Available online at www.sciencedirect.com
ScienceDirect
Journal of Discrete Algorithms 5 (2007) 330–340
JOURNAL OF DISCRETE ALGORITHMS
www.elsevier.com/locate/jda

Character sets of strings

Gilles Didier [a], Thomas Schmidt [b], Jens Stoye [c,*], Dekel Tsur [d]

[a] *Centro de Modelamiento Matematico CNRS UMR 2071 Santiago de Chile, Chile*
[b] *International NRW Graduate School in Bioinformatics and Genome Research, Center of Biotechnology, Universität Bielefeld, 33594 Bielefeld, Germany*
[c] *Technische Fakultät, Universität Bielefeld, 33594 Bielefeld, Germany*
[d] *Computer Science Department, Ben-Gurion University, Israel*

Received 19 December 2005; accepted 20 March 2006

Available online 10 August 2006

Abstract

Given a string S over a finite alphabet Σ, the *character set* (also called the *fingerprint*) of a substring S' of S is the subset $C \subseteq \Sigma$ of the symbols occurring in S'. The study of the character sets of all the substrings of a given string (or a given collection of strings) appears in several domains such as rule induction for natural language processing or comparative genomics. Several computational problems concerning the character sets of a string arise from these applications, especially:

(1) Output all the maximal locations of substrings having a given character set.
(2) Output for each character set C occurring in a given string (or a given collection of strings) all the maximal locations of C.

Denoting by n the total length of the considered string or collection of strings, we solve the first problem in $\Theta(n)$ time using $\Theta(n)$ space. We present two algorithms solving the second problem. The first one runs in $\Theta(n^2)$ time using $\Theta(n)$ space. The second algorithm has $\Theta(n|\Sigma|\log|\Sigma|)$ time and $\Theta(n)$ space complexity and is an adaptation of an algorithm by Amir et al. [A. Amir, A. Apostolico, G.M. Landau, G. Satta, Efficient text fingerprinting via Parikh mapping, J. Discrete Algorithms 26 (2003) 1–13].
© 2006 Elsevier B.V. All rights reserved.

Keywords: Character sets; Fingerprints; Combinatorial algorithms on words; Comparative genomics; Natural language processing

1. Introduction

The study of local composition of a given string S can be formalized by considering the subsets of symbols occurring in all the substrings of S. We define the character set of a given substring S' as the set of all and only those symbols having at least one occurrence in S'. The present work is devoted to answer the following two queries:

Query 1: Given a string S and a set of characters C, find all maximal substrings of S that have C as their character set.

* Corresponding author.
E-mail address: stoye@techfak.uni-bielefeld.de (J. Stoye).

1570-8667/$ – see front matter © 2006 Elsevier B.V. All rights reserved.
doi:10.1016/j.jda.2006.03.021

Query 2: Given a string S, find for each character set C that occurs at least once in S, all maximal substrings of S that have C as their character set.

Both queries can easily be extended to search in a collection of strings instead of a single string.

Previous work about this topic was motivated by two main applications: natural language processing and comparative genomics. Query 2 was not explicitly raised in these applications, but many of the problems considered are strongly related to it.

In comparative genomics, one is interested in identifying, among the genomes of two or more organisms, the regions composed of the same set of genes (sometimes called clusters), disregarding the number and order of physical occurrence of the genes of this cluster [1,2]. Here the considered strings are the ordered sequences in which the genes of the organisms appear in the DNA. Formalized with character sets, the problem becomes that of "finding all the character sets appearing in all the strings of a given collection". A first solution of this particular problem was given by Uno and Yagiura [3]. This solution is limited to the case of a collection of two strings which are both permutations of the same set of n elements (i.e. each symbol appears only once in each string). They presented an optimal $O(n + K)$ time algorithm where K is the effective number of common contiguous subsets between the two permutations (i.e. the common character sets), which are called common intervals. Heber and Stoye [4] extended this result to common intervals of $k \geqslant 2$ permutations. Unfortunately the simplicity of the model makes it unsuitable to be used on real genomic data because of the presence of multiple copies of the same gene in a genome (paralogous genes). A more general algorithm, presented by Schmidt and Stoye in [5], solves the question of finding all the common intervals of two strings (not only permutations) in $\Theta(n^2)$ time using $\Theta(n^2)$ space. In Section 4 of this paper we describe an alphabet independent algorithm solving the same question in $\Theta(n^2)$ time and $\Theta(n)$ space.

In the field of natural language processing, some techniques of automatic induction of lexical classification rules may rely on the analysis of the statistics of the character sets. The symbols considered here are the lexical categories (noun, adjective, verb, ...). We refer to [6,7] for more details. For this type of applications, Amir et al. [7] developed a general algorithm applicable both to Queries 1 and 2, using an efficient encoding of the character sets (fingerprints). After an $O(n|\Sigma| \log n \log |\Sigma|)$ time preprocessing and saving a tree structure of size $O(n|\Sigma|)$ in memory, it allows to answer Query 1 in $O(|\Sigma| \log n)$ time. In Section 3 of this paper we present another way to solve Query 1 that requires linear time and space (with respect to the length of the sequence) which can be an efficient alternative to the solution of [7] when the alphabet has a high cardinality. Using a direct adaptation of the preprocessing from [7], Query 2 can be answered in $O(n|\Sigma| \log n \log |\Sigma|)$ time and $\Theta(n)$ space. In Section 5 of this paper we improve this algorithm to require $\Theta(n|\Sigma| \log |\Sigma|)$ time and $\Theta(n)$ space.

This paper is an extended version of two conference papers containing weaker results, [8] and [5].

2. Basic definitions and statement of the problems

Given a string S over the finite alphabet $\Sigma = \{1, \ldots, |\Sigma|\}$, $|S|$ denotes the length of S, $S[i]$ refers to the ith character of S, and $S[i, j]$ is the substring of S that starts with the ith and ends with the jth character of S. For convenience, we will always assume for a string S that $S[0] = S[|S| + 1] = \delta$, where the special character δ does not occur elsewhere in S, so that border effects can be ignored when speaking of the left or right neighbor of a character in S.

Definition 1 *(character set).* Given a string S, the *character set* of a substring $S[i, j]$ is defined by

$$\mathcal{CS}(S[i, j]) = \{S[k] \mid i \leqslant k \leqslant j\}.$$

Definition 2 *(location).* Given a string S over an alphabet Σ and a subset $C \subseteq \Sigma$, the interval $[i, j]$ is a *location* of C in S if and only if $\mathcal{CS}(S[i, j]) = C$.

Definition 3 *(maximal).* A substring $S[i, j]$ of S is *left-maximal* if $S[i - 1] \notin \mathcal{CS}(S[i, j])$, it is *right-maximal* if $S[j + 1] \notin \mathcal{CS}(S[i, j])$, and it is *maximal* if it is both left- and right-maximal.

The locations of two maximal substrings having the same character set cannot overlap each other. An interval or a location $[i, j]$ is called maximal if the substring $S[i, j]$ is maximal.

Queries 1 and 2 of the Introduction can now be expressed in terms of maximal locations by the following two problems:

Problem 1. Given a string S over Σ and a character set $C \subseteq \Sigma$, output all the maximal locations of C in S.

Problem 2. Given a string S over Σ, for each character set $C \subseteq \Sigma$ with at least one location in S, output all the locations of C in S.

As pointed out in the Introduction, many related queries about character sets like "output all the character sets with at least some fixed number of locations in S" can be seen as particular instances of Problem 2. All that is needed are appropriate filters of the output.

Another variation of the problem is the case where not only one string S is given, but a collection of strings $\mathcal{S} = (S_1, S_2, \ldots, S_k)$. This can be reduced to the above problems by concatenating all the strings of the collection and inserting the special character δ between them. By convention, the positions of the special character will not be considered as positions of the string.

3. Finding all the maximal locations of a given character set

We start with a simple algorithm that solves Problem 1, i.e. given a string S of length n over the alphabet Σ and a character set $C \subseteq \Sigma$, it outputs all the maximal locations of C in S. Its complexity is linear with respect to the length of S both in time and space.

Definition 4 *(C-maximal).* Given a character set C, a substring $S[i, j]$ of a string S is *C-maximal* if $\mathcal{CS}(S[i, j]) \subseteq C$, $S[i-1] \notin C$ and $S[j+1] \notin C$.

A C-maximal substring is maximal.

Remark 1. An interval $[i, j]$ is a maximal location of a given character set C if and only if the substring $S[i, j]$ is C-maximal and $|\mathcal{CS}(S[i, j])| = |C|$.

The algorithm solving Problem 1 is now straightforward. We assume the character set C is stored in a bit array of size $|\Sigma|$. Starting at the first position, we read the characters of S in a left-to-right fashion up to position i, the first position of an occurrence of a symbol of C, i.e. the start of the first C-maximal substring of S. Next, to get the right boundary j of this C-maximal substring, we move further, as long as we find symbols of C and, e.g. using again a bit array of size $|\Sigma|$, count the number of different symbols encountered. With the preceding remark, if this last number is equal to $|C|$, $[i, j]$ is a maximal location of C and we output this interval. We iterate this process starting at position $j+1$ until the end of S is reached.

Theorem 1. *Given a string S of length n (or a collection $\mathcal{S}$ of strings of overall length n) and a character set C, all maximal locations of C in S (respectively $\mathcal{S}$) can be found in $\Theta(n)$ time and space.*

4. Finding the locations of all character sets—first algorithm

Now we consider Problem 2. The general outline of our first algorithm for solving this problem is that for each position i of the sequence S, all the locations of character sets of substrings starting at i are reported. In the following we will assume i to be fixed to some value between 1 and $n = |S|$.

4.1. Maximal substrings starting at a given position—ranks of symbols

By definition, a (left-) maximal substring starting at a position i of S cannot include the first occurrence of $S[i-1]$ following i. Denoting by L the list of symbols occurring between i (included) and the first occurrence of $S[i-1]$ following i (excluded), ordered according to their first occurrences, the character sets of maximal substrings starting at i are the sets $\{L[1], \ldots, L[m]\}$ for all integers $m \in \{1, \ldots, |L|\}$.

Definition 5 *(rank).* We define for each symbol $x \in \Sigma$ its *rank* as the index of x in L if x occurs in L, and as $+\infty$ if not.

The rank of a character x will be stored in the entry $\textsc{Rank}[x]$ of a table $\textsc{Rank}$ of size $|\Sigma|$. We fix always $\textsc{Rank}[\delta]$ to $+\infty$.

Definition 6 *(complete).* A substring $S[a,b]$ is *complete* if and only if

$$\{\textsc{Rank}[S[l]] \mid l \in [a,b]\} = \Big\{1,\ldots,\max_{l\in[a,b]}\{\textsc{Rank}[S[l]]\}\Big\}.$$

An interval $[a,b]$ is complete if $S[a,b]$ is complete.

Remark 2. A substring has the same character set as a substring starting at i if and only if it is complete.

4.2. Related maximal intervals

The number of maximal substrings grows in general as a quadratic function of the length of the sequence. We will see in this subsection that, among these, at most $|S|$ maximal substrings can be complete.

Definition 7 *(rank interval).* For each position k of S, the *rank interval* of k, denoted $\textsc{Int}[k]$, is the interval $[a,b]$ containing k such that the substring $S[a,b]$ satisfies:

(1) $\textsc{Rank}[S[l]] \leqslant \textsc{Rank}[S[k]]$ for all $l \in [a,b]$;
(2) $\textsc{Rank}[S[a-1]] > \textsc{Rank}[S[k]]$ and $\textsc{Rank}[S[b+1]] > \textsc{Rank}[S[k]]$.

Rank intervals are hierarchically nested:

Lemma 1. *If* $\textsc{Int}[k]$ *and* $\textsc{Int}[k']$ *are two rank intervals, then* $\textsc{Int}[k] \cap \textsc{Int}[k'] \in \{\emptyset, \textsc{Int}[k], \textsc{Int}[k']\}$. *More precisely, if* $\textsc{Int}[k] \cap \textsc{Int}[k'] \neq \emptyset$ *we have*:

(1) $\textsc{Int}[k] \subseteq \textsc{Int}[k']$ *if and only if* $\textsc{Rank}[S[k]] \leqslant \textsc{Rank}[S[k']]$;
(2) $\textsc{Int}[k] \supseteq \textsc{Int}[k']$ *if and only if* $\textsc{Rank}[S[k]] \geqslant \textsc{Rank}[S[k']]$;
(3) $\textsc{Int}[k] = \textsc{Int}[k']$ *if and only if* $\textsc{Rank}[S[k]] = \textsc{Rank}[S[k']]$.

Proof. We just prove the first assertion, the other ones being direct consequences.

($\Rightarrow$) By definition, the maximum rank over $\textsc{Int}[k']$ is reached in position k'. If $\textsc{Int}[k] \subseteq \textsc{Int}[k']$ then $\textsc{Rank}[S[k]] \leqslant \textsc{Rank}[S[k']]$.

($\Leftarrow$) Let $\textsc{Int}[k] = [a,b]$ and $\textsc{Int}[k'] = [a',b']$ be two rank intervals such that $\textsc{Int}[k] \cap \textsc{Int}[k'] \neq \emptyset$. Assume that $\textsc{Rank}[S[k]] \leqslant \textsc{Rank}[S[k']]$. As $\textsc{Int}[k] \cap \textsc{Int}[k'] \neq \emptyset$, we have either $a \in [a',b']$ or $b \in [a',b']$. If $a \in [a',b']$ then b cannot be strictly greater than b' because otherwise $b'+1$, whose rank is strictly greater than $\textsc{Rank}[S[k']]$, would belong to $[a,b]$, which would be in contradiction to the hypothesis. In the same way, if $b \in [a',b']$ then a cannot be strictly smaller than a'. Hence assertion (1) is proved. □

The hierarchical structure of the collection $\textsc{Int}[k]$ for all $k \geqslant i$ can be represented as a tree whose nodes are associated to the intervals $\textsc{Int}[k]$ (possibly for several indices k if they produce the same interval) and whose topology reflects the nesting structure as described by Lemma 1. Note that the set of all the leaves having a given node $\textsc{Int}[k]$ as ancestor corresponds to the set of positions of S bounded by it. In Fig. 1, we show the tree corresponding to the ranks associated to position $i = 1$ of $S = \texttt{abcdacadbab}$.

The construction of the intervals $\textsc{Int}[k]$ can be done in linear time. Initializing a stack by pushing the position 0 of infinite rank, we iterate the following process for each position k from 1 to $|S|+1$:

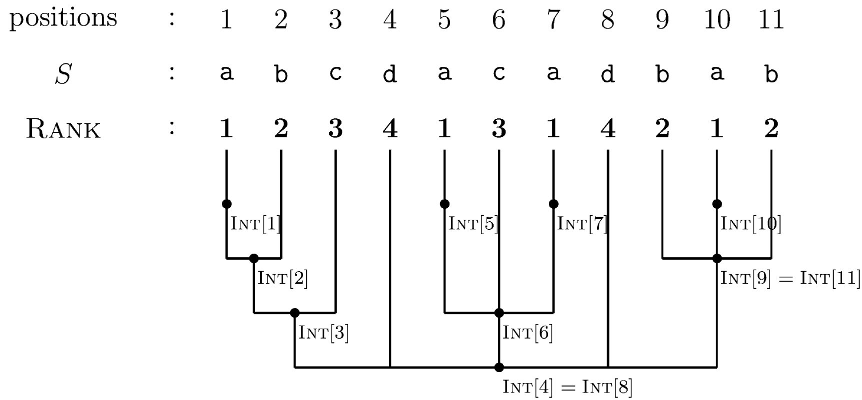

Fig. 1. Tree representation of the hierarchy of the rank intervals associated to position 1 of $S =$ abcdacadbab.

(1) Pop from the stack all the positions of rank smaller than RANK$[S[k]]$ and fix the right bound of their corresponding intervals to $k-1$ (for all these positions, k is the smallest position of greater rank).
(2) Let t be the top of the stack, i.e. the greatest position $t < k$ of rank greater or equal to RANK$[S[k]]$. If RANK$[S[k]] =$ RANK$[S[t]]$, fix the left bound of INT$[k]$ to the left bound of INT$[t]$, otherwise to $t+1$.
(3) Push the position k on the stack.

As the last iteration is done with the position $|S|+1$ of infinite rank, all the intervals of interest (i.e. not including any position of infinite rank) are computed.

Theorem 2. *If a maximal substring $S[a,b]$ is complete, then $[a,b]$ is a rank interval.*

Proof. Assume $S[a,b]$ both maximal and complete. There is $j > i$ such that $\mathcal{CS}(S[i,j]) = \mathcal{CS}(S[a,b]) = C$ (Remark 2). By definition of RANK, there is no symbol in C of infinite rank, and each integer between 1 and $m = \max\{\text{RANK}[x] \mid x \in C\}$ appears as a rank of a symbol of C. In other words, $\{\text{RANK}[x] \mid x \in C\} = \{1, \dots, m\}$ and $S[a,b]$ is complete. As $S[a,b]$ is maximal, the symbols $S[a-1]$ and $S[b+1]$ are not in C. Moreover, each rank in $\{1, \dots, m\}$ has a unique antecedent in Σ and we have necessarily that RANK$[S[a-1]]$ and RANK$[S[b+1]]$ are strictly greater than m. Let $k \in [a,b]$ such that RANK$[S[k]] = m$. It follows from the preceding and the definition of the rank interval INT$[k]$ that $[a,b] =$ INT$[k]$. □

4.3. Selecting the complete rank intervals

Definition 8 *(rank distance).* If k and k' are positions of S, the *rank distance* between k and k', denoted $\mathbf{d}(k,k')$, is the maximum rank of symbols occurring in S between k and k', both included.

Computing the rank distance between two positions of S is exactly the *Range Maximum Query Problem* and can be solved in constant time per query after $\Theta(|S|)$ time preprocessing [9].

Remark 3. Let INT$[k]$ be a rank interval, and $j \in$ INT$[k]$ a position inside this interval. The rank distance between j and a position outside INT$[k]$ is strictly greater than the rank distance between j and any position inside INT$[k]$.

Definition 9 *(rank-nearest).* Let k be a position and I a non-empty set of positions of S. Denote by $\overleftarrow{k}$ the greatest position of I smaller than k, if it exists, and 0 otherwise. Similarly, denote by $\overrightarrow{k}$ the smallest position of I greater than k, if it exists, and $|S|+1$ otherwise. The *rank-nearest* position of k in I, denoted $N(k,I)$, is:

- $\overleftarrow{k}$ if $\mathbf{d}(k,\overleftarrow{k}) \leqslant \mathbf{d}(k,\overrightarrow{k})$,
- $\overrightarrow{k}$ otherwise.

Definition 10 *(successor).* Let k be a position of S with a finite rank and P the set of positions of rank $\textsc{Rank}[S[k]]+1$. If P is not empty and if the rank-nearest position of k in P, $N(k,P)$, belongs to $\{1,\dots,|S|\}$, the *successor* of k is $N(k,P)$, otherwise k does not have a successor.

Successors will be stored in a table $\textsc{Succ}$ where for each k, $1 \leqslant k \leqslant n$, entry $\textsc{Succ}[k]$ is the successor of k. By chaining positions of S of successive rank forward and backward and using the solution of the *Range Maximum Query Problem* of [9], the computation of the table $\textsc{Succ}$ is possible in $\Theta(|S|)$ time and space.

Definition 11 *(path).* A *path* of level l is a sequence of l positions of S satisfying the following recursive definition:

- A path of level 1 is a position of the symbol of rank 1.
- A sequence $(p_1,\dots,p_{l+1})$ is a path of level $l+1$ if:
  - the sequence $(p_1,\dots,p_l)$ is a path of level l;
  - $\textsc{Rank}[S[p_{l+1}]] = l+1$;
  - denoting by E the set of the last positions of the paths of level l having p_{l+1} as successor, we have $p_l = N(p_{l+1},E)$.

We can illustrate this definition using the example of Fig. 1. If we consider the rank arising from the first position of the sequence (displayed in the third line of the figure), there are four paths of level one: (1), (5), (7) and (10). Following the above definition and considering the successors of their last positions, some of these paths can be continued to paths of level two. In particular, position 2 is the successor of three last positions of paths of level one: 1, 5 and 7. Among these, position 1 is the rank-nearest of position 2. So, the only path of the next level that includes position 2 is $(1,2)$, and the paths (5) and (7) cannot be continued.

If we iterate the same process, we can compute the whole set of paths which is given by the sequences of positions $(1,2,3,4)$, (5), (7), $(10,9,6)$ and all their non-empty prefixes.

Note that each position of S appears in at most one path.

Theorem 3. *A rank interval* $\textsc{Int}[k]$ *is complete if and only if it contains a path of level* $\textsc{Rank}[S[k]]$.

Proof. ($\Leftarrow$) By definition, if $\textsc{Int}[k]$ contains a path of level $\textsc{Rank}[S[k]]$, then it is complete.

($\Rightarrow$) Reciprocally let us assume $\textsc{Int}[k]$ to be complete and for each l, $1 \leqslant l \leqslant \textsc{Rank}[S[k]]$, let P_l be the set of paths of level l included in $\textsc{Int}[k]$. We will prove by induction that P_l is not empty for any $l \leqslant \textsc{Rank}[S[k]]$. In particular, there is at least one position of rank 1. So P_1 is not empty.

Assume that P_l is not empty for some level $l < \textsc{Rank}[S[k]]$. As $\textsc{Int}[k]$ is complete, each last position of a path in P_l has a successor in $\textsc{Int}[k]$ (Remark 3). Hence, there is at least one position $u \in \textsc{Int}[k]$ of rank $l+1$ that is a successor of a last position of a path in P_l. Let E be the set of the last positions of a path of level l having u as successor. As E includes at least one position of $\textsc{Int}[k]$ (from the preceding and Remark 3 again), the position $p_l = N(u,E)$ belongs to $\textsc{Int}[k]$. Applying recursively the same argument to $p_{l-1}, \dots, p_1$, the positions of smaller ranks of the path ending at p_l, allows us to conclude that this path is included in $\textsc{Int}[k]$. Finally $(p_1,\dots,p_l,u)$ is a path included in $\textsc{Int}[k]$ and P_{l+1} is not empty. □

4.4. Algorithm

The general strategy of the algorithm is, for each position i of S, to compute all the maximal locations of the growing character sets admitting a maximal location starting at i. As we want to output the locations of a given character set only once, the intervals computed are reported only if the first maximal location found starts at i.

Theorem 2 states that all the maximal intervals of which corresponding substrings have eventually the same character set as a maximal substring starting at i are the rank intervals associated to position i, but only those which are complete. The table $\textsc{Rank}$ of all rank intervals associated to a position i is pre-computed. In order to prepare the completeness test, at the same time also the preprocessing used for rank distance queries is performed and the successor table is filled.

A direct consequence of Theorem 3 is that a rank interval $[a, b]$ is complete if and only if there is a position $k \in [a, b]$ such that:

- INT$[k] = [a, b]$,
- k is the last position of a path included in INT$[k]$.

If we have the set of all the paths of a given level l, testing if their bounds are included in the rank interval of their last positions will give us all the maximal locations of the character set of cardinality l admitting a maximal location starting at i. To perform this test, we just need to represent each path of level l by an element recording the two following fields:

- POS, the last position of the path, and
- BOUND, the smallest interval containing the path.

We store the elements corresponding to all the paths of a given level in a chained list LIST.

The construction of the paths of growing levels is done iteratively. Before starting, LIST is initialized with all the positions of rank 1, in ascending order. At each iteration, after testing the preceding inclusion and possibly reporting the maximal intervals, the list is updated to go from level l to level $l + 1$. First one needs to remove from LIST all the elements corresponding to paths that cannot be continued. According to Definition 11, this arises in two situations: when a last position does not have a successor or when a last position is not the rank-nearest of its successor among the other candidates. If we assume that the last positions are ordered in ascending order in LIST for the level l, their respective successors are, by construction, also in ascending order. In particular, all the last positions having the same successor appear successively in LIST, and the rank-nearest from this successor is easy to determine (we just need to consider the last positions bounding this successor). Next the field POS of the elements of the list corresponding to paths that can be continued to paths of level $l + 1$ are replaced by their successors and the field BOUND. At the end of the iteration, the elements of LIST are still ordered in ascending order of their field POS. So as LIST is ordered at its initialization, it will be ordered for all the iterations.

For each position i of S, all the steps of the initializations are of complexity linear in $|S|$ both in time and space. During the execution of the loop of construction each position of S is parsed and stored at most twice: (possibly) once as successor and once as last position of a path. The construction of the paths is also linear both in space and time.

The whole procedure is summarized in Algorithm 1. Altogether, we have:

Theorem 4. *Given a string S of length n, Algorithm 1 reports the locations of all the maximal substrings of S, gathered by character sets, in $\Theta(n^2)$ time using $\Theta(n)$ space.*

5. Finding the locations of all character sets—second algorithm

In this section we show how to improve the algorithm of Amir et al. [7] for solving Problem 2 by a factor of $\log n$. We first give a short description of the original algorithm in Section 5.1, and then we present our improvement in Section 5.2. W.l.o.g., we assume in the following that $|\Sigma|$ is a power of 2.

5.1. A $\Theta(n|\Sigma|\log|\Sigma|\log n)$ algorithm

Given a string S of length n over alphabet Σ, the algorithm performs $|\Sigma|$ iterations, where in the kth iteration it finds all the substrings of S with character sets of size k. The algorithm maintains a maximal interval $[a, b]$ over S, and arrays COUNT$[1..|\Sigma|]$ and LIFE$[1..|\Sigma|]$. COUNT$[i]$ is the number of times the character i appears in $S[a, b]$, and LIFE$[i]$ is 1 if COUNT$[i] > 0$ (namely, if i is in the character set of $S[a, b]$), and 0 otherwise.

Initially, the interval $[a, b]$ spans the longest prefix of S whose character set has size k. Then, the interval is moved in the following way: First, b is increased until the character set of $S[a, b]$ has size $k + 1$, and then a is increased until the character set of $S[a, b]$ has size k (the array COUNT is used to find the new values for a and b). The interval movement is repeated until the end of S is reached. This way, all the character sets of all substrings of S can be found in $\Theta(n|\Sigma|)$ time. However, each set may be encountered several times. Therefore, the algorithm needs to identify and

```
{Main loop}
for each position i of S do
  {Initialization}
  Compute the table RANK
  Compute the table of intervals INT
  Perform the preprocessing for the Range Maximum Queries
  Compute the table SUCC
  Initialize LIST with the positions of rank 1 in increasing order

  while LIST is not empty do
    {Test if there are locations to output}
    FIRST ← the first element of LIST verifying FIRST.BOUND ⊆ INT[FIRST.POS]
    if the left end of INT[FIRST.POS] ⩾ i then
      output INT[FIRST.POS]
      PREVIOUS ← INT[FIRST.POS]
      for each element E following FIRST in LIST do
        if E.BOUND ⊆ INT[E.POS] and INT[E.POS] ≠ PREVIOUS then
          output INT[E.POS]
          PREVIOUS ← INT[E.POS]
        end if
      end for
      output newline {no more maximal substring with the same character set}
    end if
    {Compute the next level}
    for each element E in LIST do
      if SUCC[E.POS] does not exist or E.POS is not the nearest index of
      SUCC[E.POS] among the positions of the list having the same successor then
        remove E from LIST
      else
        E.POS ← SUCC[E.POS]
        {Update E.BOUND}
        if E.POS < E.BOUND.start then
          E.BOUND.start ← E.POS
        end if
        if E.POS > E.BOUND.end then
          E.BOUND.end ← E.POS
        end if
      end if
    end for
  end while
end for
```

Algorithm 1.

collate multiple occurrences of the same character set in order to solve Problem 2. This is done using additional data structures.

A subarray LIFE$[i2^l+1..(i+1)2^l]$ of LIFE ($0 \leqslant l \leqslant \log|\Sigma|$, $0 \leqslant i \leqslant |\Sigma|/2^l - 1$) will be called a *block of level* l. The main idea is to assign a *name* for every block in LIFE in all the configurations of LIFE. The names are consistent, namely two blocks of the same level with the same content are assigned the same name. In particular, in all the maximal locations of some character set C, the names assigned to the entire array (i.e., to the block of level $\log|\Sigma|$) are equal.

The naming is performed as follows: Consider the initial configuration of LIFE (i.e., the one that corresponds to the first interval). The name of a block in level 0 is the corresponding value in LIFE. Now, suppose that we assigned names for all the blocks of level $l-1$. We will assign names to the blocks of level l from left to right. A block of level l is composed of two blocks of level $l-1$. Suppose that the current block is composed of two blocks whose names are

10
8 | 9
5 | 6 | 6 | 7
2 | 3 | 2 | 4 | 2 | 4 | 4 | 3
0 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0

(a)

12
11 | 9
7 | 6 | 6 | 7
4 | 3 | 2 | 4 | 2 | 4 | 4 | 3
1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0

(b)

Fig. 2. An example of blocks naming. Figure (a) shows an example of naming for the array LIFE = 0, 1, 0, 0, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 0. Figure (b) shows the names after changing LIFE[1] to 1 (changed names appear in boldface). Note that the first block on level 2 has name 7, as the pair (4, 3) appeared previously, while the first block on level 3 has a new name (11), as the pair (7, 6) did not appear previously.

x and y. If the pair (x, y) appeared previously, then the name of the current block is the name that was assigned to the pair (x, y). Otherwise, assign the minimum unused name to the pair (x, y), and also assign this name to the current block. See Fig. 2 for an example.

After each movement of the interval $[a, b]$, two positions in the array LIFE are changed: In one position, a 1 is changed into 0, and in a second position, a 0 is changed into 1. After each of these changes, the algorithm updates the names of the blocks. This can be done efficiently as only one block changes its name in each level (see Fig. 2).

Checking whether a pair (x, y) appeared previously can be done in $O(\log n)$ time using balanced binary search trees. Thus, creating the names for the first interval takes $O(|\Sigma| \log |\Sigma| \log n)$ time, and updating the names after each interval movement takes $O(\log |\Sigma| \log n)$ time. Therefore, the total time complexity of the algorithm is $O(n|\Sigma| \log |\Sigma| \log n)$.

5.2. A $\Theta(n|\Sigma| \log |\Sigma|)$ *algorithm*

Consider some fixed k. The improved algorithm differs from the algorithm of the previous section in the order in which names are assigned to blocks: The new algorithm first assigns names to the level 1 blocks of LIFE for all possible intervals (with character set of size k). Then, the algorithm assigns names to the level 2 blocks for all possible intervals, and this is continued until the algorithm assigns names to the blocks of level $\log |\Sigma|$. As in the previous algorithm, we need to ensure that the names are consistent. For obtaining the names, the algorithm builds lists $L_1, \ldots, L_{\log |\Sigma|}$. The list L_s will contain $|\Sigma|/2^s$ elements corresponding to the $|\Sigma|/2^s$ blocks of level s in the initial configuration of LIFE (i.e., the configuration that corresponds to the first interval), and one element for each level s block whose name is changed due to a change in LIFE. Each element of L_s is a tuple (x, y, z, i) that corresponds to the ith block in level s at some configuration of LIFE, where x and y are the names assigned to the two sub-blocks of the block, and z is the name that is assigned to the block. The algorithm also maintains arrays $\text{NAME}_1, \ldots, \text{NAME}_{\log |\Sigma|-1}$. The array $\text{NAME}_s[1..|\Sigma|/2^s]$ is used to store the names of all the blocks of level s at some configuration of LIFE.

Assigning names to the blocks of level 1 is done as follows: Start with the initial configuration of LIFE, and for $j = 1, \ldots, |\Sigma|/2$, add the tuple $(\text{LIFE}[2j-1], \text{LIFE}[2j], 0, j)$ to the list L_1. Note that the third element of the tuple is 0, which means that the block has not been assigned a name yet. When a name is assigned to the block, the 0 will be replaced by that name. Recall that during the movement of the interval $[a, b]$, the cells of LIFE are changed. Every time a cell in LIFE changes its value, add the tuple $(\text{LIFE}[2\lceil i/2\rceil - 1], \text{LIFE}[2\lceil i/2\rceil], 0, \lceil i/2\rceil)$ to L_1, where i is the index of the cell.

After building L_1, create a copy L'_1 of L_1, and store pointers from each element in L'_1 to the corresponding element in L_1. Then, lexicographically sort the list L'_1, where the key of an element $(x, y, 0, i)$ of L_1 is (x, y). Afterward, traverse L'_1 from its start to its end, and assign a name to each element of L'_1 and the corresponding element in L_1: The first element of L'_1 is assigned the name 1. For an element $(x, y, 0, i)$ whose previous element is (x', y', z', i'),

assign the name z' if $(x, y) = (x', y')$, and assign the name $z' + 1$ otherwise (assigning a name z to $(x, y, 0, i)$ means that the zero is replaced by z).

To assign names to the blocks of level 2, the algorithm uses the array NAME_1. Initially, NAME_1 contains the names of the level 1 blocks of the first configuration of LIFE. That is, $\text{NAME}_1[s]$ is equal to the name assigned to the sth element in L_1. Then, the list L_2 is built as follows: For $j = 1, \ldots, |\Sigma|/4$, add the tuple $(\text{NAME}_1[2j - 1], \text{NAME}_1[2j], 0, j)$ to L_2. Furthermore, for every $k > |\Sigma|/2$, let (x, y, z, i) be the kth element of L_1. Change the value of $\text{NAME}_1[i]$ to z, and add the tuple $(\text{NAME}_1[2\lceil i/2\rceil - 1], \text{NAME}_1[2\lceil i/2\rceil], 0, \lceil i/2\rceil)$ to L_2. As before, a copy L'_2 of L_2 is generated and its elements are sorted. By traversing L'_2, the elements of L_2 are assigned names. This process is repeated for building the lists $L_3, \ldots, L_{\log|\Sigma|}$.

To analyze the time complexity, notice that the length of a list L_s is at most $|\Sigma|/2^s + 2n \leqslant 3n$ (we assume that $|\Sigma| \leqslant n$, otherwise we use the algorithm of Section 4). It follows that the names of the blocks of each level are bounded by $3n$. Using radix sort, sorting a list L'_s takes $O(n)$ time. Therefore, the time complexity of the algorithm is $\Theta(n|\Sigma|\log|\Sigma|)$. The space complexity of the algorithm is $\Theta(n)$ (note that after the list L_s is built, the lists L_{s-1} and L_s can be erased).

The time complexity of the algorithm can be further reduced to $\Theta(n|\Sigma|\log(\frac{|\Sigma|}{\log n} + 2))$: Let $s = \min(\log|\Sigma|, \lceil\log\log n\rceil)$. A block in level s can be assigned a name directly by treating the contents of the block as a number in binary representation and using this number as the name (we assume a RAM model in which arithmetic operations on integers between 0 and $n^{O(1)}$ are performed in constant time). Thus, the algorithm needs to assign names only for blocks with level greater than s. The number of levels which are processed is $\log|\Sigma| - s + 1 = O(\log(\frac{|\Sigma|}{\log n} + 2))$. We therefore have:

Theorem 5. *Given a string S of length n, the algorithm described above finds the locations of all the maximal substrings of S, gathered by character sets, in $\Theta(n|\Sigma|\log(\frac{|\Sigma|}{\log n} + 2))$ time using $\Theta(n)$ space.*

6. Conclusion

Compared to the amount of results on characterizing and finding substrings of sequences, the study of character sets of substrings of sequences is still in its infancy. In this paper we presented a very simple algorithm reporting all the maximal locations of a given character set in linear time, independent of the alphabet size, by direct parsing, and two algorithms for reporting the locations of maximal substrings gathered by character set of a given string or a collection of strings. The first of these two algorithms is alphabet independent and runs in $\Theta(n^2)$ time, the second one runs in $\Theta(n|\Sigma|\log(\frac{|\Sigma|}{\log n} + 2))$ time. As the possible number of different character sets occurring in a string of length n over an alphabet Σ is of order $\Theta(n|\Sigma|)$, none of the algorithms above solves Problem 2 in an optimal way. Moreover, their respective complexities show that the choice of using one or the other of these algorithms will depend (asymptotically) on the rate of the alphabet size relative to the length of the considered sequence. In particular, when considering a sequence of genes, we have $|\Sigma| = \Theta(n)$, favoring the first algorithm, but for natural language processing $|\Sigma|$ is *a priori* fixed with regard to the length of the texts, favoring the second algorithm.

Acknowledgement

We would like to thank Mathieu Raffinot for helpful comments and suggestions.

References

[1] T. Dandekar, B. Snel, M. Huynen, P. Bork, Conservation of gene order: a fingerprint of proteins that physically interact, Trends Biochem. Sci. 23 (1998) 324–328.

[2] I.B. Rogozin, K.S. Makarova, J. Murvai, E. Czabarka, Y.I. Wolf, R.L. Tatusov, L.A. Szekely, E.V. Koonin, Connected gene neighborhoods in prokaryotic genomes, Nucleic Acids Res. 30 (2002) 2212–2223.

[3] T. Uno, M. Yagiura, Fast algorithms to enumerate all common intervals of two permutations, Algorithmica 26 (2000) 290–309.

[4] S. Heber, J. Stoye, Finding all common intervals of k permutations, in: Proceedings of the 12th Annual Symposium on Combinatorial Pattern Matching, CPM 2001, in: Lecture Notes in Computer Science, vol. 2089, Springer-Verlag, Berlin, 2001, pp. 207–218.

[5] T. Schmidt, J. Stoye, Quadratic time algorithms for finding common intervals in two and more sequences, in: Proceedings of the 15th Annual Symposium on Combinatorial Pattern Matching, CPM 2004, in: Lecture Notes in Computer Science, vol. 3109, Springer-Verlag, Berlin, 2004, pp. 347–358.

[6] F. Karlsson, A. Voutilainen, J. Heikkilä, A. Antilla, Constraint Grammar: A Language-Independent System for Parsing Unrestricted Text, Mouton de Gruyter, 1995.
[7] A. Amir, A. Apostolico, G.M. Landau, G. Satta, Efficient text fingerprinting via Parikh mapping, J. Discrete Algorithms 26 (2003) 1–13.
[8] G. Didier, Common intervals of two sequences, in: Proceedings of the Third International Workshop on Algorithms in Bioinformatics, WABI 2003, in: Lecture Notes in Bioinformatics, vol. 2812, Springer-Verlag, Berlin, 2003, pp. 17–24.
[9] M.A. Bender, M. Farach-Colton, The LCA problem revisited, in: Proceedings of the 4th Latin American Symposium on Theoretical Informatics, LATIN 2000, in: Lecture Notes in Computer Science, vol. 1776, Springer-Verlag, Berlin, 2000, pp. 88–94.