNS-BitG is also the bit-parallel PNS algorithm. In contrast to PNS-BitC, it has no constraint on the length of core N-factors.
—PMNS utilizes the core N-factors and necessary factors to further improve pruning power.
—PMNS-BitC is the bit-parallel PMNS algorithm with a length constraint on core N-factors.
—PMNS-BitG is similar to PMNS-BitC, but it has *no* constraint on the length of core N-factors.
—PNS-BitC^H is an extension of algorithm PNS-BitC by considering the optimization technique of removing discardable core N-factors introduced in Section 5.4.
—PNS-BitC^D is another extension of algorithm PNS-BitC that implements the optimization technique of choosing a good matching direction introduced in Section 6.
—PNS-BitC$^+$ considers both the optimization of removing discardable core N-factors and the optimization of choosing a good matching direction.
—PNS-BitC^{BWT} is an extension of algorithm PNS-BitC but uses the BWT format to store the text T. In Section 4, the structure BitIndex is introduced to facilitate the identification of starting positions of matching factors, including matching prefixes, matching suffixes, matching N-factors, and matching necessary factors. Alternatively, the bit vectors for matching factors can be calculated using the method of storing the text T in a BWT format. Yang et al. [2013] propose forming an inverted list of starting positions for a substring α in a constant time ($O(|\alpha|)$) by simulating searches

[9]https://www.gnu.org/software/grep.
[10]https://github.com/google/re2/.



(a) Lengths of motifs patterns.

(b) The number of patterns with different parameter n in PROSITE patterns.

Fig. 15. Statistical results on real-life query workloads.

using BWT, which also stores the suffix array to quickly locate the positions of substrings, then computes the bit vector $\vec{\alpha}$ by setting the bit $\vec{\alpha}[p]$ as 1, where p is the starting position of α in the inverted list. Similarly, they propose computing the bit vector for ending positions of α by setting the bit $\vec{\alpha}[p + |\alpha| - 1]$ as 1. PNS-BitC^{BWT} utilizes the bit vectors calculated from the BWT format. In the same way, we can get the algorithms PNS^{BWT}, PNS-BitG^{BWT}, PMNS^{BWT}, PMNS-BitC^{BWT}, and PMNS-BitG^{BWT} by using the BWT format of T to compute bit vectors.

We extracted several subsequences of length ranging from 10 million to 100 million from the Human Genome. For the other two datasets, the size of each dataset varied from 10MB to 100MB.

Query workloads. In order to evaluate the algorithms, it is important to select a good set of REs. Since there are no standard benchmarks or "random" REs [Navarro and Raffinot 2004], we used synthetic query workloads generated by RE2. We also used two real query workloads for DNA and protein applications to evaluate our techniques.

—*Synthetic query workloads*. We utilized RE2 to generate regular expressions. RE2 supports full and partial matching for regular expressions using a finite-state machine.

This RE generator can create regular expressions according to four given parameters: (i) atom set (i.e., alphabet); (ii) operator set; (iii) maximal number of characters that appear in an RE, denoted as *maxatoms*; and (iv) maximal number of operators that appear in an RE, denoted as *maxopts*.

For the three public datasets (i.e., Human Genome, Protein sequences, and English texts), we used their alphabets to define an atom set, and use $\{\cdot, |, *, (,)\}$ to define an operator set, which is consistent with our RE definition in Section 2. In the RE2 setting, the difference between *maxatoms* and *maxopts* is not greater than 1. Thus, we only need to vary parameters *maxopts* to generate different query workloads for different datasets. Each query workload contained 50 regular expressions.

—*Real query workloads*. We also used two real query workloads as follows.

(1) Motif patterns. A sequence motif is a nucleotide or amino-acid sequence pattern, which is also exported as an RE. This query workload consisted of 1,161 patterns. We analyzed the length of motif patterns (i.e., the number of characters in a pattern). As shown in Figure 15(a), the length of motif patterns varies from 6 to 39, and 92% of the patterns have lengths from 8 to 28.
(2) PROSITE patterns. We downloaded this query workload from the PROSITE database, which consisted of a large collection of biologically meaningful signatures that are described as patterns. Generally, PROSITE patterns have the form of

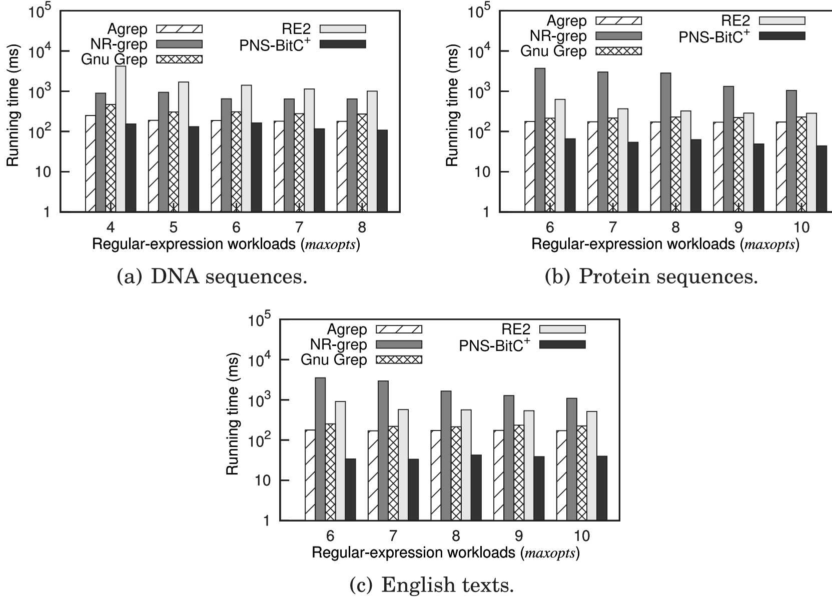


Fig. 16. Performance comparison of different algorithms on supporting RE matching ($|T| = 5 \times 10^7$).

[RK]$-x(m, n)-${DE}$-$A$-$Y, where [RK] represents (R|K) in our setting, $x(m, n)$ indicates any string of a length between m and n (e.g., $x(1, 3)$ is equivalent to $\Sigma(\Sigma|\epsilon)(\Sigma|\epsilon)$), {DE} stands for all amino acids except D and E, and A$-$Y represents A·Y in our setting. Figure 15(b) shows the number of patterns with different parameter n in PROSITE patterns, in which 52% of PROSITE patterns have values $n < 6$. The total number of PROSITE patterns was 1,308.

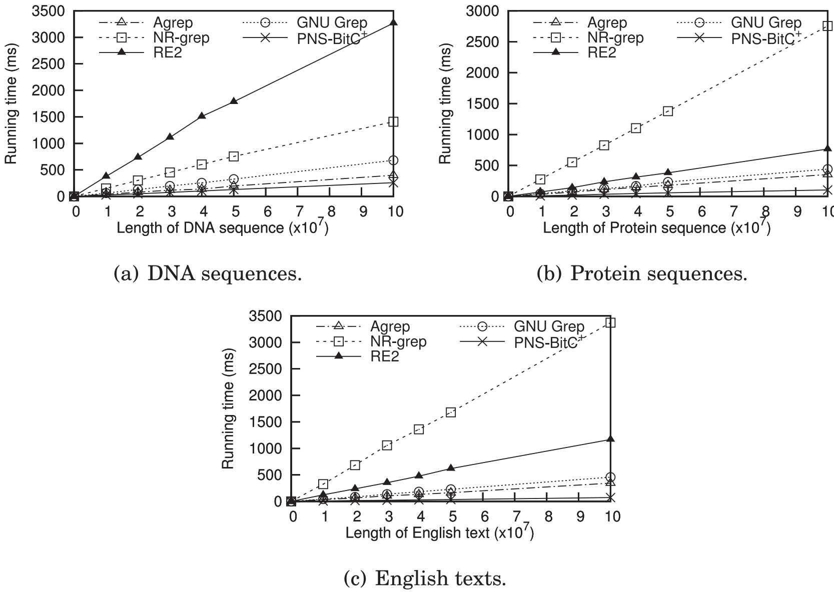
For each query workload, we recorded the average runtime of each RE as the performance metric. The runtime of matching an RE in our experiments means the whole online processing time, which mainly includes the following parts: (i) generating an automaton for each RE in the query workload, (ii) generating bit vectors for all derived positive and negative factors from the RE, (iii) generating candidates, and (iv) verifying candidates.

All the algorithms were implemented using GNU C++. The experiments were run on a PC with an Intel 3.10GHz Quad Core CPU i5 and 8GB memory with a 500GB disk, running a Ubuntu (Linux) 64b operating system. All index structures were in memory. Constructing the basic structure of BitIndex (i.e., representing the occurrences of each character using a bit vector) was done offline.

7.1. Qualitative Tests with Synthetic Query Workloads

7.1.1. Comparison of RE Matching Algorithms. In the first set of experiments, we compared the newly proposed algorithms with existing algorithms on supporting RE matching. We selected the algorithm PNS-BitC$^+$ as representative of the newly designed algorithms.

We tested the online runtime using different query workloads by varying the parameter *maxopts*. Each dataset contained sequences with a length of 50 million. Figure 16 shows the runtime of Agrep, NR-grep, Gnu Grep, RE2, and PNS-BitC$^+$ on the three datasets. Algorithm PNS-BitC$^+$ achieved the best performance. Figure 16(a) shows that, when answering the queries (*maxopts* $= 4$) on DNAs, PNS-BitC$^+$ took 153ms



(a) DNA sequences.

(b) Protein sequences.

(c) English texts.

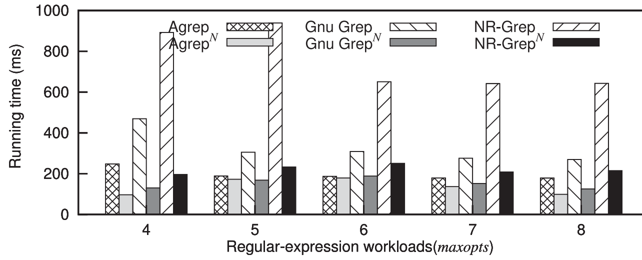
Fig. 17. Scalability of different RE matching algorithms.

only, compared to 247ms, 892ms, 469ms, and 4,186ms spent by Agrep, NR-grep, Gnu Grep, and RE2, respectively. The superiority of PNS-BITC+ over existing algorithms was even more evident on English texts. For instance, when $maxopts = 6$, PNS-BitC^{+} took 33ms only, which was 4.4, 106, 6.6, and 26.5 times faster than Agrep, NR-grep, Gnu Grep, and RE2, respectively.

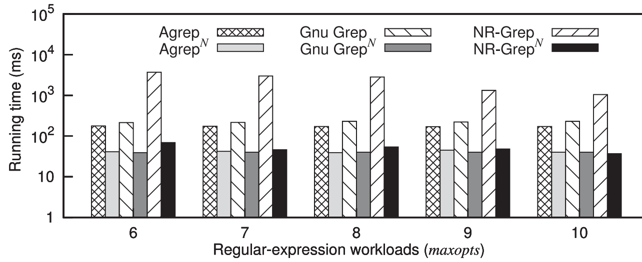
It is interesting to see that RE2 performed the worst on the DNA dataset, whereas NR-grep performed the worst on the other datasets. The runtime decreased when we increased the value of *maxopts*, since a larger *maxopts* can produce regexes with a longer pattern, resulting in fewer matching occurrences in the text.

Meanwhile, we tested the performances of different algorithms while varying data size. For each dataset, we ran the regexes in its corresponding 5 query workloads that are reported in Figure 17. Figure 17 shows the average runtime for each query. When increasing the size of datasets, the runtime of all five algorithms increased. PNS-BitC^{+} consistently outperformed the other algorithms on each dataset. For example, when answering queries on the 100MB English dataset, PNS-BitC^{+} used 71ms, Agrep used 344ms, NR-grep used 3,370ms, Gnu Grep used 458ms, and RE2 used 1,169ms. The result showed that, among the five RE matching algorithms, PNS-BitC^{+} was most stable when the size of the alphabet or string length increased.

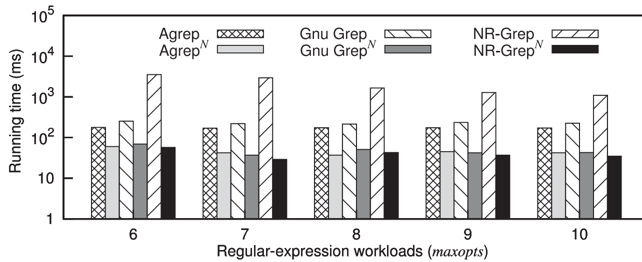
7.1.2. Improving Existing Algorithms Using N-Factors. The second set of experiments evaluates the benefit of N-factors. Instead of using our newly proposed algorithms, based on N-factors, we modified three existing algorithms by integrating N-factors and reported their performance in Figure 18. Each superscript N means that the algorithm integrated N-factors. We used the same setting reported in Section 7.1.1. It can be observed that the modified algorithms achieved a much better performance than the original ones. For instance, the N-factor improved the performance of the algorithms Agrep and Gnu Grep by about 3 times on English text. The improvement was most evident for the algorithm NR-grep, as its runtime decreased from 1,092ms to 43ms for the queries with $maxopts = 10$. For Protein sequences, Figure 18(b) shows that Agrep^{N}



(a) DNA sequences.



(b) Protein sequences.



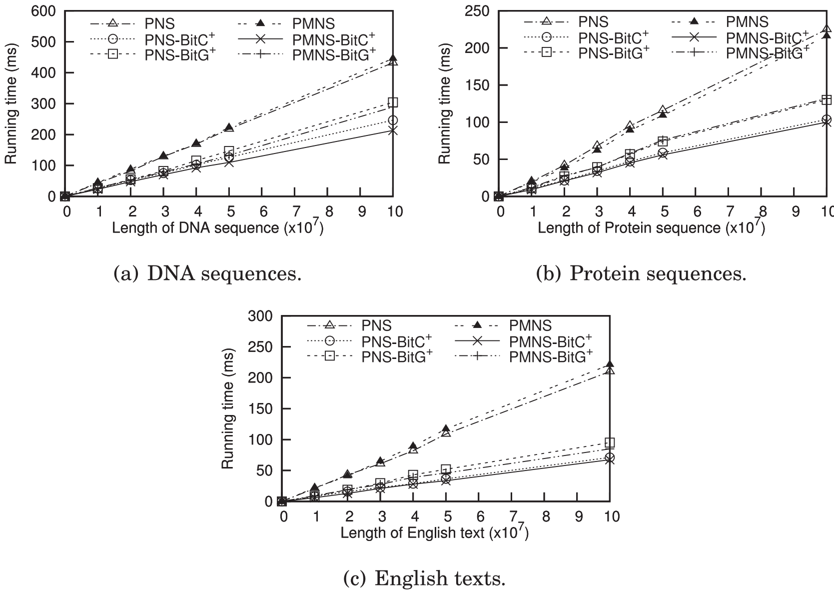
(c) English texts.

Fig. 18. Improving existing approaches by using N-factors.

reduced the time of Agrep from 170ms to 45ms, Gnu Grep^N reduced the time of Gnu Grep from 221ms to 40ms, and NR-grep^N reduced the time of NR-grep from 1,321ms to 48ms for queries with *maxopts* $= 9$. These results showed the effective pruning power of N-factors.

7.1.3. Scalability of Using N-Factors. The third set of experiments evaluates the scalability of using N-factors under input texts with different lengths. We used the same setting reported in Figure 17. Figure 19 shows that the runtime was increased slowly when we increased the length of the sequence for different algorithms of using N-factors. We can see that the bit-parallel algorithms performed much better than algorithm PNS and PMNS, with algorithm PMNS-BitC$^+$ performing the best among the six algorithms. For instance, when the length of the DNA sequence was 100 million, algorithm PMNS-BitC$^+$ spent 213ms only, compared to 432ms, 446ms, 256ms, 304ms, and 289ms spent by algorithms PNS, PMNS, PNS-BitC$^+$, PNS-BitG$^+$ and PMNS-BitG$^+$, respectively. We also get consistent results in English texts and Protein sequences.

Note that the runtime of algorithms PMNS, PMNS-BitC$^+$, and PMNS-BitG$^+$ was close to that of PNS, PNS-BitC$^+$, and PNS-BitG$^+$, respectively. The reason is that not all queries have necessary factors. For queries without necessary factors, algorithms



(a) DNA sequences.

(b) Protein sequences.

(c) English texts.

Fig. 19. Scalability of approaches using N-factors.

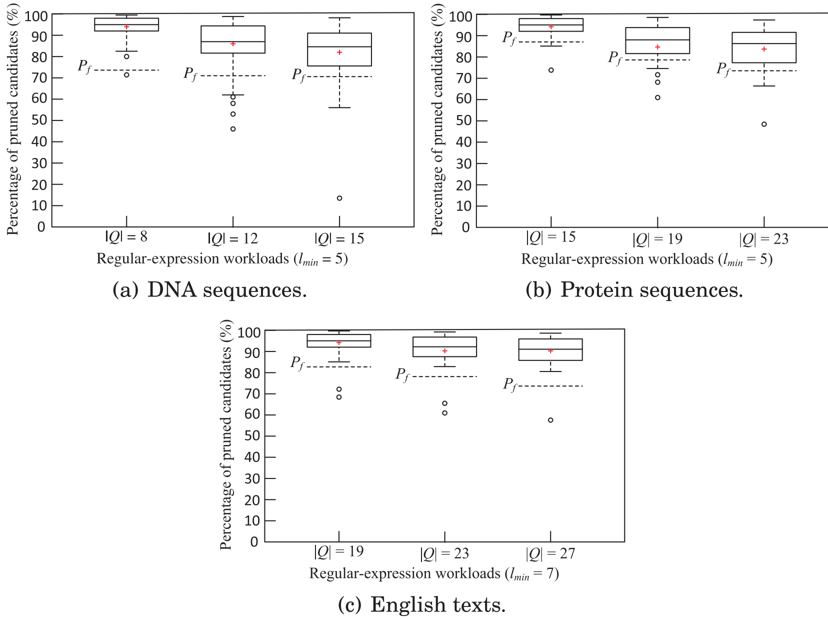
PMNS, PMNS-BitC$^+$, and PMNS-BitG$^+$ performed similar to algorithms PNS, PNS-BitC$^+$, and PNS-BitG$^+$.

7.1.4. Pruning Power of N-Factors. In Section 5.5, we analyzed the pruning power of N-factors. Our fourth set of experiments calculates the probability value p_f in Equation (11) to determine the pruning power of using N-factors for three datasets.

For DNA sequences, we used three RE workloads with query lengths 8, 12, and 15, respectively, each of which contained 100 REs, and $l_{min} = 5$. Each RE in a workload generated different numbers of suffixes and N-factors with different lengths. We ran the REs in each workload on DNA sequences with 50 million characters and calculated the average probability p_f. Figure 20(a) shows the pruning power of using N-factors on DNA sequences. The dotted lines represent the computed values p_f, which ranged from 70.071% to 73.781% in the three workloads.

Each box region in the figure represents the pruning power in the range between the first quartile (the top 25% REs) and the third quartile (the top 75% of REs). The line in the box is the median value of the pruning power. The plus sign and the circles indicate the mean value of the pruning power and the outliers, respectively. For the workload of $|Q| = 8$, the top 25% of REs in the workload could prune 98.331% of the false negatives (i.e., substrings that do not need to be verified) and the top 75% of REs could prune 92.008% of the false negatives. The other two workloads also provided significantly high pruning power.

For Protein sequences, the query lengths of three selected RE workloads were 15, 19, and 23, respectively, and l_{min} was still 5. Compared to DNA sequences, using core N-factors achieved a higher probability p_f, which ranged from 72.812% to 86.875% in the three workloads, as can be observed from Figure 20(b). The reason is that the chance that an N-factor exists in the candidate region is increased, that is, a higher value of p_2 in Equation (11), as the size of alphabets increased. Our experimental results confirm this finding. For a workload of $|Q| = 15$ on Protein sequences, the top 25% of REs and



(a) DNA sequences.

(b) Protein sequences.

(c) English texts.

Fig. 20. The ability of pruning false negatives using N-factors.

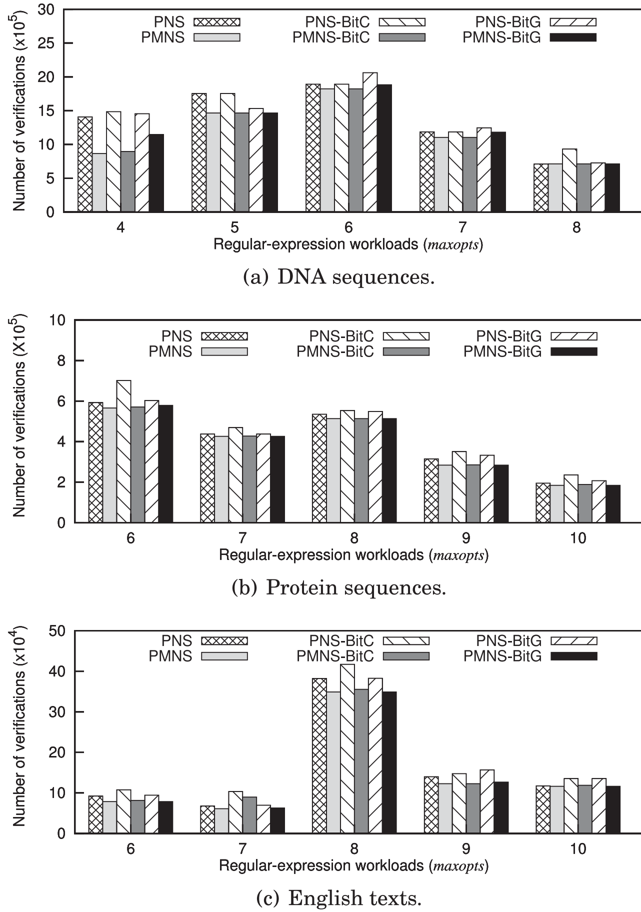
75% of REs could prune 98.87% and 91.818% of the false negatives, respectively, which were greater than the values for a workload of $|Q| = 15$ on DNA sequences.

We got similar results for English texts, as shown in Figure 20(c). The average probability value p_f for English texts ranged from 74.062% to 82.187% when varying query lengths from 19 to 27, and $l_{min} = 7$.

Note that the value p_f derived from Equation (11) for each workload on the three datasets was lower than the experimental mean value. The reason is that our analysis is based on the assumption that data is evenly distributed, which may not be true in real datasets.

We then tested the number of verifications when using different N-factor–based algorithms on the three datasets, each of which consisted of 100 million characters. As we can see from Figure 21, algorithm PMNS, which considers PNS patterns and necessary factors, and its corresponding bit-parallel algorithms PMNS-BitC and PMNS-BitG, required fewer verifications. For example, in Figure 21(a), for the queries with $maxopts = 5$, the number of verifications using PMNS-BitC was 1,465,254, compared to the verification number 1,752,900 computed by PNS-BitC. However, in some cases, the filtering advantage of PMNS-BitC was not evident. For example, in Figure 21(b), the number of verifications of algorithm PMNS-BitC was 437,924 on the queries with $maxopts = 7$, which was similar to the number for algorithms PNS-BitC and PNS-BitG, and was even higher than that of PNS.

7.1.5. Construction of Core N-Factors. We propose three different methods to construct core N-factors in Section 5, that is, algorithm NaïveCore, algorithm QuickCore, and algorithm EarlyCore. Our fifth set of experiments compares the construction time of core N-factors using different algorithms, with the experimental results listed in Table IX. We did not include the results of algorithm NaïveCore in the experiments due to its very poor performance. The construction time of algorithm QuickCore was



(a) DNA sequences.

(b) Protein sequences.

(c) English texts.

Fig. 21. Comparison of verification numbers.

Table IX. Time for Constructing Core N-factors (unit: ms)

DNA		*maxopts*=4	*maxopts*=5	*maxopts*=6	*maxopts*=7	*maxopts*=8
	QuickCore	7.38	18.52	14.33	9.46	32.79
	EarlyCore	0.22	0.79	3.20	0.73	1.22
Protein		*maxopts*=6	*maxopts*=7	*maxopts*=8	*maxopts*=9	*maxopts*=10
	QuickCore	8.96	15.77	22.49	6.34	17.18
	EarlyCore	0.34	2.10	3.06	1.30	0.83
English		*maxopts*=6	*maxopts*=7	*maxopts*=8	*maxopts*=9	*maxopts*=10
	QuickCore	41.61	16.81	18.38	8.91	13.30
	EarlyCore	1.03	1.17	0.77	1.13	1.15

not stable, varying from 6.34ms to 41.61ms. The reason is that the number of sequences to be processed could grow exponentially if there is a Kleene closure of size one. On the other hand, algorithm EarlyCore was much more efficient than QuickCore. It was also stable since it avoids generating N-factors caused by Kleene closure. In the best

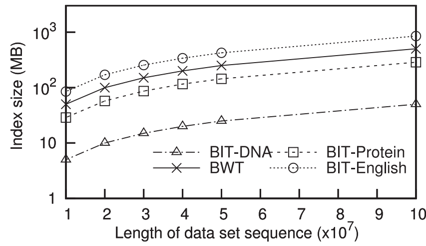


Fig. 22. Index size of BitIndex and BWT on different datasets.

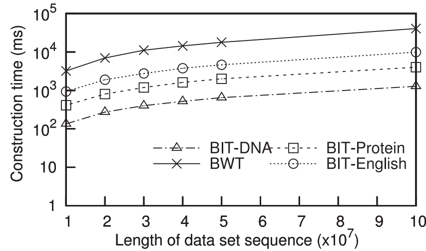


Fig. 23. The index construction time of BitIndex and BWT on different datasets.

situation, it only took 0.22ms to construct the core N-factors; the worst construction time was just 3.20ms.

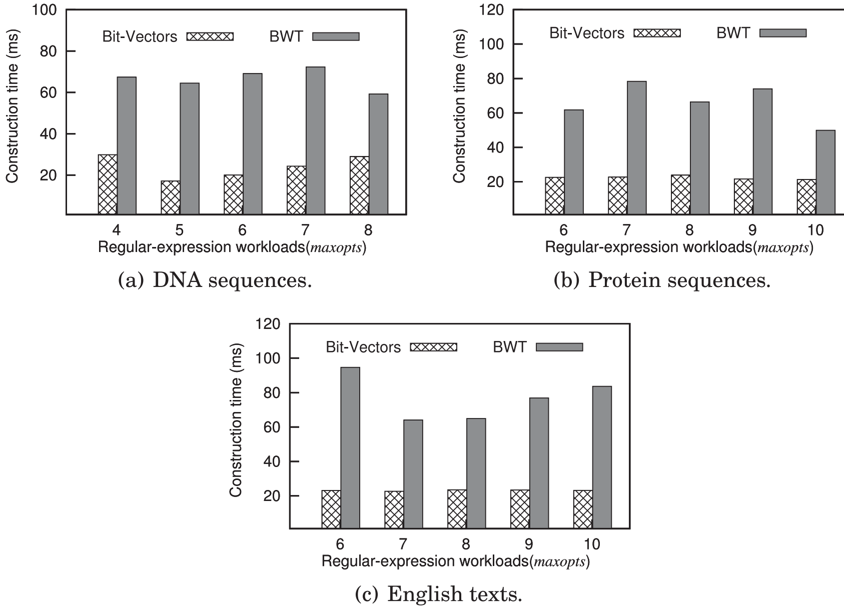
7.1.6. BitIndex *vs BWT.* In the sixth set of experiments, we demonstrated the advantage of our newly proposed index structure BitIndex by comparing its index size, construction time, and performance in forming bit vectors for matching factors with BWT.

Figure 22 shows the index size comparison of BitIndex and BWT on three datasets. Note that the sizes and construction time of BWT were the same on three datasets, since the BWT format is an alphabet-insensitive structure. Consequently, we report the result of BWT only once.

First, let us look at the index sizes of BitIndex. We can see that it needed more space to store the index as the alphabet size increased. For the DNA sequences, it had the smallest index size, which was half of the original data size, for example, the index size was 50MB when the data size was 100MB. This is because it only records a bit for each character in every position of string sequence. It is not surprising to see that it occupied larger index sizes for Protein sequences and English texts than for DNA sequences, which were 3 and 8.5 times the original data size, respectively. For example, for the 100MB English text, its index size reached 850MB, since there were 52 characters, 10 digits, white space, and 5 punctuation characters in the alphabet of English texts.

Next, let us look at the index sizes of BWT. It took more space to store the index on DNA and Protein sequences than BitIndex. For example, the index size of BWT was 500MB for the sequence with length 100 million. However, BWT occupied less space than BitIndex on English text. The BWT format took 5 times the original data size while BitIndex needed 8.5 times the original data size.

We also studied the index construction time when increasing data size on three datasets. Here, the construction time includes the file reading time and index writing time. It can be seen from Figure 23 that BitIndex needed more time to construct the index as the alphabet size increased. Regardless of the dataset, BitIndex consistently took less time than BWT since BitIndex needs only $O(n)$ time to scan the text T with



(a) DNA sequences.

(b) Protein sequences.

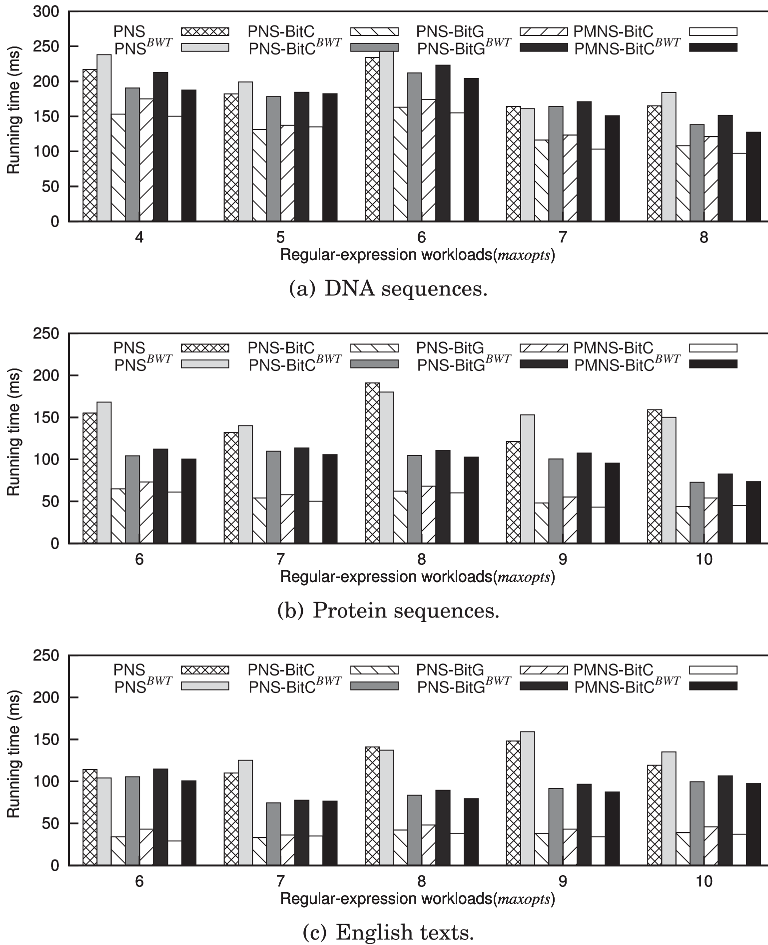
(c) English texts.

Fig. 24. Construction time for bit vectors of matching factors.

length n, while BWT takes $O(n + n \log n)$ time if using a quicksort algorithm to sort the transformed array. It cost 40,627ms to construct the BWT index for the sequence with length 100 million, which was almost 13 times the construction time of BitIndex on DNA sequences.

As described in Section 4.1, in order to prune the candidates with core N-factors by bit-parallel operations, we need to construct the bit vectors for the matching factors in an online fashion, including matching prefixes, matching suffixes, and matching core N-factors. Next, we compared the bit-vector construction time for all matching factors based on BitIndex and BWT. For the BitIndex structure, the time complexity of constructing bit vectors is $O(\frac{|T|}{w_s} \cdot (\iota \cdot l_{min} + \mu \cdot l_{min} + \nu \cdot l'))$, where w_s is the word size; ι, μ, and ν represent the number of prefixes, suffixes, and core N-factors with regard to the RE Q, respectively; and l' denotes the average length of core N-factors. For the BWT, it costs $O(\iota \cdot (l_{min} + C_P) + \mu \cdot (l_{min} + C_S) + \nu \cdot (l' + C_N))$ time to construct the bit vectors, in which C_P, C_S, and C_N, respectively, represent the average occurrence number of each matching prefix, matching suffix, and matching core N-factor in text. Figure 24 shows that BitIndex took less time to construct the bit vectors than BWT. For example, for queries with *maxopts* = 6 on DNA sequences, BitIndex took only 20ms to construct the bit vectors, while BWT took 69ms. We got similar results on Protein sequences and English texts. The reason is that the process of constructing bit vectors on BWT incurs a lot of random accesses in the memory, since the position list calculated by BWT is unordered. On the contrary, BitIndex performed well, as the access in memory is sequential while constructing bit vectors on BitIndex.

In the last part of this set of experiments, we compared the performance of different algorithms using N-factors based on BitIndex and BWT. For presentation clarity, we report only the results of algorithm PMNS-BitC as the representative for the algorithms that consider N-factor and necessary factor simultaneously. We can see from Figure 25 that all bit parallel-based algorithms on BitIndex performed better than algorithms on BWT, since they need less time to construct the bit vectors of matching factors.



(a) DNA sequences.

(b) Protein sequences.

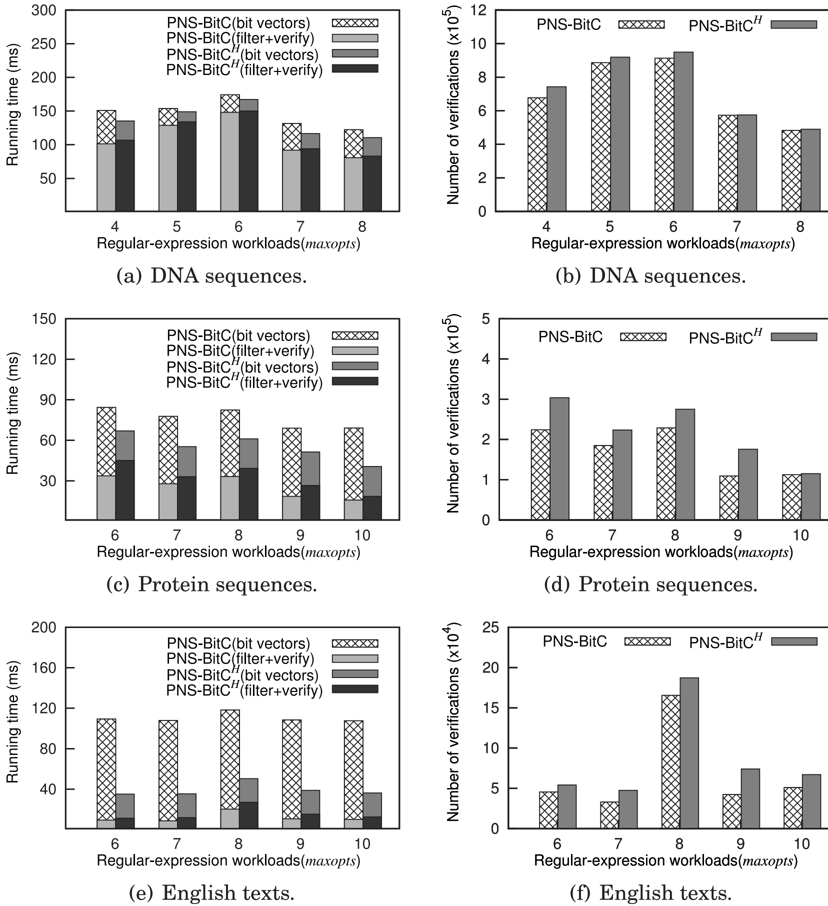
(c) English texts.

Fig. 25. Comparison of the performance of the algorithms based on BitIndex and BWT.

However, we also observe that the runtime of algorithm PNS on BitIndex could be worse than that on BWT, such as the queries with *maxopts* = 8 and 10 on Protein sequences, where the runtime on BitIndex was 191ms and 159ms, respectively, compared to 180ms and 150ms on BWT. This is because algorithm PNS utilizes the positions of matching factors and, for the BitIndex structure, it also needs to use bit operations to calculate the positions from bit vectors.

7.1.7. Effect of Removing Discardable Core N-Factors. To achieve a better matching performance, we introduced a heuristic rule to remove the discardable core N-factors in Section 5.4. As we described in Section 4, each algorithm actually consists of two phases. The first phase is to compute bit vectors of the occurrence positions of factors; the second phase is to match the initial candidates and verify them by an automaton. In this set of experiments, we compared the runtime and pruning power before and after removing discardable core N-factors by a heuristic rule to demonstrate the impact of removing discardable core N-factors.

As mentioned in Section 5.4, the main objective in removing discardable core N-factor is to improve performance (i.e., runtime). We therefore show the time of constructing bit



(a) DNA sequences. (b) DNA sequences.

(c) Protein sequences. (d) Protein sequences.

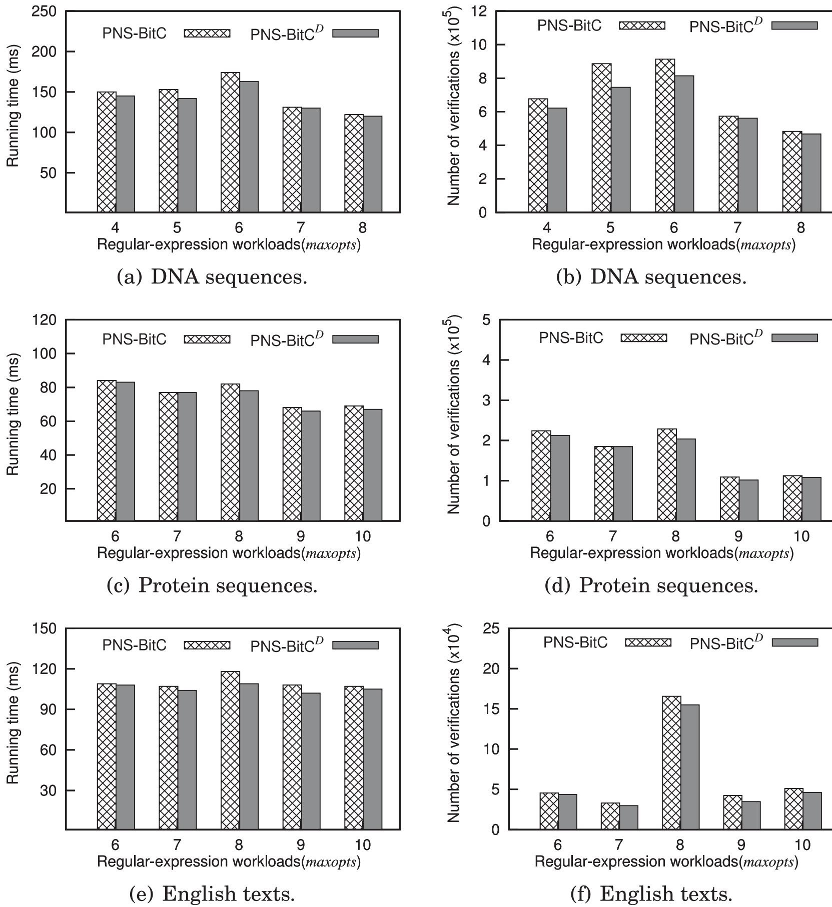
(e) English texts. (f) English texts.

Fig. 26. Effect of removing discardable core N-factors ($|T| = 5 \times 10^7$).

vectors and the time for filtering and verifying. We adopted PNS-BitC as the running algorithm, and superscript H means that the algorithm has utilized the heuristic rule to remove discardable core N-factors. Each dataset contained sequences with length 50 million.

Figures 26(a), 26(c), and 26(e) show that, when we removed discardable core N-factors, the total runtime of algorithm PNS-BitC was reduced. For the Protein and English datasets, the benefits of removing discardable core N-factors were very obvious. For example, the total runtime for queries with $maxopts = 6$ was reduced from 109.41ms to 35.15ms on English texts. This is because, after removing discardable core N-factors, we decreased the computing time for bit vectors from 100ms to 23ms, although it needed to verify a few more candidates. For the DNA sequences, the reduction in runtime was not evident, although the runtime for queries with $maxopts = 5$ and 6 was decreased to 148ms and 167ms, respectively.

Next, we studied the influence on the pruning power of core N-factors after removing discardable core N-factors. Theoretically, it will weaken the pruning power of core N-factors, since the heuristic rule reduces the number of core N-factors to accelerate the construction of their bit vectors.



(a) DNA sequences.

(b) DNA sequences.

(c) Protein sequences.

(d) Protein sequences.

(e) English texts.

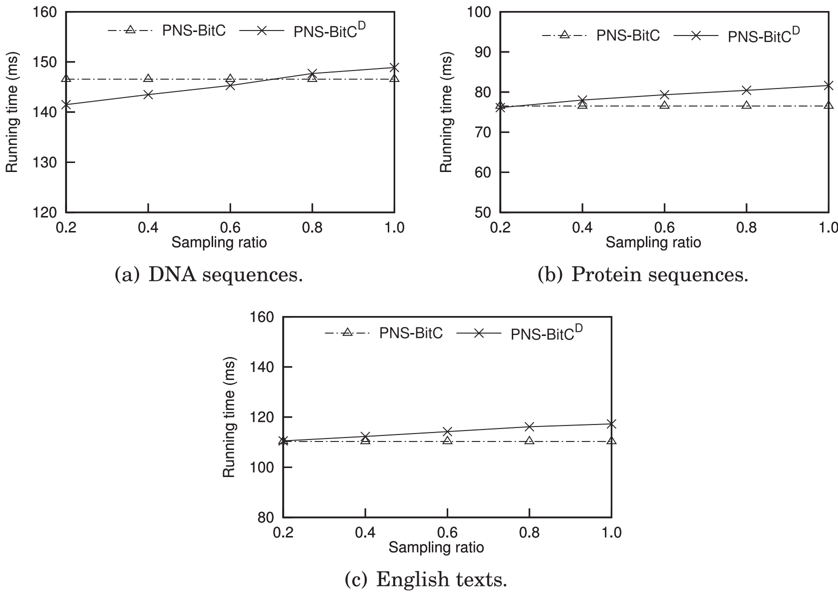
(f) English texts.

Fig. 27. Effect of choosing good matching direction.

As shown in Figures 26(b), 26(d), and 26(f), the candidate numbers for the queries have increased slightly. Queries with $maxopts = 6$ in the Protein dataset increased the most, with the candidate number increased from 224,132 to 303,867.

7.1.8. Effect of Choosing Good Matching Direction. In our last set of experiments, we evaluated the effect of choosing good matching direction. As described in Section 6, further determining matching direction can improve the efficiency of verifications; we implemented an algorithm PNS-BitC^D, which employed the optimization technique of choosing good matching direction based on algorithm PNS-BitC. Figure 27 shows the results.

Figures 27(a), 27(c), and 27(e) show the comparison results of runtime between PNS-BitC and PNS-BitC^D on three different datasets when sampling ratio $\lambda = 0.2$. We can see that the runtime decreased for some queries that utilize reverse matching, including the queries with $maxopts = 5$ and 6 in DNA sequences, queries with $maxopts = 8$ in Protein sequences, and queries with $maxopts = 8$ and 9 in English texts. For instance, the runtime for queries with $maxopts = 5$ and 6 in DNA sequences decreased to 142ms and 163ms from 153ms and 174ms, respectively. The reason is that the algorithm generated fewer candidates when using reverse matching, as shown in



(a) DNA sequences.

(b) Protein sequences.

(c) English texts.

Fig. 28. Impact of sampling ratio on performance.

Table X. Performance Comparison of Different Algorithms for Motif Patterns (unit: ms)

Algorithms	Agrep	NR-grep	Gnu Grep	RE2	PNS-BitC$^+$
Runtime	488.44	161.81	460.01	428.78	72.72

Figures 27(b), 27(d), and 27(f). For the queries with *maxopts* = 5 and 6 in DNA sequences, the candidate number decreased to 745,489 and 814,223 from 887,022 and 913,716, respectively. We also observed that the improvements of reverse matching were not evident on some queries. For example, the runtime of queries with *maxopts* = 7 and 8 on the DNA dataset only decreased to 130ms and 120ms from 132ms and 123ms, respectively. This is because the improvement on the candidate number was less, as shown in Figure 27(b).

We also report the results when varying the sampling ratio from 0.2 to 1, and employed the average runtime of five sets of queries as the performance metric. As we increased the sampling ratio, the cost of determining matching direction increased. As shown in Figure 28(a), when the sampling ratio $\lambda = 0.2$, the algorithm PNS-BitC^D took 141.5ms, compared to 146.7ms of PNS-BitC. While the sampling ratio $\lambda = 0.8$, PNS-BitC^D took more time than PNS-BitC. We got similar results on the Protein sequences and English texts, as shown in Figures 28(b) and 28(c). Consequently, we know that choosing a good matching direction can result in a better performance when the sampling ratio is small.

7.2. Real-Life Test with Real Query Workloads

We performed experiments to compare five RE matching algorithms—Agrep, NR-grep, Gnu Grep, RE2, and PNS-BitC$^+$—using two real query workloads, motif patterns, and PROSITE patterns, respectively.

Table X shows the comparison results of using motif patterns on the DNA dataset. PNS-BitC$^+$ outperformed other algorithms. For instance, the runtime of PNS-BitC$^+$ was only 72.72ms, compared to 488.44ms, 161.81ms, 460.01ms, and 428.78ms spent by Agrep, NR-grep, Gnu Grep, and RE2, respectively.

Table XI. Performance Comparison of Different Algorithms for PROSITE Patterns (unit: ms)

Algorithms	Agrep	NR-grep	Gnu Grep	RE2	PNS-BitC$^+$
Runtime	484.78	216.18	732.99	357.16	137.83

Table XI shows the comparison results of using PROSITE patterns on protein sequences. Among the five algorithms, PNS-BitC$^+$ achieved the best performance. Gnu Grep took the most time (732.99ms), which was about 5.3 times the runtime of PNS-BitC$^+$.

8. RELATED WORK

8.1. Classical Approaches to Regular Expression Matching

We first introduce the classical approaches to answering RE query in a text. Their basic idea is first transforming an RE into an automaton, then running it from each position in the text to verify if the substring is an occurrence of the regular expression. An occurrence will be reported whenever a final state of the automaton is reached [Baeza-Yates and Gonnet 1996; Mohri 1997].

Thompson proposed the definition of NFA, as well as a searching algorithm with $O(mn)$ time complexity based on the simulation of the NFA, called NFAThompson [Thompson 1968]. NFAThompson stores the set of currently active states, and for each newly read character, it looks over every currently active state to get new activated ones. Afterwards, these new states are added to a new active state set. For these new active states, NFAThompson follows all the ϵ-transitions until all the other reachable states are obtained, and adds them into the new active state set. In this way, whether an occurrence of RE is found can be determined by checking if the final state is activated.

DFAClassical [Aho et al. 1985] supports a regular expression matching by simulating the DFA, which can guarantee a linear search time of $O(n)$. Considering the approach based on the simulation of NFA, multiple states will be activated when the NFA accepts a new text character, and it needs to maintain the set of active states. On the contrary, for the simulation of DFA, only a definite state will be activated when the DFA accepts a new character, costing less compared to NFA. However, more states exist during the simulation of DFA than NFA, which makes DFAClassical cost more space than NFAThompson.

DFAModules [Myers 1992] is a compromise between deterministic and nondeterministic simulation, whose core idea is splitting the NFA into some modules, including $O(k)$ nodes, making them deterministic and maintaining an NFA consisting of $O(m/k)$ modules based on Thompson construction.

Based on Thompson's NFA simulation, a competitive algorithm [Wu and Manber 1992] called BPThompson is proposed, which agilely uses the bit-parallel operations to simulate the NFA. By packing the set of states into the bits of a computer word, the i-th state is mapped to the i-th bit, then all state transitions except the ϵ-transitions can be simulated using a precomputed table B. For the ϵ-transitions, another precomputed table Ed is used to simulate them. Another bit-parallel algorithm BPGlushkov [Navarro and Raffinot 2001, 1999; Navarro 2001] uses the Glushkov NFA [Glushkov 1961; Berry and Sethi 1986; Brüggemann-Klein 1993; Chang and Paige 1997], which costs less space than BPThompson for the automata states.

8.2. Filtering-Based Approaches to Regular Expression Matching

Because the classical RE matching algorithms need to check every character in text, the performance of matching is largely limited. An alternative method is to employ the filter-and-verify strategy, which first locates all the candidate regions in text by multiple string matching algorithms [Blumer et al. 1987; Harel 1999; Uratani and

Takeda 1993; Commentz-Walter 1979; Baeza-Yates and Gonnet 1992], then utilizes the classical RE matching algorithms to verify the candidates. After using the filtering strategies, the algorithm can avoid reading every text character. Next, we introduce several popular filtering-based approaches.

Given a regular expression, the length of shortest occurrence is denoted as l_{min}. Watson [2003] introduced the algorithm MultiStringRE, which first calculates the prefixes *Pref*(RE) of length l_{min} for all strings matching the RE, then a Commentz-Water-like algorithm is used to match the strings in *Pref*(RE). Since every occurrence of RE must start with the occurrence of a string in *Pref*(RE), it is enough to check for the occurrences of RE that start at the initial positions of *Pref*(RE) in the text.

Gnu Grep utilizes the algorithm based on necessary factors. A necessary factor divides RE into a left and right part; two automatons are constructed for verification in both directions. The selection of the best set of necessary factors consists of two stages. The first stage is an algorithm detecting the correct candidate sets. The second is a function that evaluates the cost of searching using a candidate set and the number of potential matches it produces. Actually, prefixes is a particular case of necessary factors, thus Gnu Grep could give better results than MultiStringRE since it may find the best set of necessary factors.

RegularBNDM [Navarro 2001; Navarro and Raffinot 1999] is an extension of BNDM [Navarro and Raffinot 2000] to handle regular expression queries. The idea is to modify the DFA by reversing the arrows and making all states initial, so that the resulting DFA can recognize every reversed prefix of RE. The algorithm slides a window of length l_{min} along the text, and characters in the window are read backwards by the DFA. The backward search inside the window would terminate in two cases: (1) there is no active state in the DFA, then the window will be shifted and the backward search will be restarted; (2) it reaches the starting position of the window, in which case the DFA has recognized a prefix of RE, then it will start a forward verification using the normal DFA starting from the beginning of the prefix.

In Yang et al. [2013], negative factor is first proposed to prune false negatives. It utilizes BWT to index texts. This article proposes a time efficient index structure BitIndex to support faster RE matching and shorter index construction than BWT. BitIndex requires less space than BWT when the alphabet is small. We also show that redundant core negative factors cannot improve pruning power, and require more runtime to generate vectors for them, which slows down the whole matching process. We propose an approach to identify the occurrence of redundant core negative factors and remove them to further improve the quality of selected negative factors. In addition, we analyze that different matching directions have different impacts on matching performance and propose an approach to quickly determine a proper matching direction.

8.3. Heuristic-Strategies-Based Approaches to Regular Expression Matching

Agrep [Wu and Manber 1992] is proposed to support searching for simple strings, extended strings, and REs simultaneously. The most obvious feature of Agrep is that it is capable of approximate searching; also, it has more flexible results reporting rather than just reporting lines. For the different queries, Agrep does not employ a uniform algorithm, but rather a set of heuristic strategies. As a result, Agrep can normally choose the best algorithm. For RE queries, it utilizes the bit-parallel algorithm BPThompson to handle them.

In addition, Faro and Lecroq [2013] provide an excellent survey of algorithms for exact online string matching algorithms, defined as: "Given a text t of length m and a pattern p of length m over some alphabet Σ of size σ, the exact string matching problem is to find *all* occurrences of the pattern p in the text t." This definition is different from ours of RE query (pattern). For example, the pattern p defined in this problem does

not support the Kleene closure e^*. Some BNDM-based algorithms can support simple regular expression queries without Kleene closure, like the PROSITE pattern shown in Section 7.

9. CONCLUSION

In this article, we proposed a novel technique called N-factor and developed algorithms to improve the performance of matching an RE to a sequence. We gave a full specification of this technique, and conducted experiments to compare the performance between our techniques and existing algorithms, such as Agrep, Gnu grep, NR-grep, and RE2. The experimental results demonstrated the superiority of our algorithms. We also extended Agrep, Gnu grep, and NR-grep with the N-factor technique, and showed great performance improvements.

As future work, note that negative factors will not be applicable where they provide no additional filtering power. For example, when an RE contains the subexpression ".*," every prefix match could be a match of the RE, or the candidate occurrences after filtering the automaton must touch nearly all of the text. A future study will develop an estimation to balance the filtering benefit and overhead of using negative factors.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their valuable comments and suggestions to improve the quality of this article.

REFERENCES

Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. 1985. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, MA.

Sean E. Anderson. 2005. Bit twiddling hacks. http://graphics.stanford.edu/~seander/bithacks.html.

Ricardo A. Baeza-Yates and Gaston H. Gonnet. 1992. A new approach to text searching. *Communications of the ACM* 35, 10, 74–82.

Ricardo A. Baeza-Yates and Gaston H. Gonnet. 1996. Fast text searching for regular expressions or automaton searching on tries. *Journal of the ACM* 43, 6, 915–936.

Gérard Berry and Ravi Sethi. 1986. From regular expressions to deterministic automata. *Theoretical Computer Science* 48, 3, 117–126.

Anselm Blumer, J. Blumer, David Haussler, Ross M. McConnell, and Andrzej Ehrenfeucht. 1987. Complete inverted files for efficient text retrieval and analysis. *Journal of the ACM* 34, 3, 578–595.

Anne Brüggemann-Klein. 1993. Regular expressions into finite automata. *Theoretical Computer Science* 120, 2, 197–213.

Chia-Hsiang Chang and Robert Paige. 1997. From regular expressions to DFA's using compressed NFA's. *Theoretical Computer Science* 178, 1–2, 1–36.

Beate Commentz-Walter. 1979. A string matching algorithm fast on the average. In *Proceedings of the 6th Colloquium on Automata, Languages and Programming*, Hermann A. Maurer (Ed.), Lecture Notes in Computer Science, Vol. 71. Springer-Verlag, Graz, Austria, 118–132.

M. Crochemore, A. Czumaj, L. Gasieniec, S. Jarominek, T. Lecroq, W. Plandowski, and W. Rytter. 1994. Speeding up two strings matching algorithms. *Algorithmica* 12, 4, 247–267.

Simone Faro and Thierry Lecroq. 2013. The exact online string matching problem: A review of the most recent results. *ACM Computing Surveys* 45, 2 (2013), 94–111.

V.-M. Glushkov. 1961. The abstract theory of automata. *Russian Mathematical Surveys* 16, 5, 1–53. DOI:http://dx.doi.org/10.1070/RM1961v016n05ABEH004112

David Harel. 1999. *Factor Oracle of a Set of Words*. Technical report. Université de Marne-la-Vallée.

John E. Hopcroft and Jeffrey D. Ullman. 2000. *Introduction to Automata Theory, Languages and Computation, Second Edition* (2nd ed.). Addison-Wesley, Reading, MA.

L. F. Kolakowski, J. Leunissen, and J. E. Smith. 1992. Prosearch: Fast searching of protein sequences with regular expression patterns related to protein structure and function. *BioTechniques* 13, 6, 919–921.

T. W. Lam, W. K. Sung, S. L. Tam, C. K. Wong, and S. M. Yiu. 2008. Compressed indexing and local alignment of DNA. *Bioinformatics* 24, 6, 791–797.

Mehryar Mohri. 1997. String matching with automata. *Nordic Journal of Computing* 4, 2, 217–231.

Eugene W. Myers. 1992. A four Russians algorithm for regular expression pattern matching. *Journal of the ACM* 39, 2, 430–448.

Gonzalo Navarro. 2001. NR-grep: A fast and flexible pattern matching tool. *Software: Practice and Experience* 31, 13, 1265–1312.

Gonzalo Navarro and Mathieu Raffinot. 1999. Fast regular expression search. In *Proceedings of the Algorithm Engineering, 3rd International Workshop on Algorithm Engineering (WAE'99),* Jeffrey Scott Vitter and Christos D. Zaroliagis (Eds.), Lecture Notes in Computer Science, Vol. 1668. Springer-Verlag, London, 198–212.

Gonzalo Navarro and Mathieu Raffinot. 2000. Fast and flexible string matching by combining bit-parallelism and suffix automata. *ACM Journal of Experimental Algorithmics* 5, 4.

Gonzalo Navarro and Mathieu Raffinot. 2001. Compact DFA representation for fast regular expression search. In *Algorithm Engineering, 5th International Workshop, WAE 2001, Proceedings (Lecture Notes in Computer Science)*, Gerth Stølting Brodal, Daniele Frigioni, and Alberto Marchetti-Spaccamela (Eds.), Vol. 2141. Springer-Verlag, Aarhus, Denmark, 1–12.

Gonzalo Navarro and Mathieu Raffinot. 2004. New techniques for regular expression searching. *Algorithmica* 41, 2, 89–116.

Milan Simánek. 1998. The factor automaton. In *Proceedings of the Prague Stringology Club Workshop*, Jan Holub and Milan Simánek (Eds.). Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University, Prague, Czech Republic, 102–106.

R. Staden. 1991. Screening protein and nucleic acid sequences against libraries of patterns. *DNA Sequence* 1, 6, 367–374.

Ken Thompson. 1968. Regular expression search algorithm. *Communications of the ACM* 11, 6, 419–422. DOI:http://dx.doi.org/10.1145/363347.363387

Noriyoshi Uratani and Masayuki Takeda. 1993. A fast string-searching algorithm for multiple patterns. *Information Processing and Management* 29, 6, 775–792.

Bruce W. Watson. 2003. A new regular grammar pattern matching algorithm. *Theoretical Computer Science* 1–3, 299, 509–521.

Sun Wu and Udi Manber. 1992. Fast text searching allowing errors. *Communications of the ACM* 35, 10, 83–91.

Xiaochun Yang, Bin Wang, Tao Qiu, Yaoshu Wang, and Chen Li. 2013. Improving regular-expression matching on strings using negative factors. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD*, Kenneth A. Ross, Divesh Srivastava, and Dimitris Papadias (Eds.). ACM, New York, NY, 361–372.

Received January 2015; revised August 2015; accepted September 2015