Theoretical Computer Science 589 (2015) 34–46

Contents lists available at ScienceDirect

Theoretical Computer Science

www.elsevier.com/locate/tcs

Dictionary matching with a few gaps[☆,☆☆]

Amihood Amir[a,b,1], Avivit Levy[c,*], Ely Porat[a], B. Riva Shalom[c]

[a] Department of Computer Science, Bar-Ilan University, Ramat-Gan 52900, Israel
[b] Department of Computer Science, Johns Hopkins University, Baltimore, MD 21218, United States
[c] Department of Software Engineering, Shenkar College, Ramat-Gan 52526, Israel

ARTICLE INFO

Article history:
Received 11 September 2014
Received in revised form 9 March 2015
Accepted 10 April 2015
Available online 18 April 2015
Communicated by R. Giancarlo

Keywords:
String matching
Dictionary matching
Gapped patterns

ABSTRACT

The dictionary matching with gaps problem is to preprocess a dictionary D of total size $|D|$ containing d gapped patterns $P_1, \dots, P_d$ over an alphabet Σ, where each gapped pattern P_i is a sequence of subpatterns separated by bounded sequences of don't cares. Then, given a query text T of length n over Σ, the goal is to output all locations in T in which a pattern $P_i \in D$, $1 \le i \le d$, ends. There is a renewed current interest in the gapped matching problem stemming from cyber security. In this paper we solve the problem where all patterns in the dictionary have one gap or a few gaps with at least α and at most β don't cares, where α and β are given parameters. Specifically, we show that the dictionary matching with a single gap problem can be solved in either $O(d \log d + |D|)$ preprocessing time and $O(d \log^{\varepsilon} d + |D|)$ space, and query time $O(n(\beta - \alpha) \log\log d \log^2 |D| + occ)$, where occ is the number of patterns found, or preprocessing time and space: $O(d^2 + |D|)$, and query time $O(n(\beta - \alpha) + occ)$, where occ is the number of patterns found. We also show that the dictionary matching with k gaps problem, where $k \ge 1$, can be solved in preprocessing time: $O(|D| \log |D|)$, space: $O(|D| + d\frac{(c_1 \log d)^k}{k!})$, and query time: $O((\beta - \alpha)^k (n + \frac{(c_2 \log d)^k}{k!} \log\log |D|) + occ)$, where $c_1, c_2 > 1$ are constants and occ is the number of patterns found. As far as we know, these are the best solutions for this setting of the problem, where many overlaps may exist in the dictionary.

1. Introduction

Pattern matching has been historically one of the key areas of computer science. It contributed many important algorithms and data structures that made their way to textbooks, but its strength is that it has been contributing to applied areas, from text searching and web searching, through computational biology and to cyber security. One of the important variants of pattern matching is pattern matching with variable length gaps. The problem is formally defined below.

Definition 1 *(Gapped pattern).* A Gapped Pattern P is of the form $p_1\ \{\alpha_1, \beta_1\}\ p_2\ \{\alpha_2, \beta_2\} \dots \{\alpha_k, \beta_k\}\ p_{k+1}$, where each subpattern p_j, $1 \le j \le k$ is a string over alphabet Σ, and $\{\alpha_j, \beta_j\}$ refers to a sequence of at least α_j and at most β_j don't

[☆] This research was supported by the Kabarnit Cyber consortium funded by the Chief Scientist in the Israeli Ministry of Economy under the Magnet Program.
[☆☆] A preliminary version of this paper appeared in CPM 2014.
[*] Corresponding author.
E-mail addresses: amir@cs.biu.ac.il (A. Amir), avivitlevy@shenkar.ac.il (A. Levy), porately@cs.biu.ac.il (E. Porat), rivash@shenkar.ac.il (B.R. Shalom).
[1] Partly supported by NSF grant CCR-09-04581, ISF grant 347/09, and BSF grant 2008217.

http://dx.doi.org/10.1016/j.tcs.2015.04.011
0304-3975/© 2015 Elsevier B.V. All rights reserved.

cares symbols between the subpatterns P_j and P_{j+1}, where a don't care symbol can be matched to any character in the query text.

Definition 2 *(The gapped pattern matching problem).*
Input: A text T of length n, and a gapped pattern P over alphabet Σ.
Output: All locations in T, where the pattern P ends.

The problem arose a few decades ago by real needs in computational biology applications [12,14,20,22]. For example, the PROSITE database [14] supports queries for proteins specified by gaps.

The problem has been well researched and many solutions were proposed. The first type of solutions [7,23,19] consider the problem as a special case of regular expression. The best time achieved using this method is $O(n(B|\Sigma|+m))$, where n is the text length, $B=\sum_{i=1}^{k}\beta_i$ (the sum of the upper bounds of the gaps), and $m=\sum_{i=1}^{k}p_i$ (the length of the non-gapped part of the pattern).

Naturally, a direct solution of the gapped pattern matching problem should have a better time complexity. Indeed, such a solution exists [8] whose time is essentially $O(nk)$.

A further improvement [6,18,24] analyses the time as a function of *socc*, which is the number of times all segments p_i of the gapped pattern appear in the text. Clearly $socc \le nk$. Rahman et al. [24] suggest two algorithms for this problem. In the first, they build an Aho–Corasick [1] pattern matching machine from all the subpatterns and use it to go over the text. Validation of subpatterns appearances with respect to the limits of the gaps is performed using binary search over previous subpatterns locations. Their second algorithm uses a suffix array built over the text to locate occurrences of all subpatterns. For the validation of occurrences of subpattern, they use van Emde Boas data structure [26] containing ending positions of previous occurrences. In order to report occurrences of the gapped pattern in both algorithms, they build a graph representing legal appearances of consecutive subpatterns. Traversing the graph yields all possible appearances of the pattern.

Their first algorithm works in time $O(n+m+socc\log(\max_j gap_j))$ where m is the length of the pattern (not including the gaps), *socc* is the total number of occurrences of the subpatterns in the text and $gap_j=\beta_j-\alpha_j$. The time requirements of their second algorithm is $O(n+m+socc\log\log n)$ where n is the length of the text, m is the length of the pattern (not including the gaps) and *socc* is the total number of occurrences of the subpatterns in the text. The DFS traversal on the subpatterns occurrences graph, reporting all the occurrences is done in $O(k\cdot occ)$ where *occ* is the number of occurrences of the complete pattern P in the text.

Bille et al. [6] also consider string matching with variable length gaps. They present an algorithm using sorted lists of disjoint intervals, after traversing the text with Aho–Corasick automaton. Their time complexity is $O(n\log k+m+socc)$ and space $O(m+A)$, where A is the sum of the lower bounds of the lengths of the gaps in the pattern P and *socc* is the total number of occurrences of the substrings in P within T.

Kucherov and Rusinowitch [16] and Zhang et al. [29] solved the problem of matching a set of patterns with variable length of don't cares. They considered the question of determining whether one of the patterns of the set matches the text and report a leftmost occurrence of a pattern if there exists one. The algorithm of [16] has run time of $O((|t|+|D|)\log|D|)$, where $|D|$ is the total length of keywords in every pattern of the dictionary D. The algorithm of [29] takes $O((|t|+dk)\log dis/\log\log dis)$ time, where dk is the total number of keywords in every pattern of P, and dis is the number of distinct keywords in D.

There is a renewed current interest in the gapped matching problem stemming from a crucial modern concern – cyber security. Network intrusion detection systems perform protocol analysis, content searching and content matching, in order to detect harmful software. Such malware may appear on several packets, and thus the need for gapped matching [15]. However, the problem becomes more complex since there is a large list of such gapped patterns that all need to be detected. This list is called a *dictionary*. Dictionary matching has been amply researched in computer science (see e.g. [1–5,9,10]). We are concerned with a new dictionary matching data structure enabling *dictionary matching with gaps*. Formally:

Definition 3. *The Dictionary Matching with gaps* (*DMG*) *Problem* is the following:
Preprocess: A dictionary D of total size $|D|$ over alphabet Σ consisting of d gapped patterns $P_1,\ldots,P_d$.
Query: A text T of length n over alphabet Σ.
Output: All locations in T, where a pattern P_i, for $1\le i\le d$ ends.

Note that, $|D|$ is the sum of lengths of all patterns in the dictionary, *not including* the gaps sizes.

Example. Let D be the set of patterns $P_1=a\ c\ \{2,4\}\ d\ d$, $P_2=a\ b\ \{2,4\}\ c\ d$, $P_3=a\ b\ \{2,4\}\ c$. Then, the text $T=c\ d\ a\ b\ a\ b\ e\ b\ c\ d\ a\ c$ has occurrences of P_2 and P_3 ending at locations 10 and 9, respectively.

The *DMG* problem has not been sufficiently studied yet. Haapasalo et al. [13] give an on-line algorithm for the general problem. Their algorithm is based on locating "keywords" of the patterns in the input text, that is, maximal substrings of the patterns that contain only input characters. Matches of prefixes of patterns are collected from the keyword matches, and when a prefix constituting a complete pattern is found, a match is reported. In collecting these partial matches they avoid locating those keyword occurrences that cannot participate in any prefix of a pattern found thus far. Their experiments show that this algorithm scales up well, when the number of patterns increases. They report at most one occurrence for each pattern at each text position. The time required for their algorithm is $O(n \cdot SUF + occ \cdot PREF)$, where n is the size of the text, *SUF* is the maximal number of suffixes of a keyword that are also keywords, and *PREF* denotes the number of occurrences in the text of pattern prefixes ending with a keyword.

Nevertheless, more research on this problem is needed. First, in many applications it is necessary to report all patterns appearances. Moreover, as far as we know [27], these Aho–Corasick automaton based methods fail when applied to real data security, which contain many overlaps in the dictionary patterns, due to overhead in the computation when run on several ports in parallel. Therefore, other methods should be developed and combined with the existing ones in order to design efficient practical solutions.

Model We assume working on RAM model and our space bounds are expressed in words rather than bits. The alphabet size is implicit in our bounds, however, the results are not independent of the alphabet size. The results hold for fixed size alphabets. For bigger alphabets, a factor of $\log|\Sigma|$ should be introduced due to the use of suffix trees. In some cases the query time bound is dominated by other bigger factors (such as $\log|D|$) and therefore, the bound remains unchanged even for unbounded alphabets.

Results In this paper, we indeed suggest other directions for solving the problem. We prove:

Theorem 1. *The dictionary matching with a single gap problem can be solved in:*

1. *Preprocessing time:* $O(d\log d + |D|)$.
   Space: $O(d\log^{\varepsilon} d + |D|)$, *for any small* ε.
   Query time: $O(n(\beta-\alpha)\log\log d\log^2|D| + occ)$, *where occ is the number of patterns found.*
2. *Preprocessing time:* $O(d^2 + |D|)$.
   Space: $O(d^2 + |D|)$.
   Query time: $O(n(\beta-\alpha) + occ)$, *where occ is the number of patterns found.*

Theorem 2. *The dictionary matching with k gaps problem, where* $k \ge 1$, *can be solved in:*

Preprocessing time: $O(|D|\log|D|)$.
Space: $O(|D| + d\frac{(c_1\log d)^k}{k!})$.
Query time: $O((\beta-\alpha)^k(n + \frac{(c_2\log d)^k}{k!}\log\log|D|) + occ)$, *where* $c_1, c_2 > 1$ *are constants and occ is the number of patterns found.*

Remark. One may consider to apply the multiple gap solution also to a single gap, yielding query time of $O(n + \log d\log\log|D| + occ)$, improving over the time achieved by Theorem 1. However, the single gap solution of Theorem 1 supports a negative gap, where the matched subpatterns are overlapping, whereas the solution for k gaps of Theorem 2 cannot deal with negative gaps.

The paper is organized as follows. In Section 2 we describe our basic method for dealing with a single gap and describe both the preprocessing and query algorithms. The query algorithm uses an intersection procedure. In Section 3 we suggest two methods for the intersection procedure, and complete the proof of Theorem 1. In Section 4 we describe the solution for multiple gaps and prove Theorem 2. Section 5 concludes the paper and poses some open problems.

2. Bidirectional suffix trees algorithm

The basic observation used by our algorithm is that if a gapped pattern P_i appears in query text T, then searching to the left from the start position of the gap we should find the reverse of the prefix $P_{i,1}$, and searching to the right from the end position of the gap we should find the suffix $P_{i,2}$. A similar observation was used by Amir et al. [5] to solve the dictionary matching with one mismatch problem. Their problem is different from the *DMG* problem, since they consider a single mismatch while in our problem a gap may consist of several symbols. Moreover, a mismatch symbol may appear at any location of the pattern, while in the *DMG* problem the gap is set at a certain location in each pattern. Nevertheless, we show that their idea can also be adopted to solve the *DMG* problem.

Amir et al. [5] use two suffix trees: one for the concatenation of the dictionary and the other for the reverse of the concatenation of the dictionary. Combining them with set intersection on tree paths, they solved the dictionary matching with one mismatch problem in time $O(n\log^{2.5}|D| + occ)$ where n is the length of the text, $|D|$ is the sum of the lengths of the dictionary patterns, and *occ* is the number of occurrences of patterns in the text. Their preprocessing requires $O(|D|\log|D|)$. We use their framework to design an algorithm for the *DMG* problem.

A naive method is to consider matching the first subpatterns $P_{i,1}$ for all $1 \le i \le d$, and then look for the second subpatterns $P_{i,2}$, $1 \le i \le d$, after the appropriate gap and intersect the occurrences to report the dictionary patterns matchings.

$$P_1 = a\ c\ \{2,4\}\ d\ d$$
$$P_2 = a\ b\ \{2,4\}\ c\ d$$

$$\begin{array}{rcccccccccccc} T = & c & d & e & f & \overleftarrow{a} & \overleftarrow{b} & e & b & \overrightarrow{c} & \overrightarrow{d} & a & c \\ & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \end{array}$$

Fig. 1. Matching $P_{2,2}$ starting from t_9 skipping $\alpha = 2$ symbols backwards and matching $P^R_{2,1}$ to a reverse prefix of T ending at location t_6.

P_1= cbac $\{\alpha,\beta\}$ bcbabbc
P_2= cbac $\{\alpha,\beta\}$ bcbab
P_3= bac $\{\alpha,\beta\}$ bcbabbc
P_4= bac $\{\alpha,\beta\}$ bcbabca
P_5= c $\{\alpha,\beta\}$ bcb
P_6= c $\{\alpha,\beta\}$ bcbabc
P_7= aacbac $\{\alpha,\beta\}$ b
P_8= cbac $\{\alpha,\beta\}$ bcbbbb
P_9= bac $\{\alpha,\beta\}$ bcbb
P_{10}= aacbac $\{\alpha,\beta\}$ bcbabbc

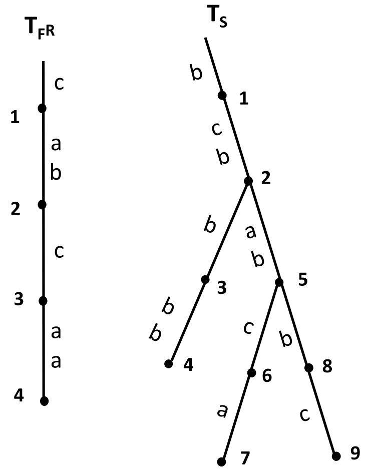


Fig. 2. The suffix trees constructed according to the dictionary. The nodes representing subpatterns are marked by numerical labels.

However, as some of the subpatterns may include other subpatterns, there may be several distinct subpatterns occurring at the same text location, each of different length. Therefore, we need to search for $P_{i,2}$, $1 \le i \le d$, after several gaps, each beginning at the end of a matched prefix $P_{i,1}$, $1 \le i \le d$. To avoid multiple searches we search all subpatterns $P_{i,1}$, $1 \le i \le d$ that *end* at a certain location. Note that in order to find all subpatterns ending at a certain location of the text we need to look for them backwards and find their reverse $P^R_{i,1}$. An example of matching both subpatterns of a pattern appears in Fig. 1.

In the preprocessing stage we concatenate the subpatterns $P_{i,2}$, $1 \le i \le d$ of the dictionary separated by the symbol $\$ \notin \Sigma$ to form a single string S. We repeat the procedure for the subpatterns $P_{i,1}$, $1 \le i \le d$, to form a single string F. We construct a suffix tree T_S of the string S, and another suffix tree T_{F^R} of the string F^R, which is the reverse of the string F. We also mark the trees nodes representing subpatterns. The marked nodes are to be numerically labeled, depending on the way we choose to solve the intersection, as discussed in the succeeding section. Nevertheless, since some patterns may share subpatterns, we do not label the tree nodes by the original subpatterns indices they represent, as was done in [5]. An example of the suffix trees constructed according to a dictionary is depicted at Fig. 2.

Given a query text, we traverse it by inserting suffixes of the text to the T_S suffix tree (i.e., constructing the suffix tree of T using Weiner's algorithm and keeping track of where each suffix of T correspond in T_S, as explained in the next paragraph). When we pass the node of T_S for which the path from the root represents $P_{i,2}$, it implies that this subpattern occurs in the text starting from the beginning of the current text suffix. We then should find whether $P_{i,1}$ also appears in the text within a legal gap from the occurrences of the $P_{i,2}$ subpattern. To this end, we go backward in the text repeatedly skipping all text symbols as required by the current value in the range from α to β, one at a time, and inserting the reversed prefix of the text to T_{F^R}. If a node representing $P_{i,1}$ is encountered, we can output that P_i appears in the text.

Navigation on the suffix trees Assume that we have found the node in T_S where $t_\ell \dots t_n$ resides. We would like to move to the node where $t_{\ell+1} \dots t_n$ appears. Similarly, if we have the node in T_{F^R} where $t_f \dots t_1$ ends, we would like to arrive at the node where $t_{f+1} \dots t_1$ ends. We do that using Weiner's algorithm [28] for suffix tree construction. Let $S' = s'_1 s'_2 \dots s'_n\$$ be a string. The Weiner construction of suffix tree starts with the suffix \$, then adds the suffixes $s'_n\$$, $s'_{n-1}s'_n\$, \dots, s'_1, \dots, s'_n\$$. The total construction time is linear for fixed size alphabets.

Consider the string $t_n t_{n-1} \dots t_1 \% F^R$, where $\% \notin F^R$. The suffix tree T_{F^R} is constructed during the preprocess phase, according to the string F^R, using Weiner algorithm. For every query text T we simply continue the Weiner construction and add $t_1 \% F^R$, $t_2 t_1 \% F^R$, ..., $t_n t_{n-1} \dots t_1 \% F^R$ to this suffix tree. Because $\% \notin F^R$ the total time for Weiner's algorithm to add all suffixes that start at the string T is $O(n)$ and the suffix tree part for F^R is precisely T_{F^R}. This in effect finds us the desired location in T_{F^R} matching T, in total linear time. When the query is over we retrace our steps and remove the query text parts from T_{F^R}.

The case for the tree T_S is similar. Consider the string $t_1 t_2 \dots t_n \% S$, where $\% \notin S$. The suffix tree T_S is constructed during the preprocess phase, according to the string S, using Weiner algorithm. Then the Weiner construction adds $t_n \% S$, $t_{n-1} t_n \% S$, ..., $t_1 t_2 \dots t_n \% S$ to this suffix tree. This also finds all locations we are interested in but the order they are encountered is

```
SINGLE_GAP_DICTIONARY $(T, D)$

Preprocessing:
1   $F = P_{1,1}\$P_{2,1}\$\cdots P_{d,1}$.
2   $S = P_{1,2}\$P_{2,2}\$\cdots P_{d,2}$.
3   $T_S \leftarrow$ a suffix tree of $S$.
4   $T_{F^R} \leftarrow$ a suffix tree of the $F^R$.
5   For every edge $(u,v) \in \{T_S, T_{F^R}\}$ with label $y\$z$, where $y, z \in \Sigma^*$
6       Break $(u,v)$ into $(u,w)$ and $(w,v)$ labeling $(u,w)$ with $y$ and $(w,v)$ with $\$z$.
7   For $i = 1$ to $d$
8       Mark the node representing $P_{i,1}$ in $T_{F^R}$.
9   For $i = 1$ to $d$
10      Mark the node representing $P_{i,2}$ in $T_S$.

Query:
1   For $\ell = \min_i\{|P_{i,1}|\} + \alpha$ to $n$
2       Insert $t_\ell t_{\ell+1} \ldots t_n$ to $T_S$.
3       $h \leftarrow$ node in $T_S$ representing suffix $t_\ell t_{\ell+1} \ldots t_n$.
4       For $f = \ell - \alpha - 1$ to $\ell - \beta - 1$
5           Insert $t_f t_{f-1} \ldots t_1$ to $T_{F^R}$.
6           $g \leftarrow$ node in $T_{F^R}$ representing $t_f t_{f-1} \ldots t_1$.
7           IntersectionQuery $(g, h)$
8               Report appearance for every $P_i$ appearing in the specified intersection.
```

Fig. 3. Dictionary matching with a single gap.

reversed. This fact can be simply circumvented by keeping an array of pointers to all the necessary nodes, and following that array backwards in lockstep with the forward movement on the tree T_{F^R}. We can have a special tag on the new nodes inserted to the suffix trees during the query in order to tell if the final nodes mean an occurrence in the original tree or not.

Observation 1. *At each of the $O(n)$ relevant locations of the text, the algorithm inserts the current suffix of the text to T_S using Weiner's algorithm [28]. For each of the prefixes defined by all $\beta - \alpha + 1$ possible specific gaps we insert its reverse to T_{F^R}. As explained in [5], the navigation on the suffix tree and reverse suffix tree can be done in amortized $O(1)$ time per character insertion. Note that each character is inserted to T_S once and to T_{F^R} $O(\beta - \alpha)$ times.*

Note that several dictionary subpatterns representative nodes may be encountered while traversing the trees. Therefore, we should report the intersection between the subpatterns found from each traversal. We do that by efficient intersection of labels on tree paths, either similar to [5], using range queries on a grid as described in Subsection 3.1 or by using a lookup table, as detailed in Subsection 3.2. The algorithms for both preprocessing and query appear in Fig. 3.

Lemma 1. *The* SINGLE_GAP_DICTIONARY *algorithm solves the DMG problem for a single gap, given that the intersection query is correctly answered.*

Proof. According to Observation 1 the algorithm inserts every possible suffix of the text into T_S and for each suffix all relevant $\beta - \alpha$ prefixes of the text are inserted into T_{F^R}. It, therefore, scans all possible appearance locations in the text. Suppose that in the current iteration the suffix of the text inserted into T_S matches the sequence represented by node h, and h is the farthest such node from the root of T_S. This implies that all $P_{i,2}$'s which are prefixes of the subpattern represented by node h, appear in the current location. Similarly, if the reversed prefix of the text inserted into T_{F^R} matches the sequence represented by node g, and g is the farthest such node from the root of T_{F^R}, it implies that all $P_{i,1}$'s which their reverse are prefixes of the subpattern represented by node g, appear within a relevant gap from the appearance of the located $P_{i,2}$'s in the current location. It remains to match subpatterns located in both trees that originate in the same dictionary pattern. This is the functionality of the intersection query, and assuming it works correctly, the algorithm then reports all occurrences of dictionary patterns in the text, thus it solves the *DMG* problem for a single gap. □

Splitting the text Usually, the input text is very long and arrives on-line. This makes the query algorithm requirement to insert all suffixes of the text unreasonable. Transforming this algorithm into an online algorithm seems a difficult problem. The main difficulty is working with on-line suffix trees construction in a sliding window. While useful constructions based on Ukkonen's [25] and McCreight's [17] suffix trees constructions exist (see [21]), no such results are known for Weiner's suffix tree construction, which our reversed prefixes tree construction depends on.

Nevertheless, we do not need to know the whole text in advance and we can process only separate chunks of it each time. To do this we take $m = \beta - \alpha + \max_i \sum_j |P_{i,j}|$ and split the text twice to pieces of size $2m$: first starting form the beginning of the text and the second starting after m symbols. We then apply the algorithms for a single gap or k-gaps for each of the pieces separately for both text splits. Note that any appearance of a dictionary pattern can still be found by the algorithms on the splitted text.

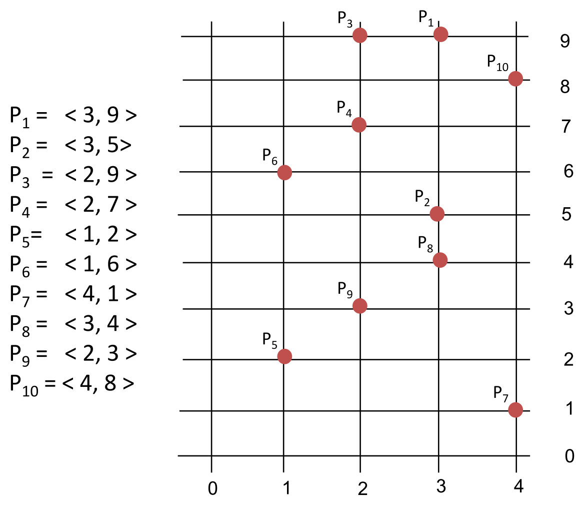


Fig. 4. The grid contains points representing the dictionary pattens.

3. The intersection procedure

In this section we suggest two methods for the intersection of patterns with first subpattern located at T_{F^R} and patterns with second subpattern located at T_S. The first has a better preprocessing time, while the second has faster query time.

3.1. Intersection by range queries on a grid

Following Amir et al. [5], we use range queries on a grid to solve the intersection between two nodes located on the suffix trees. We label the nodes in a different way, due to the fact that in the *DMG* problem it may be the case that some patterns may share subpatterns, so we mark the nodes representing subpatterns regardless of the origin of the subpatterns ending at those nodes.

We use the decomposition of a tree into vertical path for the nodes marking, suggested by [5], though we use it differently. A vertical path is defined as follows.

Definition 4. (See Definition 9.2 in [5].) A *vertical path* of a tree is a tree path (possibly consisting of a single node) where no two nodes on the path are of the same height.

Lemma 9.3 in [5] gives a logarithmic bound on the number of vertical paths in a particular vertical paths decomposition construction, while its proof describes how to construct this decomposition. After performing the decomposition, we can traverse the vertical paths of each tree and label by consecutive order all marked nodes. There are *at most* d marked nodes at each of the suffixes trees.

As the nodes of every vertical path are labeled in consecutive order, the problem of intersection of labels on tree paths can be reduced to range queries on a grid in the following way. Let the first and last mark on the relevant path in T_{F^R} be g', g and, similarly, on the path in T_S let the first and last mark be h', h. Thus, we seek points within the range $[g', g] \times [h', h]$, that is, points $< x, y >$ on the grid where $g' \le x \le g$ and $h' \le y \le h$. These points represent dictionary patterns for which both subpatterns appear at the current check. A range query can be solved using the algorithm of Chan et al. [11].

In order to be able to trace the identity of the patterns from the nodes labeling, we keep the array *point* of size d. Every entry contains two fields *first*, *second*, where $point[i].first = g$ if $P^R_{i,1}$ is represented by the node labeled by g in T_{F^R} and $point[i].second = h$ if $P_{i,2}$ is represented by the node labeled by h in T_S. Every pattern P_i is represented as a point i on a grid of size $d \times d$, denoted by $< point[i].first, point[i].second >$, that is, the x-coordinate is the label of the node representing $P^R_{i,1}$ in T_{F^R}, and the y-coordinate is the label of the node representing $P_{i,2}$ in T_S.

The grid built according to the trees depicted in Fig. 2, can be seen in Fig. 4.

Example. Suppose the query text matched node 3 of T_{F^R} and node 7 in T_S for the trees depicted in Fig. 2, implying before the gap the text prefix ends with "*cbac*" and after the gap the text suffix begins with "*bcbabca*". The path from the root of T_{F^R} to node 3 contains a single vertical path s $1-2-3$. The path from the root of T_S to node 7, according to the numbering of the tree nodes, can be decomposed to vertical paths by $1-2$ and $5-6-7$. Therefore, two grid queries are processed on the grid depicted in Fig. 4. We define ranges $[1, 3] \times [1, 2]$ and $[1, 3] \times [5, 7]$. The former reports P_5 and the latter reports P_2, P_4, P_6.

```
INTERSECTION_BY_GRID ($g$, $h$, $T_{F^R}$, $T_S$, $D$)

Preprocessing:
1  Decompose $T_{F^R}$ into vertical paths.
2  Decompose $T_S$ into vertical paths.
3  Label marked nodes on the vertical paths of $T_{F^R}$ in consecutive order.
4  Label marked nodes on the vertical paths of $T_S$ in consecutive order.
5  For $i = 1$ to $d$
6      The $i$th point $\leftarrow$ <point[$i$].first, point[$i$].second>
7  Preprocess the points for range queries.

IntersectionQuery($g, h$):
1      For every vertical path on the path $p$ from the root to $h$
2        For every vertical path $p'$ on the path from the root to $g$
3          Perform a range query on a grid with the first and last marks of $p$ and $p'$.
4          Report appearance for every $P_i$ where point $i$ appears in the specified range.
```

Fig. 5. The intersection procedure by range queries on a grid.

The procedure of intersection by grid appears in Fig. 5. Lemma 2 gives the correctness of the intersection by grid procedure.

Lemma 2. *Given the suffix trees T_{F^R} and T_S, the dictionary D and the located nodes g, h, the algorithm* INTERSECTION_BY_GRID *reports all dictionary patterns appearing in the current text location.*

Proof. By Lemma 1 we have that nodes g, h respectively represent all $P^R_{i,1}$'s and $P_{i,2}$'s, appearing in the current processed text location within a legal gap from each other. The intersection is required to match nodes from the different trees, representing two subpatterns of the same pattern. This matching process can be reduced to the problem of range queries on a grid with d points, due to the consecutive numbering of the nodes on the vertical paths and the creation of points representing the patterns of the dictionary.

According to Definition 4, the unique path from the root to nodes g, h in both trees, may consist of several vertical paths. We call such vertical paths – relevant paths. Hence, when considering the intersection of labeled nodes on the path from the root till node g in T_{F^R} and the labeled nodes on the path from the root till node h in T_S, we need to check the intersection of all the labeled nodes of every relevant vertical path from T_{F^R} with all the labeled nodes from every relevant vertical path from T_S. Therefore, the procedure applies several range queries as the product of the number of the relevant vertical paths of both trees, thus, reporting all dictionary patterns for which both their subpatterns appear in the text separated by a legal gap. □

Lemma 3. *The intersection between labels of subpatterns appearing at location t_ℓ and labels of reversed subpatterns ending at $t_{\ell-gap-1}$ can be computed in time $O(occ + \log\log d \log^2 |D|)$, where occ is the number of patterns found. The preprocessing requires $O(|D| + d\log d)$ time and $O(|D| + d\log^\varepsilon d)$ space, for any small ε.*

Proof. Since the intersection of subpatterns occurrences is reduced to the problem of range queries on a grid with d points, using [11] each range query requires $O(occ + \log\log d)$, where occ is the number of points within the range, i.e., the number of patterns for which both subpatterns appear in the text separated by a legal gap.

By [5], the number of vertical paths intersecting a path from the root to a certain node is bounded by $\log|D|$, where $|D|$ is the number of nodes in the suffix tree. As we need to perform a range query of every vertical path from the path reaching node g with every vertical path from the path reaching node h, we perform up to $\log^2 |D|$ range queries. All in all, we have $O(occ + \log\log d \log^2 |D|)$ time for finding the patterns occurring at a certain location of the text.

In the preprocessing, we decompose the suffix trees into vertical paths and mark the nodes representing subpatterns in time linear in the size of the trees, which is $O(|D|)$. The preprocessing of d points on a grid for range queries using the Chan et al. algorithm requires $O(d\log d)$ time and space $O(d\log^\varepsilon d + |D|)$, for arbitrary small ε [11]. The lemma then follows. □

Lemma 3 and Observation 1 conclude the proof of the first part of Theorem 1.

3.2. Intersection by lookup table

If a very fast query time is crucial and we are willing to pay in preprocessing time, we can solve the problem of intersection between the appearances of subpatterns on the paths of T_{F^R}, T_S using a lookup table. This can be done, since all nodes in the suffix trees representing subpatterns of the dictionary are known in preprocessing time, and they are the subject of the intersection queries.

The *inter* table is of size $d \times d$, where $inter[g, h]$ refers to the set of all indices i of patterns P_i such that $P^R_{i,1}$ appears on the path from the root of T_{F^R} till the node marked by g and $P_{i,2}$ appears on the path from the root of T_S till the node marked by h. We fill the table using dynamic programming procedure. Consequentially, labeling the nodes representing subpatterns, can be done by any numbering system, guaranteeing that nodes closer to the root are labeled by smaller numbers than nodes farther from the root, such as the BFS order. As in the previous subsection, the algorithm construct the array *point* of size d. Every entry contains the fields *first*, *second*, where $point[i].first = g$ if $P^R_{i,1}$ is represented by the node labeled by g in T_{F^R} and $point[i].second = h$ if $P_{i,2}$ is represented by the node labeled by h in T_S.

Saving at every entry all the relevant pattern indices causes redundancy in case subpatterns include others as their prefix or suffix. In order to save every possible occurrence only once, we save pattern index i only at entry $inter[g, h].index$ where g, h are the nodes respectively representing both subpatterns of P_i in the suffix trees. Note that at most one index can be saved at $inter[g, h].index$ as two patterns are bound to differ by at least one subpattern.

We hereafter prove that besides the *index* field, merely three links are required for every $inter[g, h]$:

1. A link to $inter[g', h]$ in case node h represents the subpattern $P_{i,2}$ and g' is the maximal labeled ancestor of node g, representing $P_{i,1}$. We call this link an *up* link.
2. A link to cell $inter[g, h']$ in case node g represents the subpattern $P_{i,1}$ and h' is the maximal labeled ancestor of node h representing $P_{i,2}$. We call this link a *left* link.
3. A link to cell $[prev^*(g), prev^*(h)]$, where $prev^*(g)$ and $prev^*(h)$ are the closest marked ancestors of the nodes marked by g and h where $inter[prev^*(g), prev^*(h)]$ has a pattern index or non null *up* or *left* link.

Lemma 4 describes the recursive rule for constructing the lookup table.

Lemma 4 *(The recursive rule).*
Let prev(x) be the maximal labeled ancestor of the node labeled by x.

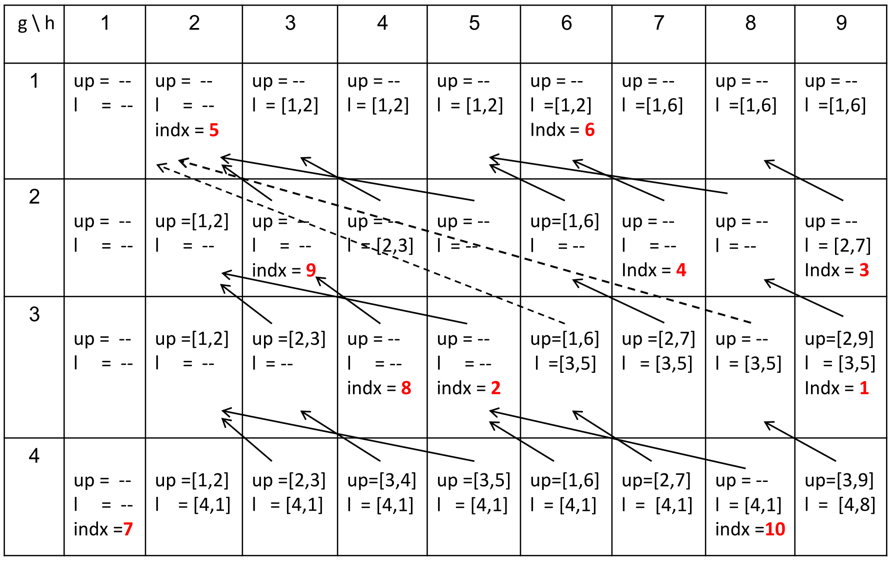
$$inter[g,h].up = \begin{cases} [prev(g), h] & \textbf{\textit{if}}\ inter[prev(g), h].index \neq null \\ inter[prev(g), h].up & \textbf{\textit{otherwise}} \end{cases}$$

$$inter[g,h].left = \begin{cases} [g, prev(h)] & \textbf{\textit{if}}\ inter[g, prev(h)].index \neq null \\ inter[g, prev(h)].left & \textbf{\textit{otherwise}} \end{cases}$$

$$inter[g,h].prev = \begin{cases} [prev(g), prev(h)] & \textbf{\textit{if}}\ inter[prev(g), prev(h)].index \neq null \\ & \quad or\ inter[prev(g), prev(h)].up \neq null \\ & \quad or\ inter[prev(g), prev(h)].left \neq null \\ inter[prev(g), prev(g)].prev & \textbf{\textit{otherwise}} \end{cases}$$

Proof. Every $inter[g, h]$ entry for $1 \le g, h \le d$, has to contain links to all entries containing indices of patterns whose first subpattern is represented by node g or its ancestors in T_{F^R} and its second subpattern is represented by node h or its ancestors in T_S. Assume, without loss of generality, that the marks on the path from the root of T_{F^R} till the node marked by g are $g'_1, g'_2, \ldots, g'_a, g$ and the marks on the path from the root of T_S till the node marked by h are $h'_1, h'_2, \ldots, h'_b, h$. There are four cases of relevant entries, for each we prove the correctness of the recursive rule.

1. Case 1: In case the pair of labels $< g, h >$ represent pattern P_i, then assign $inter[g, h].index$ by i during the preprocessing. No recursion is required.
2. Case 2: Some $\{g'_x\}$ represents the reverse of first subpattern of some dictionary patterns (as theses subpatterns include others as their suffixes), and these patterns share the second subpattern where h represents this second subpattern. In such a case $inter[g, h]$ should be linked to all entries $\{inter[g'_x, h]\}$. Nevertheless, saving a link to the ancestor node with maximal label, is sufficient, since all other relevant patterns can be reached by recursively following *up* links starting from $inter[g'_x, h]$. Therefore, we consider $prev(g)$ as the maximal labeled ancestor and if $inter[prev(g), h]$ includes an index, we assign *up* with $[prev(g), h]$. Otherwise we seek the same ancestor $inter[prev(g), h]$ looked for, for its *up* link, hence we assign *up* with $inter[prev(g), h].up$.
3. Case 3: Some $\{h'_y\}$ represents the second subpattern of some dictionary patterns (as theses subpatterns include others as their prefixes), and these patterns share the first subpattern where g represents the reverse of this first subpattern. Due to similar arguments to those in the previous case, we assign the *left* link either with $[g, prev(h)]$ or with $inter[g, prev(h)].left$.
4. Case 4: Some ancestors of nodes g and h represent both subpatterns of dictionary patterns. Note that it must be ancestors to both nodes as the previous cases dealt with representations of patterns using the nodes g or h themselves. We need to enable $inter[g, h]$ to follow all such entries, to this end we look for the closest such ancestors. We check whether $inter[prev(g), prev(h)]$ includes an index or an *up* or *left* link. If it does, link the *prev* link to $[prev(g), prev(h)]$. If it does not, it means that no pattern is represented either by node $prev(g)$ and a node on the path from the root of T_S till node $prev(h)$ or by a node on the path from the root of T_{S^R} till node $prev(g)$ and the node $prev(h)$. Therefore we can step back in both paths of the trees to $prev(prev(g))$ and $prev(prev(h))$. Such a scenario can repeat, yet $inter[prev(g), prev(h)].prev$



g \ h	1	2	3	4	5	6	7	8	9
1	up = -- l = --	up = -- l = -- indx = 5	up = -- l = [1,2]	up = -- l = [1,2]	up = -- l = [1,2]	up = -- l =[1,2] Indx = 6	up = -- l =[1,6]	up = -- l =[1,6]	up = -- l =[1,6]
2	up = -- l = --	up =[1,2] l = --	up = -- l = -- indx = 9	up = -- l = [2,3]	up = -- l = --	up=[1,6] l = --	up = -- l = -- Indx = 4	up = -- l = --	up = -- l = [2,7] Indx = 3
3	up = -- l = --	up =[1,2] l = --	up =[2,3] l = --	up = -- l = -- indx = 8	up = -- l = -- indx = 2	up=[1,6] l =[3,5]	up =[2,7] l = [3,5]	up = -- l = [3,5]	up=[2,9] l = [3,5] Indx = 1
4	up = -- l = -- indx =7	up =[1,2] l = [4,1]	up =[2,3] l = [4,1]	up=[3,4] l = [4,1]	up =[3,5] l = [4,1]	up=[1,6] l = [4,1]	up=[2,7] l = [4,1]	up = -- l = [4,1] indx =10	up=[3,9] l = [4,8]

Fig. 6. The lookup table built according to the trees depicted in Fig. 2. The arrows represent the *prev* links.

which is $[prev^*(g), prev^*(h)]$, was already computed, due to the numbering system, and the relevant information is bound to appear at that entry, so we can follow it by assigning $prev = inter[prev(g), prev(h)].prev$. □

Example. An example of using the recursive rule, filling the *inter* table can be seen in Fig. 6 for the trees depicted in Fig. 2 and the following patterns points. $P_1 =< 3, 9 >$, $P_2 =< 3, 5 >$, $P_3 =< 2, 9 >$, $P_4 =< 2, 7 >$, $P_5 =< 1, 2 >$, $P_6 =< 1, 6 >$, $P_7 =< 4, 1 >$, $P_8 =< 3, 4 >$, $P_9 =< 2, 3 >$, $P_{10} =< 4, 8 >$. (Although the numbering of marked nodes are not in BFS order, the numbering used maintains the requirement of a predecessor having a higher label than its ancestor.) Note the dashed *prev* arrows emanating from $inter[3, 8]$ and from $inter[3, 6]$ to $inter[1, 2]$. Due to the lack of information in $inter[2, 5]$, which is the entry of the immediate ancestors of $[3, 8]$ and $[3, 6]$, their *prev* links are directed to the entry $inter[2, 5].prev$ refers to.

Answering LookUp Queries A query of a node g from T_{F^R} and node h from T_S is answered by consulting entry $inter[g, h]$. We output $inter[g, h].index$ if exists, which means there is a pattern whose first subpattern is represented by node g and its second subpattern is represented by node h. In order to report all relevant patterns, that their subpatterns are represented by g or its ancestors and by h or its ancestors in the suffix trees, we follow the links saved at the current entry, as detailed in the procedure appearing at Fig. 7. The correctness of the intersection by lookUp table is given in Lemma 5.

Lemma 5. *Given the suffix trees T_{F^R} and T_S, the dictionary D and the located nodes g, h, the algorithm* INTERSECTION_BY_LOOKUP_TABLE *reports all dictionary patterns appearing in the current text location.*

Proof. By Lemma 1 we have that nodes g, h respectively represent all $P^R_{i,1}$s and $P_{i,2}$s, appearing in the current processed text location within a legal gap between them. In particular, we can say that h represents the longest $P_{i,2}$ appearing at the beginning of the current text suffix. Similarly, g represents the longest reversed $P_{i,1}$, appearing at the ending of the current reversed text prefix. Using the LookUp table, when given the intersection query, we check whether there exists a pattern that its subpatterns are represented by nodes g and h or their ancestors in the suffix trees. Suppose pattern P_i is represented by the nodes labeled by u, v. In case $u = g$ or $u = prev(g)$ or $v = h$ or $v = prev(h)$ Lemma 4 assures us we report P_i, by following the links emanating from $inter[g, h]$.

For other cases, suppose that there are nodes $g_1, \ldots, g_a$ between u and g, on the path from the root of T_{F^R} (there are some subpatterns that are the suffixes of the subpattern represented by g, yet they are longer than the subpattern represented by u) and suppose there are nodes $h_1, \ldots, h_b$ between v and h, on the path from the root of T_S (there are some subpatterns that are prefixes of the subpattern represented by h, yet they are longer than the subpattern represented by v). Let $inter[g, h].prev^*$ denote recursively following the *prev* link. By Lemma 4 and by the fact that all g_j are ancestors of g and predecessors of u and all h_f are ancestors of h and predecessors of v, we have three cases:

1. Case 1: $inter[g, h].prev^* = [g_j, h_f]$ for $1 \le j \le a$, $1 \le f \le b$, and $inter[g_j, h_f].prev = [u, v]$.
2. Case 2: $inter[g, h].prev^* = [g_j, v]$ for $1 \le j \le a$, and $inter[g_j, v].up = [u, v]$.
3. Case 2: $inter[g, h].prev^* = [u, h_f]$ for $1 \le f \le b$, and $inter[u, h_f].left = [u, v]$.

```
INTERSECTION_BY_LOOKUP_TABLE ($T_{F^R}$, $T_S$, $D$, $g$, $h$)

Preprocessing:
1   Label the marked nodes representing $P_{i,1}$s in $T_{F^R}$ using BFS order.
2   Label the marked the nodes representing $P_{i,2}$s in $T_S$ using BFS order.
3   For $i = 1$ to $d$
4       The $i$th point ← <point[$i$].first, point[$i$].second>
5   For $i = 1$ to $d$, where the $i$th point is $< g, h >$
6       $inter[g, h].index = i$.
7   For $g = 1$ to highest label in $T_{F^R}$
8       For $h = 1$ to highest label in $T_S$
9           Fill $inter[g, h]$ according to Lemma 4.

IntersectionQuery($g, h$):
1       If $inter[g, h].index \neq null$
2           Output $inter[g, h].index$.

3       If $inter[g, h].prev \neq null$
4           Let $[g', h'] \leftarrow inter[g, h].prev$.
5           IntersectionQuery( $g', h'$ )

6       Let $G \leftarrow g$.
7       While ($inter[g, h].up \neq null$ ).
8           Let $[g', h] \leftarrow inter[g, h].up$.
9           Output $inter[g', h].index$.
10          $g \leftarrow g'$.

11      While ($inter[G, h].left \neq null$).
12          Let $[G, h'] \leftarrow inter[G, h].left$.
13          Output $inter[G, h'].index$.
14          $h \leftarrow h'$.
```

Fig. 7. The intersection procedure by LookUp table.

By recursively following *prev* links and the links emanating from the *prev* entries, as detailed in Fig. 7, we are bound to reach entry $inter[u, v]$ to find index i and report it. Thus, all indices of patterns which appear in the table and both of their subpatterns are represented by ancestors of the nodes labeled by g, h are reported. □

Lemma 6 gives the preprocessing and query time guarantee.

Lemma 6. *Preprocessing to build the inter table requires $O(|D| + d^2)$ time and $O(d^2)$ space. Using the inter table, the intersection between labels of subpatterns appearing at location t_ℓ and labels of the reversed subpatterns ending at $t_{\ell-gap-1}$ can be computed in time $O(occ)$, where occ is the number of patterns found.*

Proof. The preprocess requires labeling both suffix trees in BFS order all in $O(|D|)$. Filling d $inter[g, h]$ entries with the index of the pattern which nodes g, h represent can be done in at most $O(d^2)$. Filling each of the d^2 entries of the table $inter[g, h]$ can be performed in $O(1)$ due to Lemma 4.

The query procedure is based on following links and reporting indices found. Every step of following an *up* or *left* link implies that another pattern is reported, as those links connect two subpatterns including one another, where both should be reported. The *prev* link either directs us to an entry including a pattern index, needs to be reported or it directs us to an entry representing a pattern or containing an *up* or *left* links. Hence, by following at most two links we encounter an index needed to be reported. Consequently, the time of following links is attributed to the size of the output.

The lookup table has d^2 entries, each consists of 4 fields, yielding $O(d^2)$ space requirement. □

As the Lookup Query procedure can replace the intersection procedure, which is executed $n(\beta - \alpha)$ times, Observation 1, Lemmas 5 and 6 conclude the proof of the second part of Theorem 1.

4. Algorithm for dictionary with a few gaps

In this section we study the more general case where each pattern of the dictionary may consist of k equally bounded gaps and, therefore, $k + 1$ subpatterns. Hence, every dictionary pattern is of the form

$$P_{i,1}\ \{\alpha, \beta\}\ P_{i,2}\ \{\alpha, \beta\} \ldots \{\alpha, \beta\}\ P_{i,k+1}.$$

A simple solution is a generalization of [24] using dk van Emde Boas structures, each preserving appearances of a $P_{i,j}$. We construct the Aho Corasick automaton, built upon all the dictionary subpatterns and go over the text using the

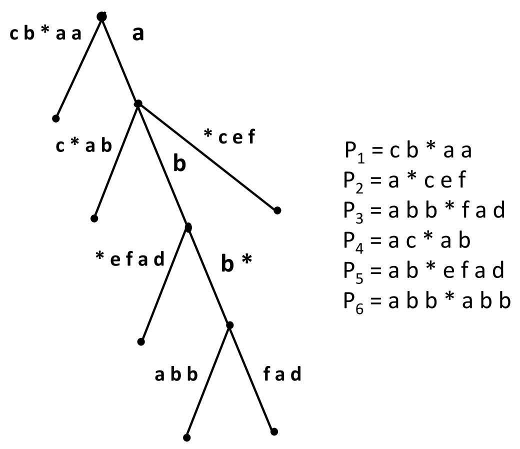


Fig. 8. A dictionary and its generalized suffix tree.

automaton. On detecting appearance of $P_{i,j}$ at location t_ℓ, we search the $(i, j-1)$-th structure for appearance of the corresponding $P_{i,j-1}$, ending at location $t_{\ell'}$, where $t_\ell - |P_{i,j}| - \beta \le t_{\ell'} \le t_\ell - |P_{i,j}| - \alpha$. In case such a $P_{i,j-1}$ exists in the structure, we consider the current $P_{i,j}$ occurrence as valid and insert it to the (i, j)-th structure. Naturally, when a $P_{i,k+1}$ subpattern occurrence is validated, we report an occurrence of the pattern P_i. The time complexity of this solution is $O(n + |D| + socc \cdot ovr \cdot \lg\lg(kn))$, where $socc$ is the total number of subpatterns occurrences in the text and ovr is the maximal overlap in the dictionary (i.e. a subpattern that appear in several patterns in the dictionary), since the construction of the automaton is linear in $|D|$ and the traversal over the text is linear in n and for every subpattern occurrence we perform a query and maybe an insertion to the data structure (or several insertions to several data structures in case of overlaps in the dictionary) each requiring $\lg\lg(kn)$ time. It should be noted, however, that in case there are a few subpatterns occurrences, or there is no overlap between the subpatterns, this simple algorithm is rather efficient, yet its worst case may be $O(|D| + dn \lg\lg(kn))$. We therefore seek other solutions.

4.1. Suffix tree with don't cares solution

Cole et al. [10] solved the problems of dictionary matching and indexing with k don't cares. Since a gap may be considered as a consecutive sequence of don't cares, their solution can solve the *DMkG* problem if k is defined as the sum of the gaps lengths. However, their complexity is exponential in k (the number of don't cares) and the sum of gaps lengths might be very large, hence, such a solution is usually infeasible. Nevertheless, we show that it is possible to encode each gap as a single don't care and thus substantially reduce the cost.

Suppose for simplicity, that all gaps in the dictionary are of fixed size g. The strategy is to build a generalized suffix tree of all dictionary patterns where each gap is replaced by a new symbol, $*$, not included in the dictionary alphabet. Then, consider every suffix of the text as a query q and search it in the suffix tree. In case $*$ is encountered, consider it as matching to the current symbol in the text query. Yet, as we deal with gaps of size g instead of matching the don't care with a single character (as done in [10]), we continue matching $g-1$ more symbols ahead in the text. This search of the query is not sufficient as Observation 2 points out.

Observation 2. *Suppose the path p in the suffix tree S matched to query q is composed of nodes $v_1, v_2, \ldots, v_i, \ldots$ and the label of node v_i is $*$, then every node v_j for $1 < j < i$ that has a branch labeled by $*$ must be considered as well for the continuation of the traversal.*

An example for the observation is depicted in Fig. 8. Suppose the gap denoted by * is of length three, and the text query is $a\ b\ b\ e\ c\ e\ f\ a\ d$, then the output should be $\{P_2, P_3, P_5\}$.

Time complexity As in [10], the time required to search a query q in the dictionary suffix tree is $O(|q|^{k+1}\Sigma^k)$, as every $*$ reached in the suffix tree, implies possibly $|q|$ more branches needed to be searched, due to Observation 2. A procedure that can recursively repeat k times. Every new search from $*$ symbol implies up to Σ subtrees rooted at $*$ that need to be searched. In order to improve the time complexity, Cole et al. [10] suggested using centroid path decomposition of the suffix tree. The query is fully searched only once on centroid paths. All the trees rooted by $*$ branching from the centroid path are searched by LCP queries in time $O(\log\log d)$ (see Lemma 17 in [10]), instead of matching nodes on the path with the current symbols in the query. Since many subtrees may need to be searched, the $*$ rooted subtrees hanging from the relevant centroid path are grouped in hierarchy. The hierarchy takes into account the number of leaves of a subtree, where a subtree containing many leaves is included in a small number of trees groups, and is organized so that the total number of trees group needed to be searched at every query is $O(\log d)$.

The preprocessing includes lexicographically ordering the leaves of the dictionary suffix tree S, building Longest Common Ancestor (LCA) structure for S and a Measured Ancestor structure, all done in time linear to D. They also build LCP structures for the trees group using LCA structure and Van Emde Boas data structure. The preprocess requires $|D|\log|D|$ time and space. Query time is $O(n + \frac{(c_2 \log d)^k}{k!} \log\log|D| + occ)$, where $c_1, c_2 > 1$ are constants.

Space complexity As in [10], the size of the data structure required is $O(|D| + d\frac{(c_1 \log d)^k}{k!})$.

This solution indeed deals with multiple gaps, however, all gaps should be of the same length. In our case, however, each gap is bounded by thresholds. Therefore, every possible gap length requires a different query in order to avoid the problem of overlapping patterns including different gaps. The same suffix tree is to be traversed for every gap length, as the gap is always denoted by a single don't care. Nevertheless, performing $\beta - \alpha$ queries, each setting a certain length to the gap for the entire traversal of the tree, may not yield a complete output. It may be the case that $P_i = P_{i,1}\{g_1\}P_{i,2}\{g_2\}P_{i,3}$, that is a pattern including three subpatterns, appears in the text where the gap between the first and second subpatterns is of length g_1 and the gap between the second and third subpatterns is of length g_2 where, without loss of generality, $\alpha \le g_1 < g_2 \le \beta$. The multiple gap algorithm setting $g = g_1$ finds only a partial match, and when setting $g = g_2$, a different partial match is found, so no match is reported. To avoid these false negative reports, we need to perform a query for every possible combination of gaps lengths for the k gaps appearing in each tree path, yielding $(\beta - \alpha)^k$ iterations of the multiple gaps algorithm. This concludes the proof of Theorem 2.

5. Conclusions and open problems

We showed that combinatorial string methods other than Aho–Corasick automaton can be applied to the *DMG* problem yielding efficient algorithms. In this paper we focused on solving *DMG*, where only a few gaps exist in all patterns in the dictionary. We showed an efficient solution for the case of a single gap that can handle also negative gap, i.e., enable overlaps between occurrences of dictionary subpatterns. We also described an efficient solution for the case of more than one gap that does not support negative gaps. Since the complexity of this solution grows exponentially in the number of gaps, it can be used when there are only a few gaps. We also relaxed the problem so that all patterns in the dictionary have the same gap bounds. It is an interesting open problem to study the general problem without these relaxations.

References

[1] A.V. Aho, M.J. Corasick, Efficient string matching: an aid to bibliographic search, Commun. ACM 18 (6) (1975) 333–340.
[2] A. Amir, G. Calinescu, Alphabet independent and dictionary scaled matching, J. Algorithms 36 (2000) 34–62.
[3] A. Amir, M. Farach, R. Giancarlo, Z. Galil, K. Park, Dynamic dictionary matching, J. Comput. System Sci. 49 (2) (1994) 208–222.
[4] A. Amir, M. Farach, R.M. Idury, J.A. La Poutré, A.A. Schäffer, Improved dynamic dictionary matching, Inform. and Comput. 119 (2) (1995) 258–282.
[5] A. Amir, D. Keselman, G. Landau, M. Lewenstein, N. Lewenstein, M. Rodeh, Indexing and dictionary matching with one error, J. Algorithms 37 (2000) 309–325, preliminary version appeared in WADS 99.
[6] P. Bille, I.L. Gørtz, H.W. Vildhøj, D.K. Wind, String matching with variable length gaps, Theoret. Comput. Sci. (443) (2012) 25–34.
[7] P. Bille, M. Thorup, Faster regular expression matching, in: Proc. 36th International Colloquium on Automata, Languages and Programming, ICALP, in: LNCS, vol. 5555, Springer, 2009, pp. 171–182.
[8] P. Bille, M. Thorup, Regular expression matching with multi-strings and intervals, in: Proc. 21st Annual ACM–SIAM Symposium on Discrete Algorithms, SODA, 2010, pp. 1297–1308.
[9] G.S. Brodal, L. Gasieniec, Approximate dictionary queries, in: Proc. 7th Annual Symposium on Combinatorial Pattern Matching, CPM 96, in: LNCS, vol. 1075, Springer, 1996, pp. 65–74.
[10] R. Cole, L. Gottlieb, M. Lewenstein, Dictionary matching and indexing with errors and don't cares, in: Proc. 36th Annual ACM Symposium on the Theory of Computing, STOC, ACM Press, 2004, pp. 91–100.
[11] T.M. Chan, K.G. Larsen, M. Pătraşcu, Orthogonal range searching on the ram, revisited, in: Proc. 27th ACM Symposium on Computational Geometry, SoCG, 2011, pp. 1–10.
[12] K. Fredriksson, S. Grabowski, Efficient algorithms for pattern matching with general gaps, character classes and transposition invariance, Inf. Retr. 11 (4) (2008) 338–349.
[13] T. Haapasalo, P. Silvasti, S. Sippu, E.S. Soininen, Online dictionary matching with variable-length gaps, in: Proc. 10th Intl. Symp. on Experimental Algorithms, SEA, in: LNCS, vol. 6630, Springer, 2011, pp. 76–87.
[14] K. Hofmann, P. Bucher, L. Falquet, A. Bairoch, The PROSITE database, Nucleic Acids Res. (27) (1999) 215–219.
[15] M. Krishnamurthy, E.S. Seagren, R. Alder, A.W. Bayles, J. Burke, S. Carter, E. Faskha, How to Cheat at Securing Linux, Syngress Publishing, Inc., Elsevier, Inc., 30 Corporate Dr., Burlington, MA 01803, 2008, e-edition: http://www.sciencedirect.com/science/book/9781597492072.
[16] G. Kucherov, M. Rusinowitch, Matching a set of strings with variable length don't cares, Theoret. Comput. Sci. 178 (1–2) (1997) 129–154.
[17] E.M. McCreight, A space-economical suffix tree construction algorithm, J. ACM 23 (2) (1976) 262–272.
[18] M. Morgante, A. Policriti, N. Vitacolonna, A. Zuccolo, Structured motifs search, J. Comput. Biol. 12 (8) (2005) 1065–1082.
[19] G. Myers, A four-Russian algorithm for regular expression pattern matching, J. ACM 39 (2) (1992) 430–448.
[20] G. Myers, G. Mehldau, A system for pattern matching applications on biosequences, Comput. Appl. Biosci. 9 (3) (1993) 299–314.
[21] J.C. Naa, A. Apostolico, C.S. Iliopoulos, K. Park, Truncated suffix trees and their application to data compression, Theoret. Comput. Sci. 304 (3) (2003) 87–101.
[22] G. Navarro, M. Raffinot, Fast and simple character classes and bounded gaps pattern matching, with applications to protein searching, J. Comput. Biol. 10 (6) (2003) 903–923.
[23] G. Navarro, M. Raffinot, New techniques for regular expression searching, Algorithmica 41 (2) (2004) 89–116.
[24] S. Rahman, C.S. Iliopoulos, I. Lee, M. Mohamed, W.F. Smyth, Finding patterns with variable length gaps or don't cares, in: Proc. 12th Annual Conference on Computing and Combinatorics, COCOON, 2006, pp. 146–155.
[25] E. Ukkonen, On-line construction of suffix trees, Algorithmica 14 (3) (1995) 249–260.

[26] P. van Emde Boas, Preserving order in a forest in less than logarithmic time, in: Proceedings of the 16th Annual Symposium on Foundations of Computer Science, 1975, pp. 75–84.
[27] Verint, Packet intrusion detection, Personal communication, 2013.
[28] P. Weiner, Linear pattern matching algorithm, in: Proc. 14 IEEE Symposium on Switching and Automata Theory, 1973, pp. 1–11.
[29] M. Zhang, Y. Zhang, L. Hu, A faster algorithm for matching a set of patterns with variable length don't cares, Inform. Process. Lett. 110 (2010) 216–220.