The Compressed Overlap Index

Rodrigo Cánovas\textsuperscript{1,2}, Bastien Cazaux\textsuperscript{1,2}, and Eric Rivals\textsuperscript{1,2}

\textsuperscript{1} L.I.R.M.M. Université de Montpellier, CNRS, U.M.R. 5506, Montpellier, France
\textsuperscript{2} Institut Biologie Computationnelle, Montpellier, France

\texttt{canovas-ba@lirmm.fr, cazaux@lirmm.fr, rivals@lirmm.fr}

\begin{abstract}
For analysing text algorithms, for computing superstrings, or for testing random number generators, one needs to compute all overlaps between any pairs of words in a given set. The positions of overlaps of a word onto itself, or of two words, are needed to compute the absence probability of a word in a random text, or the numbers of common words shared by two random texts. In all these contexts, one need to compute or to query overlaps between pairs of words in a given set. For this sake, we designed COvI, a compressed overlap index that supports multiple queries on overlaps: like computing the correlation of two words, or listing pairs of words whose longest overlap is maximal among all possible pairs. COvI stores overlaps in a hierarchical and non redundant manner. We propose an implementation that can handle datasets of millions of words and still answer queries efficiently. Comparison with a baseline solution – called FullAC – relying on the Aho-Corasick automaton shows that COvI provides significant advantages. For similar construction times, COvI requires half the memory FullAC, and still solves complex queries much faster.
\end{abstract}

\section{Introduction}

A text, a word, or a string $u$ is a sequence of letters taken from an alphabet $\Sigma$. Given two words $u, v$ over $\Sigma$, $u$ overlaps $v$ if a suffix of $u$ equals a prefix of $v$. Overlaps between the words of a given set are crucial in numerous applications: for computing word statistics or superstrings, for text compression, for analysing text algorithms, or for testing random number generators. In those various contexts, it is valuable to offer a versatile and scalable solution to compute such sets of overlaps. Here, we propose a data structure, dubbed COvI, which computes and indexes the overlaps of a set of words, and can thus be later queried to obtain the desired overlaps. Before, exposing our solution, let us dwell on the motivations for computing overlaps, starting with application in word statistics.

Consider a finite Bernoulli random text $T$ over $\Sigma$ (i.e., with independent and identically distributed symbols), and two words $u$ and $v$ of the same length over $\Sigma$. Do $u$ and $v$ have the same probability of occurring in $T$? In general, the answer is no since it depends on the self-overlaps of each word. For instance over $\Sigma = \{0, 1\}$, the words $u := 00$ and $v := 01$ have respectively a probability of $3/8$ and of $1/2$ of occurring in a text of length $3$. Moreover, $u$ can have two occurrences in such a text, while $v$ cannot.

\* This work is supported by the Institut de Biologie Computationnelle (ANR-11-BINF-0002) and Défi MASTODONS C3G from CNRS.
The difference is due to the fact that occurrences of $u$ can overlap themselves, while those of $v$ cannot. More generally, the absence probability, the waiting time of a word (the number of symbols before the first occurrence), the return probability (the number of positions between successive occurrences), the total number of occurrences in a random text of length $n$, all depend on the self-overlaps of this word [21]. All self-overlap positions can be encoded in a single binary vector called the autocorrelation: for instance the autocorrelation of $u :=$ abracadabra is $c(u) := 1000001001$, where a 1 denotes a self-overlap (see [9] for a definition). In [22], derives a formula for the absence probability in function of the autocorrelation. Such statistics are useful in many contexts, for instance to evaluate the significance of finding a word in a collection of text (in Information retrieval) or of finding that many occurrences of a DNA binding motif in a genome sequence (in Bioinformatics).

Now, because of possible overlaps, the occurrences of distinct words in a text are also interdependent. Like for a single word, when studying a set of words, their occurrences can overlap, and this influences their probability of occurring together or being absent together in random texts. These probabilities depend on the mutual overlaps of the words [18], which are encoded in binary vectors called correlations.

**Example** For the two words $u :=$ atatat, $v :=$ tggata over $\Sigma := \{a, c, g, t\}$, their autocorrelations are $c(u) := 101010$, $c(v) = 100000$, and their correlations are $c(u,v) = 000001$, and $c(v,u) = 000101$, where $c(x,y)$ denotes the correlation of $x$ over $y$, that is the binary encoding of the position where $x$ overlaps $y$ from the left. Note that with this notation, $c(u) = c(u,u)$.

Such probabilities, whose computation requires to list the correlations of pairs of words in the set, are used in many contexts. For instance, the number of common words of length $q$ (shared by two texts) serves to approximate the distance or the similarity between texts [12,4], and co-occurrence probabilities are then employed to optimise filtration criteria in similarity search algorithms[19,20]. Last, such word statistics are heavily used for testing random number generators [14], when considering the set of all possible words of length $q$ (which is a very specific case).

In [20], the computation of correlation vectors for a set of words is listed as an interesting open problem, which was solved by hand for small cases. An algorithm for the special case of a set containing all possible words of length $q$ was proposed in [14]. Hence, the goal of COvI is to provide a general, versatile, and scalable tool to compute word correlations for any set of words.

In bioinformatics, computation of superstrings models the question of DNA assembly, which aims at inferring a target DNA sequence from a set of short overlapping DNA fragments (also called reads). However, in practice, one considers approximate overlaps because of sequencing errors, and also overlaps between reverse complementary fragments since the DNA is double stranded. Specialised, efficient data structures have been designed to compute the longest overlap for any read pair and to represent them in a graph (e.g. [24]). A theoretical model is the Overlap Graph: a complete, weighted digraph in which each input sequence is a node, and the arc linking two nodes is weighted by the length of their longest overlap. COvI computes a data structure that subsumes the Overlap Graph, but requires much less space, and can be queried to build the Overlap Graph.
Fig. 1: Running example with $P = \{ATAT, ATTA, TAAT, TTAA, TTAT\}$. Left: the trie of $P$. Right: Aho-Corasick automaton (AC) of $P$, with the LOUDS and BP representations of the AC tree topology.

Below, we introduce known data structures required for the construction of COvI, whose construction and overlap queries are explained in Section 2. We investigate the performance of COvI in Section 3, and conclude in Section 4.

### 1.1 Basic Concepts

Let $\Sigma$ denote a finite alphabet of cardinality $\sigma$. A string or word $u$ over $\Sigma$ is a sequence of characters from $\Sigma$ of length $|u|$. For any integer $1 \leq i \leq j \leq |u|$, we denote the the $i$-th character of $u$ by $u[i]$, and the substring comprised between positions $i$ and $j$ by $s[i,j]$. A substring of $u$ is a prefix (resp. suffix) if it starts at position 1 (resp. ends at position $|u|$). A suffix (resp. prefix) of $u$ is said proper if it differs from $u$. Now we define the non symmetrical notion of overlaps between two strings. Let $u, v$ be strings over $\Sigma$. $u$ overlaps $v$ if a non empty suffix of $u$ equals a prefix of $v$ (i.e., there exists an integer $k$ such that $u[|u| - k + 1, |u|] = v[1,k]$). Then the string $v[1,k]$ is a right overlap for $u$ or a left overlap for $v$. The longest overlap of the pair $u$ onto $v$ is denoted $ov(u,v)$.

Throughout this work, the input consists in a set $P := \{s_1, \ldots, s_p\}$ of $p$ finite words. Then, we denote by $Ov^+(P)$ the set of all overlaps between any two words of $P$.

**Trie** Consider $P$ and assume that no word of $P$ is prefix of another (which can be achieved by appending a special symbol to each word). The trie is a tree, labelled on its arcs, designed to store a set of words [13] (see Figure 1, which is a running example). It has $p$ leaves and it spells out each word $s_i$ on a distinct branch ($s_i$ equals the concatenation of the labels from the root to its leaf). Each node $v$ uniquely represents a distinct prefix of the words in $P$, with the root being the empty prefix. Hence, two nodes share a common ancestor node $v$ if the string $v$ is a common prefix of both.

**Aho-Corasick Automaton** In 1975, Aho & Corasick proposed the first algorithm solving the exact set pattern matching problem, which for a set $P$ and text $T$ finds all the
occurrences of words of $P$ within $T$ \cite{1,10}. Their algorithm builds an automaton for $P$ and then processes $T$ using it – see Figure 1 for the Aho-Corasick automaton (AC) of our running example. In AC, the states are exactly the nodes of the trie $T$ of $P$, and the $goto$ transitions are the arcs of $T$, which link a node to its children. Hence, to a state $s$ is associated to the prefix represented by the corresponding node of $T$. The AC also includes failure links. The failure link of state $s$ (or simply of $s$) points to state $r$ if and only if $r$ represents the longest proper suffix of $s$ among all the states. Hence, one gets a characterisation of nodes representing overlaps:

**Proposition 1** \cite{25}. Let $s, r$ be two nodes/states in AC($P$). If $r$ is the node pointed by the failure link of node $s$, and $w$ is a leaf in the subtree of $r$ in $T$, then $r$ is an overlap from the string represented by $s$ onto the string represented by $w$ (i.e., $s$ overlaps $w$).

If $s$ is a leaf, we can reach the nodes representing all its overlaps with over words of $P$ by following the chain of failure links starting in $s$ and ending at the root of $T$. Conversely, nodes that are not on such chains (starting at leaves) are not overlaps between words of $P$.

**Rank and Select** To build COvI, we use binary vectors that support $\text{rank}$ and $\text{select}$ operations. Given a sequence $S$ over \{0, 1\}, and a position $i$ such that $1 \leq i \leq |S|$, then for any symbol $a$ in \{0, 1\}, $\text{rank}_a(S, i)$ returns the number of occurrences of $a$ until position $i$ in $S$, while $\text{select}_a(S, i)$ finds the position in $S$ of the $i$-th occurrence of $a$.

For a binary vector (also called bitmap) of length $n$, it is possible to answer $\text{rank}$ and $\text{select}$ in constant time using $n + o(n)$ bits \cite{16,6} ($n$ bits for storing the bitmap itself and $o(n)$ additional bits for the data structures). In practice, such structure requires 5% extra space over the bitmaps size \cite{8}.

**Tree Representation** The topology of a general tree $T$ of $n$ nodes can be represented succinctly using $2n$ bits, and by adding the support of $\text{rank}$ and $\text{select}$, one can support many tree operations in constant time \cite{2}. In this work we used two different succinct tree representations: LOUDS and BP.

The level-ordered unary degree sequence or LOUDS \cite{3,11} is built by traversing the nodes of $T$ in level-wise order, and for a node with $i$ children, we write $i$ zeros and a one into the bitmap. The balanced parentheses sequence or BP \cite{11,17} is built from a depth-first preorder traversal of $T$, writing an zero (opening parenthesis) when arriving to a node for the first time, and a one (closing parenthesis) when going up (after traversing the subtree of the node). Figure 1 shows an example of LOUDS and BP.

Experiments have shown that LOUDS is very efficient when only the simplest operations are needed, like accessing a node’s parent or going to an arbitrary child of a node, while BP also supports in constant time complex operations, such as computing the depth of a node \cite{2}.

## 2 Compressed Overlap Index

Here, we explain the algorithm to build the index COvI, and the procedures that implement the following queries. Let $x, y$ be two strings of $P$, and $q$ be a positive integer.
- correlation\((x, y)\): gives \(c(x, y)\), i.e. the correlation of \(x\) over \(y\).
- max-ov\((x, y)\): gives the length of the maximum overlap from \(x\) to \(y\) (i.e., \(|ov(x, y)|\)).
- all-right-ov\((x)\): gives an array of size \(|P|\) containing \(|ov(x, z)|\) for all \(z\) in \(P\).
- all-left-ov\((y)\): gives an array of size \(|P|\) containing \(|ov(z, y)|\) for all \(z\) in \(P\).
- global-max-ov\(()\): gives all words \(x\) in \(P\), such that there exists \(z_e \in P\) such that \(|ov(x,z_e)| = max\{|ov(w,z)|\text{ for all }w, z \in P\}\). It gives the words having the longest possible maximum overlaps among all possible pairs of \(P \times P\).
- threshold-right-ov\((x, q)\): gives an array of pairs \((z, l)\) where \(z \in P, |ov(x,z)| = l\) and \(l > q\). The query threshold-left-ov\((x, q)\) is defined similarly.

The queries correlation and max-ov works for a pair of words, while queries all-right-ov\((x)\) works of a given word \(x\) onto all words of \(P\). Finally, the query global-max-ov\(()\) is global since it evaluates all possible pairs. Note that if \(x = y\), correlation returns the autocorrelation of \(x\).

By definition of the Aho-Corasick automaton (AC), its structure is in fact the trie equipped with failure links. Above, we mentioned that one can find all overlaps within the Aho-Corasick automaton of \(P\) (Prop. 1) by traversing the chain of failure links starting at each node that represents a word of \(P\). However, some nodes do not represent overlaps between words of \(P\). In fact, COvI implements a reduced Aho-Corasick automaton where precisely those nodes have been removed, and the appropriate arcs compacted (arcs corresponding to goto and failure links). Hence, COvI’s underlying structure is not that of the AC, but a graph that we termed Extended Hierarchical Overlap Graph. The latter is a variant of the Hierarchical Overlap Graph defined in [5], except that its node set is \(\{P \cup Ov^+(P)\}\) (instead of containing only the union \(P\) with all maximal overlaps). In fact, this digraph has two types of arcs: those being the contractions of arcs from the trie, and those corresponding to failure links in the AC. Somehow, the underlying structure is a tree [25].

Experiments of Section 3 show that, because of this reduction, COvI can contain substantially less nodes than the full AC.

2.1 Construction
To construct COvI, we first build a succinct Aho-Corasick automaton, and then remove all nodes that do not correspond to an overlap between words of \(P\), and finally compact the arcs between remaining nodes. Given a set of words \(P\), our construction algorithm is decomposed in four steps: 1) build the trie \(T\) of \(P\), 2) compute the failure links of nodes of \(T\), 3) mark the nodes corresponding to overlaps in a array \(M\) and 4) build a new structure on the nodes in \(M\).

Building the trie We assume that \(P\) is given as a single text file, where the words are concatenated and separated by a special end-of-word symbol. In contrast to existing implementations, we choose a minimal structure for representing the trie: it has three components each stored in an array. For each node, we store the letter that connects it with its parent, a bit indicating if the node is a leaf, and a pointer to its right neighbour (if it exists). The neighbor array gives the position (in the array) of the neighbourhood of the current node, or 0 otherwise. Additionally, the first child of each node must be in the adjacent array position of the current node (unless it is a leaf).
Computing failure links Using $T$, which is trie of $P$, we compute the failure links following the original algorithm of Aho and Corasick, which visits the nodes in level-wise order [1].

Marking the nodes corresponding to overlaps All the overlaps between the words of $P$ (nodes that belong to the COvI) can be obtained by traversing the chain of failure links of each node that represents a word of $P$ in the trie $T$ until reaching the root. We start by creating an array $M$ to mark the nodes of $T$. After that, for each node that represents a word of $P$, we mark this node in $M$ and we traverse its chain of failure links until reaching either a marked node or the root. Finally, it marks all nodes of COvI and only these.

Building the new structure The last step of our algorithm consists in computing the components used to represent the COvI of $P$. We traverse the neighbour array of the trie (with nodes in depth-first order), and encode in a LOUDS array the new tree with the marked nodes (with nodes in breadth-first order) (see Fig. 2a). In addition, we store in an array Failure-Link the new failure links only for marked nodes of $T$. Note that the failure link of a marked node necessarily points to another marked node. Finally in the array Depths, each position gives the depth of a node represented by this position in $T$ (i.e. the length of the word corresponding to this node).

2.2 Supporting COvI Queries

To access easily to a node of COvI representing a word of $P$, we store an array $L$ corresponding to the mapping from the words of $P$ to the nodes of COvI. Here, we give the algorithm for each query and its complexity. Let $m$ denote the number of nodes in COvI.

Query $\text{max-ov}(x,y)$ Complexity in $O(|x| + |y|)$
To compute the maximum right overlap between $x$ and $y$, we need to find the deepest node on the tree such that it is an ancestor of the node representing $y$ and, at the same time, it can be reached using iteratively the failure links path starting from the node representing $x$. In the worst case, this takes $|x| + |y|$ node queries. In practice, we access the nodes $n_x$ and $n_y$ representing $x$ and $y$ respectively, and update $n_x$ by Failure-Link[$n_x$] and $n_y$ to Parent[$n_y$] (using the LOUDS array). Then, at each step, we check the depth of $n_x$ and $n_y$. If they have the same depth $d$, and are the same node or $d$ is zero, then return $d$. If not, in the case that $n_x$ is deeper than $n_y$, move $n_x$ to Failure-Link[$n_x$], otherwise move $n_y$ to Parent[$n_y$], and repeat the checking process (see Fig. 2) until eventually reaching the root.

Query $\text{correlation}(x,y)$ Complexity in $O(|x| + |y|)$
We use the same algorithm of $\text{max-ov}(x,y)$, but instead of stopping after finding the first overlap, we continue the process reporting each time we find a new overlap (until $n_x$ or $n_y$ arrive at the root) (see Fig. 2b).
Fig. 2: (a) shows how we traverse the trie to create the LOUDS array. Only the coloured nodes will be kept in COvI. (b): COvI of P and example of the max-ov(x,y) and the correlation(x,y) queries for x = ATTA and y = ATAT (x is the second word of P, y is the first). Starting from leaves k and l, the first node at the intersection of blue and dotted red paths is node b, whose depth equals one. The backward path of goto transitions is in bold blue, while the failure links is in dotted bold red arcs.

**Query all-right-ov(x) Complexity in O(m)**

We use an array O of size m (the number of nodes of the COvI tree) initialised with a predefined empty value. Then we access the node nx representing x and compute n_x := Failure-Link[n_x]. From this point, we recursively set O[n_x] equal to the depth of n_x and update n_x := Failure-Link[n_x] until arriving at the root of the tree. Finally, for each y a word of P, we move to the parent of the node ny (using the LOUDS array), and check the value of o := O[ny]. If o is not the empty value, then we know that o is the maximum overlap between x and y, otherwise we move ny to its parent and check the O array again. Before reporting the o overlap between two words, our methodology needs to assign the value o to O[n_y'] for all the nodes n_y' visited until that point.

**Query all-left-ov(y) Complexity in O(m)**

This query is similar to all-right-ov(x), because we have symmetrical definition of the failure and parent links.

**Query global-max-ov() Complexity in O(|P|)**

We know that every internal node of the COvI tree represents an overlap between at least two (different or the same) words in P. Then, for each word x in P we move to its parent and mark that node as a candidate to be a maximum overlap in the set. After, we initialise the maximum depth d = 0, and for each word y in P we move to its failure link node ny only checking the ones that are marked as candidate nodes. If the depth of ny is lower than d, we move to the next word. Otherwise, if it is equal to d, we add y to the set of answers. If the depth of ny is greater than d, we erase the current set of answers, add y to the set and update d to the depth of ny.

**Query threshold-right-ov(x,q) Complexity in O(m)**

We can use a similar approach to all-right-ov(x). The only difference is, that we
add an extra condition where we check if the depth of \( n_y \) is lower than \( q \). If that is the case, then we stop the current search at that point and update the values of the \( O[n'_y] \) assigning then a zero value. Finally, at the end of the overlap search of each \( y \) we only report the ones with depth greater than or equal to \( q \). The algorithm for query \textit{threshold-left-ov}(x,q)\) is similar.

3 Experiments

We assess the performance of COvI and compare it with a baseline solution based on the Aho-Corasick automaton. Given the lack of an existing implementation, we created FullAC, which stores the components of the Aho-Corasick automaton that are required for supporting the queries presented in Section 2. To be fair when comparing with COvI, FullAC only stores the topology of the generated trie, the failure link of each node, and a mapping between \( P \) and the nodes/leaves of the trie. In FullAC, the arcs are labelled by a single symbol. Thus, we choose to store the trie topology in BP format, which allows us to compute the depth of a node in constant time, thereby avoiding another array to store this information.

All experiments were performed on a Intel(R) Xeon(R) CPU E5-2623 at 3.00 GHz and 125 GB of main memory. The operating system was Ubuntu 14.04.1, version 3.19.0-59-generic Linux kernel. Our COvI and FullAC data structures are implemented in C++11, using version 5.4.30 of the g++ compiler and the sdsl library [7].

Datasets The test data used was generated from the DNA, PROTEINS and ENGLISH text of size 100 Megabytes, obtained from the Pizza&Chili Corpus\(^\text{3}\). From each of these files we created three datasets containing all the \( k\)-mers (with \( k \in \{25, 50, 100\} \)) extracted from each of the words stored in each text using a random, one to ten, skip step to compute the next \( k\)-mer. These datasets give us overlaps of different lengths between the words, depending on the value of \( k \) (with a maximum overlap length of \( k-1 \)). Before indexing, all words were alphabetically ordered and duplicates removed (using sort and uniq commands). Although COvI and FullAC do not require this preprocessing, we opt for it because it is then simpler to assess the influence of the number of input words on memory usage. Table 1 lists the datasets we used and their stats.

Note that for COvI these datasets are not ideal: they contain multiple overlaps for each word, but no duplicated words. This tends to lessen the difference in number of nodes between COvI and FullAC. In practice this case is not expected. For example in Bioinformatics generally the set of words (reads) consist of multiple short words with multiple repetitions and overlaps, or a set of long words with no repetitions and only very small overlaps. Both previous described cases are ideal for COvI given the construction explained in Section 2.1.

Difference in number of nodes. By design, COvI differs from FullAC in their numbers of nodes. The last three columns of Table 1 give the number of nodes of each structure, and the ratio of the number of nodes kept in COvI. For all datasets, COvI keeps only 20 to 32 percents of the nodes of FullAC. It also appears that the larger the \( k\)-mers,
| File Name | k-mer | $\sigma$ | Size (Megabytes) | Number of Words | Thousands of nodes FullAC | COvI | Ratio (%) |
|-----------|-------|---------|-----------------|----------------|--------------------------|------|----------|
| DNA       | 25    | 16      | 447             | 18,028,835     | 248,356                  | 70,357 | 28       |
| DNA       | 50    | 16      | 916             | 18,844,684     | 713,491                  | 159,625 | 22       |
| DNA       | 100   | 16      | 1,830           | 19,000,429     | 1,661,261                | 335,429 | 20       |
| PROTEINS  | 25    | 27      | 371             | 14,975,787     | 304,470                  | 89,581 | 29       |
| PROTEINS  | 50    | 27      | 685             | 14,090,763     | 628,268                  | 161,891 | 25       |
| PROTEINS  | 100   | 27      | 1,147           | 11,908,565     | 1,111,355                | 242,678 | 21       |
| ENGLISH   | 25    | 239     | 438             | 17,692,377     | 288,569                  | 92,402 | 32       |
| ENGLISH   | 50    | 239     | 868             | 17,864,478     | 734,042                  | 200,620 | 27       |
| ENGLISH   | 100   | 239     | 1,729           | 17,960,037     | 1,630,202                | 415,286 | 25       |

Table 1: Text files used in our experiments and their stats. The last three columns show the numbers of nodes in the FullAC, in COvI, and the percentage of the nodes of FullAC that are kept in COvI.

the fewer the nodes kept in COvI. Indeed, the compression achieved by COvI increases with $k$, as the number of nodes in FullAC increases with the length of input words. Naturally, this compression impacts the size of all three arrays of COvI.

Construction time and space. The final space occupied by each structure and the time spent to build COvI and FullAC are reported for each dataset in Table 2. The user time is in seconds and the final storage space in Megabytes (Mb). The construction of FullAC follows the same first two steps than for COvI (see Section 2.1), except that the BP representation can be computed during step two. One observes that most of the computational time is spent by the first two steps (that is, calculating the full trie and its failure links). As these steps are the same for FullAC and COvI, it follows that the extra time needed to transform Aho-Corasick structure into COvI amounts to less than 1% of the total time. Regarding memory consumption, COvI uses in the worse case $\sim 1.2$ times the size of the dataset, while only requiring between 20% to 40% of the FullAC size. Shall the datasets include repetitions, both the sizes FullAC and COvI would decrease similarly in function of the input size. This would impact the size of array $L$, which stores the mapping, but not their topology.

Query times. Finally we measured the queries time performances using COvI and FullAC. In the case of FullAC the queries are solved similarly as for COvI. The main difference is that, depending of the query, while the number of failure link transitions is the same for both approaches, the number of parent transitions in FullAC would increase in function to the number of nodes that are part of FullAC and are not in COvI.

We opted to present the times obtained only for the general queries (excluding queries threshold-right-ov($x,q$) and threshold-left-ov($x,q$) that are more difficult to compare because of the extra parameter $q$).

For max-ov($x,y$) and correlation($x,y$), we randomly selected 100,000 pairs of word indexes and reported the average time per query (in microseconds). Similarly, for all-right(left)-ov($x$), we randomly selected 100,000 word indexes and also reported the average time per query (in seconds). Last, we tested global-max-ov($x$), by
running this query 1,000 times and reporting the average time obtained (in seconds). Table 3 shows the results obtained for each of these queries.

From Table 3 one sees that query times for max-ov(x,y) and correlation(x,y) are similar: they depend on the words’ length. For these queries COvI is always between 2 to 4 times faster than FullAC. The times obtained are around 10 fold the time for computing directly the overlap between two words, because of the use of LOUDS and BP, respectively, to move within the tree. The advantage of keeping COvI in this case is that, in general finding and extracting the two words to be compared from a plain text takes longer than using directly our data structure.

For the queries all-right(left)-ov(x) COvI also performs better than FullAC being 2 to 4 times faster than FullAC. Notice that a naive approach would compute the right(left) overlaps of x against each of the words in a set: it would take p times the time of computing one overlap. We can deduce (given the number of words in each set) that our method improves over a naive approach by taking ≃ 20 to 30 percents of its time. Moreover, given our approach, if the set contains repetitions, COvI would only compute the overlaps once, while a naive approach would recompute these multiple times.

An advantage of COvI is that it contains all overlap information without the unnecessary nodes of FullAC. This avoids recomputing overlaps that are shared between different pairs of words. A clear proof of that, is the performance displayed for global-max-ov(). While a naive approach would require p² overlap computations, COvI only uses 2p queries. Notice that for our datasets, computing this query naively would be prohibitive (taking on the order of weeks to finish), while COvI takes only a few seconds.

**Scalability** To probe the scalability and efficiency of COvI on real genomic data, we compared on a larger server COvI and FullAC on a set of 49 million reads of 75 nucleotides each (≃ 3.5 gigabytes Gb of sequences; available at http://pbi1.univ-lyon1.fr/pub/logiciel/kissplice/TWAS_paper/GeuvadisFastq.tar.gz). FullAC had 1.392 million nodes and occupied 5.74 Gb of memory, while COvI stored 334 million nodes in 1.74 Gb, which is circa half the original input size. COvI scales up and offers even higher compression on genomic data than on our benchmark datasets.
Table 3: Average times to perform five queries defined in Section 2. The times for the first two queries are in microseconds, the others are in seconds.

|             | DNA 25 | DNA 50 | DNA 100 | PROTEINS 25 | PROTEINS 50 | PROTEINS 100 | ENGLISH 25 | ENGLISH 50 | ENGLISH 100 |
|-------------|--------|--------|---------|-------------|-------------|--------------|-----------|-----------|-------------|
| **k-mer**   |        |        |         |             |             |              |           |           |             |
| Time (microseconds) |        |        |         |             |             |              |           |           |             |
| max-ov(x,y) |        |        |         |             |             |              |           |           |             |
| FullAC      | 16.90  | 17.71  | 37.49   | 12.33       | 14.06       | 22.09        | 13.50     | 16.88     | 26.69       |
| CDvI        | 3.38   | 5.26   | 9.25    | 3.11        | 6.38        | 11.11        | 4.47      | 7.63      | 11.92       |
| correlation(x,y) |        |        |         |             |             |              |           |           |             |
| FullAC      | 16.71  | 17.96  | 36.87   | 12.25       | 14.02       | 21.73        | 13.46     | 16.98     | 26.53       |
| CDvI        | 3.43   | 5.36   | 9.34    | 3.17        | 6.42        | 11.20        | 4.55      | 7.72      | 11.93       |
| Time (seconds) |        |        |         |             |             |              |           |           |             |
| all-right-ov(x) |        |        |         |             |             |              |           |           |             |
| FullAC      | 17.65  | 53.12  | 126.95  | 22.31       | 47.34       | 85.40        | 22.00     | 55.54     | 126.54      |
| CDvI        | 6.55   | 14.52  | 31.87   | 8.09        | 15.12       | 23.78        | 9.10      | 20.17     | 43.68       |
| all-left-ov(y) |        |        |         |             |             |              |           |           |             |
| FullAC      | 22.21  | 62.42  | 151.04  | 30.91       | 64.00       | 101.98       | 30.12     | 81.44     | 182.89      |
| CDvI        | 8.46   | 19.27  | 41.21   | 10.55       | 19.46       | 29.74        | 11.05     | 24.31     | 51.84       |
| global-max-ov() |        |        |         |             |             |              |           |           |             |
| FullAC      | 8.07   | 9.84   | 10.70   | 7.39        | 7.62        | 6.63         | 8.16      | 9.64      | 10.56       |
| CDvI        | 2.16   | 2.59   | 2.93    | 2.08        | 2.30        | 2.22         | 2.52      | 2.92      | 3.30        |

4 Conclusions

Here, we introduced CDvI: to our knowledge, the first available compressed representation of an overlap index. It exploits a reduced version of Aho-Corasick automaton, and can store all (maximal and non maximal) overlaps between any pair of input words in linear space. CDvI offers multiple types of queries: on overlaps between two words, or between one word onto any other word, and a query for retrieving the pairs of words having the longest overlap over all possible pairs. Experiments showed the performance of CDvI both in terms of construction time, of space usage, and of querying times for different kinds of texts and alphabets. It outperforms a solution based on a version of the Aho-Corasick automaton. Future work include improving some query algorithms (e.g., max-ov(x,y)) to reach a linear complexity, creation of queries addressing a subset of the input words, or making CDvI dynamic to allow insertions/deletions of words without requiring reconstruction from scratch. We also consider studying how to implement and modify a recent work of G. Manzini [15], which may lower memory usage but increase query times. In the same line, it would be interesting to assess CDvI performance in large scale experiments, as in [23].

An interesting perspective is to use CDvI to implement (approximation) algorithms for the shortest superstring problem (and its variants), and to test their scalability and their ability to solve this difficult question for various inputs and alphabet sizes.

Finally, all implementations from this article are publicly available at https://github.com/rcanovas.
References

1. A. V. Aho and M. J. Corasick. Efficient String Matching: An Aid to Bibliographic Search. 
   *Communications of the ACM*, 18(6):333–340, 1975.
2. D. Arroyuelo, R. Cánovas, G. Navarro, and K. Sadakane. Succinct Trees in Practice. In 
   *Proc. Workshop on Algorithm Engineering & Experiments*, ALENEX’10, pages 84–97, 2010.
3. D. Benoit, E. Demaine, I. Munro, R. Raman, V. Raman, and S. Rao. Representing Trees of 
   Higher Degree. *Algorithmica*, 43(4):275–292, 2005.
4. S. Burkhardt, A. Crauser, P. Ferragina, H.-P. Lenhof, E. Rivals, and M. Vingron. q-gram 
   Based Database Searching Using a Suffix Array (QUASAR). In *3rd An. Int. Conf. Computational 
   Molecular Biology*, pages 77–83. ACM Press, 1999.
5. B. Cazaux, R. Cánovas, and E. Rivals. Shortest DNA cyclic cover in compressed space. In 
   *Data Compression Conference (DCC)*, pages 536–545. IEEE Computer Society Press, 2016.
6. D. Clark. *Compact Pat Trees*. PhD thesis, University of Waterloo, Ontario, Canada, 1998.
7. S. Gog, T. Beller, A. Moffat, and M. Petri. From Theory to Practice: Plug and Play with 
   Succinct Data Structures. In *13th International Symposium on Experimental Algorithms*, 
   (SEA 2014), pages 326–337, 2014.
8. R. González, S. Grabowski, V. Mäkinen, and G. Navarro. Practical Implementation of Rank 
   and Select Queries. In *Proc. 4th Workshop on Experimental Algorithms*, pages 27–38, 2005.
9. L.J. Guibas and A.M. Odlyzko. Periods in strings. *J. Comb. Theory A*, 30:19–42, 1981.
10. D. Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computa-
    tional Biology*. Cambridge University Press, 1997.
11. G. Jacobson. Space-efficient static trees and graphs. In *Proc. 30th Symposium on Founda-
    tions of Computer Science (FOCS)*, pages 549–554, 1989.
12. P. Jokinen and E. Ukkonen. Two algorithms for approximate string matching in static texts. 
    In A. Tarlecki, editor, *Proc. 16th Symp. Math. Found. Comp. Sci.*, number 520 in LNCS, 
    pages 240–248. Springer-Verlag, Berlin, 1991.
13. D. Knuth. The Art of Computer Programming, Volume 3: Sorting and Searching. Addison-
    Wesley, 1973.
14. P. Leopardi. Testing the Tests: Using Random Number Generators to Improve Empirical 
    Tests. In *Monte Carlo and Quasi-Monte Carlo Methods*, pages 501–512. Springer, 2009.
15. G. Manzini. XBWT tricks. In *Proc. 23th SPIRE*, pages 80–92, 2016.
16. I. Munro. Tables. In *Proc. 16th Foundations of Software Technology and Theoretical Com-
    puter Science (FSTTCS)*, pages 37–42, 1996.
17. I. Munro and V. Raman. Succinct representation of balanced parentheses and static trees. 
    *SIAM Journal on Computing*, 31(3):762–776, 2001.
18. S. Rahmann. Word statistics in random texts and applications to computational molecular 
    biology. Master’s thesis, Ruprecht-Karls Univ., Heidelberg, RFA, July 2000. 171 pages.
19. S. Rahmann and E. Rivals. Exact and Efficient Computation of the Expected Number of 
    Missing and Common Words in Random Texts. In *Proc. of the 11th Symp. CPM*, volume 
    1848 of LNCS, pages 375–387. Springer, 2000.
20. S. Rahmann and E. Rivals. The number of missing words in random texts. *Combinatorics, 
    Probability and Computing*, 12:73–87, 2003.
21. S. Robin and J.-J. Daudin. Exact distribution of word occurrences in a random sequence of 
    letters. *J. Appl. Probab*, 36:179–193, 1999.
22. R. Sedgewick and P. Flajolet. *Analysis of Algorithms*. Addison-Wesley, 1996.
23. J. T. Simpson and R. Durbin. Efficient construction of an assembly string graph using the 
    FM-index. *Bioinformatics*, 26(12):i367, 2010.
24. JT Simpson and R. Durbin. Efficient de novo assembly of large genomes using compressed 
    data structures. *Genome Research*, 22(3):549–556, 2012.
25. Esko Ukkonen. A Linear-Time Algorithm for Finding Approximate Shortest Common Superstrings. *Algorithmica*, 5:313–323, 1990.