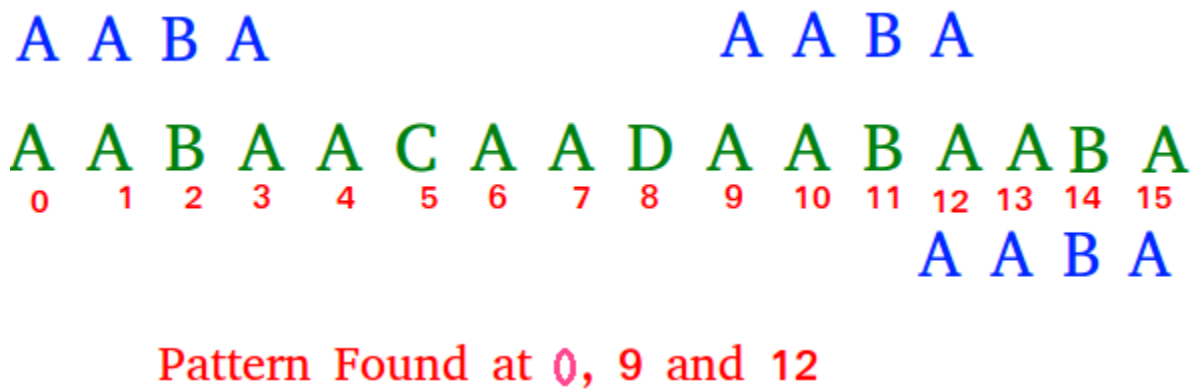
Reducing Time and Space in Indexed String Matching by Characters Distance Text Sampling

Simone Faro and Francesco Pio Marino

Dipartimento di Matematica e Informatica,
Università di Catania, viale A.Doria n.6, 95125, Catania, Italia
faro@dmi.unict.it

String Matching

String Matching consists in finding all occurrences of a given pattern x, of length m, in a large text y, of length n, where both sequences are composed by characters drawn from an alphabet Σ of size σ.



Exist two types of solutions, called *Online* and *Offline*:

- Solutions based on the first approach assume that the text is not pre-processed and thus they need to scan the text online, when searching.
- Solutions based on the second approach tries to drastically speed up searching by pre-processing the text and building a data structure that allows searching in time proportional to the length of the pattern.

Sampled String Matching

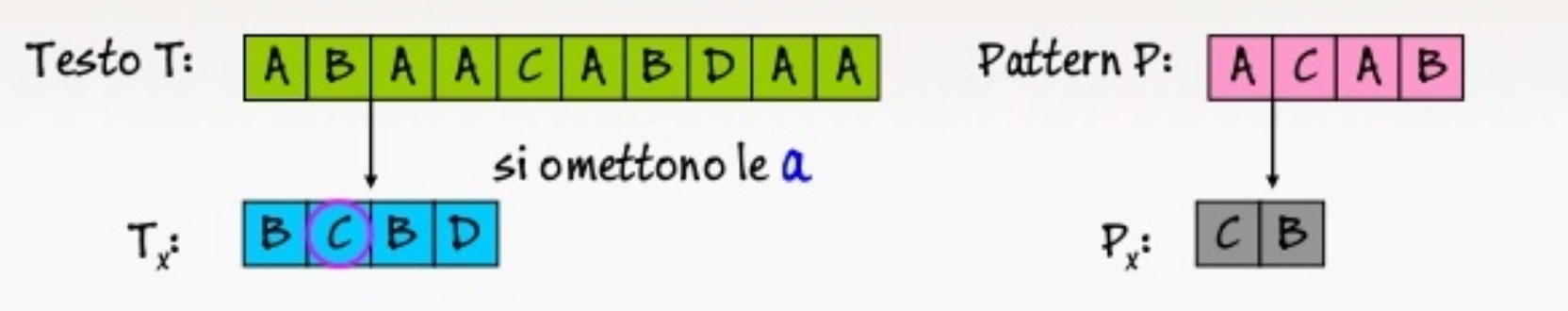
The task of the sampled string matching problem, introduced in 1991 by Vishkin, is to find all occurrences of a given pattern x in a given text y, assuming that a fast and succint pre-processing of the text is allowed in order to build a data-structure, which is used to speed-up the searching phase. For its features we call such data structure a ***partial-index*** of the text, the partial index should:

- Be succint
- Be fast to build
- Allow fast search
- Allow fast update

Apart the theoretical result of Vishkin, a more practical solution to sampled string matching has been recently introduced by Claude et al. based on an alphabet reduction.

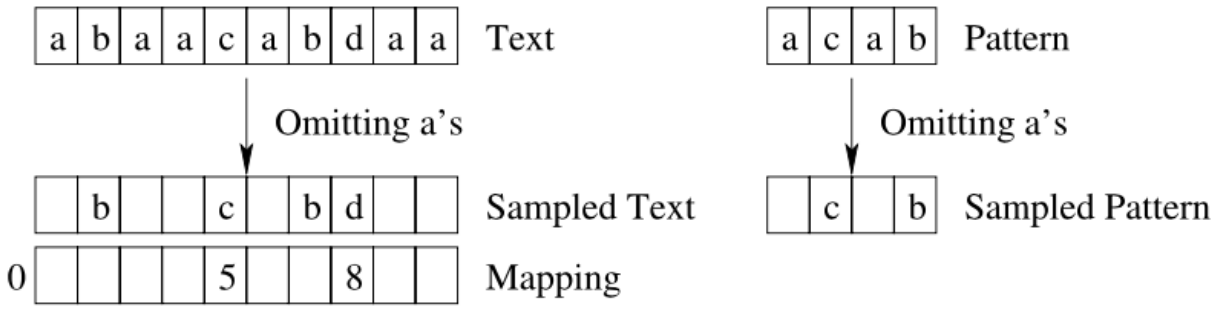
Occurrence Text Sampling

Let y be the input text, of length n, and let x be the input pattern, of length m, both over an alphabet Σ of size σ. The main idea of their sampling approach is to select a subset of the alphabet, $\hat{\Sigma} \subset \Sigma$ (the sampled alphabet), and then to construct a partial-index as the subsequence of the text (the sampled text) $\hat{y}$, of length $\hat{n}$, containing all (and only) the characters of the sampled alphabet $\hat{\Sigma}$. More formally $\hat{y}[i] \in \hat{\Sigma}$, for all $1 \leq i \leq \hat{n}$.



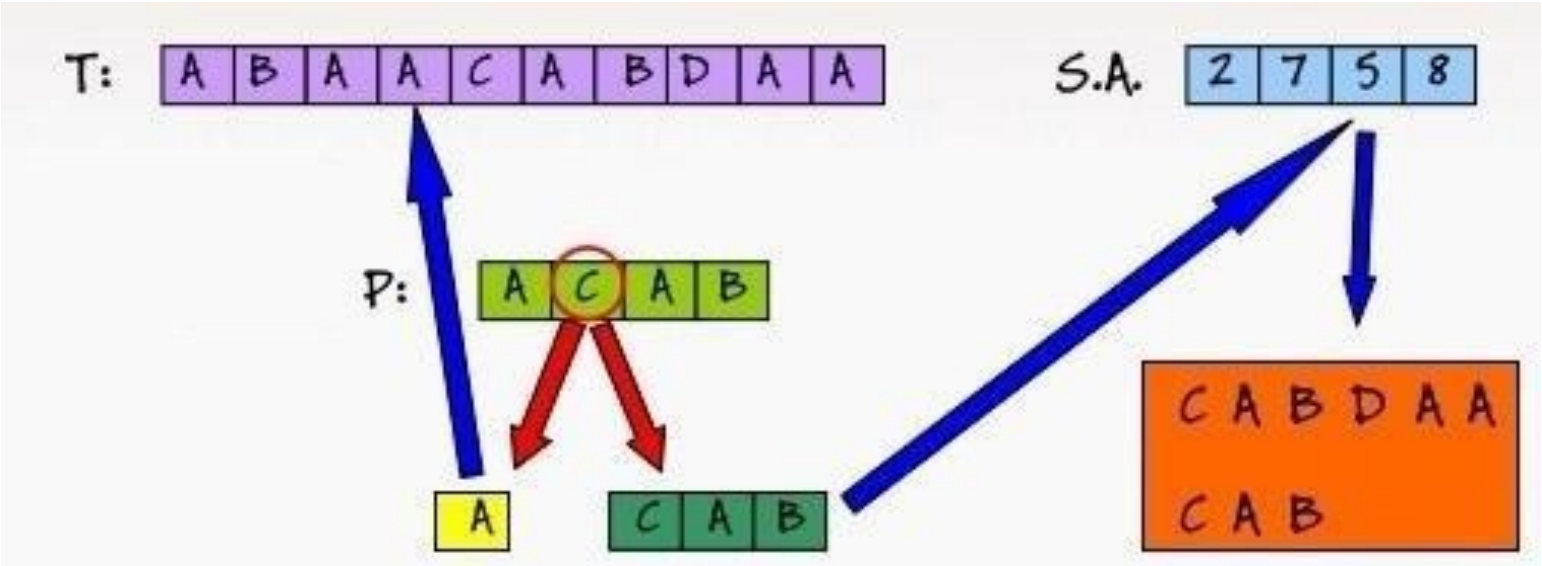
Position Mapping

During the searching phase of the algorithm a sampled version of the input pattern, $\hat{x}$, of length $\hat{m}$, is constructed and searched in the sampled text. Since $\hat{y}$ contains partial information, for each candidate position i returned by the search procedure on the sampled text, the algorithm has to verify the corresponding occurrence of x in the original text. For this reason a table ρ is maintained in order to map, at regular intervals, positions of the sampled text to their corresponding positions in the original text. The position mapping ρ has size $\lfloor \hat{n}/q \rfloor$, where q is the *interval factor*, and is such that $\rho[i] = j$ if character $y[j]$ corresponds to character $\hat{y}[q \times i]$. The value of $\rho[0]$ is set to 0. In their paper, on the basis of an accurate experimentation, the authors suggest to use values of q in the set $\{8, 16, 32\}$



Occurrence Text Suffix Array

To turn the sampling approach into an index*, Claude et al*. use a suffix array to index the sampled positions of the text. When constructing the suffix array, only suffixes starting with a sampled character will be considered, but the sorting will still be done considering the full suffixes. The resulting sampled suffix array is like the suffix array of the original text where suffixes starting with unsampled characters have been omitted.



Characters Distance Solution

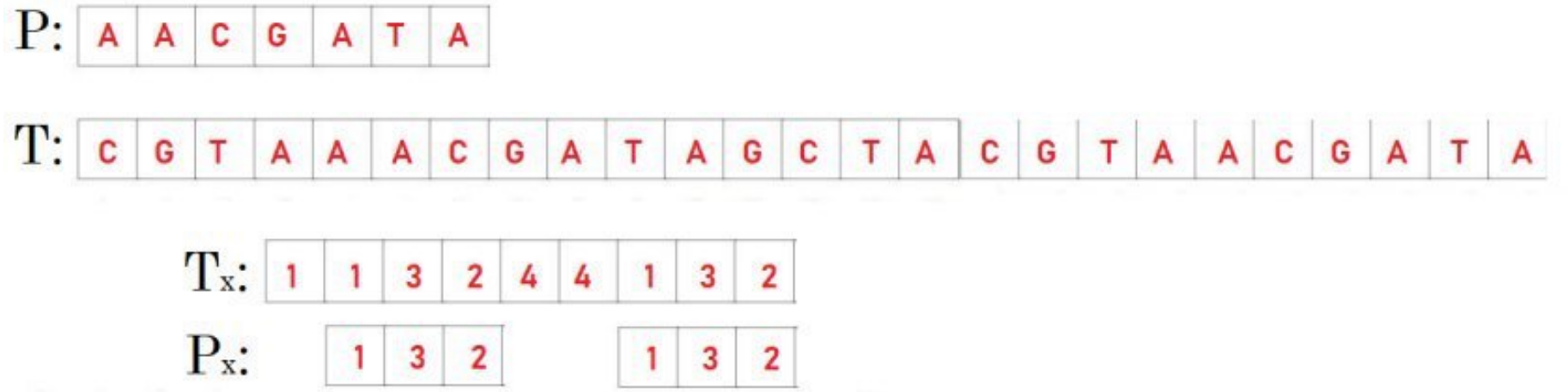
Definition 2.3 (The Characters Distance Sampling)**.** Let $c \in \Sigma$ be the pivot character, let $n_c \leq n$ be the number of occurrences of the pivot character in the text y and let δ be the position function of y. We define the *characters distance function* $\Delta(i) = \delta(i) - \delta(i-1)$, for $1 \leq i \leq n_c - 1$, as the distance between two consecutive occurrences of the character c in y.

The *characters-distance sampled version* of the text y is a numeric sequence, indicated by $\bar{y}$, of length $n_c - 1$ defined as

$$\bar{y} = \langle \Delta(1), \Delta(2), .., \Delta(n_c - 1) \rangle. \tag{2.1}$$

SAMPLING(y, n_c)
1. $j \leftarrow 0$
2. for $i \leftarrow 1$ to n do
3. if $y[i] \in \bar{\Sigma}$
4. $\bar{y}[j] = \delta(j) - \delta(j-1)$
5. $j = j + 1$

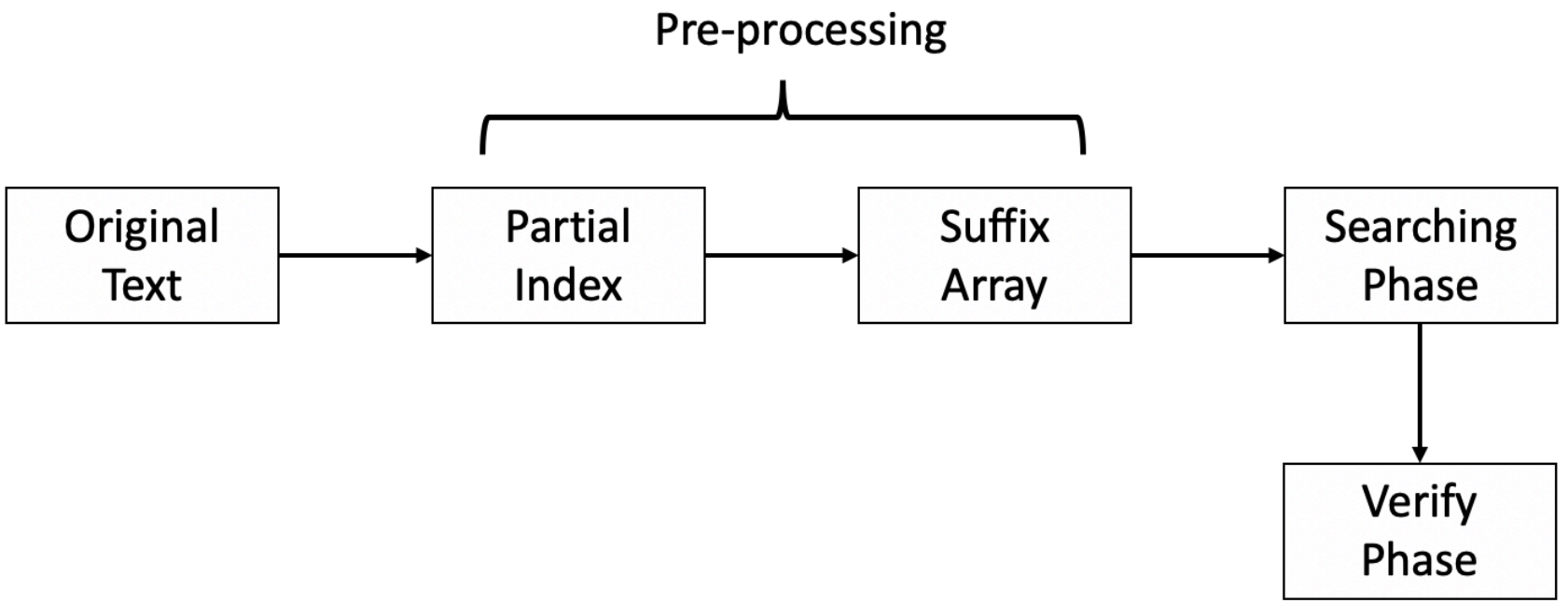
Characters Distance Solution



VERIFY(x, m, y, s)
1. $i \leftarrow 1$
2. while $i \leq m$ and $x[i] = y[s+i]$ do $i \leftarrow i+1$
3. if $i > m$ then return TRUE
4. return FALSE

Indexed Characters Distance

The algorithm proposed in this section is divided into two phases: a first *pre-processing phase* which consists in the construction of a suffix array of the sampled version of the text and a *searching phase* which is used to search any pattern x of length m in y making use of the suffix array $s_{\bar{y}}$ and the sampled text $\dot{y}$.



Indexed Characters Distance

Let y be an input text of length n over an alphabet Σ of size σ and let $C \subseteq \Sigma$ be the set of pivot characters. During the preprocessing phase the algorithm builds and stores the position sampled text $\dot{y}$ of y. This requires $O(n)$-time and $O(n_c)$-space, where n_c is the number of occurrences of any pivot character in y. Subsequently a suffix array of $\bar{y}$ is constructed.

However when constructing the suffix array of $\bar{y}$, the algorithm takes into account only suffixes beginning with a pivot character in the original text, drastically reducing the space requirement for maintaining the whole index. Apart from this detail, all other features of the data structure remain unchanged.

Example 7. Let y = "agaacgcagtata" be a text of length 13, over the alphabet $\Sigma = \{a,c,g,t\}$. Let $C = \{a\}$ be the pivot character. Thus the character distance sampling version of y is $\bar{y} = \langle 2, 1, 4, 3, 2 \rangle$.

$$
\begin{aligned}
s_{\bar{y}}[0] &= 1 \rightarrow \langle 1, 4, 3, 2 \rangle \\
s_{\bar{y}}[1] &= 0 \rightarrow \langle 2 \rangle \\
s_{\bar{y}}[2] &= 4 \rightarrow \langle 2, 1, 4, 3, 2 \rangle \\
s_{\bar{y}}[3] &= 3 \rightarrow \langle 3, 2 \rangle \\
s_{\bar{y}}[4] &= 2 \rightarrow \langle 4, 3, 2 \rangle
\end{aligned}
$$

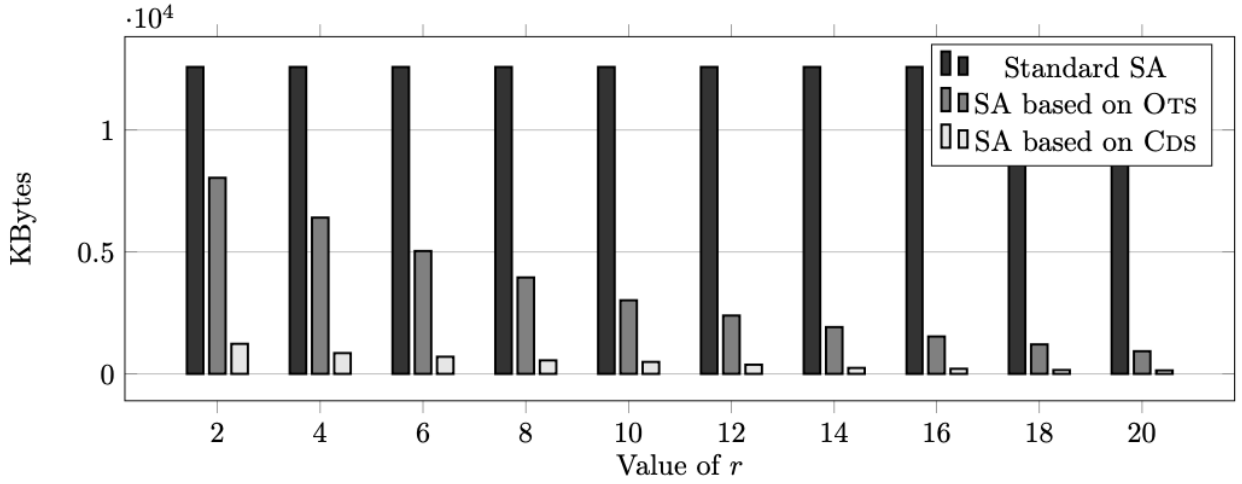
Indexed Characters Distance

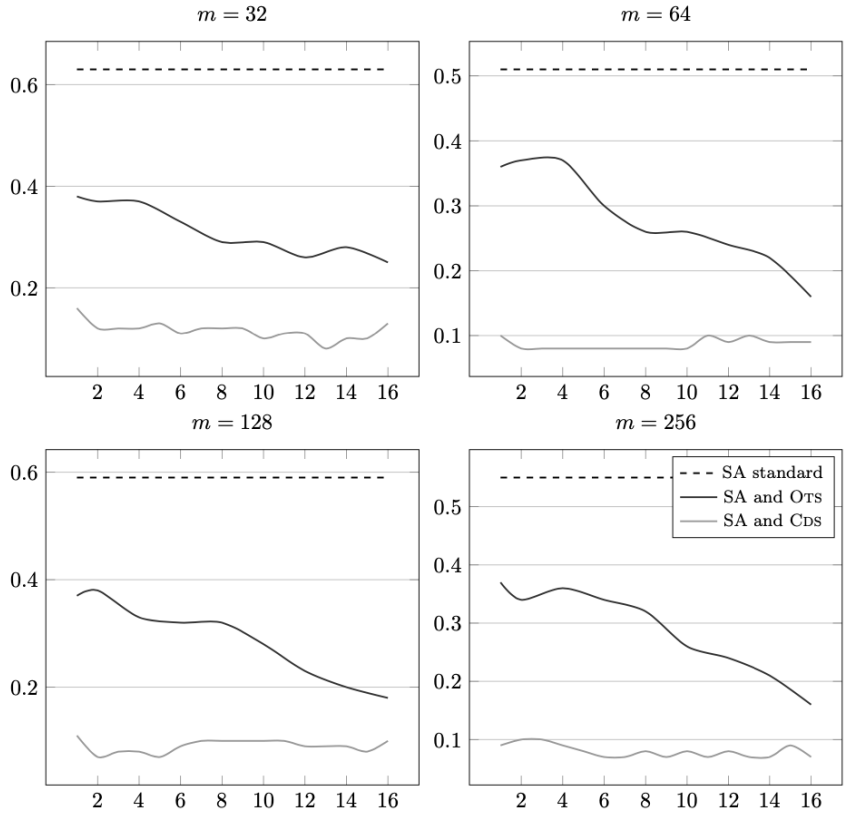
During the searching phase the algorithm uses the suffix array of the sampled text $s_{\bar{y}}$ as an index to quickly locate every occurrence of a sampled pattern $\bar{x}$ in $\bar{y}$. Each of these occurrences is treated as a candidate occurrence of x in y, and as such it will be verified by a comparison procedure.

Thanks to the lexicographical ordering of the suffix array, all such suffixes are grouped together and can be found efficiently with a single binary search, which locates the starting position of the interval. All other occurrence are then grouped together close the first one.

Space requirement and Searching time

As expected, the function which describes memory requirements follows a decreasing trend while the value of *r* decreases. Specifically the benefit in space consumption obtained by the algorithms based on character distance sampling ranges from 70% to 80% when compared with the *OTS* approach.
From such experimental results it turns out that the indexed searching approach based on *OTS* reaches a speed-up between 61% and 74%, while new proposed solution reaches a speed-up between 80% and 91%. In addition the best results are obtained in all cased by the approach based on character distance sampling.





Future Works

The previous sampling approaches to exact string matching prove to work efficiently only in the case of natural language texts or, in general, when searching on input sequences over large alphabets, while their performances degrade when the size of the underlying alphabets decreases.

- We are working on an extension of the approach previously introduced to small alphabets which turns out to be much more feasible in the case of biological data like genome or protein sequences
- A possible idea could be extend the previous indexed searching using such different data structures like Suffix tree or FM-Index

THANKS YOU
FOR YOUR ATTENTION

Any questions?