Efficient String Matching
with Wildcards and Length Constraints

Gong Chen, Xindong Wu, Xingquan Zhu, and Abdullah N. Arslan

Department of Computer Science,
University of Vermont,
Burlington VT 05405, USA
\{gchen, xwu, xqzhu, aarslan\}@cs.uvm.edu

Abstract. This paper defines a challenging problem of pattern matching with wildcards and length constraints, and designs an efficient algorithm to return each pattern occurrence in an on-line manner. Given a pattern $P$ and a text $T$, a substring $S$ in $T$ is a matching pattern of $P$ if (1) the number of wildcards between each two consecutive letters in $S$ and (2) the length of $S$, are both bounded by the user’s specifications. We design a complete algorithm, SAIL\(^1\) that returns each matching pattern as soon as it appears in $T$ in an $O(klmn)$ time with an $O(lm)$ space overhead, where $k$ is the frequency of $P$’s last letter occurring in $T$, $l$ is the user-specified maximum length for $S$ in which $P$ occurs, $m$ is the length of $P$, and $g$ is the maximum difference between the user-specified maximum and minimum numbers of wildcards allowed between each two consecutive letters in $S$. SAIL is inherently an on-line algorithm where $T$ is generated as the algorithm is running. It returns an occurrence of the given pattern $P$ as soon as it appears in the input of $T$.

1 Introduction

Given a pattern $P$ and a text $T$, classical pattern matching returns each location in $T$ where $P$ matches a substring of $T$. The matching can be exact, or with some errors (known as approximate pattern matching). In either case, the problem is usually complicated by taking wildcards\(^2\) into the matching process. Not only has pattern matching with wildcards a theoretical research value but also it owns a significant impact on many real-world applications such as text indexing, biological sequence analysis, stream data mining and sensor networking.

- Text Indexing. The Internet is awash with an abundance of textual information due to large and growing collections of databases and articles. It is essential to be able to retrieve correct information that satisfies the user’s queries. Text indexing is one of the common techniques to serve this purpose, and a challenging task in this regard is to locate positions where the user

\(^1\) String matching with wildcards and length constraints.
\(^2\) A wildcard is also called a don’t care value in some references. They are used interchangeably in this paper.
specified pattern occurs in the text, with possible wildcards in the pattern [2].

- **Sequence Analysis.** The DNA sequence TATA is a common promoter that often occurs after the sequence CAAATCT within 30 to 50 wildcards [9,1]. So matching patterns with wildcards becomes especially crucial in exploring valuable information from DNA sequences. In addition, one can find many examples about patterns with wildcards in biological sequences and their corresponding motivations in [5].

- **Stream Data Mining.** Streaming data are of growing importance in many new database applications such as data warehousing and sensor networking. Mining dependencies or correlations among a large number of data streams becomes more and more critical [10,16], where subsequence matching with wildcards is the very first (and also the most important) step.

There are many other applications that involve (or crucially depend on) pattern matching with wildcards, and research from theoretical complexity analysis has provided many solutions to solve different kinds of problems. Among them, Fischer and Paterson [4] were the first who generalized pattern matching with wildcards: given a pattern P and a text T, either of which may contain wildcards, denoted by φ, the goal is to locate all P’s occurrences in T. φ can match any literal in a given alphabet. Several improvements in terms of time complexity have been proposed, but the following problems are still open.

1. **Complex Local Constraints.** When searching for occurrences of a pattern P in a text T, the user may want to specify a different range of wildcards between each two consecutive letters of P, as shown in **Example 1** below. With such a range specification, the user would have more flexibility to control queries. Unfortunately, no existing efforts have provided a good solution for this problem³.

2. **Global Length Constraints.** In many situations, the user may want to constrain the length of each matching substring of T in which P occurs. Although the local constraints have an impact on this length to some degree, no effective mechanism exists for the user to control it.

We will integrate both local and global constraints for effective pattern matching.

**Example 1** Given a pattern P=ωφ₀₋₃ bφ₁₋₂ c and a text T=adaddbccc, where φ is a wildcard, which can match any letter in the text. The range of wildcards between a and b specified by the user is [0,3], and between b and c is [1,2]. Suppose the range of each matching substring’s length specified by the user is [7,8]. The following alignment shows one occurrence.

| T: a d a d b c c c | P: a φ φ b φ φ c |

³ One efficient solution exists when the length of the pattern is a small constant, and this solution will be reviewed in Section 2.
Actually the above occurrence of $P$ in $T$ is the only one. The reader may think about the following alignment:

$$
T: \text{a d a d b c c c}
$$

$$
P: \quad a \phi \phi b \phi c
$$

If we only consider local constraints, the above alignment is fine. However, since the length of the matching substring in which $P$ occurs is 6 and the global length constraints are not satisfied, thus no occurrence of $P$ happens in the above alignment.

Based on the above observations and inspired by the existing work in [4, 9, 8, 1, 11, 6, 7, 2], we will define a problem in Section 3, which is different from all existing research efforts and can be used to deal with Example 1, and propose an efficient algorithm as its solution. In our problem definition, the user can specify the range of wildcards between each two consecutive letters of the pattern and the range of the length of a matching substring in which the pattern occurs.

The remainder of this paper is organized as follows. Section 2 provides a survey on related research efforts and analyzes their differences from our problem. Section 3 presents our problem statement. Section 4 describes the design of our algorithm, SAIL, and provides a running example. Section 5 analyzes the correctness and complexity of SAIL. Section 6 draws our conclusions.

2 Related Work

The problem of string matching with wildcards was first studied by Fisher and Paterson [4]. A wildcard $\phi$ can match any letter in a given alphabet. The objective of their work was to find all locations of occurrences of a pattern $P$ in a text $T$. The user can specify a fixed number of $\phi$'s between every two consecutive letters in $P$ when matching with $T$. Several improvements in terms of time complexity have been made by [11, 6, 7]. [2] considered a slightly different problem, where instead of fixing the number of $\phi$'s between two consecutive letters in $P$ and $T$, they fixed the total number of $\phi$'s in $P$. The most critical disadvantage of these research efforts is that the number of $\phi$'s is a constant but not a range, so it provides very limited flexibility for the user's queries.

To alleviate the problem of a fixed number of $\phi$'s, Kucherov [8] proposed a solution to allow an unbounded number of $\phi$'s between two consecutive letters in a given pattern. Given a set of such patterns, their objective is to find whether any of these patterns matches some substring of the text that does not contain any $\phi$. Obviously, allowing an unbounded number of $\phi$'s still does not offer the users enough flexibility to control their queries.

In real-world applications, such as text searching, many queries can be represented by the fly (followed-by) operator. For example, we may try a query 'mice fly fog' which means mice followed by fog, separated by at most $g$ (a user-defined constant) literals. This inevitably involves queries with a range of $\phi$'s. To this end, Manber [9] proposed an algorithm for string matching with a sequence of wildcards. This approach considered the following problem: given two pattern
strings P and Q, each of which consists of literals, and an integer $g$, all occurrences of the form $P_\phi Q$ in the text are returned. The number of $\phi$'s between P and Q is in the range of $[0,g]$, and the text does not contain any $\phi$. This problem was called exact string matching with variable-length don't cares in [1] which studied approximate string matching with variable-length don’t cares based on edit distances and dynamic programming. Manber’s work [9] is the closest to our problem in this paper, but since this method was developed to handle two patterns only, it is not clear how effective this method could be when dealing with multiple patterns (like the example in Example 1). If both P and Q consist of only one literal and there are three or more pattern strings $P_1 \phi_0 Q_1, P_2 \phi_0 Q_2 \ldots P_m \phi_0 Q_m$, this method can possibly first find occurrences of the first two patterns, and then check for the third pattern, and so on. Nevertheless, a serious flaw exists for such a naive solution: in the worst case, the number of all possible occurrences is in fact exponential, for example, it can be greater than $g^{m-1}$ where $g_{\text{max}}$ is $\max \{g_i\}(i=1,2,\ldots,m-1)$. When the number of patterns is relatively small, two or three as Manber mentioned, this method may be an efficient solution. But when the numbers of patterns and wildcards are large, its efficiency becomes a problem.

In summary, with the problem in [4,11,6,7], the number of $\phi$'s between every two consecutive literals is fixed. In [2], the total number of $\phi$'s is a constant. In [9, 1], the number range of $\phi$'s can be specified but only one sequence of $\phi$'s can occur between two patterns. In [8], $\phi$'s can occur anywhere in the pattern but the number of $\phi$'s in every occurrence is not bounded. We consider a challenging problem that the number range of $\phi$'s between every two consecutive literals in the pattern can be specified separately. This consideration gives much more flexibility for the user to specify query patterns than other research efforts. In addition, we note that a range of $\phi$’s is just a local constraint for a query pattern and the user may also want to specify a range for the total length of a matching substring in which the pattern occurs. So we also take global length constraints into consideration.

We have considered the feasibility of applying some common approaches on our pattern matching problem. When the one-off condition in Section 3 is not required, and there are no global length constraints, finding all occurrences of a pattern in a text is a well-studied problem (see [3] for example). A classical approach is to construct a finite automaton. We can design a finite automaton with $O(gm)$ states that recognizes a single occurrence with wildcards and local length constraints where $g$ is the maximum number of wildcards allowed between any two consecutive letters in the pattern. However, if we also want to incorporate the global length constraints and the one-off condition in Section 3, then this approach increases the number of necessary states exponentially because there are exponentially many ways of assigning the letters of the text to occurrences. Therefore, this approach does not offer a tractable solution.

Suppose we would like to design a context free grammar and reduce our pattern matching problem to a parsing problem for context free languages. This
time we need exponentially many production rules because the symbols of the occurrences can interleave.

Dynamic programming deals with pairwise sequence alignment in an efficient way [14], but does not consider either local constraints between consecutive matching letters or global length constraints in the pattern description. Therefore, it cannot provide a tractable solution for our pattern matching problem.

3 Problem Statement

Definition 1. Given

- a pattern $P = p_0...p_{m-1}$, a text $T = t_0...t_{n-1}$, where $p_j \neq \phi$, $0 \leq j \leq m - 1$, $t_i \neq \phi$, $0 \leq i \leq n - 1$, and $m$ and $n$ are the lengths of $P$ and $T$ respectively;
- $[\text{min}_j, \text{max}_j]$, the number range of $\phi$’s between $p_j$ and $p_{j+1}$, where $0 \leq j \leq m - 2$;
- minLen and maxLen, the minimum and maximum length constraints for a $T$’s substring in which $P$ occurs;\footnote{Due to $[\text{min}_j, \text{max}_j]$, the following inequality should be satisfied: $m + \sum_{j=0}^{m-2} \text{min}_j \leq \text{minLen} \leq \text{maxLen} \leq m + \sum_{j=0}^{m-2} \text{max}_j$.}

if there exists a sequence of position indices $i_0, ..., i_j, ..., i_{m-1}$ where $0 \leq i_j \leq n - 1$, $0 \leq j \leq m - 1$, $i_{j-1} < i_j$ when $1 \leq j \leq m - 1$ such that

1. $t_{i_j} = p_j$,
2. $\text{min}_{j-1} \leq i_j - i_{j-1} - 1 \leq \text{max}_{j-1}$,
3. $\text{minLen} \leq i_{m-1} - i_0 + 1 \leq \text{maxLen},$

the sequence $\{i_0, i_1, ..., i_{m-1}\}$ is an occurrence of $P$ in $T$, and the substring $t_{i_0}t_{i_0+1}...t_{i_j}...t_{i_{m-1}}$ is a matching substring. The second condition is a local constraint and the third condition is a global constraint.

Problem 1. Our objective is to find the maximum number of occurrences of $P$ in $T$, on conditions that every literal in $T$ can only be used once for matching $p_j (0 \leq j \leq m - 1)$ and as soon as there exists one occurrence of $P$ in $T$ when $T$ is being scanned from left to right it will be returned.

We call the condition that every literal in $T$ can only be used once for matching $p_j (0 \leq j \leq m - 1)$ one-off condition. We call the condition that as soon as there exists one occurrence of $P$ in $T$ when $T$ is being scanned it will be returned on-line condition. Under the above two conditions, an occurrence of $P$ that can maximize the total number of occurrences of $P$ in $T$ is an optimal occurrence. Position $i_j$ of an optimal occurrence is an optimal position for $p_j$, denoted by $i_j$.

For example, given $P=\text{abc}$, $T=\text{abc}$, $\text{min}_0 = 0$, $\text{max}_0 = 1$, $\text{min}_1 = 0$, $\text{max}_1 = 2$, $\text{minLen} = 3$, and $\text{maxLen} = 4$, $t_0t_1t_2$ is a matching substring. After it has
been used to match one occurrence of \( P \), \( t_0 t_1 \) cannot be used again under the one-off condition. Therefore \( P \) occurs only once in \( T \).

This one-off condition has both theoretical and practical significance. If literals in \( T \) can be used to match the occurrence of \( P \) for more than once, the number of occurrences of \( P \) in \( T \) can possibly be greater than \( k g^{m-1} \) where \( k \) is the frequency of \( P \)'s last letter occurring in \( T \), \( m \) is \( P \)'s length, and \( g \) is \( \max\{\max_j - \min_j\} \) \((j = 0, ..., m - 2)\). Thus, an efficient algorithm for returning all occurrences of \( P \) is impossible. In reality, it might not be necessary to find “all” those occurrences. For example, in sequential pattern analysis in data mining [13, 15, 12], the user is more interested in how frequent a candidate shopping pattern appears. If the number of occurrences of a pattern in the transaction database is greater than a user-specified threshold, this pattern is of the user’s interest. In the previous example, \( T \) can be treated as a sequence of transactions that one customer buys, in the temporal order of item \( a \), item \( b \), item \( c \) and item \( d \); and \( P \) can be treated as a candidate shopping pattern. The user is interested to know how frequent this shopping pattern \( P \) occurs in a set of transactions. In this situation, it is unreasonable to count \( P \) twice in \( T \). Also, when the user uses a web search engine to retrieve documents, \( P \) can be treated as a list of query keywords, and the more frequently \( P \) occurs in one document, the higher the rank of the document in the returned document list. It is obvious that when we count the number of occurrences of pattern \( P \) (abc) in a document \( T \) (abcc), it makes sense to count each occurrence for only once.

4 Algorithm Design

4.1 Issues and Procedures

To support efficient pattern matching and other on-line applications, we have taken three important issues into consideration to design SAIL:

1. On-line searching. SAIL should output an occurrence of the given pattern as soon as it appears in the input of \( T \) so far.
2. Backtracking. A naive algorithm [13] would be as follows. In a forward phase, find successive elements of \( P \) in \( T \) as long as the difference between the positions of the element \( p_j \) just found and \( p_{j-1} \) is in the range of \([\min_{j-1}, \max_{j-1}]\). If it is more than \( \max_{j-1} \), the algorithm switches to a backward phase and “pulls up” \( p_{j-1} \)'s position to satisfy the range. Pulling up \( p_{j-1} \) may necessitate pulling \( p_{j-2} \) because the difference between \( p_{j-1} \) and \( p_{j-2} \) may be more than \( \max_{j-2} \); and a series of pulling-ups may be incurred till \( p_0 \) is pulled up. Then the algorithm switches to the forward phase to find the next element of \( P \). This procedure is repeated till all \( P \)'s elements are found. That is an “occurrence” is found.\(^5\) From the above description, one can see that this naive solution is very inefficient because it involves an expensive backtracking procedure. Therefore, SAIL should not use this costly procedure.

\(^5\) Here the global constraints are omitted.
3. **Optimization.** Under the one-off condition, SAIL should determine which occurrence is an optimal one if multiple occurrences exist for a \( p_{n-1} \)'s position. For example, \( P = \text{abc} \), \( T = \text{aabbcc} \), \([\text{min}_0, \text{max}_0] = [0, 1], [\text{min}_1, \text{max}_1] = [0, 1], \) minLen=3 and maxLen=5. When the first \( p_2(c) \) at position 4 comes, four possible occurrences of \( P \) exist. They are \( \{0,2,4\}, \{1,3,4\}, \{0,3,4\} \) and \( \{1,2,4\} \). Which one is the optimal occurrence? As we can see, another \( p_2(c) \) in position 5 will come. Under the constraints, the only possible occurrence for this position is \( \{1,3,5\} \). As a result, \( \{0,2,4\} \) is the only optimal occurrence for \( p_2(c) \) in position 4 since other occurrences consume literals of a future matching substring.

Before we go into the details of SAIL, we provide a simple example to explain the main idea of SAIL\(^6\). It shows how SAIL finds an optimal occurrence. Basically, SAIL conducts two phases, a **forward phase** and a **backward phase**. In the **forward phase**, SAIL sequentially scans the input of \( T \) for possible \( p_j(j = 0, 1, \ldots, m-1) \)'s positions in \( T \) and constructs a search table, as shown in Figure 1. This table is constructed until it contains at least one occurrence of \( P \). Then the **backward phase** is triggered to locate one optimal occurrence\(^7\). In this phase, literals used for matching this occurrence are marked to avoid that they will be used again. Then SAIL re-initializes the search table with 0 in every cell for future matching.

To conduct efficient matching based on a constructed search table, we shall resolve two important issues: (1) how SAIL constructs every cell of the search table and (2) when it stops the **forward phase** and switches to the **backward phase**. The answers will be provided by the following procedures of SAIL.

1. List \( p_0, p_1, \ldots, p_j, \ldots, p_{n-1} \) and start from \( t_0 \) to locate every pattern literal \( p_j \)'s possible position in \( T \).

<table>
<thead>
<tr>
<th>( T )</th>
<th>( t_0 )</th>
<th>( t_1 )</th>
<th>( t_2 )</th>
<th>( t_3 )</th>
<th>( t_4 )</th>
<th>( t_5 )</th>
<th>( t_6 )</th>
<th>( t_7 )</th>
<th>( t_8 )</th>
<th>( t_9 )</th>
<th>( t_10 )</th>
<th>( t_11 )</th>
<th>( t_12 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( P )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( p_0 )</td>
<td>a</td>
<td>a</td>
<td>d</td>
<td>f</td>
<td>e</td>
<td>e</td>
<td>f</td>
<td>h</td>
<td>b</td>
<td>b</td>
<td>g</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( p_1 )</td>
<td>e</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>( p_2 )</td>
<td>f</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>( p_3 )</td>
<td>b</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>( p_4 )</td>
<td>a</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Fig. 1. A Search Table for \( P = \text{e}^3 \text{f}^3 \text{g}^3 \) and \( T = \text{a} \text{d} \text{f} \text{e} \text{c} \text{f} \text{b} \text{b} \text{g} \) and \( T = \text{a} \text{d} \text{f} \text{e} \text{c} \text{f} \text{b} \text{b} \text{g} \)

2. SAIL sets "1" to the corresponding cell in Figure 1 to indicate a seed position*(see Definition 2 below). This mechanism guarantees that as long as a

\(^6\) To help a better understanding, we ignore the global constraints in this example and some details are omitted.

\(^7\) Several optimal occurrences may exist and the occurrence SAIL returns is definitely one of them.
sequence $i_0, i_1, \ldots, i_j$ exists in $T$, which satisfies the corresponding local constraints, the cell for a seed position of $p_j$ is marked in a search table. This will be proved by Lemma 2 in Section 5.1.

3. SAIL keeps searching seed positions of every pattern literal $p_j$ till one seed position for the last pattern literal $p_{m-1}$ is found\(^8\). Once one seed position of $p_{m-1}$ is found, the **backward phase** is triggered to locate an optimal occurrence of $P$ in $T$. Actually one seed position of $p_{m-1}$ indicates that there is a sequence $i_0, i_1, \ldots, i_{m-1}$ which satisfies the local constraints, so in the **backward phase** an optimal occurrence is guaranteed to be returned.

4. In the **backward phase**, using $i_{m-1}$, a seed position of $p_{m-1}$, which trivially becomes an optimal position of $p_{m-1}$, SAIL finds $i_{m-2}$. Using $i_{m-2}$, SAIL finds $i_{m-3}$. This procedure is repeated until all optimal positions of one optimal occurrence are located. The search table is re-initialized for the next round search. In Figure 1, we have highlighted the trace of one optimal occurrence.

---

**Definition 2. Seed Position**

- When $j = 0$, if $t_i = p_j$ position $i$ is a seed position of $p_j$; and
- when $1 \leq j \leq m - 1$, if $t_i = p_j$ and $\exists$ a seed position $k$ of $p_{j-1}$ such that $\text{min}_{j-1} \leq k - i - 1 \leq \text{max}_{j-1}$, position $i$ is a seed position of $p_j$.

Note when the global constraints are considered, the definition of $p_0$’s seed position is slightly different from what we define now, which is demonstrated at a later stage.

With the above procedures, all three important issues, on-line searching, backtracking and optimization, are seamlessly integrated for efficient pattern matching. The example in Figure 1 provides some inspirations for understanding SAIL and the global constraints have not been considered yet. In the next subsection, we will combine both global and local constraints to provide a complete description of SAIL.

### 4.2 Proposed Algorithm

SAIL scans the input of $T$. For every position $i$ where $t_i = p_{m-1}$, it checks whether there is an occurrence of $P$ in $T$ and outputs an optimal occurrence if there exists one. SAIL first considers global constraints, and then deals with local constraints. That is, SAIL conducts search by using the technique in the previous subsection under length constraints.

Throughout the paper, we use notations as follows: $p[i][] := p_j$ and $t[i][] := t_i$ indicate literals of the pattern and text respectively; $\text{used}[i][] := \text{true}$ once $t_i$ has been used in an optimal occurrence of $P$; $\text{range}[j][\text{min}] = \text{min}_{j}$ and $\text{range}[j][\text{max}] = \text{max}_{j}$ specify the range of $\phi$’s number between $p_j$ and $p_{j+1}$; and occurrence$[j][]$ records $i_j$.

---

\(^8\) This step is slightly different when the global constraints are included, which can be seen from line 10 of the pseudo code in Figure 5.
as long as there exists an optimal occurrence of $P$ in $T$. pos[0,...,m-1][0,...,len-1] will be explained when it is used.

The one-off condition in Section 3 is considered with marks of used[]. Since it is simply a condition-check which can be seen from pseudo codes in the figures to follow, we will omit the description for checking this condition.

Figure 2 presents Procedure main of SAIL. It starts from the beginning of $T$ to search position $i$ where $t[i]=p[m-1]$. After that, SAIL initiates Procedure getOptOcc to locate one optimal occurrence of $P$ in $T$ if there exists one for some position $i$.

**Procedure main**

1. $i \leftarrow 0$
2. **while** $t[i] \neq \text{End}$ **do**
3. **if** $t[i]=p[m-1]$ **then** getOptOcc($i$);
4. $i \leftarrow i+1$

**Fig. 2.** Pseudo code for Procedure main

In Figure 3, given a possible position of $p[m-1]$, end, lines 1 to 4 calculate the range of possible positions of $p[0]$ by considering the global constraints. The range is $[\text{start, startLimit}]$. Then SAIL scans $T$ from the smallest possible position of $p[0] \text{ start}$ to the given position end. Two phases are going to be involved in the scanning process, as we have discussed in the above subsection. The forward phase checks whether any occurrence of $P$ exists in $T$ with $p[m-1]$ at the position end. If one or several occurrences exist, the backward phase chooses an optimal one and stores the corresponding optimal positions in occurrence[]. Finally, **Procedure output** outputs the occurrence[].

**Procedure getOptOcc(end)**

1. $\text{start} \leftarrow \text{end-maxLen+1}$;
2. $\text{startLimit} \leftarrow \text{end-minLen+1}$;
3. **if** $\text{startLimit}<0$ **then** **return**;
4. **if** $\text{start}<0$ **then** $\text{start} \leftarrow 0$;
5. **for** $i=\text{start}$ **to** end **do**
6. **forward phase**
7. **if** flagOfOcc **then**
8. **backward phase**
9. **output**(occurrence[]);

**Fig. 3.** Pseudo code for Procedure getOptOcc

Before describing the forward phase and the backward phase, we need to provide an explanation on $\text{pos[0,...,m-1][0,...,len-1]}$ since it is the key to un-
derstand these two phases. By lines 1 to 4 of GETOPTOC, we can get the smallest possible position of $p[0]$, start, given a possible position of $p[m-1]$ end. $len = end - start + 1$, which is the possible maximum length of a matching substring. So pos[$j$][i-start] actually associates with $p[j]$ and position $i$ (Figure 4). It indicates whether position $i$ is a seed position of $p[j]$. Every time when a new possible position of $p[m-1]$ comes, pos[0,...,m-1][0,...,len-1] are initialized to 0s.

**Definition 3. Seed Position (with global constraints)**

- When $j=0$, $\exists$ position $i$ such that $t[i]=p[j]$ and start $\leq i \leq$ startLimit, position $i$ is a seed position of $p[j]$; and
- when $1 \leq j \leq m - 1$, $\exists$ position $i$ such that $t[i]=p[j]$ and a seed position $k$ of $p[j-1]$ such that range.min[$j-1$] $\leq (k - i - 1) \leq$ range.max[$j-1$], position $i$ is a seed position of $p[j]$.

\[ 0 \ldots st \quad st+1 \ldots i \ldots \quad \text{end} \ldots n-1 \]

\[
\begin{array}{cccccc}
\text{pos}[0][0] & \text{pos}[0][1] & \ldots & \text{pos}[0][\text{st}] & \ldots & \text{pos}[0][\text{len}-1] \\
\text{pos}[1][0] & \text{pos}[1][1] & \ldots & \text{pos}[1][\text{st}] & \ldots & \text{pos}[1][\text{len}-1] \\
\vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\
\text{pos}[j][0] & \text{pos}[j][1] & \ldots & \text{pos}[j][\text{st}] & \ldots & \text{pos}[j][\text{len}-1] \\
\vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\
\text{pos}[m-1][0] & \text{pos}[m-1][1] & \ldots & \text{pos}[m-1][\text{st}] & \ldots & \text{pos}[m-1][\text{len}-1] \\
\end{array}
\]

**Fig. 4.** Corresponding relationship between pos[] and positions in T. The first line is positions in T and corresponding elements of pos[] are aligned below. st stands for start.

In the FORWARD PHASE (Figure 5), SAIL starts from $p[0]$ to $p[m-1]$ to search seed positions of every pattern literal $p[j]$. If position $i$ is a seed position of $p[0]$, pos[0][i-start] is set to 1. For any other pattern literal $p_j$ ($j \neq 0$), if $p[j]=t[i]$, SAIL should further check by line 8 to make sure that there exists at least one position $k$ such that $t[k]=p[j-1]$ and the local constraints between $p[j-1]$ and $p[j]$ are satisfied by $k$ and $i$. If such a $k$ exists, SAIL sets the corresponding cell of pos[] to ‘1’ to indicate that position $i$ is a seed position of $p[j]$. When SAIL treats position end as one seed position for pattern literal $p_{m-1}$, i.e., pos[m-1][end-start] is set to 1 and flagOfOcc is set to true, the BACKWARD PHASE is

---

9 We could use pos[0,...,m-1][0,...,n-1] so that pos[j][i] associates with $p[j]$ and position $i$ directly. This approach has more space costs than what we use. For an easy understanding, readers should associate pos[j][i-start] with $p[j]$ and position $i$.

10 Note we re-define $p[j]$'s seed position by Definition 3.
triggered to output an optimal occurrence by line 11 of the \textsc{forward phase} and line 7 of \textsc{procedure getOptOcc}.

\textsc{forward phase}

\begin{enumerate}
\item for \( j \leftarrow 0 \) to \( m-1 \) do
\item if \( \text{\texttt{\textasciicircum used[i]} \text{\texttt{\textasciicircum i}} \leq \text{\texttt{\textasciicircum startLimit}}} \)
\quad and \( t[i]=p[j] \quad \text{\texttt{\textasciicircum and}} \quad j=0 \) then
\quad \text{pos}[j][i-start]←1;
\item if \( \text{\texttt{\textasciicircum used[i]} \text{\texttt{\textasciicircum t[i]}=p[j]} \quad \text{\texttt{\textasciicircum and}} \quad j>0 \) then
\item localMin←range[i-1].min;
\item localMax←range[i-1].max;
\item for \( k \leftarrow \text{localMin} \) to \( \text{localMax} \) do
\item if \( i-k \geq \text{start} \)
\quad and \( \text{pos[i-1][i-k-start]}=1 \)
\quad and \( \text{\texttt{\textasciicircum used[i-k-1]} \text{\texttt{\textasciicircum then}}} \)
\quad \text{pos}[j][i-start]←1;
\item if \( j=m-1 \) and \( \text{\texttt{\textasciicircum i\textasciicircum end}} \) then
\item flagOfOcc←true;
\item break;
\end{enumerate}

\textbf{Fig. 5.} Pseudo code for the Forward Phase

In the \textsc{backward phase} shown in Figure 6, SAIL first stores position \( i \) in \text{\texttt{\textasciicircum occurrence[m-1]} \text{\texttt{\textasciicircum to register the optimal position of pattern literal}} \text{\texttt{\textasciicircum p[m-1]}} \). Then SAIL searches backward to register the optimal position of other pattern literals \( p[m-r] (2 \leq r \leq m) \) by considering the \text{\texttt{\textasciicircum range[m-r].min}} and \( \text{\texttt{\textasciicircum range[m-r].max}} \) (lines 4 to 12). When searching for an optimal position of \( p[m-r] \), SAIL selects the smallest position within the local constraints (lines 7 to 9) to guarantee optimum.

\subsection{4.3 A Running Example}

In this subsection, we show how SAIL works with a running example where \( P \), \( T \) and constraints are given as follows.

\begin{verbatim}
p[0...4]=\{babac\};
t[0...22]=\{cccccbabbbfadacabbabec\}; range[0].min=0, range[0].max=3;
range[1].min=0, range[1].max=4;
range[2].min=1, range[2].max=1;
range[3].min=0, range[3].max=3;
minLen=11;maxLen=14.
\end{verbatim}

In \textsc{procedure main}, SAIL first scans \( T \) to find a possible position of the last pattern literal \( p[m-1] \), which is 'c' in this example. As we can see, \( t[0]=p[m-1] \), so position 0 is checked first by \textsc{procedure getOptOcc}. Before SAIL conducts the \textsc{forward phase}, \( \text{\texttt{\textasciicircum maxLen(14)}} \) and \( \text{\texttt{\textasciicircum minLen(11)}} \) are used to constrain
**Backward Phase**

1. `maxPosPosi`←i;
2. `used[maxPosPosi]`←true;
3. `occurrence[m-1]`←`maxPosPosi`;
4. for `j←m-2 downto 0` do
   5. `localMin`←range[i],min;
   6. `localMax`←range[i],max;
   7. for `kt`←`localMin` to `localMax` do
      8. if `maxPosPosi-kt>start` and `pos[i][maxPosPosi-kt-start]=1` and `!used[maxPosPosi-kt]` then
         9. `maxPrePosi`←`maxPosPosi-kt`;
     10. `occurrence[i]`←`maxPrePosi`;
     11. `used[maxPrePosi]`←true;
     12. `maxPosPosi`←`maxPrePosi`;

**Fig. 6.** Pseudo code for the Backward Phase

the range of possible positions of p[0]. Due to the `minLen` constraint, it is impossible for p[0] to occur before position 0. So position 0 is not going to be checked further by lines 2 and 3 in `GETOPToc`. The same situation happens for positions 1, 2, 3 and 4 in `T`.

For position 14, the situation is now different. SAIL uses `maxLen` and `minLen` to calculate a meaningful range for p[0]'s possible positions. By lines 1 and 2 of `PROCEDURE GETOPToc`, `start=1` and `startLimit=4`. So p[0]'s possible position is in [1,4]. Then SAIL starts the FORWARD PHASE. Since `t[1]=t[2]=t[3]=t[4]=c`, no cell of `pos[0][i-start]`, i=1...,4 will be set to 1 by line 2 of FORWARD PHASE. Thus any other element of `pos[]` will not be set to 1 by the second condition of line 8 of FORWARD PHASE in this case. Therefore, no occurrence exists for position 14. If `minLen` were 10 not 11, p[0]'s possible positions would include position 5 and there would be an (optimal) occurrence {5,6,9,11,14} for position 14.

For position 21 (see Table 1), the range of p[0]'s possible positions is [8,11] ([`start, startLimit`]). In the FORWARD PHASE, SAIL checks every position from `start to end` to find seed positions of p[0](0 ≤ j ≤ m - 1). In this case, position 8 is a seed position of p[0] (by line 2 of FORWARD PHASE). Therefore, `pos[0][8-start]` is set to 1. Position 8 is not a seed position of p[2] (by lines 4 to 8 of FORWARD PHASE) though t[8]=p[2]. The same situation happens for position 9. Note that position 10 has no chance to become a seed position of any pattern literal because t[10] is different from p[0](j=0,...,4).

For position 11, t[11]=p[1]. But before SAIL sets `pos[1][11-start]` to 1 to indicate a seed position of p[1], this position is further checked to see whether the condition in Definition 3 is satisfied. So SAIL goes through lines 5 to 8 of FORWARD PHASE to check whether there exists a seed position of p[0] indicated by `pos[0][10-start], pos[0][9-start] and pos[0][8-start]`. As long as one of them is 1,
Table 1. The constructed search table pos[][] when end=21, start=8. The column index is (i-start), start ≤ i ≤ end; and the row index is j, 0 ≤ j ≤ m − 1. Since pos[j][i-start] associates with t[i] and p[j], the corresponding literals are listed in the second column and the third row. The value of i is listed in the second row.

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>b</td>
<td>b</td>
<td>f</td>
<td>a</td>
<td>d</td>
<td>a</td>
<td>c</td>
<td>b</td>
<td>b</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>a</td>
<td>c</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

The condition is satisfied, pos[1][11-start] is set to 1 due to pos[0][9-start]=1. This checking guarantees that before treating position 11 as a seed position of p[1], there exists a seed position such that p[0] at that position and p[1] at position 11 satisfy the local constraints between p[0] and p[1]. By lines 1 and 4 of FORWARD PHASE, SAIL finds that t[11] also matches p[3]. A similar procedure is followed to check whether position 11 is a seed position of p[3]. Since pos[2][9-start] is 0, which means no p[2] exists within the local constraints between p[2] and p[3] if p[3] is at position 11, thus pos[3][11-start] will not be set to 1.

Position 12’s situation is the same as position 10.


For position 14, t[14] matches p[4], so SAIL checks pos[3][13-start,...,10-start]. As a result, pos[4][14-start] cannot be set to 1.

For position 15, t[15] matches p[0] and p[2]. Line 2 of FORWARD PHASE checks position 15. It cannot be a seed position of p[0] since it is out of the range [start,startLimit] which is [8,11]. SAIL sets the corresponding pos[2][15-start] to 1 since there exists one seed position of p[1](pos[1][13-start]=1).

For positions 16 through 21, a similar checking procedure is repeated and the corresponding cells in pos[][] are set to 1. Once pos[4][21-start] is set to 1, the BACKWARD PHASE is triggered by line 11 of the FORWARD PHASE and line 7 of PROCEDURE GETOPTOC. In BACKWARD PHASE, SAIL sets position 21 as p[4]’s optimal position by line 3. Then pos[3][20-start,...,17-start] are checked (due to the local constraints range[3]_min=0, range[3]_max=3) and the smallest k for which pos[3][k-start] is 1 will be chosen as p[3]’s optimal position by lines 5 to 8 of BACKWARD PHASE. In this case, position 17 is chosen as p[3]’s optimal position. Similarly, positions 15, 11 and 8 are chosen as optimal positions of p[2], p[1] and p[0] respectively. Accordingly, {8, 11, 15, 17, 21} is selected as an optimal occurrence of the pattern.

In PROCEDURE MAIN, position 22 is checked finally (see Table 2). All used elements (marked as * in Table 2) of T are never considered for further matching
\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|}
\hline
 & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 \\
\hline
0 & b & f & a* & d & a & c & b* & b & a* & b & c & a & c* & c \\
\hline
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\hline
2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\hline
3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\hline
4 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
\hline
\end{tabular}
\caption{The constructed search table pos{}[][] when \text{end=22}, \text{start=9}. The column index is (i-start), \text{start} \leq i \leq \text{end}; and the row index is \text{j}, 0 \leq j \leq m-1. Since pos{}[][] associates with \text{t[i]} and \text{p[j]}, the corresponding literals are listed in the second column and the third row. The value of \text{i} is listed in the second column. Literals used for matching previously are marked with symbol *.
\label{table:pos_table}}
\end{table}

Again. Once pos[][22-start] is set to 1, the \textsc{backward phase} is triggered. It will finally determine one optimal occurrence of \text{P}, which is \{9, 13, 18, 20, 22\}.

Actually the selected occurrence \{9, 13, 18, 20, 22\} is the only possible occurrence of \text{P} in \text{T} for \text{p[4]} at \text{position 22}. This situation is different from the one for \text{p[4]} at \text{position 21}, where three other possible occurrences of \text{P} exist in \text{T} as shown in Table 3, except the one that has been selected, \{8, 11, 15, 17, 21\}. As shown in Table 3, if any of these three occurrences is selected (instead of \{8, 11, 15, 17, 21\}), it will consume at least one position (the one underlined in Table 3) which will be used for the only occurrence for \text{p[4]} at \text{position 22}, \{9, 13, 18, 20, 22\}. In that case, the occurrence for \text{p[4]} at \text{position 22} will not occur under the one-off condition. This is why \text{SAIL} does not select any occurrence in Table 3 as an optimal occurrence for \text{position 21}.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|c|}
\hline
\text{p[0]} & \text{p[1]} & \text{p[2]} & \text{p[3]} & \text{p[4]} \\
\hline
\text{position 9} & 9 & 11 & 15 & 17 & 21 \\
\hline
\text{position 2} & 9 & 13 & 15 & 17 & 21 \\
\hline
\text{position 2} & 9 & 13 & 18 & 20 & 21 \\
\hline
\end{tabular}
\caption{Three possible occurrences for \text{p[4]} at \text{position 21}}
\end{table}

5 Analysis

5.1 Correctness and Completeness

To demonstrate the correctness and completeness of SAIL, we need to prove four lemmas first. Note that because the one-off condition is trivial, checking this condition is omitted in the proofs.
**Lemma 1.** Given an appearance of the last pattern letter \( p[m-1] \), end, Procedure getOptOcc takes into account all possible occurrences of \( P \) in \( T \) with \( p[m-1] \) at position end and all these possible occurrences satisfy the global constraints.

*Proof.* Given minLen and MaxLen, and an appearance of \( p[m-1] \) at position end, the range of possible positions of \( p[0] \) is:

\[
[max(0, end - maxLen + 1), end - minLen + 1],
\]

Any possible occurrence ending at position end should start at a position in this range. In PROCEDURE getOptOcc, SAIL checks this through the following procedures:

By lines 1 and 4 of PROCEDURE getOptOcc:

\[
start := max(0, end - maxLen + 1)
\]

By line 2 of PROCEDURE getOptOcc:

\[
startLimit := end - minLen + 1
\]

By line 3 of PROCEDURE getOptOcc, we guarantee that

\[startLimit \geq 0\]

which indicates that no possible \( p[0] \) exists for position end, so SAIL returns and checks the next appearance of \( p[m-1] \).

By the second condition of line 2 of FORWARD PHASE, SAIL guarantees that \( p[0] \)'s seed position is in a valid range \([start, startLimit]\) which is \([max(0,end-maxLen+1), end-minLen+1]\). In addition, by line 5 of PROCEDURE getOptOcc, SAIL scans from start to end. Therefore, the maxLen and minLen constraints are satisfied. All possible occurrences under the global constraints have been taken into consideration.

**Lemma 2.** In the FORWARD PHASE, if \( \exists \) a sequence \( \{i_0, i_1, ..., i_j\} \) where \( p[0] = t[i_0], start \leq i_0 \leq startLimit \), \( p[k]=t[i_k], i_{k-1} < i_k \), and \( range[k-1].min \leq i_k - i_{k-1} - 1 \leq range[k-1].max \) when \( 1 \leq k \leq j \), then \( pos[j][k].start \) will be set to 1 where \( 0 \leq j \leq m-1 \) and this sequence has been marked in the form of \( pos[0][i_0].start]=1 \), \( pos[k][i_k].start]=1 \).

*Proof.* Due to the fact that \( start \leq i_0 \leq startLimit \) and \( p[0]=t[i_0] \), the condition on line 2 of FORWARD PHASE is satisfied. Accordingly \( pos[0][i_0].start \) is set to 1 by line 3 of FORWARD PHASE.

It is obvious that when \( 1 \leq k \leq j \), \( p[k]=t[i_k], i_{k-1} < i_k \), \( range[k-1].min \leq i_k - i_{k-1} - 1 \leq range[k-1].max \) and \( pos[0][i_0].start]=1 \), \( p[1]=t[i_1] \) which satisfies the condition on line 4 of FORWARD PHASE, \( i_0 < i_1 \), \( range[0].min \leq i_1 - i_0 - 1 \leq range[0].max \), and \( pos[0][i_0].start]=1 \) which satisfies the condition on line 8 of the FORWARD PHASE. Therefore \( pos[1][i_1].start \) will be set to 1 by line 9 of the FORWARD PHASE. Similarly, due to the fact that \( pos[1][i_1].start]=1 \) and the same conditions in Lemma 2 hold for \( k=2 \), \( pos[2][i_2].start \) will be set to 1 by line 9 of the FORWARD PHASE. When \( k=3,4,...,j \), a similar proof can be made. Accordingly, Lemma 2 is true.
Lemma 3. In the forward phase, when \( \text{pos}[j][\text{start}], 0 \leq j \leq m-1 \), is set to 1, any sequence \( \{i_0, i_1, ..., i_{j-1}, i\} \), where \( \text{start} \leq i_0 \leq \text{startLimit} \), \( i_{k-1} < i_k < i \), \( 1 \leq k \leq j-1 \), \( p[0]=t[i_0], p[k]=t[i_k], p[j]=t[i], \) range\(|k-1|.min \leq i_k - i_{k-1} - 1 \leq \) range\(|k-1|.max \) and range\(|j-1|.min \leq i - i_{j-1} - 1 \leq \) range\(|j-1|.max \), has been marked in form of \( \text{pos}[0][\text{start}] = 1 \), \( \text{pos}[k][\text{start}] = 1 \) and \( \text{pos}[j][\text{start}] = 1 \).

Proof. We prove this lemma by contradiction. Assume when \( \text{pos}[j][\text{start}], 0 \leq j \leq m-1 \), is set to 1, \( \exists \) a sequence \( i_0, i_1, ..., i_{j-1}, i \) which satisfies all conditions in Lemma 3, but at least one element \( i_r \) of this sequence has not been marked.

\( i_r \) cannot be \( i_0 \), because when \( p[0]=t[i_0] \) and \( \text{start} \leq i_0 \leq \text{startLimit} \), the condition on line 2 of the forward phase is satisfied. Therefore, \( \text{pos}[0][i_0] \) is set to 1 by line 3 of the forward phase.

\( i_r \) cannot be \( i \) since \( \text{pos}[j][\text{start}] \) is set to 1 by the first condition in Lemma 3.

The only possible case is that for \( 1 \leq r \leq j-1 \), \( \text{pos}[r][\text{start}] \) is not set to 1. If this is the case, no \( i_{r-1} \) exists by lines 4-9 of the forward phase, so that \( i_{r-1} < i < i, p[m-1]=t[i_{r-1}], p[m]=t[i_r], \) range\(|r-1|.min \leq i_r - i_{r-1} - 1 \leq \) range\(|r-1|.max \). Contradiction with the conditions of Lemma 3 is achieved.

Therefore, Lemma 3 is true.

Lemma 4. In the backward phase, one optimal occurrence of \( P \) in \( T \) is selected and stored into occurrence[].

From Table 3, we know that when position 21 is treated as \( p[4]'s \) position, in addition to the optimal occurrence, three other possible occurrences exist too.

If the backward phase had selected any of these three occurrences, the only occurrence of the pattern at position 22 (where position 22 is treated as \( p[4]'s \) position) becomes impossible. When position 21 is selected as \( p[4]'s \) position in Table 1, the backward phase does not select positions 20, 13 and 9 as the positions of \( p[3], p[2] \) and \( p[1] \) though positions 20, 13 and 9 satisfy their local constraints respectively. Intuitively, given a \( p[i]'s \) position, \( i_j \), the backward phase pushes \( p[j-1]'s \) position, \( i_{j-1} \), as far from \( i_j \) as possible. This procedure gives a future \( p[j]'s \) position \( i'_{j} \) as much chance as possible to find a \( p[j-1]'s \) position \( i'_{j-1} \) with the local constraints.

Proof. To be succinct, when we say \( x \) can consume \( y \), it means that \( p[j-1]=t[y], p[j]=t[x], \) and range\(|j-1|.min \leq x - y - 1 \leq \) range\(|j-1|.max \) where \( 1 \leq j \leq m-1 \).

In Figure 7, the important positions have been marked. We will prove a claim to demonstrate that \( i_j \) consuming \( i_{j-1} \) maximizes the chance\(^{11} \) that \( i'_{j} \) finds a \( p[j-1]'s \) position to consume.

---

\(^{11}\) Here the term \( \text{chance} \) has a slightly different meaning from its usual use. When the number of \( p[j-1]'s \) positions that \( i'_{j} \) can consume is changed from 0 to 1, the chance is increased. But when the number of \( p[j-1]'s \) positions that \( i'_{j} \) can consume is more than or equal to 1, the chance is treated as the same since \( i'_{j} \) can only consume one of them.
$$0...i_{j-1}^{'}...i_{j-1}^{''}...i_{j-1}...i_j...i_j^{'}...i_j^{''}...n-1$$

**Fig. 7.** Important sequential positions: \(1 \leq j \leq m-1; t[i_j] = t[i_j] = t[i_j] = p[j-1], t[i_j] = p[j]; 0 \leq i_j^{''} < i_{j-1} < i_j < i_j^{'} \leq n - 1; i_{j-1} < i_j^{'} < i_j^{''}; i_{j-1} \text{ is the smallest position so that range}[j-1].max \leq i_j - i_{j-1} - 1 \leq range[j-1].max.$$

First, we prove that \(i_j^{''}\) cannot consume \(i_j^{'''}\) by contradiction. Assume \(i_j^{''}\) can consume \(i_j^{'''}\). Since

\[i_j^{''} - i_j^{'''} - 1 \leq range[j-1].max\]

and

\[i_j < i_j^{''}\]

thus,

\[i_j - i_{j-1}^{'''} - 1 < range[j-1].max. \quad (1)\]

Since

\[i_j - i_{j-1} - 1 \geq range[j-1].min\]

and

\[-i_{j-1}^{'''} > -i_{j-1}\]

therefore,

\[i_j - i_{j-1}^{'''} - 1 > range[j-1].min. \quad (2)\]

By Eqs. (1) and (2), \(i_{j-1}^{'''}\) can be consumed by \(i_j^{''}\). Contradiction is achieved since \(i_j^{'''} < i_{j-1}\) and \(i_{j-1}\) is the smallest position that can be consumed by \(i_j^{''}\). Therefore, \(i_j^{''}\) can consume \(i_j^{'''}\) or some positions \(i_{j-1}^{'''}\) after \(i_{j-1}\).

Now we consider all possible cases of \(i_j^{''}\)’s consumption on \(i_j^{'}\)’s consumption.

**Case 1.** \(i_j^{''}\) and \(i_j^{'}\) can only consume \(i_{j-1}\). \(i_j^{'}\) has to consume \(i_{j-1}\) since \(i_j^{'} \leq i_j^{''}\) and the claim is trivially true.

**Case 2.** \(i_j^{''}\) can only consume \(i_{j-1}\) and \(i_j^{'}\) can have at least another choice \(i_{j-1}^{''}\) to consume. Since

\[i_j^{''} - i_{j-1}^{''} - 1 \leq range[j-1].max\]

and

\[i_{j-1} < i_j^{''}\]

thus,

\[i_j^{''} - i_{j-1}^{''} - 1 < range[j-1].max. \quad (3)\]

Since

\[i_j - i_{j-1}^{''} - 1 \geq range[j-1].min\]

and

\[i_j^{''} > i_j\]
therefore,
\[ i'_j - i'_{j-1} - 1 > \text{range}[j-1].\text{min}. \quad (4) \]

By Eqs. (3) and (4), \( i'_j \) can have another choice \( i'_{j-1} \) to consume. Contradiction is achieved. This case does not exist at all.

In the above two cases, \( i'_j \) can only consume \( i_{j-1} \). Now we consider the cases that \( i'_j \) can consume at least one position \( i'_{j-1} \) after \( i_{j-1} \). Assume that \( i_j \) has other choices to consume besides \( i_{j-1} \). Otherwise \( i_j \) has to consume \( i_{j-1} \) and the claim is trivially true.

Case 3. \( i'_{j-1} \) is not in other choices of \( i_j \). In this case, whatever \( i_j \) consumes will not affect the fact that \( i'_j \) can always consume \( i'_{j-1} \) in the future. The claim is trivially true.

Case 4. \( i'_{j-1} \) is in other choices of \( i_j \) and \( i'_j \) can consume \( i_{j-1} \). In this case, whichever \( i_j \) consumes will not affect the fact that \( i'_j \) can always consume the other in the future. That is, if \( i_j \) has consumed \( i_{j-1} \), \( i'_j \) will consume \( i'_{j-1} \); if \( i_j \) has consumed \( i'_{j-1} \), \( i'_j \) will consume \( i_{j-1} \). The claim is trivially true.

Case 5. \( i'_{j-1} \) is in other choices of \( i_j \) while \( i'_j \) cannot consume \( i_{j-1} \) but has another choice besides \( i'_{j-1} \). In this case, whichever \( (i_{j-1} \) and \( i'_{j-1} \) \( i_j \) consumes will not affect the fact that \( i'_j \) can always have at least one choice to consume in the future. The claim is trivially true.

Case 6. \( i'_{j-1} \) is in other choices of \( i_j \) while \( i'_j \) cannot consume \( i_{j-1} \) and has no other choices except \( i'_{j-1} \). In this case, whichever \( (i_{j-1} \) and \( i'_{j-1} \) \( i_j \) consumes \( i'_{j-1} \), the chance that \( i'_j \)'s consumption is decreased. But if \( i_j \) consumes \( i_{j-1} \), the chance that \( i'_j \)'s consumption is not decreased. The claim is true.

By the above proof, the **claim** that \( i_j \) consuming \( i_{j-1} \) maximizes the chance that \( i'_j \) finds a \( p[j-1] \)'s position to consume is true. Since this claim is valid for \( 1 \leq j \leq m-1 \), Lemma 4 is true.

**Claim.** Given a possible position of \( p[m-1] \), \textit{end}, if \( \exists \) one optimal occurrence of \( P \) in \( T \) with \( p[m-1] \) at end, Procedure getOptOcc will output it.

**Proof.** Given a position of \( p[m-1] \), \textit{end}, with Lemma 1, occurrences satisfying the global constraints will not be omitted by \textsc{procedure getOptOcc}. Since an optimal occurrence satisfies the length constraints by the definition, it will not be omitted by \textsc{procedure getOptOcc}.

Since an optimal occurrence is a sequence \( \{i_0, i_1, \ldots, i_k, \ldots, i_{m-1}\} \) where \( p[0] = t[i_0], \text{start} \leq i_0 \leq \text{startLimit}, p[k] = t[i_k], i_k < i_k, \text{and range}[k-1].\text{min} \leq i_k - i_{k-1} - 1 \leq \text{range}[k-1].\text{max}, 1 \leq k \leq m - 1, \text{pos}[m-1][i_{m-1}]-\text{start} \) will be set to 1 by Lemma 2. Note \( i_{m-1} = \text{end} \) since this optimal occurrence has \( p[m-1] \) at the position \textit{end} by the condition of Claim 1.

When \( \text{pos}[m-1][\text{end}-\text{start}] \) is set to 1, any sequence \( \{i_0, i_1, \ldots, i_k, \ldots, i_{m-2}, \text{end}\} \), where \( \text{start} \leq i_0 \leq \text{startLimit}, i_k < i_k < \text{end}, 1 \leq k < m - 2, p[0] = t[i_0], p[k] = t[i_k], p[m-1] = t[\text{end}], \text{range}[k-1].\text{min} \leq i_k - i_{k-1} - 1 \leq \text{range}[k-1].\text{max} \)
and range[m-2]min ≤ end - i_{m-2} - 1 ≤ range[m-2]max, has been marked in the form of pos[i_{i_0-start}] = 1, pos[k] [i_k-start] = 1 and pos[m-1][end-start] = 1 by Lemma 3. This means that all possible occurrences are considered during the backward phase including optimal occurrences. (Among these occurrences, the backward phase should choose an optimal occurrence.)

Since pos[m-1][end-start] is set to 1, the backward phase is triggered by lines 10-11 of the forward phase and line 7 of procedure getOptOcc. By Lemma 4, occurrence[i] stores an optimal occurrence. By line 9 of procedure getOptOcc, it outputs occurrence[].

Therefore, the claim is true.

By lines 2 and 3 of procedure main, all possible positions of p[m-1] in T will be checked by procedure getOptOcc. With Claim 1, if there exists one optimal occurrence for the appearance of p[m-1], it will be output. Our target in the problem definition is achieved by definition of the optimal occurrence. Therefore, correctness and completeness of SAIL have been proved.

5.2 Complexity

We assume a generic one-processor, random access machine (RAM) model of computation as our implementation technology. A constant amount of time is required to execute every time every line of our pseudo code\textsuperscript{12}.

The dominant factor of execution time in procedure main is from procedure getOptOcc. In getOptOcc, since lines 1-4 can be done in a constant time, the dominant factor is the execution time of loop 5-9.

We analyze the forward phase first. The worst case is that all p_j, (1 ≤ j ≤ m-1), are the same and the number of literals between p_{j-1} and p_j in T is max_j. Thus, by lines 1 and 7 of the forward phase, the time complexity of the forward phase is O(mg) where m is P's length and g is the max{max_k - min_k}(0 ≤ k ≤ m - 2).

With lines 4 and 7, we know that the time complexity of the backward phase is O(mg). Procedure output can be done in O(m).

In procedure getOptOcc, with line 5, it is obvious that the forward phase is executed for every position between start and end. The difference between start and end can be as large as maxLen. We use l to denote maxLen. Thus, for a possible p_{m-1}'s position end, the forward phase can run as many times as l. Since the backward phase and procedure output run only when there exists an optimal occurrence of P in T with p_{m-1} at the position end (by line 7 of getOptOcc), for every end the backward phase and procedure output run at most once. Thus, the time complexity of procedure getOptOcc is O(lmg).

All positions i where t_i = p_{m-1} are checked by procedure main. Thus, getOptOcc can run as many as k times, which is the number of p_{m-1}'s occurrences in T. In the worst case, k can be as large as n. Accordingly, the time

\textsuperscript{12} We separate the process of calling a subroutine from the process of executing the subroutine.
complexity of SAIL is \(O(nlm)\) and its space overhead is \(O(tm)\) (which is used for the search table).

6 Conclusions

In this paper, we have solved a problem of string matching with wildcards, where complex local constraints and global length constraints are seamlessly integrated to facilitate efficient on-line pattern matching. With the proposed approach, a search engine can give the users extensive flexibility to specify their queries, such as providing different granularities for different consecutive pattern literals and controlling the length of a matching substring. We have solved the problem with an algorithm, SAIL, whose time complexity is \(O(nlm)\). The analysis of correctness and completeness verifies the efficiency and performance of SAIL. Therefore, we believe the proposed problem and the SAIL algorithm have a practical significance for real-world applications like text indexing, sequence analysis and stream data mining.

SAIL is available on the Web at http://www.cs.uvm.edu/~gchem/String-Matching/onlineSmApplet.html. We invite the readers to try it and provide feedback.

References


