Shortest DNA cyclic cover in compressed space

Bastien Cazaux, Rodrigo Cánovas and Eric Rivals

L.I.R.M.M. & Institut Biologie Computationnelle
Université de Montpellier, CNRS F-34392 Montpellier Cedex 5, France
cazaux@lirmm.fr, canovas-ba@lirmm.fr, rivals@lirmm.fr

Abstract

For a set of input words, finding a superstring (a string containing each word of the set as a substring) of minimal length is hard. Most approximation algorithms solve the Shortest Cyclic Cover problem before merging the cyclic strings into a linear superstring. A cyclic cover is a set of cyclic strings in which the input words occur as a substring. We investigate a variant of the Shortest Cyclic Cover problem for the case of DNA. Because the two strands that compose DNA have a reverse complementary sequence, and because the sequencing process often overlooks the strand of a read, each read or its reverse complement must occur as a substring in a cyclic cover. We exhibit a linear time algorithm based on graphs for solving the Shortest DNA Cyclic Cover problem and propose compressed data structures for storing the underlying graphs. All results and algorithms can be adapted to the case where strings are simply reversed but not complemented (e.g. in pattern recognition).

Introduction Computing a shortest superstring for a set of input words is a way to compress a language. Indeed, overlaps between the words allow to save characters. The Shortest Superstring problem is crucial in text compression. Unfortunately, its hardness prevents us from finding, not only an optimal, but even an approximate solution that is arbitrarily near to optimal [1, 2]. Rather than for a single linear superstring, it is thus better to opt for a compressed version that consists in a set of cyclic superstrings, whose total length is minimal. This representation is called a shortest cyclic cover. Indeed, the length of a shortest cyclic cover is smaller than that of a shortest linear superstring. These problems are related to DNA or genome assembly in bioinformatics: A highly redundant set of sequencing reads is given as input and one wishes to reconstruct the DNA sequence. As DNA is made of two complementary and reverse strands, a read can come from any strand, and thus either word or its reverse complement should occur in the superstring/genome. Moreover, chromosomes can be linear or cyclic molecules, and a sequenced sample may contain a mixture of cyclic genomes. We investigate the Shortest DNA Cyclic Cover problem and present a linear time algorithm for it. We also propose a compressed data structure for storing the underlying graphs used to solve the problem. All results described here can be adapted to the case where strings are simply reversed, and applied for instance to pattern matching of 2D shapes [3].

Outline First, we define the problem of finding a shortest DNA cyclic cover of a set strings and show the optimality of a `greedy` algorithm (Section 1). Then, we introduce a graph, the *Hierarchical Overlap Graph* (HOG), which encodes all the information of the Overlap Graph using linear space, and also present a variation of the HOG adapted to

1068-0314/16 $31.00 © 2016 IEEE
DOI 10.1109/DCC.2016.79

the DNA case. In the latter, nodes representing a word and its the reverse complementary sequence are glued together in a single node (Section 2). We then show that the Shortest DNA cyclic cover problem can be solved in linear time with this graph. Finally, we provide a construction algorithm for the HOG and a compressed representation that reduces the memory requirement (Section 3) before concluding.

1 Shortest DNA Cyclic Cover of Strings

1.1 Notation and problem definition

We introduce a notation for linear and cyclic strings/words.

Let Σ be a finite alphabet. Let $\Sigma^\star$ denote the set of finite (linear) strings over Σ, and ε denote the empty string. Let u,v,w,x be linear strings of $\Sigma^\star$. We denote by $|u|$ the length of u. If $x = uvw$ then u,v and w are all substrings of x, u is a prefix of x, and w a suffix of x. Furthermore, we say that v is a *proper* substring of x if $v \neq x$ (the definitions of a proper prefix/suffix are similar). An overlap from u over v is a linear string that is a proper suffix of u and a proper prefix of v. We denote by $ov(u,v)$ the longest overlap from u to v (also termed maximal overlap). Overlaps are not symmetrical. The *prefix from u to v*, denoted by $pr(u,v)$ is the string satisfying $u = pr(u,v)ov(u,v)$, while the *suffix from u to v*, denoted $su(u,v)$, is the substring of v satisfying $v = ov(u,v)su(u,v)$.

A DNA sequence is viewed as a string over $\Sigma = \{A,T,C,G\}$. As DNA is a double stranded molecule, any sequence also occurs as its reverse complement on the other strand, *i.e.* read from right to left and complemented ($A \leftrightarrow T$, $C \leftrightarrow G$). We denote the *reverse complement* of a string w by $\overleftarrow{w}$. This notion and the notation are extended to any finite set of strings. The operation of reverse complementing a string is the composition of an inversion with an order 2 permutation over Σ.

The following key property links the notions of maximal overlap and reverse complement.

Property 1 *Let u and v be two linear strings. Then,* $ov(\overleftarrow{u}, \overleftarrow{v}) = \overleftarrow{ov(v,u)}$.

Throughout the text, let $P := \{s_1, \ldots, s_n\}$ be a finite set of linear words over Σ. We term the norm of P the quantity $\sum_{s_i \in P} |s_i|$, and denote it by $\|P\|$. Without loss of generality, we assume that P is substring free: no input word is a substring of another.

Computing a superstring for a set words, which is a string that includes each word as a substring, is a key question in stringology and data compression, but also in bioinformatics where it relates to DNA assembly [4]. DNA assembly, when the initial words are "unstranded", consists in finding a superstring that includes either the word or its reverse complement for each input word. Therefore, it is interesting to design algorithms for finding superstrings and generalisation thereof for the DNA case (even without considering sequencing errors or mutations).

Superstrings obtained by merging words of P using their maximal overlaps [1] can be described by a permutation on $\{1,\ldots,n\}$ that for any s_i gives its successor in the superstring. Conversely, given a permutation one can easily compute the superstring *induced* by this

[1] Superstrings built using non maximal overlaps are disregarded because of their non optimality.

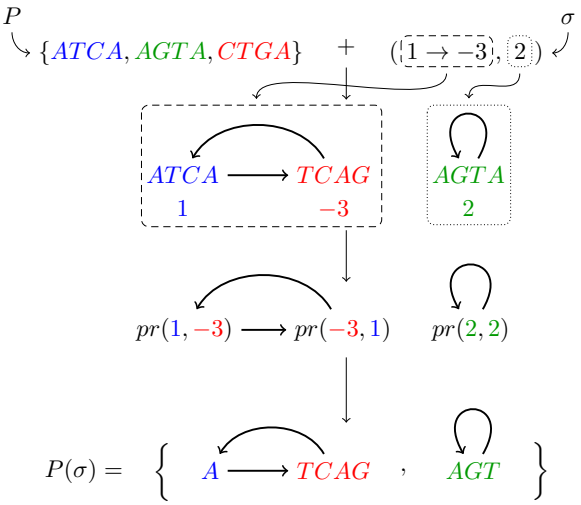

Figure 1: Example of permutation and cyclic cover for the input set $P := \{ATCA, AGTA, CTGA\}$. An instance of a DNA cyclic cover of P obtained with a permutation σ. We obtain the cyclic cover $P(\sigma) = \{ATCAG, AGT\}$, where $ATCAG$ and AGT are cyclic words. Note that this cyclic cover is not optimal.

permutation. Similarly, a cyclic superstring or a cyclic cover can also be described by a permutation.

Finding a shortest superstring of a set P is known to be difficult [1, 4], and approximation algorithms often use the slightly more general concept of *cyclic cover*. A *cyclic cover* is a set C of cyclic strings such that each word of P is a substring of a cyclic string of C. We introduce the variant for the DNA case, the *DNA Cyclic Cover*: it is a cyclic cover C such that for each s of P, there exists $c \in \mathcal{C}$ such that s or $\overleftarrow{s}$ is a substring of c. Let ***DCC*(P)** be the set of DNA cyclic covers for the strings of P. In the DNA case, the substring freeness hypothesis is extended to $P \cup \overleftarrow{P}$.
We introduce a variation of the Shortest Cyclic Cover problem adapted to the DNA case.

Problem 1 (Shortest DNA Cyclic Cover of Strings (SDCCS)) *Let P be a set of linear strings. Find C, a DNA cyclic cover of strings in P, minimising $\|C\|$.*

For the input P, we denote by **OPT(P)**, the set of optimal solutions for SDCCS.

1.2 Permutations for DNA cyclic covers

To describe a DNA cyclic cover by a permutation, we need a permutation on the set $\{-|P|, \ldots, -1, 1, \ldots, |P|\}$, where, a positive element i stands for s_i, while $-i$ represents $\overleftarrow{s_i}$. As a cyclic string for DNA can be read in both directions, the permutation describes the cyclic cover in both directions; a constraint enforces that in each direction either s_i or $\overleftarrow{s_i}$ is used in the induced DNA cyclic cover. Figure 1 shows an example for a cyclic cover (see definition below).

Example 1 *Consider $P := \{ATCA, AGTA, CTGA\} := \{s_1, s_2, s_3\}$. One can form the DNA cyclic cover $\{TCAGTA\}$ (with a single cyclic string) as the induced DNA cyclic cover of*

P of the permutation $\sigma := (1 \to -3 \to 2, -2 \to 3 \to -1)$. *This means merging* s_1 *with* $\overleftarrow{s_3}$ *with* s_2*, or equivalently merging* $\overleftarrow{s_2}$ *with* s_3 *with* $\overleftarrow{s_1}$. *The permutation* σ *can also be written as* $(\mathbf{2}, 3, -2, -\mathbf{3}, \mathbf{1}, -1)$*, where the numbers for the cycle* $1 \to -3 \to 2$ *appear in bold, and the others in italic.*

Clearly for an input P, the set of DNA cyclic covers that are induced by permutations (using maximal overlaps), which we denote by **PDCC(P)**, is a subset of all the DNA cyclic covers of P. Surely, an optimal DNA cyclic cover can be induced by a permutation. Hence, we get **OPT(P)** $\subseteq$ **PDCC(P)** $\subseteq$ **DCC(P)**.

1.3 Optimal and Greedy DNA cyclic covers

The classical Shortest Cyclic Cover of Strings (SCCS) has been investigated because of its central role in nearly all approximation algorithms for the Shortest Superstring Problem, starting with [2]. It can be solved in polynomial time by computing a minimum assignment on the distance graph in $O(\|P\| + |P|^3)$ [5]. However, a greedy algorithm also yields an optimal solution [6], for which a linear time version was recently proposed [7]. Algorithm 1 below gives the `greedy` algorithm for the SDCCS problem.

Algorithm 1: The `greedy` algorithm for SDCCS

Input: P a set of linear words **Output**: C a DNA cyclic cover of strings of P;
$C := \emptyset$
while *P is not empty* **do**
    u and v two elements of $P \cup \overleftarrow{P}$ which have the longest overlap (u can be equal to v but not to $\overleftarrow{v}$)
    w is the merge of u and v
    $P := P \setminus \{u, \overleftarrow{u}, v, \overleftarrow{v}\}$
    if $u = v$ *(*i.e. *w is a cyclic string)* **then** $C := C \cup \{w\}$ **else** $P := P \cup \{w\}$
return C

We denote by **GREEDY(P)**, the set of solutions of the `greedy` for SDCCS (see Algorithm 1). The `greedy` algorithm for SCCS and SDCCS are similar. Since the line of proof applied to SCCS remains valid for SDCCS, we obtain that **GREEDY(P)** $\subset$ **OPT(P)**. This proof relies on the properties of subset systems and on the Monge condition for string overlaps [6]. In summary, we obtain the following theorem, which implies that greedy DNA cyclic covers are optimal.

Theorem 1 (Figure 2) *For any set of linear strings P, we have:*

$$\textbf{\textit{GREEDY(P)}} \subseteq \textbf{\textit{OPT(P)}} \subseteq \textbf{\textit{PDCC(P)}} \subseteq \textbf{\textit{DCC(P)}}$$

2 Hierarchical Overlap Graph

In the sequel, we use graph and data structures in which each node is associated with a string. For simplicity, we confound the node and the string it represents.

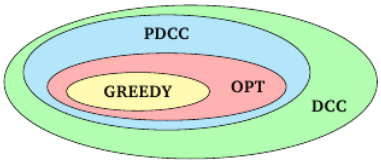

Figure 2: Inclusions of sets of solutions for SDCCS into the set of DNA Cyclic Covers.

The Shortest Superstring problem can be solved by computing a maximum weight Hamiltonian path in the *Overlap Graph* [2].

Definition 1 (Overlap Graph) *The* Overlap Graph *(OG) of P is a complete, oriented graph, weighted on its arcs, whose nodes are the strings of P, and in which the weight of an arc* (u,v) *equals the length of the maximal overlap from string u to string v.*

The overlap graph for our running example is shown on Figure 4.

The number of arcs in the OG is quadratic in the number of words. However, Gusfield *et al.* have exhibited an algorithm for computing the maximal overlaps in $O(\|P\|)$ time, plus $O(n^2)$ time for outputting the weights [8]. However, the OG does not take advantage from the fact that several words can share the same or similar overlaps. A clear improvement would be a graph summarising the same information than OG, but would take $O(\|P\|)$ space. For this sake we propose the *Hierarchical Overlap Graph*.

2.1 Hierarchical Overlap Graph

Consider a Generalised Suffix Tree (ST) for P equipped with Suffix Links (SL) (see [4] for the definition and properties of Suffix Trees, including their generalised version). For practical reasons, we extend the notion of SL to leaves of the ST. To ease the distinction, we say that the suffix links are red, while arcs of the ST are blue. Consider two leaves representing say s_i and s_j for some $1 \leq i,j \leq n$. The maximal overlaps of s_i over s_j is a suffix of s_i and a prefix of s_j: it is a repeated string within P and thus an explicit node that is an ancestor of s_j in the ST. Hence, there exists a chain of suffix links linking the leaf s_i to the node representing $ov(s_i,s_j)$, while there exists a blue chain from $ov(s_i,s_j)$ to s_j. Note that this property remains valid for any two nodes in the ST. We call this path the *Red-Blue Path* between two nodes.

Let $Ov(P)$ be the set of maximum overlaps from a string of P to another string or the same string of P. Let us define the *Hierarchical Overlap Graph* as follows (see an illustration for our running example in Figure 4).

Definition 2 *The* Hierarchical Overlap Graph *of P, denoted by* $HOG(P)$*, is the oriented graph* $(P \cup Ov(P), R \cup B)$ *where:*

$$R = \{(x,y) \in (P \cup Ov(P))^2 \mid y \textit{ longest suffix of } x \textit{ in } V\},$$

$$B = \{(y,x) \in (P \cup Ov(P))^2 \mid y \textit{ longest prefix of } x \textit{ in } V\}.$$

The HOG is reminiscent of the "Hierarchical Graph" of [9], where the arcs also encodes inclusion between substrings of the input words. However, in the Hierarchical Graph, each

arc extends or shortens the string by a single symbol, making it larger than the HOG. The HOG is in fact embedded in the ST of P, but successive blue arcs may be contracted in the HOG (the same for red arcs). We will see in Theorem 3 on page 8 that $HOG(P)$ takes linear space in $\|P\|$.

Property 2 *Let u and v be two strings of P. In the HOG, there exists a Red-Blue path from u to v such that the largest node that is an ancestor of u and v is the overlap from u to v. The length of this path is at most* $|pr(u,v)| + |su(u,v)|$.

Since, the HOG is embedded in the Generalised Suffix Tree of P, we obtain the following space bound.

Corollary 1 *The total length of Red-Blue paths on the HOG is linear in* $\|P\|$. *The total length of the Red-Blue cycles used in a cyclic cover is linear in the norm of the cyclic cover.*

2.2 Glued Hierarchical Overlap Graph

Now, we want to find a solution of *SDCCS* and we are considering the graph $HOG(P \cup \overleftarrow{P})$ (not simply $HOG(P)$).

Because we considered for each word its normal sequence and its reverse complementary sequence, a sought superstring must include at least one of these. For each input word, we take into account the two forms of the word, and the corresponding nodes in the OG, forming a class. The OG is partitioned into n classes, where n is the number of words [6]. Hence, a "DNA" superstring corresponds to a partitioned path in the OG, while a DNA cyclic cover is a partitioned cyclic cover in the OG.

Let X be a set such that $X \cup \overleftarrow{X} = (P \cup \overleftarrow{P}) \cup Ov(P \cup \overleftarrow{P})$. X can be seen as a quotient, *i.e.* the set of representatives, of the equivalence relation of the reverse complement: Two different elements are in the same equivalence class, if the first is the reverse complement of the second. In the sequel, we take X as the set of minimal lexicographically strings between w and $\overleftarrow{w}$ for all strings of $(P \cup \overleftarrow{P}) \cup Ov(P \cup \overleftarrow{P})$. Note that the construction can be adapted to the case where X is another quotient of the equivalence relation of the reverse complement.

For a string w of $(P \cup \overleftarrow{P}) \cup Ov(P \cup \overleftarrow{P})$, we set $a(w)$ as the representative element of X of the string w, and $d(w)$ as the element of $\{-1,0,1\}$ such that, $d(w) = 0$ if $w = \overleftarrow{w}$, $d(w) = 1$ if $w \in X$, or $d(w) = -1$ if $w \notin X$.

Definition 3 (Glued Hierarchical Overlap Graph) *The* Glued Hierarchical Overlap Graph *of P, denoted by* $GHOG(P)$, *is the undirected graph* (V,E,l) *labeled on its edges, such that* $V = X$ *and for all* $(u,v) \in B$, $(a(u),a(v)) \in E$ *and* $l((a(u),a(v))) : \begin{cases} a(u) \mapsto d(u) \\ a(v) \mapsto d(v) \end{cases}$ *where B is the set of blue arcs of* $HOG(P \cup \overleftarrow{P})$.

The Glued Hierarchical Overlap Graph for our example is shown on Figure 4. By Property 1, there exists a bijection from B onto R, *i.e.* between the set of blue arcs and the set of red arcs of $HOG(P \cup \overleftarrow{P})$. Hence, in Definition 3, we only consider arcs of B to

define the edges of the GHOG(P). The function l applies to edges of GHOG(P), and $a(u)$ tells whether or not one needs to complement the string u. In other words, $d(u)$ records of the original string we had when entering the edge: if $d(u) = 1$ then u represents $a(u)$, if it equals -1 then u is equal to $\overleftarrow{a(u)}$.

We define the *RF-path* between two elements of $(P \cup \overleftarrow{P}) \cap X)$ as the smallest path $p = (e_1 = (u_1, u_2), \ldots, e_m = (u_m, u_{m+1}))$ of the $GHOG(P)$ such that there exists i between 2 and m such that:

- for all $j < i$, $|u_j| \geq |u_{j+1}|$ and $l((u_{j-1}, u_j))(u_j) = l((u_j, u_{j+1}))(u_j)$,
- for all $j \geq i$, $|u_j| \leq |u_{j+1}|$ and $l((u_{j-1}, u_j))(u_j) = l((u_j, u_{j+1}))(u_j)$,
- $l((u_{i-1}, u_i))(u_i) = -l((u_i, u_{i+1}))(u_i)$.

An example of a RB-path in the HOG and a RF-path in the GHOG are shown on Figure 3.

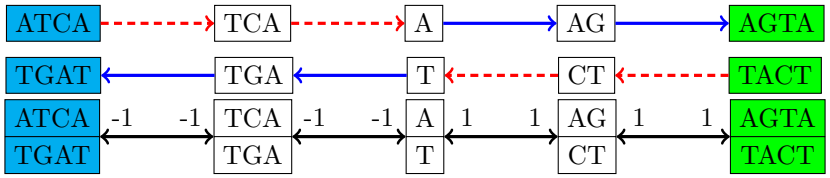

Figure 3: An example of RB-path in the HOG and a RF-path in the GHOG.

Theorem 2 *There exists an algorithm that exactly solves SDCCS in linear time in $\|P\|$.*

Proof 1 *Theorem 1 states that the* `greedy` *algorithm solves exactly SDCCS. The idea is to build a greedy solution for SDCCS in linear time in $\|P\|$ using the $GHOG(P)$. To achieve that we adapt the* Superstring Graph *designed for solving the SCCS problem to the DNA case [7]. In fact the Superstring graph is a subgraph of the HOG in which each arc has a multiplicity that counts the number of traversal for that arc. The multiplicity are set to make the Superstring Graph Eulerian. Its main property is to encode all greedy solutions in linear space. For SCCS, paths in the Superstring Graph are Red-Blue Paths between the leaves representing the input words (*i.e. *the s_i for $1 \leq i \leq n$). For SDCCS, the RF paths replace the Red-Blue paths in the Superstring Graph built on the GHOG.*

3 Data structures and construction algorithms for the GHOG

3.1 Representing HOG(P) using a compressed Suffix tree

Consider a Generalised Suffix Tree (ST) for P equipped with Suffix Links (SL) [4]. As previously explained, for maximal overlaps between two words of P, each element of $Ov(P)$ is an explicit, internal node of the ST of P. Hence, $P \cup Ov(P)$ takes a space that is linear in $\|P\|$. Now, as both $(P \cup Ov(P), B)$ and $(P \cup Ov(P), R)$ are trees, we get that $HOG(P)$ is also linear in $\|P\|$. Gusfield *et al.* gave a characterisation of overlap nodes in the ST of P, with which they solved the all pairs suffix-prefix problem [8]. With this characterisation, one can mark in the ST of P the nodes that belong to $HOG(P)$ in linear time in $\|P\|$.

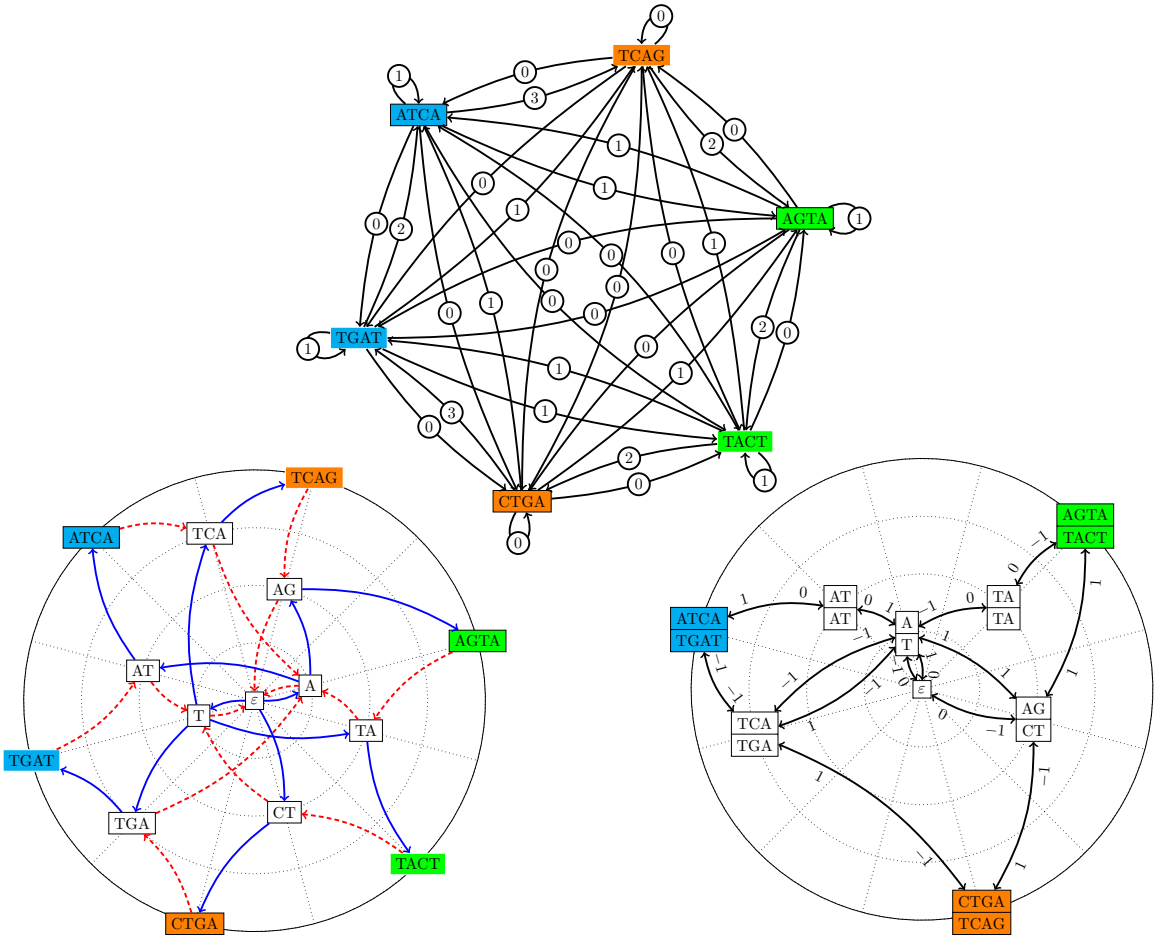

Figure 4: Running example with $P := \{ATCA, AGTA, CTGA\} := \{s_1, s_2, s_3\}$. We have $\overleftarrow{P} = \{TGAT, TACT, TCAG\}$. The figure shows for $P \cup \overleftarrow{P}$ – *i.e.* in DNA case – (Top) the Overlap Graph, (bottom left) the Hierarchical Overlap Graph, and (right) the Glued Hierarchical Overlap Graph. In the HOG and GHOG, input words appear on the outer circle, and their substrings are ordered on inner circles according to their length, with the empty word ε at the centre. In the HOG, one sees the Red-Blue paths, which become RF paths in the GHOG. In the GHOG, the -1, 1 or 0 labels store whether the traversal implies a change in the DNA strand or not.

Theorem 3 *The graph $HOG(P)$ can be built in linear time in $\|P\|$. The algorithm uses $O(\|P\|)$ space, and finally $HOG(P)$ occupies a space that is linear in $min(|P|^2, \|P\|)$.*

The Suffix Tree occupies large space in practice. Currently, several implementations of Compressed Suffix Tree (CST) data structures exist that support all classical operations including Suffix Links, and occupy between 5 and 12 bits per symbol [10, 11]. These implementations offer an advantageous trade off between storage and efficacy of operations. Using a CST and an additional membership table that marks which nodes belongs to the HOG, it is possible to emulate the traversal of blue and red arcs between nodes of the HOG. Remind that a blue arc is path of one or more arcs of the ST dictated by the labels, while a red arc is a chain of one or more SL starting from a node.

Let v be a node of CST(P) that belongs to HOG(P). To emulate a red arc starting from v, we iteratively traverse the chain of SL until we reach a marked node. At most this ends up at the root, taking in the worst case, $|v| \times \text{complexity}(SL)$ time.

Now to traverse a blue arc of the HOG using a label, we iteratively use the operation Child of v with the given label. If the node reached belongs to HOG(P) then the path is valid, taking in the worse case, $|v| \times \text{complexity}(Child)$ time.

To obtain all possible outgoing blue arcs from v, we first check whether there exist marked nodes in its subtree and interrupt the search if there are none (we assume that checking the existence of marked nodes within a sub-tree can be done in constant time). Otherwise, we compute the Children of v in the ST and output those children that belong to the HOG. For the remaining children of v, we iterate the process in their subtree until all branches have been considered. This procedure stops at most when it reaches a leaf representing a word of P. It takes $((\max_{1\leq i\leq n}|s_i|) - |v|) \times \text{complexity}(Children)$ time.

Alternative solution To encode HOG(P), one could use either a bi-directional wavelet index [12, 13], which is a compressed representation of the affix array [14]. This would almost double the space compared to a single CST, since the bi-directional wavelet index stores the trees for P and for the reverse of P. However, it would simplify the traversal of a red arc, since we could use the parent function on the reverse of P. Note that because the Superstring graph is a subgraph of the HOG, it can also be emulated on top of the compressed data structure used for $HOG(P)$.

3.2 Representing GHOG(P)

To represent GHOG(P) we need to build the HOG for the set $P \cup \overleftarrow{P}$ and to glue pairs of substrings that are the respective reverse complement of each other. Each node encodes the lexicographically smallest substring of the pair, but represents both of them. Above, we detail how to build and emulate the HOG using a CST. To emulate GHOG(P) with the same data structure, we propose to add a new operation called Dlink (for DNA link), which links a node/string to the node representing its reverse complementary string. A Dlink is needed only for the nodes of GHOG(P) (not for all the ST nodes); this can be stored either using a permutation of the nodes, or by traversing the CST. Using the CST structure, Dlink and the marked nodes, one can emulate all the edges going out of a node of GHOG(P). Determining whether a string is the representative of its node is done by comparing it once with its reverse complement using Dlink, and the result is recorded in an array. This procedure also yields the edge's label (*i.e.* in $\{-1,0,1\}$). One traversal of the GHOG is enough to compute the representative string for each node and all labels.

Theorem 4 *The graph $GHOG(P)$ can be built in linear time in $\|P\|$ using a Compressed Suffix Tree data structure.*

4 Conclusion and perspectives

DNA cyclic covers are a natural generalisation of cyclic covers to handle the double strandedness of DNA. We propose a linear time algorithm for finding the Shortest DNA Cyclic Cover of a set of words using compressed data structures. In the general case, shortest

cyclic covers are used to compute approximate shortest superstrings. Hence, our work opens the way to approximation algorithms for linear or cyclic shortest superstrings in the DNA case. Currently for the DNA case, computing the Overlap Graph takes too much space. The HOG and GHOG reduce the memory needed and are thus valuable alternatives. Our algorithm to build these graphs with compressed data structures is another step further in term of improved memory requirement.

Our proposal is flexible in several aspects. The graph used to encode the greedy solutions to the problem can be adapted to the cases where one imposes thresholds on the overlap lengths. With real sequencing data, an overlap must be long enough compared to the word size to be considered as reliable. Moreover, the graph could incorporate information on the multiplicity of input words, *i.e.* the minimal number of occurrences each word should have in the cyclic cover. Of course, extensions of our algorithms to approximate overlaps is a challenging perspective. Both graphs, the HOG and GHOG, which we introduced to compute cyclic covers, can be updated dynamically in linear time. Conceiving re-optimisation algorithms, which can recompute a solution once the instance has been modified, for example after adding or deleting a word is motivating line of research.

Acknowledgements: This work is supported by ANR Colib'read (ANR-12-BS02-0008) and Défi MASTODONS SePhHaDe from CNRS.

References

[1] J. Gallant, D. Maier, and J. A. Storer, "On finding minimal length superstrings," *Journal of Computer and System Sciences*, vol. 20, pp. 50–58, 1980.

[2] A. Blum, T. Jiang, M. Li, J. Tromp, and M. Yannakakis, "Linear approximation of shortest superstrings," in *ACM Symposium on the Theory of Computing*, 1991, pp. 328–336.

[3] H. Bunke and U. Buehler, "Applications of approximate string matching to 2D shape recognition," *Pattern Recognition*, vol. 26, no. 12, pp. 1797–1812, 1993.

[4] D. Gusfield, *Algorithms on Strings, Trees and Sequences*. Cambridge Univ. Press, 1997.

[5] C. H. Papadimitriou and K. Steiglitz, *Combinatorial optimization : algorithms and complexity*, 2nd ed. Dover Publications, Inc., 1998, 496 p.

[6] B. Cazaux and E. Rivals, "The power of greedy algorithms for approximating Max-ATSP, Cyclic Cover, and superstrings," *Discrete Applied Mathematics*, p. doi:10.1016/j.dam.2015.06.003, 2015.

[7] ——, "A linear time algorithm for Shortest Cyclic Cover of Strings," LIRMM Research report, p. 12, Feb. 2015, hal.lirmm.fr.

[8] D. Gusfield, G. M. Landau, and B. Schieber, "An efficient algorithm for the all pairs suffix-prefix problem," *Inf. Proc. Letters*, vol. 41, no. 4, pp. 181–185, 1992.

[9] A. Golovnev, A. S. Kulikov, and I. Mihajlin, "Solving SCS for bounded length strings in fewer than 2^n steps," *Inf. Proc. Letters*, vol. 114, no. 8, pp. 421–425, 2014.

[10] K. Sadakane, "Compressed suffix trees with full functionality," *Theory Comput. Syst.*, vol. 41, no. 4, pp. 589–607, 2007.

[11] G. Navarro and L. M. S. Russo, "Fast fully-compressed suffix trees," in *Data Compression Conference, DCC 2014*, 2014, pp. 283–291.

[12] T. Schnattinger, E. Ohlebusch, and S. Gog, "Bidirectional search in a string with wavelet trees and bidirectional matching statistics," *Inf. Comput.*, vol. 213, pp. 13–22, 2012.

[13] T. W. Lam, R. Li, A. Tam, S. C. K. Wong, E. Wu, and S. Yiu, "High throughput short read alignment via bi-directional BWT," in *Proc. IEEE BIBM*, 2009, pp. 31–36.

[14] D. Strothmann, "The affix array data structure and its applications to RNA secondary structure analysis," *Theor. Comput. Sci.*, vol. 389, no. 1-2, pp. 278–294, 2007.