Pattern Analysis and Applications (2018) 21:1151–1165
https://doi.org/10.1007/s10044-018-0733-0

INDUSTRIAL AND COMMERCIAL APPLICATION

Multi-pattern matching with variable-length wildcards using suffix tree

Na Liu[1,2] · Fei Xie[3] · Xindong Wu[1,4]

Received: 12 October 2017 / Accepted: 19 July 2018 / Published online: 25 September 2018
© Springer-Verlag London Ltd., part of Springer Nature 2018

Abstract
Multi-pattern matching with variable-length wildcards is an interesting and important problem in bioinformatics, information retrieval and other domains. Most of the previously developed multi-pattern matching methods, such as famous Aho–Corasick and Wu–Manber algorithms, aimed to solve some classical string matching problems. However, these algorithms are not efficient for patterns with flexible wildcards or do-not-care characters. In this paper, we propose two efficient algorithms for multi-pattern matching with variable-length wildcards based on suffix tree, called MMST-L and MMST-S, according to the length of exact characters in a pattern. Experimental results show that the two MMST algorithms, in most cases, outperform other various versions of comparing algorithms.

Keywords Multi-pattern matching · Wildcards · Suffix tree

1 Introduction

Pattern matching, which discovers substrings as patterns in a string or a database, is an interesting and important issue in bioinformatics, information retrieval, knowledge graph, intrusion detection and other domains. For example, pattern matching has been widely used in the analysis of DNA sequences and disease detection. Additionally, it has also been applied in automatic question and answering systems, entity matching, etc. With the increasing demand of applications, pattern matching research varies from single-pattern matching to multi-pattern matching and from exact pattern matching to approximate pattern matching. It is worth mentioning that regular expressions, gaps, wildcards or do-not-care characters, are added onto the patterns to be matched, which to some extent broadens the applications of pattern matching. Therefore, the problem of multi-pattern matching with variable-length wildcards is a very important and valuable research topic.

In the classical multi-pattern matching problem, we are given a set of patterns $\mathcal{P} = \{P_1, P_2, \ldots, P_k\}$ and a text, and aim to search all patterns in the same manner and read the text only once. Many algorithms for exact pattern matching from a single pattern may be extended for multi-pattern matching, with more or less successes [1]. The simplest solution of a single-pattern matching algorithm extended to multi-pattern matching is to repeat n searches, which leads to the worst-case complexity of O(N) in a single-pattern matching enlarged to O(k*N) in multi-pattern matching. In order to improve the efficiency of multi-pattern matching algorithms, many solutions and their variations have been presented. These algorithms can be roughly classified into three categories, including (1) prefix-based approaches including the Aho–Corasick [2] and Multiple Shift-And [3] algorithms; (2) suffix-based approaches including the

Electronic supplementary material The online version of this article (https://doi.org/10.1007/s10044-018-0733-0) contains supplementary material, which is available to authorized users.

✉ Na Liu
liuna3546@163.com

Fei Xie
xiefei9815057@sina.com

Xindong Wu
xwu@hfut.edu.cn

1 School of Computer Science and Information Engineering, Hefei University of Technology, Hefei, China

2 School of General Education, Beifang Minzu University, Yinchuan, China

3 Department of Computer Science and Technology, Hefei Normal University, Hefei, China

4 School of Computing and Informatics, University of Louisiana, Lafayette, LA, USA

Commentz–Walter [4] and Wu–Manber [5] algorithms; (3) factor-based approaches including the Multiple BNDM [6], Set Backward Dawg Matching (SBDM) and Set Backward Oracle Matching (SBOM) [7] algorithms. These aforementioned approaches mainly focus on exact multi-pattern matching. However, when it comes to practice, patterns with more sophisticated forms, such as wildcards, gaps or regular expressions, will have more applications.

Pattern matching with regular expressions is complex in programming and costly in processing time. These limits constrain this approach to be only used when it is necessary. However, the wildcards, gaps or do-not-care characters can be solved in a more efficient way in some applications with simpler algorithms. A wildcard means that this position can match with any arbitrary character, and is denoted with *. A wildcard is also called a gap constraint [8] or a do-not-care character [9]. Variable-length wildcards refer to those continuous positions, which carry arbitrary characters. The problem of pattern matching with wildcards was first introduced by Fischer and Paterson [10] in 1974, who focused on a fixed number of wildcards. Afterward, many subsequent studies have been explored efficient and sophisticated solutions for variable-length wildcards, even arbitrary length wildcards where the number of wildcards can be infinite or negative. In order to solve these complex problems, people have adopted corresponding approaches, such as dynamic programming [11], bit parallelism [12], Aho–Corasick automation [13], directed acyclic word graph (DAWG) [14] and fast Fourier transform (FFT) [15]. However, the study based on suffix tree to solve pattern matching with variable-length wildcards is less extensive compared with those studies exploring the approaches mentioned above.

Suffix tree is a powerful and popular data structure, which is similar to the trie structure. The trie structure can be employed to handle some exact string matching problems. The tree structure has also been applied in natural language processing and textual data representation. Recently, Zhang et al. [16] introduced a tree-structured representation for author recommendation in 2016. In the research, the tree structure was proposed to represent the rich features of each author. Suffix tree was first introduced by Weiner [17] in 1973, which is a compacted trie structure storing all the suffixes of a given string in linear time and space. Therefore, the suffix tree has wide applications in many domains regarding string or text processing. For example, in exact string matching, a suffix tree can be used to find the longest common prefix, the longest repeated substring, the palindrome [18] or the recognition of DNA contamination. A significant feature of the suffix tree is that all characters of a string are stored by suffixes in a tree, while for each suffix tree node, the path string is obtained by concatenating the sequence of labels encountered along the path from the root to the node [19]. Additionally, once the suffix tree is built, it can be used repeatedly if the given string is not changed. Therefore, this feature is assumed as suitable for multi-pattern matching. However, the majority of existing researches about the suffix tree focus on exact single-pattern matching. Can we develop an approach that may absorb the advantages of the suffix tree in exact single-pattern matching and try to fulfill in multi-pattern matching with wildcards? This can be a practical and meaningful research issue.

Hence, in this paper, we propose two efficient suffix-tree-based algorithms, namely MMST-S and MMST-L, according to the length of the exact characters in the given pattern. Two different approaches fulfilling the approximate multi-pattern matching with variable-length wildcards are adopted. The core problem of the algorithms is to achieve a single-pattern matching with variable-length wildcards and then extend to multi-pattern matching. In order to simplify the research issue, the condition of infinite or negative wildcards is not taken into consideration. To further improve the performance, we choose a dynamic programming approach to fulfill matching, called MMST-S for the 'short-length' exact characters (e.g., a*[2,3]b*[0,2]c). In contrast, when the length of exact characters in a pattern is long (e.g., abc*[2,3]cde*[0,2]abcd), we adopt an editing distance approach to fulfill matching, which is called MMST-L. Based on the experimental results, these two MMST algorithms substantially outperform existing solutions in most cases.

The contributions of this paper are summarized as follows:

1. To extend the application of the suffix tree into approximate pattern matching and multi-pattern matching. There are other theoretical studies on the suffix tree in approximate pattern matching in the present researches, and our study will fulfill the experimental demonstration.
2. To design two efficient algorithms according to the length of exact characters in patterns, which can be applied in bioinformatics, such as DNA and protein sequences.

The rest of the paper is organized as follows. In Sect. 2, related work about multi-pattern matching with wildcards is described. Afterward, Sect. 3 defines the research problems on multi-pattern matching and pattern matching with wildcards. The proposed algorithms MMST-S and MMST-L are presented in Sect. 4. Section 5 presents our experimental results and the performance analysis of our proposed algorithms in real bioinformatics data. Finally, Sect. 6 summarizes the findings of the study and also gives suggestions about our future research.

2 Related work

Many existing studies contributed to the pattern matching, which are roughly divided into single-pattern matching and multi-pattern matching according to the number of pattern; or exact pattern matching and approximate pattern matching according to the accuracy of matching. This paper mainly presents an analysis about the problem of approximate multi-pattern matching with wildcards.

Compared with the single-pattern matching, the study of multi-pattern matching is relatively less. Some algorithms are extended from single-pattern matching. However, there are still some good solutions presented for multi-pattern matching. For example, Aho and Corasick [2] proposed the Aho–Corasick automaton algorithm, which serves as an extension of the classical single-pattern matching algorithm Knuth–Morris–Pratt for a set of patterns in 1975. The idea of the AC automaton is to traverse all prefixes of the given text to build a trie from shorter to longer for each prefix. Additionally, all the suffixes which are in the pattern set need to be found [20] (e.g., 'he,' 'she,' 'her'). The AC algorithm and its extended variant algorithms have been widely used for multi-pattern matching, the drawback of which is that these algorithms require a large amount of memory space. The Commentz–Walter algorithm [4] is never faster than Aho–Corasick algorithm or other multi-pattern matching algorithms. However, it is historically important because it was the first expected sub-linear multi-pattern matching algorithm. The algorithm of Wu and Manber [5] resolved the obstacle of poor performance of the extension of Horspool algorithm in 1994. The Wu–Manber algorithm is found to be practical and simple for the reduction in the probability that each character block appears in one of the patterns by reading these blocks of characters. The shortage of Wu–Manber algorithm is that too much memory is consumed if the length of character blocks becomes larger. Additionally, some general factor-based approaches can also be extended to multi-pattern matching, such as MultiBDM [21] in 1997 and Dawg–Match [22] in 1999. However, they are complicated. Meanwhile, the performance is poor in practice. Some algorithms based on the bit parallelism are efficient when the set of pattern is small.

There are few algorithms for approximate multi-pattern matching targeting on those more complicated problems being proposed. Muth and Manber [23] proposed a good solution called MultiHash for one error in 1996. Baeza-Yates and Navarro [24] extended the PEX algorithm to multi-patterns called MultiPEX in 1997, which splits each pattern into $k+1$ pieces and performs a multi-pattern exact search for all pieces. If a piece matched with more than one pattern, then the algorithm needs to check the corresponding pattern about whether it satisfies the permissible errors.

Pattern matching with wildcards was first proposed by Fischer et al. [10] in 1974, with a fixed number of wildcards in a single-pattern matching. Wildcard is also called gap or do-not-care character, which matches with an arbitrary character or a group of characters solving more sophisticated problems in some applications, such as a word misspelling or a DNA mutation. A series of efficient and sophisticated methods regarding the pattern matching with wildcards have been promoted. The number of wildcards changes from one character to more, from the fixed length to the flexible length, even arbitrary length. For example, Cole and Hariharan [25] improved the time efficiency of pattern matching with constant wildcards with the time complexity O($nlogn$) in 2002. Rahman et al. [26] promoted an algorithm for the problem of pattern matching with variable-length wildcards in a certain range in 2006. Haapasalo et al. [27] proposed an algorithm based on the classical Ahot–Corasick automaton in which the range covers not only the variable length but also the arbitrary one in 2011. All of the aforementioned methods focus on the problem of single-pattern matching with wildcards.

Browsing through previous studies, it can be found that researches about multi-pattern matching with wildcards are quite limited. In 1997, Kucherov and Rusinowitch [28] solved the problem of multi-pattern matching with variable length do not cares based on DAWG (directed acyclic word graph) in dynamic dictionary matching. The main idea is to scan the text from the left to right using the automaton and then find the leftmost location during the matching process. Therefore, when the matched pattern in the leftmost is in the worst case, it scans continuously the rest text to find all of the possibilities that results in the bad time complexity. In 2007, Kulekci [29] proposed a new multi-pattern matching algorithm called TARA, which performs matching fixed-length pattern with do-not-care characters based on bit parallelism. The main idea is to slide a window on the string and then have a check of any occurrence of given patterns in the window via bitwise operations by *Alignment, Mask* and *Shift* three matrices and scan the order of these positions in the window. However, the bit parallelism algorithms are limited as it can be only applied for the model of finite length. When the worst case occurs, it can be time-consuming. In 2011, Zhang et al. [30] presented three algorithms for multi-pattern matching with wildcards based on fast Fourier transforms (FFT). The first one finds the matches of a small set of patterns, the second finds the occurrences of patterns based on a prime number encoding of the pattern set and the text, and the third is based on Hamming distance between bit vectors when the number of wildcards in a pattern is very small. However, in these three algorithms, the number of wildcards is fixed or constant.

Suffix tree was first proposed by Weiner [17] based on the trie structure in 1973. However, the construction

process is complicated. In 1976, McCreight [31] proposed a more efficient method to construct the suffix tree according to the reverse order. It is extremely difficult to understand the two original approaches. Until 1995, Ukkonen [32] developed an efficient and different linear-time method for the construction of the suffix tree, which is easier to understand while the complexity of the algorithm is only O(N). Since then, suffix tree efficiently solves the pattern matching problem in many applications.

Similarly, there are more researches about the suffix tree in exact pattern matching and single-pattern matching compared with those about the approximate pattern matching and multi-pattern matching. Gusfield [33] thinks that the suffix tree has provided a bridge between exact pattern matching problems and inexact pattern matching problems. In 2005, Chattaraj et al. [34] introduced the inexact-suffix tree data structure to detect the extensible patterns. The time complexity in worse case is O(2^n), when the size of the input string n is big. Meanwhile, it is very time-costive. In 2009, Ukkonen [35] proposed a method to find maximal and minimal equivalent representations for gapped and non-gapped motifs. In this research, a motif is a pattern that uses suffix-tree-based linear-time algorithms to non-gapped motifs. Bille et al. [36] also conducted a variety of researches for the problem of the pattern matching with wildcards. In 2014, they provided two approaches for string indexing in the pattern with variable-length wildcards based on suffix tree search. The main idea is to find all occurrences of a pattern in a top-down traversal starting from the root and to build a compressed suffix tree storing all possible modifications of all suffixes containing the wildcards. However, the complexities of time and space have increased corresponding to the increase in the number of wildcards or gaps in the pattern. In 2016, Thankachan et al. [37] described a method to extend the generalized suffix tree model to incorporate a selected bounded set of perturbed suffixes for complex approximate sequence matching problem. However, the methods promoted above are theoretically described and hardly usable in practice.

To sum up, these approaches proposed in the above researches can solve one side of the problem. However, there are two algorithms in this research based on suffix tree combined with dynamic programming and editing distance to solve the problem.

3 Preliminaries

In this section, we give a formal definition of the multi-pattern matching with variable-length wildcards. Meanwhile, these algorithms used some primitives.

3.1 Sequence and pattern

Definition 1 Let a sequence be composed of characters $S = s_0 s_1 \dots s_{n-1}$, where S is called an object sequence, n is the length of S and $s_i \in \Sigma\ (0 \le i \le n-1) \cdot \Sigma$ is an alphabet, containing all of the different symbols of S, and $|\Sigma|$ is the size of Σ. For example, in DNA sequence, Σ is {A,C,G,T} and the size of DNA alphabet is 4 denoted by $|\Sigma| = 4$.

Definition 2 $S_i^j = s_i s_{i+1} \dots s_j$, is called the subsequence of S, denoted by $Sub(S)$, where $0 \le i \le j \le n-1$, when $i=0$, $\mathrm{S}_\mathrm{i}^\mathrm{j} = s_0 s_1 \dots s_j$ is the prefix of S, denoted by $Prefix(S)$; when $j=n-1$, $S_i^{n-1} = s_i s_{i+1} \dots s_{n-1}$ is the suffix of S, denoted by $Suffix(S)$.

***Example* 1** Given an object sequence $S = s_0 s_1 s_2 s_3 s_4 s_5$ = gtccgc, $S_i^j = s_2 s_3 s_4$ = ccg is one of the subsequences of S, $S_i^j = s_0 s_1 s_2 s_3$ = gtcc is one of the prefixes of S and $S_i^j S_i^j = s_2 s_3 s_4 s_5$ = ccgc is one of the suffixes of S.

Definition 3 The subsequence of $P = p_0 p_1 \dots p_{m-1}$ is called a pattern, where m is the length of P, and $p_j \in \Sigma (0 \le j \le m \le n-1)$.

3.2 Wildcards

Definition 4 If one of the positions p_j in the pattern P can match arbitrary character, it is called a wildcard denoted with *.

Definition 5 Let a pattern $P = p_0 * [l_0, h_0] p_1 \dots * [l_{j-1}, h_{j-1}] p_j \dots * [l_{m-1}, h_{m-1}] p_m$ be constructed by the character p_j and wildcard *, where $p_j \in \Sigma\ (0 \le j \le m)$, $*\ [l_{j-1}, h_{j-1}]$ is the gap constraint between two exact characters, l_{j-1} and h_{j-1} refer to integer numbers, representing the minimum and maximum numbers of wildcards between the characters p_{j-1} and p_j.

- When $0 \le l_{j-1} \le h_{j-1}$ and $h_{j-1} \ne \infty$, the pattern P is called a pattern with variable-length wildcards, especially in the case when $l_{j-1} = h_{j-1}$ presenting the length of wildcard gap is constant;
- The pattern P is called a pattern with arbitrary length wildcards, if $l_{j-1} \le h_{j-1}$ while l_{j-1} and h_{j-1} can be a negative integer, or when l_{j-1} is an arbitrary integer, and $h_{j-1} = \infty$,.
- When $l_0 = l_1 \dots = l_{j-1} \dots l_{m-1}$, and $h_0 = h_1 \dots = h_{j-1} = \dots = h_{m-1}$, the pattern P is called a pattern with periodic length wildcards.

***Example* 2** For the pattern $P = p_0 {*} [l_0, h_0] p_1 {*} [l_1, h_1] p_2 {*} [l_2, h_2] p_3$ = gt*[0,1]c*[0,0]c*[1,∞]gc, where the $p_0 = gt$, $p_1 = c$,

$p_2 = c$, $p_3 = gc$, the sub-pattern gt*[0,1]c means that the minimum of the position range of wildcards between $p_0 = gt$ and $p_1 = c$ is 0 while the maximum is 1. This sub-pattern is a pattern with variable-length wildcards. When it comes to the sub-pattern c*[0,0]c, the minimum and maximum of the position range of wildcards between $p_1 = c$ and $p_2 = c$ are 0. When there is no character between p_1 and p_2, c*[0,0]c = cc. The sub-pattern c*[1,∞]gc means that the number of characters between $p_2 = c$ and $p_3 = gc$ is an arbitrary integer. Such sub-pattern is a pattern with arbitrary length wildcards.

This study only focuses on the problem of pattern matching with variable-length wildcards. Because this problem is more sophisticated, we need to consider more sub-problems.

Definition 6 G denotes the gap constraint size of wildcards. In a variable-length wildcard constraint *[l,h], l and h are positive integer numbers, l is the minimum of the gap and h is the maximum, then $G = h - l + 1$.

Definition 7 $M_{\min}$ denotes the minimum length of the single pattern $P = p_0 * [l_0, h_0]\, p_1 \ldots * [l_{j-1}, h_{j-1}] p_j \ldots * [l_{m-1}, h_{m-1}] p_m$ and $M_{\max}$ denotes the maximum length of P. Then, $M_{\min} = m + 1 + \Sigma_{j=0}^{m-1} l_j$, $M_{\max} = m + 1 + \Sigma_{j=0}^{m-1} h_j$, where $m + 1$ means the number of the exact characters, $\Sigma_{j=0}^{m-1} l_j$ means the sum of the minimum of each wildcard and $\Sigma_{j=0}^{m-1} h_j$ means the sum of the maximum of each wildcard.

***Example* 3** Given a pattern $P = $ g*[0,1]g, the gap length of wildcards $G = h - l + 1 = 1 - 0 + 1 = 2$. In this case, the minimum length of P $M_{min} = 2 + 0 = 2$, while the maximum length of P $M_{max} = 2 + 1 = 3$.

3.3 Pattern matching

Definition 8 Given an object sequence $S = s_0 s_1 \ldots s_{n-1}$ and a pattern $P = p_0 p_1 \ldots p_{m-1}$. If there are some positions $i_1, \ldots, i_m$ satisfying the following equation:

$$\begin{cases} S_{i_j} = p_j \\ l_{j-1} \le i_j - i_{j-1} - 1 \le h_{j-1} \Rightarrow l_{j-1} + 1 \le i_j - i_{j-1} \le h_{j-1} + 1 \\ M_{\min} \le i_m - i_1 + 1 \le M_{\max} \le n \end{cases}$$

where $0 \le j \le m - 1$ and $0 \le i_1 \le \ldots \le i_m \le n$, then the sequence $i_1, \ldots, i_m$ can be called as an occurrence.

***Example* 4** Given a DNA sequence $S = s_0 s_1 s_2 s_3 s_4 \ldots s_9 =$ ggcgtccgcg, $n = 10$, and a pattern $P = $ g*[1,2] cg*[1,4]c, find all of the occurrence positions of pattern P in S.

For the pattern, $P = $ g*[1,2]cg*[1,4]c, $M_{\min} = 4 + 1 + 1 = 6$, $M_{\max} = 4 + 2 + 4 = 10$, $G_0 = 2 - 1 + 1 = 2$ and $G_1 = 4 - 1 + 1 = 4$. As shown in Fig. 1, the occurrence positions of $p_0 = g$ are 0,1,3,7,9 in the object sequence S, while the occurrence positions of $p_1 = cg$ are (2,3), (6,7) and (8,9). When $p_0 = 1$, $p_1 = 2$, $l_{j-1} \le i_j - i_{j-1} - 1 \le h_{j-1}$, $2 - 1 - 1 = 0 < l_{j-1}$, this equation fails. Meanwhile, the other positions cannot satisfy the Eq. 8. When $p_1 = $ cg = (8,9), the end position of $p_1 = 9$ added to $G_1 = 4$ is equal to $13 > M_{\max}$ and also bigger than $n = 10$. So this position also does not satisfy Eq. 8.

Therefore, the positions satisfying the gap range [1,2] are $p_0 = 0$ and $p_1 = (2,3)$. By the same token, when $p_1 = (2,3)$, the satisfied positions of p_2 are 5 and 6. Therefore, occurrence matching the pattern P in S is 2, while the occurrence positions are {0,2,3,5} and {0,2,3,6}.

3.4 Multi-pattern matching

Definition 9 Given an object sequence $S = s_0 s_1 \ldots s_{n-1}$ and a pattern set $\mathcal{P} = \{P_0, P_1, \ldots, P_{k-1}\}$, all of them were constructed by alphabet Σ where n is the length of S and k is the size of the pattern set $\mathcal{P}$. The key of the multi-pattern matching is to find all the occurrences of each pattern in the given sequence.

If multiple object sequences and the multiple patterns are given, it is the problem of multi-pattern matching in multiple sequences. This paper only focuses on the problem of one object sequence and multiple patterns. If each of the patterns in the pattern set is the exact pattern, it is the problem of exact multi-pattern matching. In the pattern set of this research, each one is the pattern with variable-length wildcards (see Definition 5).

According to the realistic application, each pattern can get different matching results through the logical operators, such as AND, OR and NOT. Specifically, there are two patterns, including A and B. The relation A AND B suggests that the output results match with patterns A and B; the relation A OR B means that the output results either match pattern A or match pattern B; the relation A NOT B means that the output results only match pattern A and do not match pattern

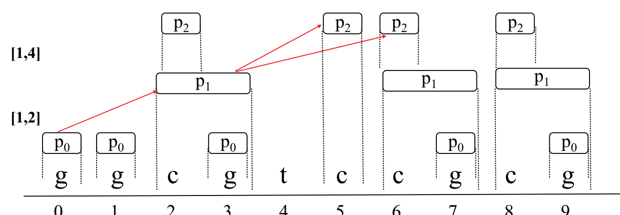

Fig. 1 An example of the pattern matching with variable-length wildcards

B. The OR relation is adopted in this paper for multi-pattern matching to output the matching result of each pattern and the total matching numbers.

3.5 Suffix Tree

Definition 10 Let *S* be a sequence of *n* characters. According to the Ukkonen's approach [31], the last character '\$' denoting the end marker for the sequence is added. The path from the root to any leaf represents a suffix for the sequence denoted SuffixTree(S), commonly abbreviated Suf(S). Such as the object sequence $S = s_0 s_1 \ldots s_i \ldots s_{n-1}$, the $\mathrm{Suf(S_i)} = s_i s_{i+1} \ldots \mathrm{s_{n-1}}\$$.

***Example* 5** Given a DNA sequence $S = s_0 s_1 s_2 s_3 s_4 \ldots s_{19}$ = ggcgtccgcgcacacctccc, $n = 20$, the special terminal symbol '\$' is added to construct the suffix tree, which is shown in Fig. 2. The root node is denoted $(-1, -1)$ while altogether there are 21 paths form the root to the leaves corresponding to the 21 suffix sequences altogether including the '\$.' The number in the leaf node gives the start position of the corresponding suffix, denoted by Suf(S_i). The path is a subsequence which, from the middle node to the leaf node, denoted (*u, v*), where *u* is the start position of the middle node and *v* is the end position of the leaf node. The suffix links (character-\$) belong to compressed path.

Lemma 1 If a sequence *S* with length *n* is constructed to form a suffix tree (include the '\$'), then the suffix tree will have these following properties:

- It has $n + 1$ leaves numbered from 0 to *n*.
- Except for the root node, every internal node has at least two children nodes.
- Every edge is labeled with a non-empty subsequence of *S*.
- Subsequences represented by sibling nodes must begin with different characters.

Lemma 2 If there are two children nodes of the root node of Suf(*S\$*), namely A and B, A and B must be the one of the elements of Σ or the terminal symbol '\$' denoted $\mathrm{A, B} \in \Sigma \cup \{\$\}$ and $\mathrm{A} \neq \mathrm{B}$. Therefore, it can be assumed that the number of branches for the root node is equal to the size of $|\Sigma|$ in addition to \$.

***Example* 6** As shown in Fig. 2, the length *n* of the DNA sequence *S* is 20 while the \$ is added. The suffix tree Suf(*S\$*) has 21 leaves which are numbered from 0 to 20; two characters *a, c* are the children nodes of the root node, so *a, c* $\in \Sigma$. Besides, the number of the branches of this root node can be obtained in this way, which is the size of the alphabet $|\Sigma| + 1 = 4 + 1 = 5$.

Lemma 3 If *T* is the subsequence of sequence *S*, *T* must be a prefix of another subsequence *T*' in the suffix tree Suf(*S\$*). Additionally, the occurrence number of *T* must be the number of the children nodes of this non-leaf node of *T*'.

Lemma 4 If T is the subsequence of sequence *S*, then *T* is the deepest non-leaf node from the root node, and *T* must be the longest common subsequence or the longest repeat subsequence.

***Example* 7** As shown in Fig. 2, the subsequences T_1 = cgc and T_2 = gcg both are the deepest non-leaf nodes from the root node in the DNA sequence Suf(*S\$*), while both T_1 and T_2 are the longest common subsequences, the length of which is 3. T_1 has two leaf nodes 6 and 8, and $\mathrm{T_2}$ also has two leaf nodes 1 and 7. This suggests that the two subsequences also occurred two times in *S*. The number of the leaf node is the start position of two subsequences, respectively.

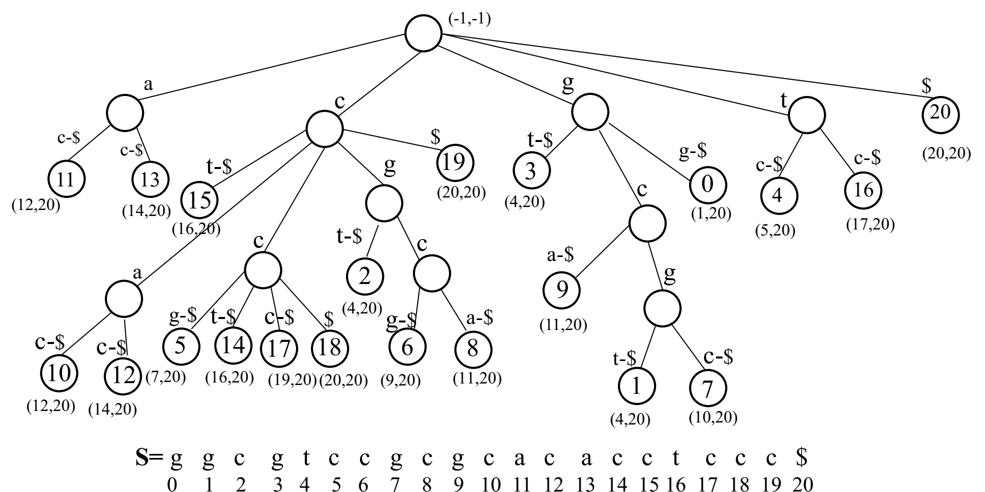

Fig. 2 The suffix tree for the DNA sequence *S*

4 Algorithm design

The efficient algorithms for multi-pattern matching with variable-length wildcards based on the suffix tree are presented in this section. The algorithm involves three important phases. In the first phase, the suffix tree is constructed based on the Ukkonen's method [32]. During the second phase, multiple patterns are pre-processed and classified using the approach in the next step. When it comes to the third phase, which is also the important phase, the problem of multi-pattern matching with variable-length wildcards is solved using our two algorithms (MMST-S and MMST-L) in detail. Finally, it is the complexity analysis of two algorithms.

4.1 Suffix tree construction

Recently, the approach to constructing the suffix tree is mostly based on the Ukkonen's method [32], the basic idea of which is to start from an empty tree, which is the root node, and then insert nodes into the tree in the order of the characters of the object sequence. Such node is a new branch node, and each of the branch nodes is an active node during the suffix tree construction period. It inserts characters from left to right. It needs to determine the position of the path when the letter of new character has existed at the current active node. A new branch node would be created if the letter of new character did not exist. All of the characters of the object sequence are inserted in this suffix tree before the terminal symbol '$'appears, which is shown in Fig. 2. The time complexity of this method is only O(n), while n represents the length of the object sequence.

Our algorithm to construct the suffix tree also takes the above approach. We traverse the tree in the top-down order. Each of the paths from the root to the leaf node is the suffix subsequence of the object sequence. The number of the leaf nodes for the corresponding branch node and the start position of the edge of the compact path are stored. The suffix tree was constructed in Algorithm 1, which is shown as follows.

Algorithms 1: Suffix Tree Construction

Input: *an objected sequence text*
Output: *SuffixTree(text)*

```
1. root=new SfxNode();
2. Add('$',S[n]);
3. temp= SfxNode(suffix_link=root);
4.root.add_link(S[0],longest);
5. for ( i=0; i   n;i++)
6.      current=temp; previous=None;
7.      while T[i] is not in the current.children do
8.          NewNode= new SfxNode();
9.          current.add_link(T[i],NewNode);
10.         If previous node is not null then
11.            previous.suffix_link=NewNode;
12.         end if
13.         previous=NewNode;
14.         current=current.add_link;
15.     end while
16.     if current is root
17.         previous.suffix_link=root;
18.     else previous.suffix_link=current.children[S[i]];
19.     temp=temp.children[S[i]];
20. end for
21. return root;
```

Once the suffix tree is constructed, the tree can be repeatedly applied if there is no change occurring over the object sequence. This property is appropriate for multi-pattern matching.

4.2 Multiple-pattern pre-processing

Holding the purpose of improving the efficiency of our algorithm, some pre-processing measures have been taken for the pattern set. The pseudo-code was, respectively, described in Algorithms 2 and 3. According to the length of exact characters in the pattern, there are two approaches.

The short form is that the exact characters of a pattern are composed of single or short characters, such as P_1 = a*[1,2]b; P_2 = c*[0,1]de*[2, 3]f. The pre-process measure of multi-patterns is simple for short exact characters. The patterns are classified based on the exact character alphabetical order. Because of the short form, we will take dynamic programming from front to back in the suffix tree. According to the property of the suffix tree, if the first character of the suffixes P_1 and P_2 is same, these two suffixes must be in the same alphabetic branch node, by Lemma 2. Hence, the sort of the patterns in the pre-process phase will improve the efficiency of the algorithm MMST-S.

The long form is that exact characters of a pattern are composed of a group of characters, such as P_1 = abcd*[1,2]bbbbb and P_2 = cacbd*[0,1]dec*[2,3]fght; it requires to split the exact characters into groups and gaps. Meanwhile, they need to be stored in the array, respectively. Afterward, the patterns are classified according to the alphabetical order of the last exact characters' group. Because we will take editing distance for these exact groups from back to front in the algorithm MMST-L, and in the matching phase, the approach based on suffix is usually faster than the approach based on prefix. For example, if a subsequence does not satisfy the last group but it satisfies all of the front groups, it must not be the correct occurrence position. Meanwhile, if a subsequence satisfies the last group, it is possible the correct occurrence position.

4.3 Algorithm MMST-S

As mentioned in the pre-processing phase, the algorithm MMST-S is suitable for the short form of the multi-pattern

with short exact character. After the pre-processing phase, the multi-pattern set $\mathcal{P}=\{P_0,P_1,\ldots,P_{k-1}\}$ where $P_i=p_0*[l_0,h_0]p_1\ldots*[l_{j-1},h_{j-1}]p_j\ldots*[l_{m-1},h_{m-1}]p_m$, are sorted according to the alphabetical order of the first character p_0 in each pattern. Firstly, the algorithm MMST-S was judged to p_0, which is the element of the alphabet Σ. Besides, the dynamic programing measures have been adopted to search the next exact character p_1, in the branch of the letter of p_0, to find its children node and leaf node which satisfied the gap range $[l_0,h_0]$. If one of the possible occurrence positions is found, then it requires to continue to find p_2 until the last exact character p_m was found.

As shown in Fig. 3, the suffix tree is constructed for the object sequence $S=s_0s_1s_2s_3s_4\ldots s_{19}=$ ggcgtccgcg-cacacctccc and two patterns $P_0=p_0*[l_0,h_0]p_1=$ g*[0,1]g, $P_1=p_0*[l_0,h_0]p_1=$ c*[0,1]c. The first step is to sort these patterns according to the alphabetical order of each first exact character. Because letter *c* is in the front of letter g, so P_1 is matched first. Afterward, the occurrence position must be in the branch node or leaf node in the first character $p_0=$ *'c,'* while other branches such as *'a,' 'g,' 't'* are excluded. During the third stage, it needs to find the occurrence position of $p_1=$ *'c,'* which satisfies the gap range [0,1]. Figure 3 makes it relatively easy to get the occurrence positions matching pattern P_1; when the minimum number of wildcards is 0, the positions are the children or leaf nodes of the subsequence *'cc'*:(5,6),(14,15),(17,18),(18,19); when the maximum number of wildcards is one, the positions are the children or leaf nodes of the subsequence *'c*c'*:(6,8),(8,10),(10,12),(12,14),(17,19). Therefore, the occurrence number of P1 in the object sequence S is nine. Similarly, when the pattern is $p_0=$ g*[0,1]g, the occurrence positions must appear in the branch of the first farther node '*g,'* which satisfies the wildcard condition: (0,1),(1,3),(7,9). The occurrence number is three.

According to the above description and the example of the algorithm MMST-S, Algorithm 2 shows the pseudo-code.

Algorithms 2: MMST-S

Input: *root, Pattern set* $=\{P_0,P_1,\ldots,P_{k-1}\}$
Output: *Match position and march number*

1. *initialize the arrays and parameters*;
2. *node=root;*
3. *sort the set and store in the array P*
4. for *(i=0;i k;i++) // k is the size of the Patterns*
5. for *(j=0;j P[i].length; j++)*
6. if(p[j]) then
7. *search p[j] in node;*
8. *save temp=node[p[j]].children;*
9. else (p[j]) *// p[j] is the wildcard;*
10. *save min[j]=l_j, max[j]=h_j;*
11. *save next=getchar(p[j+1]);*
12. for *(t=0;t temp.length ;t++)*
13. *search the next in the temp;*
14. if*(next.position match the gap of the min and max)* then
15. *save the Position;*
16. *temp=Position;*
17. else *continue*
18. end for
19. *save MatchPosition[i] and MatchNumber[i];*
20. end for
21. end for

4.4 Algorithm MMST-L

Algorithm MMST-L is proposed for the long form of the multi-pattern with long exact characters. The main idea is to make a comparison of the occurrence positions of each exact character group by the editing distance and judge the gap range of these groups whether to satisfy the variable length of wildcards. Specifically, after the pre-processing phase, the multiple patterns set $\mathcal{P}=\{P_0,P_1,\ldots,P_{k-1}\}$, where $P_i=p_0*[l_0,h_0]p_1\ldots*[l_{j-1},h_{j-1}]p_j\ldots*[l_{m-1},h_{m-1}]p_m$ are stored in the array according to the alphabetical order of the first letter of the last exact group p_m in each pattern. Then, we get the start position from the leaf nodes of the last exact

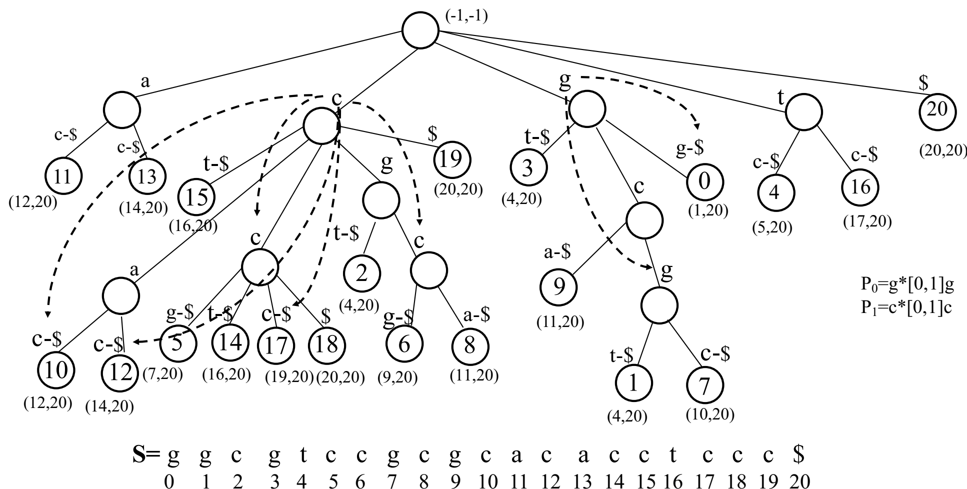

Fig. 3 An example of MMST-S algorithm

characters' group of the first pattern from the suffix tree. Lemma 3 shows that the occurrence number is the total number of leaf nodes in a branch node, while the occurrence position is the start number of each leaf node. Additionally, the end position of p_{m-1} is calculated accordingly, which is equal to the start position of p_{m-1} added by the length of $p_{m-1}-1$. Finally, to further improve the efficiency of the algorithm, we take the measure from back to front to judge the gap range of the start position of the first character of p_m minus the end position of the last character of p_{m-1}. If the range satisfies the length of wildcard $[l_{m-1},h_{m-1}]$, continue to judge the rang of p_{m-1} and p_{m-2}, until p_0. The same approach is adopted to judge the next pattern if the range does not satisfy the length.

As shown in Fig. 4, given the object sequence $S=s_0s_1s_2s_3s_4\ldots s_{19}=$ ggcgtccgcgcacacctccc we construct a suffix tree of two patterns $P_0=p_0{*}[l_0,h_0]p_1=$ gcg*[0,6]cc and $P_1=p_0{*}[l_0,h_0]p_1=$ cgc*[1,2]ca. Firstly, sort these patterns according to the alphabetical order of the last exact group. Since *'ca'* is in front of *'cc,'* pattern P_1 is first. In the pre-processing phase, we need to split these exact character groups and their gap ranges of wildcards. Secondly, we get the start positions of $p_m=p_1=$ *'ca'* from the suffix tree which are 10 and 12. Afterward, we calculate the end position of $p_{m-1}=p_0=$ *'cgc.'* We need to get the start positions of the *'cgc,'* which are 6 and 8, while the length of the $p_0=3$. Therefore, the end positions of *'cgc'* are $6+3-1=8$ and $8+3-1=10$. Thirdly, the distance of two groups was evaluated to see whether they can satisfy the wildcard $*[l_0,h_0]=[1,2]$. The occurrence positions are ([6,8][10,11]), ([8,10][12,13]). In ([*m,n*]), *m* refers to the start position of the group and *n* is the end position of the group. When $m=n$, it suggests that there is only a single character in the group. The number of occurrences is two. Then, the position of the next pattern P_0 is obtained through a similar method. The start positions of *'cc'* are 5,14,17,18, while the end positions of *'gcg'* are 3 and 9, and the occurrences satisfying the variable length of wildcard *[0,6] are ([1,3][5,6]),([7,9][14,15]).

Algorithm 3 is the pseudo-code of algorithm MMST-L.

Algorithms 3: MMST-L

Input: *root, Pattern set* $=\{P_0,P_1,\ldots,P_{k\text{-}1}\}$
Output: *Match position and march number*

```
1. initialize the arrays and parameters;
2. node=root;
3. split the exact character group and store in the array P;
4.sort the P;
5.for (i=0;i  k;i++)    // k is the number of the Patterns
6.    for (j= P[i].length; j   0 ;j--)
7.        search p[j] in node;
8.        save subp[j]=p[j].leafnode;
9.        if(subp[j].position)>subp[j-1].position) then// optimize the algorithm
10.            gap=subp[j].get(0)-subp[j-1].get(1);
            // the start position of the latter one minus the end position of the previous one
11.            if(gap match the rang of the min and max) then
12.                save the Position;
13.            else continue
14.        else continue
15.    end for
16.    save MatchPosition[i] and MatchNumber[i];
17. end for
18. return MatchPosition and MatchNumber;
```

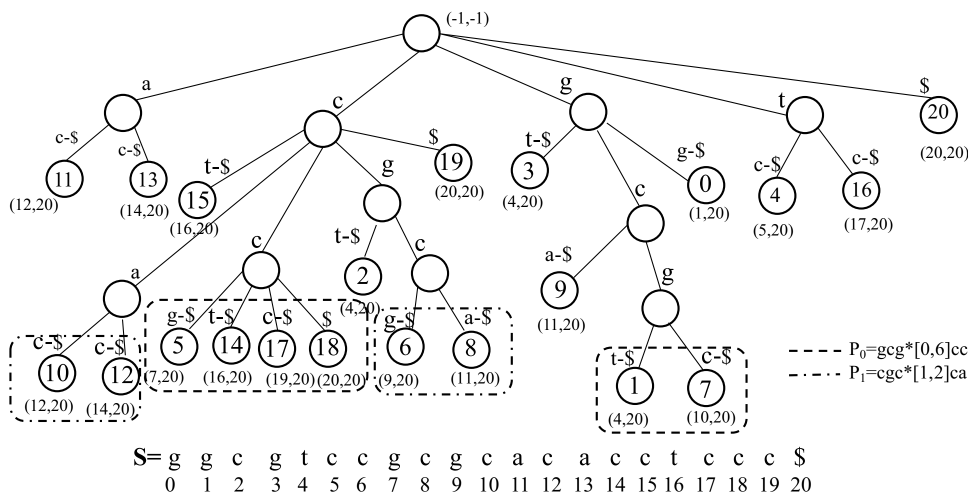

Fig. 4 An example of MMST-L algorithm

4.5 Complexity analysis

This section focuses on the complexity and the extensibility of the aforementioned algorithms. Let the length of the object sequence S be n, the alphabet size be $|\Sigma|$, the multi-pattern set $\mathcal{P}=\{P_0,P_1,\ldots,P_{k-1}\}$, where k is the number of the patterns, and $P_i=p_0*[l_0,h_0]p_1\ldots*[l_{j-1},h_{j-1}]p_j\ldots *[l_{m-1},h_{m-1}]p_m$, where m is the average length of the patterns. The algorithm of constructing suffix tree is based on the Ukkonen's method. Both the time and space complexities are O(n) [31]. However, the algorithms MMST-S and MMST-L are based on different approaches, so the time complexities are different.

For the algorithm MMST-S, it consists of three *'for'* loops: The time of the first loop is the number of patterns k, the second loop is the average length of the patterns m and the third loop is the average children nodes of the characters $n/|\Sigma|$. It needs to make a comparison about the wildcards. In the worst case, the number of the wildcards is $m-1$, so the G is the gap range of each wildcard. Beginning with the construction of suffix tree, the overall time complexity is $O\left(n+k(n/|\Sigma|)\left(\sum_{i=0}^{m-1}G_i\right)\right)$. When $|\Sigma|$ is big, the $n/|\Sigma|$ tends to be a constant number which is less than n. According to the first character, it can exclude all of the branch nodes in other character branches.

For the algorithm MMST-L, it has two *'for'* loops: The time of the first loop is also the number of patterns k, while the second loop is the number of the exact characters' group w, and the number of the wildcards is $w-1$; the G is the gap range of each wildcard. Therefore, the general time complexity is $O\left(n+k(n/|\Sigma|)\left(\sum_{i=0}^{m-1}G_i\right)\right)$. Compared with the algorithm MMST-S, the group w is less than the number of patterns m. Thus, when w equals to m that means the pattern has the short form, the algorithm consumes more time.

The suffix tree is a data structure, which can improve the time efficiency through space–consumption. If the length of a sequence is n, when constructing the suffix tree, the tree contains at most n leaves and $2n$ nodes, the space complexity is O(n). In the two algorithms MMST-S and MMST-L, the cost of the memory is mainly used to store some probability occurrence positions in the array. The number of these positions is less than n, which can be absolutely ignored. Thus, both of the space complexity of two MMST algorithms are O(n).

5 Experimental evaluation

In this section, we mainly discuss the time performance of the two algorithms, namely MMST-S and MMST-L. We first give the experimental environment and test data and then report the experimental results by testing various parameters in real biological data against other existing algorithms.

5.1 Experimental environment and data sets

All experiments were conducted on a laptop with Intel Core i5 2.7 GHz CPU and 8G main memory, running on OS X EI Capitan Operating System. The algorithms of MMST-S, MMST-L and also the comparison algorithms of WM-gap, BG-gap and ST-TWEC-gap are written in JAVA language. The algorithms WM-gap and BG-gap were recomposed by the algorithms WM and BG promoted by Salmela et al. [38] and the thought of Ukkonen [39]. The WM-gap is based on the famous Wu–Manber [5] algorithm, the BG-gap is based on the Multiple BNDM [6] algorithm, and the ST-TWEC-gap is recomposed by the algorithms ST-TWEC [40].

This experiment chose the real biological data, the DNA sequence AX829174 and the protein sequence AJM00528

as the experimental data, which are downloaded from the NCBI (National Center for Biotechnology Information) Web site [41]. Compared with the natural language texts, Gonzalo et al. [1] think that DNA sequences are comparatively more difficult to deal with than natural language texts. This is because DNA sequences contain more intrinsic repetitions. For example, the size of the alphabet is $|\Sigma| = 4$ in DNA sequences. The length of the DNA sequence AX829174 is 10,011; the protein sequence AJM00528 consists of 34,350 characters. In our experiments, we will pick some length-L segments from the two sequences as the object sequence for providing the various values of L.

5.2 Experimental results

In this section, we first provide the experimental results obtained from the measurement of the running time of two algorithms MMST-S and MMST-L in comparison with WM-gap, BG-gap and ST-TWEC-gap under the condition of different variables in two different sequences. Afterward, it presents a deep analysis regarding the reason why our algorithms outperform other algorithms. For changing the different various parameters, we take the tool to generate the multi-pattern set $\mathcal{P} = \{P_0, P_1, \ldots, P_{k-1}\}$, where k is the number of patterns. The formation of the pattern is $P = p_0 l_0, h_0 p_1 \ldots l_{j-1}, h_{j-1} p_j \ldots l_{m-1}, h_{m-1} p_m$, where $p_0 \ldots p_j \ldots p_m \in \Sigma$ are the exact characters from the alphabet of experimental data, while m refers to the length of the pattern. The $l_0, h_0 \ldots l_{j-1}, h_{j-1} \ldots l_{m-1}, h_{m-1}$ are the positive integer numbers which denote the variable-length wildcards. In the experiment, we only set the value of the G, because $G = h - l + 1$, and the values of h and l are generated randomly.

5.2.1 DNA sequence data

This study presents a comparison of the running time on the DNA sequence AX829174 with varied sequence lengths, varied pattern sizes, varied wildcard numbers, varied gap sizes and varied exact characters' lengths. When we compare a variable, other parameters are given.

The first test of our algorithms is on the varied object sequence with length n changing from 1000, 2500, 5000 to 10,011. The multiple pattern set is $\mathcal{P} = \{P_0, P_1, \ldots, P_{k-1}\}$ where k is 10. For each single pattern $P_i = p_0 l_0, h_0 p_1 \ldots l_{j-1}, h_{j-1} p_j \ldots l_{m-1}, h_{m-1} p_m$, $p_0 \ldots p_j \ldots p_m \in \sum = \{a,g,c,t\}$ and m is 3. Besides, the number of wildcards w is 2. The maximum of gap range G is 9. Figure 5 presents the experimental results to better compare the runtime. It is clear that the algorithm MMST-S is faster than other algorithms, and the algorithm BG-gap uses the bit parallelism method which limited the length of word size of memory. Since the tested patterns are in the short forms, the algorithm MMST-L needs to compare many repeat positions that reduced efficiency.

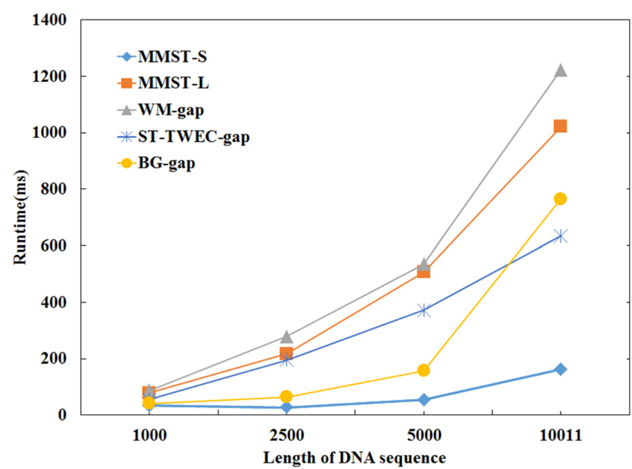

Fig. 5 Comparison of the runtime by varying lengths of DNA sequence

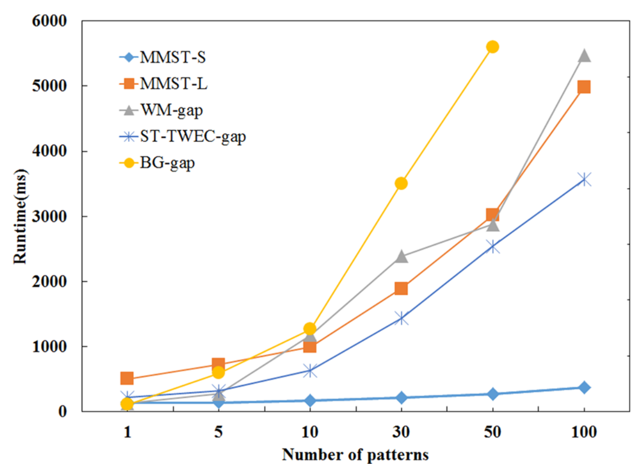

Fig. 6 Comparison of the runtime by varying numbers of patterns

Figure 6 displays the time performance of four algorithms with different numbers of patterns on DNA sequence. In this experiment, the length of the object sequence is 10,011, while the patterns in this study adopt the short exact character, and there are two wildcards in each pattern; the G is also 9. The number of patterns k changes from 1 to 100. When $k = 1$ for the single pattern, it consumes some time for the two algorithms MMST-S and MMST-L to construct the suffix tree. Therefore, the time performance is not as good as the comparison algorithms. However, the suffix tree is repeatedly used once constructed. With the number of patterns increasing, the time performance of MMST-S is better than other algorithms. The algorithm MMST-L is not suitable for the short pattern form.

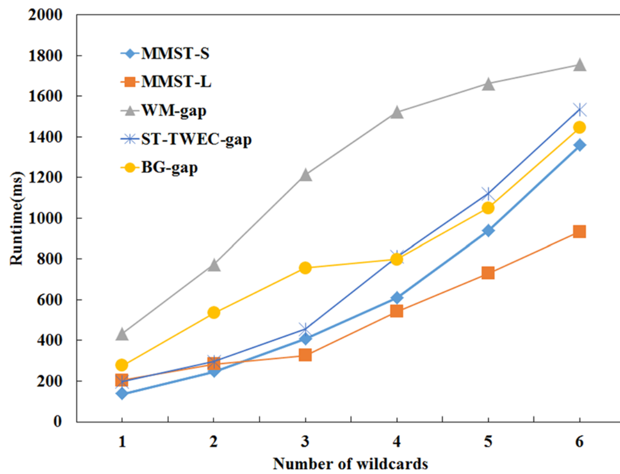

Fig. 7 Comparison of the runtime by varying numbers of wildcards

The third test of four algorithms is on the varied number of wildcards. The number of wildcards w varies from 1 to 6, and the maximum of gap range G is also 9. However, the form of patterns is long exact character, such as each single pattern P_i = gcg0,2cct1,9cgc. The number of patterns is 10, while the object sequence is also the DNA sequence AX829174, whose length is 34,350. Figure 7 shows the time performance of four algorithms under different numbers of wildcards. With the increase in the wildcard number, the algorithms BG-gap, WM-gap and ST-TWEC-gap consume more time than our two algorithms. Even though there will no matching result with the rise of the wildcards, all of the positions in sequences have still been searched by these three compared algorithms. Our algorithms MMST-S and MMST-L will decrease time by some optimized judgments, and the performance of MMST-L is better than of MMST-S in long exact character patterns.

The fourth test is the comparison of four algorithms with different gap ranges of wildcards. As shown in Fig. 8, the length of the object sequence n is 10,011 while the number of pattern k is 10. The number of wildcard w is 1. Besides, the form of pattern is long exact character, while the maximum of gap range G changes from 1 to 100. None matching results is obtained for these five algorithms when the gap is small. With the increase in gap range, the number of matching results is also increasing. Similar to the third test, the algorithms MMST-S and MMST-L consume less time than compared algorithms by optimized judgments.

5.2.2 Protein sequence data

The size of the alphabet in protein sequence $|\Sigma|$ is 20. It includes $\{a,c,d,e,f,g,h,i,k,l,m,n,p,q,r,s,t,\ v,w,y\}$. Compared with the DNA sequence, the size of the alphabet is closer to the natural language. For example, the size of alphabet is 26 in English language. Then, a comparison will be made about the time performance of five algorithms on the real protein sequence AJM00528. The length n is 34,350. With the decrease in repeat positions, the number of matching results will decrease. It has been proven that the algorithms of MMST-S and MMST-L have more advantages than the comparing algorithms.

Similar to Sect. 5.2.1 where other parameters are given, we compare the time performance by varying sequence lengths, varying pattern sizes, varying wildcard numbers, varying gap sizes and different lengths of exact characters in a pattern.

As shown in Fig. 9, the comparison of four algorithms is under different lengths of object sequence, $n = 2500$, 5000, 10,000, 20,000 and 34,350. Other parameters are similar to the first test in DNA sequence where the pattern set $\mathcal{P} = \{P_0, P_1, \ldots, P_{k-1}\}$, $k = 10$, and each single pattern $P_i = p_0 l_0, h_0 p_1 \ldots\ l_{j-1}, h_{j-1} p_j \ldots l_{m-1}, h_{m-1} p_m$. Here, $p_0 \ldots p_j \ldots p_m$

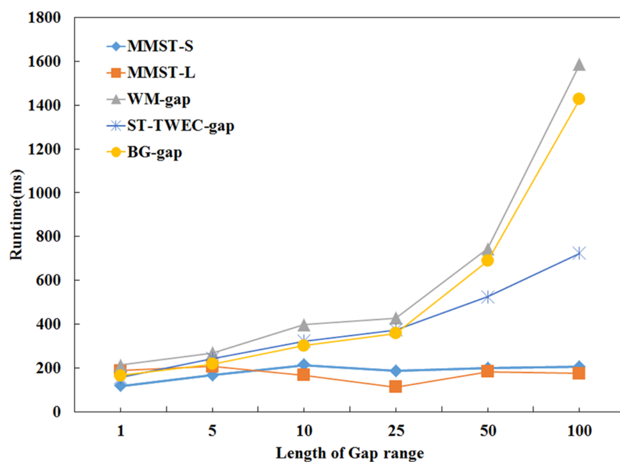

Fig. 8 Comparison of the runtime by varying gap ranges of wildcards

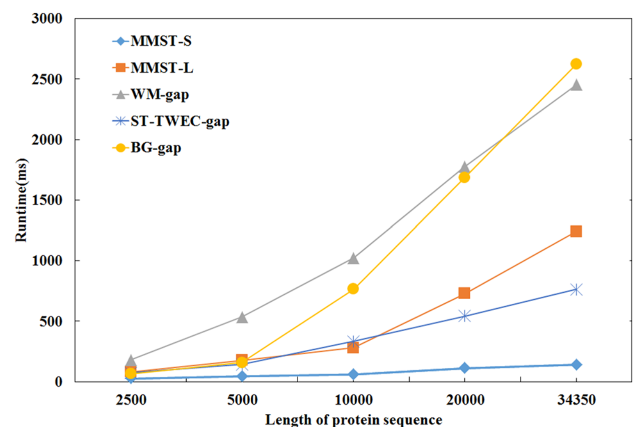

Fig. 9 Comparison of the runtime by varying lengths of protein sequence

$\in \sum = \{a, c, d, e, f, g, h, i, k, l, m, n, p, q, r, s, t, v, w, y\}$, m is 3, the number of wildcards $w = 2$ and the maximum of gap range $G = 9$. The form of each pattern is short exact character. The result is similar to that in DNA sequence. These three compared algorithms consume more time with the increase in the length of the object sequences. The difference is that the algorithm MMST-S is faster when the size of alphabet character increases. This is because more impossible positions are excluded. When it comes to the algorithm MMST-L, the number of leaf nodes is also increasing with the length of the object sequence increasing under the short pattern form.

Figure 10 shows the runtime of four algorithms with the increase in the number of the patterns. Here, k changes from 1 to 100, n is 34,350, and other parameters are similar to the first test in protein sequence. In particular, when the size of patterns k is big, the algorithm BG-gap will be error.

Because there are less repeat characters in the protein sequence, the number of occurrences becomes less corresponding to the increase in the number of wildcards. In this experiment, we also take the short form pattern to get more occurrences. As shown in Fig. 11, the number of wildcards w increases from 1 to 6, when $n = 34{,}350$, $k = 10$, and $G = 9$. The algorithm MMST-S is the fastest among other algorithms. It takes the MMST-L more time to judge the added possible occurrence positions with the increase in w.

Figure 12 shows the comparison of runtime with different gap ranges of wildcards. The maximum value of gap range G is from 1 to 100, when $n = 34{,}350$, $k = 10$, and $w = 1$. The gap and occurrence present a positive correlation. With the increase in the gap, the number of occurrences also increases correspondingly. So we take the long form pattern in this test. Compared with the DNA sequence, the protein sequence contains 20 characters, and since our algorithms MMST-S and MMST-L can exclude more impossible positions, these two algorithms are much more efficient than WM-gap, BG-gap and ST-TWEC-gap.

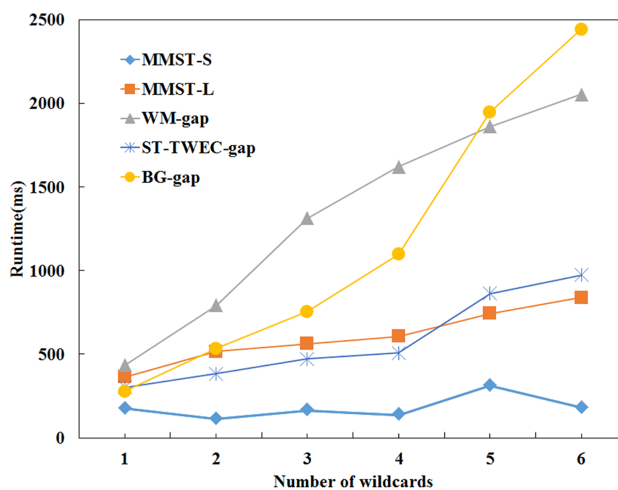

Fig. 11 Comparison of the runtime by varying numbers of wildcards

5.2.3 Performance analysis

With the above time performance experiments on DNA sequence and protein sequence, we conclude that the algorithms MMST-S and MMST-L have better performance than compared algorithms. Although the construction of suffix tree is time-consuming, the tree can be used repeatedly. It is an advantage in multi-pattern matching. Additionally, MMST-S and MMST-L exclude more impossible positions

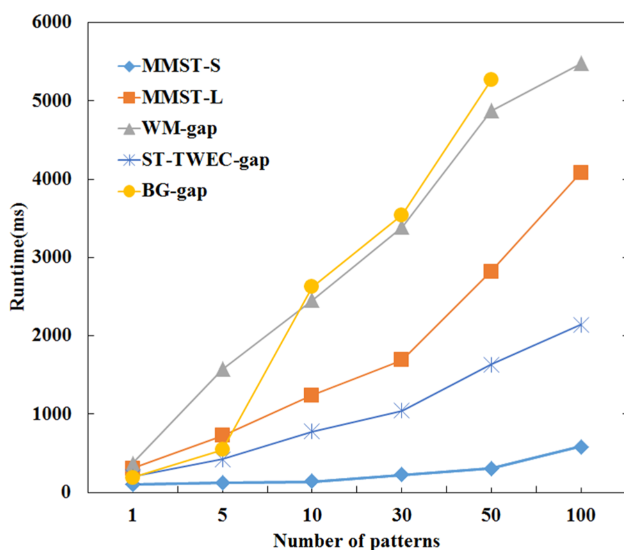

Fig. 10 Comparison of the runtime by varying numbers of patterns

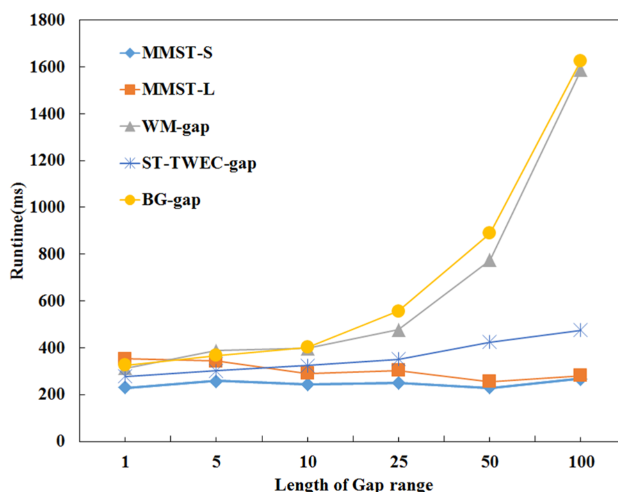

Fig. 12 Comparison of the runtime by varying gap ranges of wildcards

by the properties of suffix tree. Combined with our optimized judgments, MMST-S and MMST-L have comparative higher efficiency in pattern matching with variable-length wildcards.

The compared algorithm BG-gap is based on the bit parallelism, which is limited by processor word size. This algorithm is only valuable for a small set of patterns or small object sequence. In Figs. 6 and 10, the BG-gap is overflow when the number of patterns k is 100. Moreover, there are two major difficulties sabotaging the re-composition of the BG-gap algorithm for the variable-length wildcards. One is to shift the set of patterns safely to avoid skipping an occurrence; the other is to recognize the character block in the object sequence.

WM-gap recomposed WM and split the segments by wildcards, leading to the differences in the length of the segments. Besides, it will consume more time to choose the reasonable size of shifts of the blocks. It reduces the efficiency of the algorithm.

ST-TWEC-gap is recomposed ST-TWEC algorithm which presented a suffix-tree-based tweet clustering algorithm. Because this compared algorithm is also based on a suffix tree index structure, the efficiency of the algorithm is similar to our algorithms MMST-S and MMST-L than other compared algorithms in sometimes. However, MMST-S and MMST-L algorithms are more focused on multi-pattern matching with wildcards than ST-TWEC-gap algorithm.

Consequently, our algorithms MMST-S and MMST-L based on suffix tree are more efficient than compared algorithms in multiple patterns matching with variable-length wildcards.

6 Conclusion and future work

In this paper, we have proposed two algorithms, namely MMST-S and MMST-L, based on the suffix tree for solving the problem of the multi-pattern matching with variable-length wildcards. The results of our empirical study suggest that our algorithms are more efficient than compared algorithms in terms of time performance. According to the length of exact characters in patterns, we designed MMST-S based on dynamic programming for short form patterns and MMST-L based on editing distance for long form patterns. It is demonstrated, respectively, by a series of experiments in DNA sequences and protein sequences.

There are several interesting issues that will be studied in our future work. For example, the problem of arbitrary length wildcards needs to consider negative or infinite values. Additionally, the problem of mining approximate patterns based on the suffix tree, and developing applications in other domains such as document representation and indexing [16] are also worthwhile topics. The mined patterns can be used to reflect semantic relations between phrases in documents.

Acknowledgements This research is supported by the National Key Research and Development Program of China (Grant No. 2016YFB1000900) and National Natural Science Foundation of China (NSFC) (Grant Nos. 61503116 and 61229301).

References

1. Gonzalo N, Mathieu R (2007) Flexible pattern matching in strings: practical on-line search algorithms for texts and biological sequences. Publishing House of Electronics Industry, Beijing
2. Aho AV, Corasick MJ (1975) Efficient string matching: an aid to bibliographic search. Commun ACM 18(6):333–340
3. Baeza-Yates R, Gonnet GH (1992) A new approach to text searching. Commun ACM 35(10):74–82
4. Commentz-Walter B (1979) A string matching algorithm fast on the average. Automata, languages and programming, pp 118–132
5. Wu S, Manber U (1994) A fast algorithm for multi-pattern searching. Department of Computer Science, University of Arizona, Tucson
6. Raffinot M (1997) On the multi backward dawg matching algorithm (MultiBDM). In: Proceedings of the 4th South American workshop on string processing. Carleton University Press, pp 149–165
7. Allauzen C, Raffinot M (1999) Factor oracle of a set of words. Technical report 99-11
8. Rahman MS, Iliopoulos CS, Lee I et al (2006) Finding patterns with variable length gaps and don't cares. In: Proceedings of the 12th annual international computing and combinatorics conference, vol 8, pp 146–155
9. Akutsu T (1996) Approximate string matching with variable length don't care characters. IEICE Trans Inf Syst 79(9):1353–1354
10. Fischer MJ, Paterson MS (1974) String-matching and other products. In: Proceeding of the 7th SIAM AMS complexity of computation, Cambridge, USA, pp 113–125
11. Min F, Wu XD, Lu ZY (2009) Pattern matching with independent wildcard gaps. In: Proceedings of the 8th IEEE international conference on dependable, autonomic and secure computing. Chengdu, China, IEEE, pp 194–199
12. Guo D, Hong XL, HuX G, Gao J, Liu YL, Wu GQ, Wu XD (2011) A bit-parallel algorithm for sequential pattern matching with wildcards. Cybernet Syst 42(6):382–401
13. Bille P, Gørtz IL, Vildhøj HW, Wind DK (2012) String matching with variable length gaps. Theoret Comput Sci 443(1):25–34
14. Inenaga S, Hoshino H, Shinohara A, Takeda M, Arikawa S, Mauri G, Pavesi G (2001) On-line construction of compact directed acylic word graphs. In: Proceedings of the 12th annual symposium on combinatorial pattern matching, pp 169–180
15. Zhang M, Zhang Y, Hu L (2010) A faster algorithm for matching a set of patterns with variable length don't cares. Inf Process Lett 110(6):216–220
16. Zhang H, Chow TW, Wu QM (2016) Organizing books and authors by multilayer SOM. IEEE Trans Neural Netw Learn Syst 27(12):2537
17. Weiner P (1973) Linear pattern matching algorithm. In: 14th annual IEEE symposium on switching and automata theory, pp 1–11

18. Giegerich R, Kurtz S (1997) From Ukkonen to McCreight and Weiner: a unifying view of linear-time suffix tree construction. Algorithmica 19(3):331–353
19. Grossi R, Italiano GF (1993) Suffix trees and their applications in string algorithms. In: Proceedings of the 1st South American workshop on string processing, pp 57–76
20. Zhou Z, Zhang T, Chow SSM, Zhang Y, Zhang K (2016) Efficient authenticated multi-pattern matching. In: Presented at the 11th ACM, ACM Press, New York, USA, pp 593–604. http://doi.org/10.1145/2897845.2897906
21. Raffinot M (1997) On the multi backward Dawg matching algorithm (MultiBDM). In: Baeza-Yates R, (ed) Proceedings of the 4th South American workshop on string processing, Valparaíso, Chile. Carleton University Press, pp 149–165
22. Crochemore M, Czumaj A, Gasieniec L, Lecroq T, Plandowski W, Rytter W (1999) Fast practical multi-pattern matching. Inf Process Lett 71(3/4):107–113
23. Muth R, Manber U (1996) Approximate multiple string search. In: Proceedings of the 7th annual symposium on combinatorial pattern matching, number 1075 in lecture notes on computer science, Springer, Berlin, pp 75–86
24. Baeza-Yates RA, Navarro G (1997) Multiple approximate string matching. In: Proceedings of the 5th workshop on algorithms and data structures, number 1272 in lecture notes in computer science, Springer, Berlin, pp 174–184. Extended version to appear in Random Structures and Algorithms (Wiley)
25. Cole R, Hariharan R (2002) Verifying candidate matches in sparse and wildcard matching. In: Proceedings of the 34th annual ACM symposium on theory of computing, May 2002, pp 592–601
26. Rahman MS, Iliopoulos CS, Lee I et al (2006) Finding patterns with variable length gaps or don't cares. In: Proceedings of the 12th annual international computing and combinatorics conference, August 2006, pp 146–155
27. Haapasalo T, Silvasti P, Sippu S, Soisalon-Soininen E (2011) Online dictionary matching with variable-length gaps. In: Proceedings of the 10th international symposium, SEA Kolimpari, Chania, Crete, Greece. Springer, Berlin, pp 76–87
28. Kucherov G, Rusinowitch M (1997) Matching a set of strings with variable length don't cares. Theoret Comput Sci 178(1–2):129–154
29. Kulekci MO (2007) TARA: an algorithm for fast searching of multiple patterns on text files. In: 22nd international symposium on computer and information sciences, pp 136–141
30. Zhang M, Zhang Y, Tang J, Bai X (2011) Multi-pattern matching with wildcards. J Softw 6(12):2391–2398
31. McCreight EM (1976) A space-economical suffix tree construction algorithm. J ACM 23(2):262–272
32. Ukkonen E (1995) On-line construction of suffix trees. Algorithmica 14(3):249–260
33. Gusfield D (1997) Algorithms on strings, trees, and sequences Computer Science and Computational Biology. Cambrigde University Press, Cambridge
34. Chattaraj A, Parida L (2005) An inexact-suffix-tree-based algorithm for detecting extensible patterns. Theoret Comput Sci 335(1):3–14
35. Ukkonen E (2009) Maximal and minimal representations of gapped and non-gapped motifs of a string. Theoret Comput Sci 410(43):4341–4349
36. BilleP Gørtz IL et al (2014) String indexing for patterns with wildcards. Theory of Computing Systems 55(1):41–60
37. Thankachan SV, Apostolico A, Aluru S (2016) A provably efficient algorithm for the k-mismatch average common substring problem. J Comput Biol 23(6):472–482
38. Salmela L, Tarhio J, Kytöjoki J (2007) Multi-pattern string matching with q-grams. J Exp Algorithm 11(1):1–19
39. Ukkonen E (1992) Approximate string-matching with q-grams and maximal matches. Theoret Comput Sci 92(1):191–211
40. Arın İnanç, Erpam MK, Saygın Y (2018) I-TWEC: interactive clustering tool for Twitter. Expert Syst Appl 96:1–13
41. NCBI: http://www.ncbi.nlm.nih.gov/