A linear time algorithm for Shortest Cyclic Cover of Strings

Bastien Cazaux, Eric Rivals

Journal of Discrete Algorithms 37 (2016) 56–67

Contents lists available at ScienceDirect

Journal of Discrete Algorithms

www.elsevier.com/locate/jda

A linear time algorithm for Shortest Cyclic Cover of Strings

Bastien Cazaux [a,b], Eric Rivals [a,b,*]

[a] LIRMM, CNRS and Université de Montpellier, 161 rue Ada, 34095 Montpellier Cedex 5, France
[b] Institut de Biologie Computationnelle, CNRS and Université de Montpellier, 860 rue Saint Priest, 34095 Montpellier Cedex 5, France

ARTICLE INFO

Article history:
Available online 19 May 2016

Keywords:
Greedy
Shortest cyclic cover of strings
Superstring
Overlap
Minimum assignment
Graph

ABSTRACT

Merging words according to their overlap yields a superstring. This basic operation allows to infer long strings from a collection of short pieces, as in genome assembly. To capture a maximum of overlaps, the goal is to infer the shortest superstring of a set of input words. The Shortest Cyclic Cover of Strings (SCCS) problem asks, instead of a single linear superstring, for a set of cyclic strings that contain the words as substrings and whose sum of lengths is minimal. SCCS is used as a crucial step in polynomial time approximation algorithms for the notably hard Shortest Superstring problem, but it is solved in cubic time. The cyclic strings are then cut and merged to build a linear superstring. SCCS can also be solved by a greedy algorithm. Here, we propose a linear time algorithm for solving SCCS based on a Eulerian graph that captures all greedy solutions in linear space. Because the graph is Eulerian, this algorithm can also find a greedy solution of SCCS with the least number of cyclic strings. This has implications for solving certain instances of the Shortest linear or cyclic Superstring problems.

1. Introduction

The possibility of merging two words into a longer one according to their overlap – for instance merging *abcde* with *cdeba* into *abcdeba* – allows to infer the sequence of a target molecule from the short reads produced by sequencing machines. In computer science, merging results in a *superstring* of the input words. The action of merging is heavily used in any genome assembler (although not necessarily on the complete reads, sometimes only on substrings of the reads) and given the redundancy of the sequencing, which yields a high density of reads, the objective is to compute a shortest superstring of a set of input words [10]. In bioinformatics, the task of genome assembly is worsened by the inaccuracy of the sequencing and by the structure of the genome which contains repeated sequences [10]. However, progresses on the model question, called *Shortest (linear) Superstring Problem* (SLiS) – or often Shortest Common Superstring – still shed light on the difficulty of assembly. Moreover, the target molecule can be a linear or a circular structure. It is also interesting to address the question of the *Shortest Cyclic Superstring* (SCyS). Finding superstrings also has application in data compression. Indeed, as a superstring represents a condensed representation of the initial language, it can serve as a compressed version of the input set.

Let $P := \{s_1, \ldots, s_{|P|}\}$ be a set of input words and denote the sum of their lengths by $\|P\|$, *i.e.* the norm of P. Without loss of generality, we always assume that P is factor free, *i.e.* for any two strings of P, none is a substring of the other. From the computational viewpoint, finding a shortest superstring is known to be hard to solve (NP-hard even on a binary

* Corresponding author at: LIRMM, CNRS and Université de Montpellier, CC477, 161 rue Ada, 34095 Montpellier Cedex 5, France. Tel.: +33 4 67 41 86 64.
E-mail addresses: Bastien.Cazaux@lirmm.fr (B. Cazaux), rivals@lirmm.fr (E. Rivals).

http://dx.doi.org/10.1016/j.jda.2016.05.001
1570-8667/

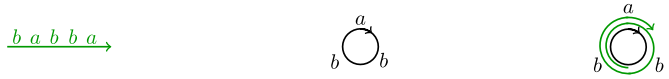
alphabet) and to approximate (Max-SNP hard) [7,1]. The research in the last 25 years has mainly focused on polynomial time approximation algorithms for SLiS (see [8] for a recent list). Despite these efforts, the algorithm `greedy` (Algorithm 1), an algorithm that iteratively updates the input set by merging two maximally overlapping strings until one string is left, was conjectured to approximate the length of the shortest superstring with ratio 2 [17,1]. This is better than most known algorithms, and the conjecture is still valid. For linear superstrings on the constant size alphabets, Ukkonen proposed a linear time implementation of the algorithm `greedy` based on the Aho–Corasick automaton [19]; later, another implementation using the suffix tree was given in [12,13].

The question of approximating the superstring can be reformulated on a complete weighted digraph, the so-called *distance graph*: each input word is a vertex and an arc encodes the directional merging of the two words. The weight of an arc from u to v equals the length of the prefix of u that is not covered by the overlap of v (*e.g.*, the corresponding prefix of *abcde* to *cdeba* is *ab*, and the weight would be 2). Then a shortest linear superstring is associated with a Hamiltonian path on this graph, and a shortest cyclic superstring with a Hamiltonian cycle [1]. Yet those problems are also hard to approximate and most approximation algorithms for SLiS resort to a relaxed question, namely *Shortest Cyclic Cover of Strings* (SCCS). These algorithms solve SCCS by computing a minimum assignment on the distance graph in $O(\|P\| + |P|^3)$ [16], and then transform the cycles into cyclic strings. Our work focuses on SCCS. We propose an algorithm that uses only linear time in $\|P\|$, and for this sake introduce the superstring graph, which captures all solutions of `greedy` for SCCS (see Sections 4 and 3). These greedy solutions are called *greedy superstrings*. Exploring the graph properties, we show that it gives, for a subset of instances of the shortest linear and cyclic superstring problems, interesting approximations or even exact solutions (Section 4).

1.1. Notation and problem definition

For any rooted tree T, V_T is the set of nodes of T. For a node v of V_T, $Children_T(v)$ is the set of children of v in T. We denote by $[i, j]$ the set of integers between i and j.

We consider two kinds of strings: *linear* and *cyclic strings* (see Figs. 1a and 1b). For a string s, the length of s is $|s|$. For a linear string s and $i \leq j$ in $[1, |s|]$, $s[i, j]$ is the linear substring of s beginning at the position i and ending at the position j, $s[i]$ is the substring $s[i, i]$, $s[1, j]$ is a prefix of s and $s[i, |s|]$ is a suffix of s. A prefix (or suffix) s' of s is proper if s' is different of s. For another linear string t, the overlap from s to t, denoted by $ov(s, t)$, is the longest substring which is a proper suffix of s and a proper prefix of t. The prefix from s to t, denoted by $pr(s, t)$, is such that $s = pr(s, t)ov(s, t)$. The *merge* of s with t is the linear word $pr(s, t)t$ if $s \neq t$, and the cyclic word $pr(s, t)$ otherwise. A *linear superstring* of a set P of linear strings is a linear string w such that each string of P is a substring of w. We say that a linear string w is a substring of a cyclic string c if there exists w_c a linear permutation of c such that w is a substring of w_c^∞ (where $w_c^\infty = w_c w_c \ldots$) – see Fig. 1c. A *cyclic cover* of P is a set C of cyclic strings such that each linear string of P is a substring of a cyclic string of C.



(a) linear word (b) cyclic word (c)

Fig. 1. (c) Shows how the linear word (a) is included in the cyclic word (b).

We define the following problem:

Problem 1 *(Shortest cyclic cover of strings).*

Input: A set of linear strings P.
Output: A cyclic cover C of P that minimises $\|C\|$.

Let σ be a permutation of $\{1, \ldots, |P|\}$, we can split up σ in cyclic permutations: $\sigma = \sigma_1 \circ \ldots \circ \sigma_m$. For each cyclic permutation σ_i of length n_i, we denote the cyclic string by:

$$P(\sigma_i) := pr(s_{\sigma_i(1)}, s_{\sigma_i(2)}) \ldots pr(s_{\sigma_i(n_i-1)}, s_{\sigma_i(n_i)})\ pr(s_{\sigma_i(n_i)}, s_{\sigma_i(1)}),$$

and

$$P(\sigma) := \{P(\sigma_1), \ldots, P(\sigma_m)\}\ (\text{See Fig. 2}).$$

It is easy to see, that for each permutation σ of $\{1, \ldots, |P|\}$, $P(\sigma)$ is a cyclic cover of P and for each optimal solution w of Problem 1 (SCCS), there exists σ a permutation of $\{1, \ldots, |P|\}$ such that $w = P(\sigma)$.

1.2. Related works

Let us briefly recall some crucial works in the study of shortest superstrings and of Shortest Cyclic Cover of Strings.

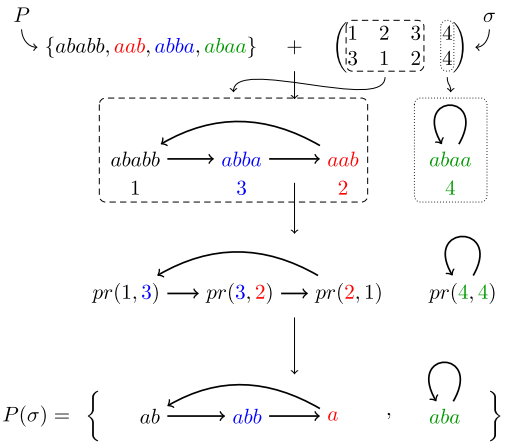


Fig. 2. Running example with the input set $P := \{ababb, aab, abba, abaa\}$. Instance of a cyclic cover of P obtained with a permutation σ. We obtain the cyclic cover $P(\sigma) = \{ababba, aba\}$ ($ababba$ and aba are cyclic words).

Since SLiS does not admit a PTAS, numerous works on SLiS have reported polynomial time approximation algorithms with a constant ratio ranging from 4 to > 2 (see [8, Table 1]). A majority of these model SLiS as a minimum weight Hamiltonian path problem on the distance graph G of P (where an arc (s_i, s_j) is weighted by $|pr(s_i, s_j)|$ for any valid i, j). Note that if instead it is weighted by $|ov(s_i, s_j)|$, we obtain the *Overlap Graph*, and SLiS becomes equivalent to finding a maximum weight Hamiltonian path [1]. Clearly, the weight of a cyclic cover of G (*i.e.*, the sum of the weight of all cycles) is lower than that of a Hamiltonian cycle, which itself gives a lower bound of the weight of a Hamiltonian path on G. Hence, most approximation algorithms for SLiS rely on solving SCCS, which is polynomial. Blum et al. [1] were the first to exhibit such an algorithm and to prove that it achieves a constant ratio of 3 for SLiS. This algorithm first solves SCCS and in a second step combines the cyclic strings by running a greedy algorithm on them. Teng and Yao improved this ratio by an algorithm that recursively solves SCCS [18]. After that, many authors have investigated the combinatorial properties of words and of some specific cycles, which enabled them to further improve the approximation (*e.g.* [3,14,15]), but all need to solve SCCS. To do so, they compute a *Minimum Assignment* on the distance graph. An assignment (or cyclic cover) of a digraph G is a set of cycles such that every vertex of G belongs to exactly one cycle. To solve SCCS on an instance P, approximation algorithms for SLiS first 1/ build the distance graph of P, with $|P|$ nodes and $|P|(|P| - 1)$ arcs, which requires $O(\|P\| + |P|^2)$ time and space using the Generalised Suffix Tree (GST) of P [11], then 2/ compute a minimum cyclic cover using the Hungarian algorithm in $O(|P|^3)$ time [16]. Once the cyclic cover is known, the algorithm in [18] also requires to compute a particular assignment called a canonical assignment, which takes an additional $O(\|P\|)$ using the suffix tree based algorithm of [9]. However, the complexity bottleneck of algorithms using a cyclic cover remains the $O(\|P\| + |P|^3)$ time to build the graph and compute the cover. It was also shown that `greedy` (Algorithm 1) solves SCCS exactly [4]. To the best of our knowledge, there is currently no linear time algorithm for solving SCCS.

Algorithm 1: The algorithm `greedy` for Shortest Cyclic Cover of Strings.

```
Input: P a set of linear words;
Output: S a set of cyclic strings that cover P;
S := ∅
while P is not empty do
    u and v two elements of P having the longest overlap (u can be equal to v)
    w is the merge of u and v
    P := P \ {u, v}
    if u = v (i.e. w is a cyclic string) then S := S ∪ {w} else P := P ∪ {w}
return S
```

Theorem 1 *([6]). For Shortest Cyclic Cover of Strings, Algorithm `greedy` yields an optimal solution.*

1.3. Explanations about algorithm `greedy`

An overlap is said to be *greedy-feasible* if a greedy solution takes it. Let u_1, u_2, v_1 and v_2 be four strings. We say that the overlaps $ov(u_1, u_2)$ and $ov(v_1, v_2)$ are *dependent* if and only if $u_1 = v_1$ or $u_2 = v_2$. When Algorithm `greedy` merges u with v, it chooses the overlap between u and v and forbids any overlap beginning by u or ending by v, *i.e.* it forbids all overlaps that are dependent of $ov(u, v)$. We say that a set of overlaps is *independent* if each overlap is not dependent of another.

Thus, Algorithm `greedy` makes a series of choices to select a sequence of greedy-feasible and independent overlaps:

- In the first step of the Algorithm `greedy` chooses an overlap among the longest overlaps.
- In the second step, it chooses an overlap among the longest overlaps which are independent of the first overlap.
- At each further step, it takes a maximum overlap that is independent of the previously selected overlaps.
- In the end, it outputs a set of overlaps that are maximal (for the inclusion as a base in a matroid). This set of overlaps is independent.

As the set of overlaps at the end of `greedy` is maximal, there exists a permutation σ of $\{1,\dots,|P|\}$, such that for all i in $\{1,\dots,|P|\}$, $ov(s_i, s_{\sigma(i)})$ is an overlap of our set of overlaps. Thus, we obtain the following property which is similar to a well-known property on the solution of the greedy algorithm for SLiS [20, Def. of Induced (π, S) and Def. 2]:

Proposition 2. *Let w a greedy solution of SCCS. There exists a permutation σ of $\{1,\dots,|P|\}$ such that $w = P(\sigma)$.*

2. Generalised Suffix Tree, overlap, merge, and Red–Blue paths

The suffix tree of a word w is a data structure that indexes all substrings of w. Each arc is labelled with a substring of w such that the concatenation of the labels along a branch from the root to a leaf spells out a suffix of w. A constraint requires that the labels of arcs from a node to its offspring start with different symbols. Thus, each node represents a prefix of a suffix, *i.e.* a substring of w. In fact, each internal node is the longest common prefix of all leaves in its subtree. Traversing an arc extends the represented string on its right. The *word depth* of a node is the length of the string it represents. An example of (generalised) suffix tree is shown in Fig. 3: the sole suffix tree structure is displayed in Fig. 3a. For instance, the node 3 represents the string *aa*. Going from node 3 to node 1 extends the string represented by a *b* to yield the string *aab*.

ST are equipped with additional arcs called *Suffix Links* (SL): a SL links a node representing au (with a a letter and u a string) to the node representing u. Let us label the SL by the letter a. The notion of SL is usually restricted to internal nodes, but it extends naturally to leaves. We consider that each node has its SL. Traversing a SL removes the first letter of the word: it shortens the represented word at the front. To each SL, we associate a label, which is the letter it removes. For instance, in Fig. 3, traversing the SL of node 1 shortens the word *aab* to *ab*, which is the string represented by node 2. The label of that SL is *a* (labels of SL are not represented in this figure). It is known that, if SLs are reversed (considered in the other direction), they form a tree rooted in the root of the ST, and all leaves are on the same branch of this tree [5]. Note that the chain of SL starting from any node goes up to the root of T (For simplicity, assume the root has a SL to itself). The tree composed by the suffix links is displayed on Fig. 3b. Moreover, in the figures, the SL are drawn as an arc from a child to its parent in L (see Fig. 3).

For simplicity, we say that a suffix tree arc is *blue*, while SL are *red*. So, we have the blue tree, denoted T, and the red tree, denoted L, on the same nodes.

The GST generalises this principle to several words. From now on, the generalised suffix tree of P is the pair (T, L). An extensive introduction to GST and their applications can be found in [10]. The distinction between explicit and implicit nodes is skipped here, but explained in [10,2].

2.1. Overlaps

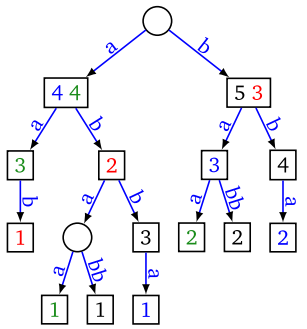
Basically, all suffixes and all substrings of words of P are represented in the GST. If a word u overlaps a word v, then a suffix of u equals to a prefix of v. Hence, one expects the overlap to be represented somewhere in the GST. Indeed, if an internal node representing the suffix of u appears on the branch spelling v, then u overlaps v. This internal node exists since this substring occurs at least twice (in u and in v), and represents the overlap. Let us call such node an *overlap node* from u to v. In our representation, the internal node is labelled with the starting position of that suffix in u on the branch leading to the leaf of v. For example, in Fig. 3 the node 2 is located on the branch of 1. It is also on the branches of 1 and 1. It represents the string *ab*. So we know that *aab* overlaps *abba*, *abaa* and *ababb*, with the overlap *ab*.

Clearly, all overlap nodes from u to v are ancestors of another. Moreover, the word depth of an overlap node gives the length of the corresponding overlap. Hence, the maximum overlap between two words is the deepest overlap node of those words. For instance, both nodes 4 and 3 are ancestors of 1 meaning that *abaa* overlaps *aab* with two possible overlaps: *a* and *aa*, of length 1 and 2 respectively.

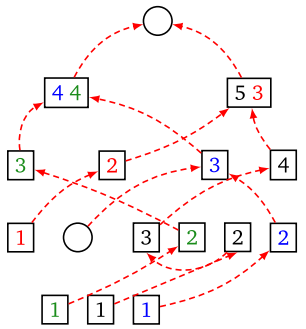
All these notions are known. Indeed, it is known that one can compute efficiently all overlaps for each pair of words in P using the GST of P. In fact, one usually computes pairwise the maximum overlaps for P [11], to build the prefix or the overlap graph of P (as mentioned in introduction).

2.2. Merge and Red–Blue paths

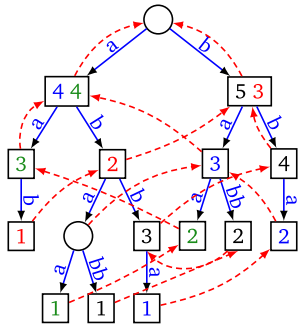
Above the overlap node was defined only using T; the SL (hence L) were not involved. Let us now give an alternative definition using T and L.



(a) T: tree structure of the suffix tree of P



(b) L: the tree made by the suffix links of T



(c) The GST of P is made of T and L

Fig. 3. (c) The Generalised Suffix Tree (GST) of $P := \{ababb, aab, abba, abaa\}$. The GST of P is in fact composed of two trees: (a) the tree T, which is the basic structure of the suffix tree – in blue arcs, and (b) L, the tree made by the suffix links of T – in red arcs. Square nodes represent words that occur as a suffix of some input word, circle nodes are the other nodes of T. Each square node stores its positions of occurrences in S; for simplicity, we display the starting position as a number and the word of S in which it occurs as its colour, instead of showing the list of pairs (i, j). The square node labelled by a red 2, which represents the string *ab*, means that this string occurs as the suffix starting at position 2 in the red word *aab*. A square node labelled with a 1 represents an input word. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

If u overlaps v, then a suffix of u equals a prefix of v. We have seen that the overlaps is a node on the branch spelling v. We can also find this precise suffix of u starting at the node representing u by following enough suffix links. This means going up in the red tree from the node u. Hence, one sees that an overlap node from u to v is an ancestor of u in L (the red tree), and an ancestor of v in T (the blue tree). We obtain an alternative definition: an overlap node from u to v is a *red* ancestor of u and a *blue* ancestor of v. For simplicity, let us say it is a *red–blue* ancestor of the ordered pair (u, v).

Now, we see that the maximum overlap of (u, v) is the lowest red–blue ancestor of (u, v). Also, if the red path reaches the root, then the overlap is the empty word (and their merge is simply their concatenation).

In other words, any overlap for a pair (u, v) corresponds to a *Red–Blue path* from u to v in (T, L).

The merge of u and v using their maximum overlap $ov(u, v)$ is the word $w := pr(u, v)ov(u, v)su(u, v)$, where $u := pr(u, v)ov(u, v)$ and $v := ov(u, v)su(u, v)$. We state this definition with the maximum overlap, but it is valid with any overlap; the larger the overlap, the shorter the merge. We denote by RB-path(u, v) the Red–Blue path which corresponds to the maximum overlap between u and v.

Assume we want to spell out the (linear) merge of u and v using the overlap node w in (T, L). It can be done in three steps:

1. traverse T from the root to u: this spells out u with the labels of blue arcs;
2. go up L from u to w following the red arcs: it is possible since w is a red ancestor of u (here we skip one by one all letters of $pr(u, v)$);
3. go down T from w to v using blue arcs, which is possible since v is a descendant of w; this spells out $su(u, v)$.

A merge of (u, v) is a linear superstring of $\{u, v\}$. The process above can be generalised to spell out any cyclic superstring of a cyclic permutation of words $(w_1, w_2, \ldots, w_n)$ using some given pairwise overlaps: one starts from the node representing w_1, then performs steps 2 and 3 using the appropriate Red–Blue path from w_1 to w_2 to merge them, then iterates only steps 2 and 3 for each following word until one reaches w_1 again. This circuit on the GST is called a *Red–Blue circuit*.

Example 1. For instance in Fig. 3, the Red–Blue circuit corresponding to the cyclic merge of the words *ababb*, *abba*, and then *aab* in this order, contains three Red–Blue paths: for the pairs (*ababb*, *abba*), (*abba*, *aab*) and then for (*aab*, *ababb*). This

circuit starts at node 1 goes up with SL to node 2 and then 3 which is a red–blue ancestor of (*ababb*, *abba*), then goes down with blue arcs to 1. This is the Red–Blue path for (*ababb*, *abba*), which spells out *ab* with the SL labels. Then the circuit goes up with SL to 2, 3, and 4, which is an ancestor of 1. This spells out *abb* with SL labels. Then, it goes down to 1. This makes up the Red–Blue path for (*abba*, *aab*). Then, the circuit goes up to 2, which is an ancestor of 1; this spells out *a* with one red label. Then, it goes down and back to 1, which was the starting node. Altogether the sequence of red labels spells out *ababba*, while the sequence of blue labels yields *aababb*: both write the same cyclic word.

Now, let us state a more subtle property. For a Red–Blue circuit, writing all blue labels in order or writing all red labels in order spell out the same superstring (a red label is a label of a SL). Hence, the number of blue arcs is smaller than the number of red arcs, which is the superstring length. A proof of a slightly different version of this property can be found in [5, Section 4.3 on p. 18].

3. The superstring graph

Here, we are going to show that the choices made by Algorithm `greedy` for SCCS are not really choices. In fact, we will show that each greedy solution can be represented by a unique graph embedded in the Generalised Suffix Tree. As this graph turns out to be computable in linear time, one can derive a greedy solution for SCCS in linear time.

3.1. Equivalence of greedy solutions

Let w be a greedy solution of SCCS. By Proposition 2, there exists a permutation σ such that $w = P(\sigma)$. For each cyclic permutation σ_i of σ, we denote by $c(\sigma_i)$ the Red–Blue circuit obtained from σ_i in the GST.

We denote by $G_P(\sigma) = (V, R, B)$ the directed multigraph built with the arcs of the Red–Blue circuits of all the $c(\sigma_i)$, where R denotes the set of *Red* arcs, and B denotes the set of *Blue* arcs of all the $c(\sigma_i)$.

A graph is said to be *Eulerian*, if there exists a Eulerian path on this graph, *i.e.* a path that visits each arc of the graph exactly once. As $G_P(\sigma)$ is a collection of Red–Blue circuits, which are made of Red–Blue paths, each connected component of $G_P(\sigma)$ is Eulerian. (With each additional circuit, the graph remains Eulerian.)

For each permutation of $\{1, \ldots, |P|\}$, we can build the corresponding multigraph $G_P()$. We have the following proposition.

Proposition 3. *Let w_1 and w_2 be two greedy solutions of SCCS and let σ_1 and σ_2 be their respective permutations of $\{1, \ldots, |P|\}$. Then $G_P(\sigma_1) = G_P(\sigma_2)$.*

Proposition 3 states a key property of greedy solutions for SCCS. Below, we give two lines of proof: the first one explains why the property is valid (and better conveys the intuition to the reader), while the second gives a more formal, but less explanatory, demonstration.

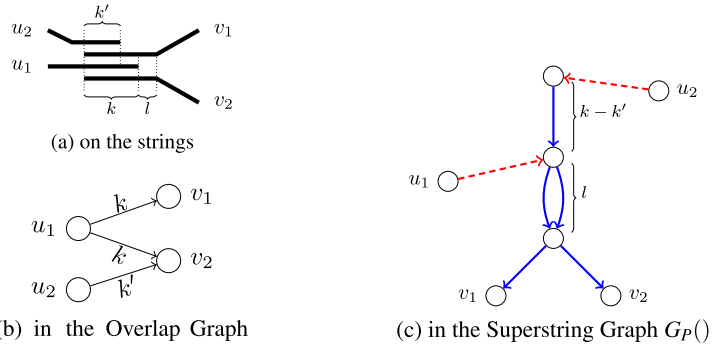


Fig. 4. Example with three overlaps: two dependent ones, between u_1 and v_1 and between u_1 and v_2, and another overlap between u_2 and v_2: shown on the strings (a), in the overlap graph (b), and in the Superstring Graph (c).

Proof. Let $w = P(\sigma)$ be a greedy solution of SCCS. Let us show that the choices made by Algorithm `greedy` for SCCS do not influence $G_P(\sigma)$. `greedy` faces a choice when several overlaps of the same length are dependent (Clearly, independent overlaps can be swapped in a permutation and thus do not influence the graph).

Let $ov(u_1, v_1)$ and $ov(u_1, v_2)$ be two dependent and greedy-feasible overlaps. Assume that $ov(u_1, v_1)$ is the overlap choosen by `greedy` to obtain w, and let u_2 in P be the successor of v_2 in σ. It implies that $ov(u_2, v_2)$ is the overlap that merges u_2 and v_2 in w. By the definition of `greedy`, we have that $|ov(u_1, v_1)| \geq |ov(u_2, v_2)|$. Let us show that there exits a Red–Blue path from u_2 to v_1 in $G_P(\sigma)$.

As $|ov(u_1, v_1)| \geq |ov(u_2, v_2)|$, we have that $ov(u_2, v_2)$ is a prefix of $ov(u_1, v_2)$, and thus a prefix of $ov(u_1, v_1)$. Hence in the GST, the node $ov(u_2, v_2)$ is an ancestor of $ov(u_1, v_1)$, and $ov(u_1, v_1)$ is a node in the Red–Blue paths from u_1 to v_1 and from u_2 to v_2. Thus, we have both Red–Blue paths from u_1 to v_2 and from u_2 to v_1 are in $G_P(\sigma)$ (see Fig. 4). □

An alternative proof. Let σ_1 and σ_2 be two permutations of P corresponding to two distinct greedy solutions for SCCS. We set $G_p(\sigma_1) := (V_1, R_1, B_1)$ and $G_p(\sigma_2) := (V_2, R_2, B_2)$.

We proceed by contraposition and assume that $G_p(\sigma_1) \neq G_p(\sigma_2)$; then $((R_1 \cup B_1) \setminus (R_2 \cup B_2)) \cup ((R_2 \cup B_2) \setminus (R_1 \cup B_1)) \neq \emptyset$. As the role of both permutations is symmetrical, we assume that $(R_1 \cup B_1) \setminus (R_2 \cup B_2) \neq \emptyset$. Let $e := (u, v)$ be an arc of $(R_1 \cup B_1) \setminus (R_2 \cup B_2)$ such that $max(|u|, |v|)$ is maximal among all possible arcs of $(R_1 \cup B_1) \setminus (R_2 \cup B_2)$. According to the definition of $G_p(\sigma_1)$, there exist i and j such that $\sigma_1(i) = j$ and e is an arc on the Red–Blue path linking s_i to s_j (formally $e \in \text{RB-path}(s_i, s_j)$). Two cases arise.

Case $e \in B_1$:
Let i_2 denote the predecessor of j in σ_2, in other words, the element of σ_2 satisfying $j = \sigma_2(i_2)$. But, one gets that $ov(s_{i_2}, s_j)$ is a prefix of $ov(s_i, s_j)$, since otherwise there would exist $j_1 := \sigma_1(i_2)$, such that $\left|ov(s_i, s_j)\right| < \left|ov(s_{i_2}, s_{j_1})\right|$, which would imply that e is not maximal. It follows that e belongs to RB-path(s_{i_2}, s_j), and thus to B_2, which contradicts our hypothesis.

Case $e \in R_1$:
Let j_2 denote the successor of i in σ_2, that is $j_2 := \sigma_2(i)$. Clearly, $ov(s_i, s_{j_2})$ must be a suffix of $ov(s_i, s_j)$, since otherwise there would exist i_1 such that $j_2 = \sigma_1(i_1)$ and $\left|ov(s_i, s_j)\right| < \left|ov(s_{i_1}, s_{j_2})\right|$, which would mean that e is not maximal. One obtains that $e \in$ RB-path(s_i, s_{j_2}), and thus e belongs to R_2, which is a contradiction.

This ends the proof. □

By Proposition 3, the graph $G_P()$ represents all the solutions of the greedy algorithm for SCCS. We called this graph, the *Superstring Graph* of P, and denote it by G_P. An example of Superstring Graph for $P := \{abb, bbb, bbc\}$ is shown in Fig. 7.

To sum up, we know that for each greedy solution w, there exists a permutation of $\{1, \ldots, |P|\}$ that can be translated into a set of Red–Blue circuits, $c(w) = \{c_1(w), \ldots, c_m(w)\}$. Remember that each Red–Blue circuit is made by chaining Red–Blue paths. Let c be a multi-cycle of G. We say that c *covers* G_P if each arc of G_P is included in a single cycle of c. By Proposition 3, for each greedy solution w we have that $c(w)$ covers G_P.

By the following proposition, we have that each multi-cycle that covers G comes from a greedy solution.

Proposition 4. $\{$*multi-cycle* $p \mid p$ *covers* $G_P\} = \{c(w) \mid w$ *is a greedy solution*$\}$.

To prove Proposition 4, we first need a lemma. A *crossing* in G_P is a node that has red in- and out-going arcs and blue in- and out-going arcs.

Lemma 5 *(Absence of crossing).* G_P *does not contain any crossing.*

Proof of Lemma 5. It is enough to see that the only nodes that can have a red out-going arc and a blue in-going arc correspond to the words of P. But if $v \in P$, v is a leaf in T and in L by definition of the GST, and thus v has neither a blue arc going out, nor a red arc going in. □

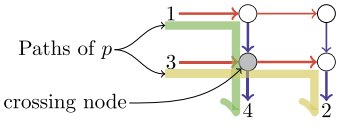


Fig. 5. Illustration for the proof of Proposition 4. In the proof of $\subseteq$, the argument is the existence of a crossing node in a non-greedy solution. This figure displays the four nodes $v_1, \ldots, v_4$ and the nodes (circles) representing their overlaps. Then, the node representing $ov(v_3, v_4)$ (grey node) is a crossing, *i.e.* it has blue and red in-going and out-going arcs. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

Proof of Proposition 4. To understand this equality, it is sufficient to note that there is a relation between choosing the longest overlap in `greedy` and the absence of crossing in G_P.

Proof of $\subseteq$.

Let us show $\{$multi-cycle $p \mid p$ covers $G_P\} \subseteq \{c(w) \mid w$ is a greedy solution$\}$ by contraposition. Assume that there exists a multi-cycle p covering G_P and there does not exist a greedy solution w of P such that $p = c(w)$. By hypothesis, there exist four nodes v_1, v_2, v_3 and v_4 in P that do not satisfy the greedy property. In other words, although $|ov(v_1, v_2)| <$

$|ov(v_3, v_4)|$, both $Path(v_1, v_4)$ and $Path(v_3, v_2)$ are in p. Hence, the node $ov(v_3, v_4)$ is a crossing in G_P (see Fig. 5). As there does not exist any crossing in G_P (Lemma 5), we obtain a contradiction, and get

$$\{\text{multi-cycle } p \mid p \text{ covers } G_P\} \subseteq \{c(w) \mid w \text{ is a greedy solution}\}.$$

Proof of $\supseteq$.

Reciprocally, assume that there exists a greedy solution w of P and there does not exist a multi-cycle p that covers G_P such that $p = c(w)$. Thus, there exists an arc of $c(w)$ which is not an arc of G_P. Among those arcs, let $e = (u, v)$ be the one with the deepest node (*i.e.* longest word) v in T. By the definition of G_P, e belongs to R (*i.e.* is a red arc), because it is the deepest of its kind. Let v_1 and v_2 be the nodes of P such that e belongs to $Path(v_1, v_2)$. As e is not in G_P, for each u in P such that $Path(v_1, u)$ is in G_P, $u \neq v_2$. Thus, there exists v_4 in P such that $Path(v_1, v_4)$ is in G_P and $|ov(v_1, v_2)| < |ov(v_1, v_4)|$. As w is a greedy solution of P, there exists v_3 in P and not in G_P such that $Path(v_3, v_4)$ is in $c(w)$ and $|ov(v_1, v_4)| \leq |ov(v_3, v_4)|$. Thus, there exists an arc in $c(w)$ that is not in G_P with a deeper extremity than e. This contradicts our hypothesis and we get $\{c(w) \mid w \text{ is a greedy solution}\} \subseteq \{\text{multi-cycle } p \mid p \text{ covers } G_P\}$. □

3.2. Characterisation of the Superstring Graph and a construction algorithm

Now, we are going to characterise the Superstring Graph and give a linear time algorithm to build it from a set of words. For this algorithm, we define and compute weights (respectively $n(.)$ and $d(.)$) for the red and for the blue arcs of the GST. Finally, the SG will be composed of those arcs having a strictly positive weight (and of the nodes visited by them).

As `greedy` selects overlaps in order of decreasing length, one can see it as a recursive algorithm that progresses along the branches of the GST of P. On the GST, `greedy` progresses from the leaves toward the root, *i.e.* in order of decreasing word depth of the nodes. Let us use this recursive side of `greedy` to explain the Superstring Graph.

Let v be a node of the GST; when `greedy` sees the overlaps of length $|v|$, it has already considered overlaps of larger lengths. We define two weights $n()$ and $d()$ such that:

- $n(v)$ is the number of strings of P which have v as suffix and are not merged on their right with another string of P with a larger overlap.
- $d(v)$ is the number of strings of P which have v as prefix and are not merged on their left with another string of P with a larger overlap.

In other words, $n(v)$ is the number of leaves of the subtree of v in L that have not yet been merged on their right, and $d(v)$ is the number of leaves of the subtree of v in T that have not yet been merged on their left.

First, it can be seen easily seen that a node v which has no other leaf both in its subtrees in L and in T, is a string of P. Thus, for these nodes we set, $n(v) = d(v) := 1$.

For the other nodes, we use the recursion of `greedy`. In fact, we can determine how many strings of P have not yet been merged with another string by looking the weights $n(.)$ and $d(.)$ of the children of v in L and of its children in T (see Fig. 6).

Let $a(v)$ be the difference between the sum of the $n(u')$ of the children u' of v in L and the sum of the $d(u)$ of the children u of v in T:

$$a(v) := \sum_{u' \in Children_L(v)} n(u') - \sum_{u \in Children_T(v)} d(u).$$

Hence, the absolute value of $a(v)$ is the largest number of strings of P that `greedy` can merge with other strings of P using the overlap v. With $a(v)$, we can calculate $n(v)$ and $d(v)$:

$$\text{If } a(v) \geq 0, \text{ we have } \begin{cases} n(v) := a \\ d(v) := 0, \end{cases}$$

$$\text{if } a(v) < 0, \text{ we have } \begin{cases} n(v) := 0 \\ d(v) := -a. \end{cases}$$

With this recursive definition of the weights $n()$ and $d()$, we can construct the Superstring Graph from the GST of P. Whenever the weight of an arc of the GST is equal to 0, this arc does not belong to the Superstring Graph. Similarly, if for a node, say v, all of its in-going and out-going arcs have a zero weight, v is isolated and is not included in the Superstring Graph – this corresponds to the subset U in Definition 1 (which states another definition of the Superstring Graph).

Definition 1. The Superstring Graph $G_P = (V, R, B)$ where

$$V = V_T \setminus U$$

$$R = \{(u, w)^{n(u)} \mid u \in V_T,\ w \text{ the parent of } u \text{ in } L\}$$

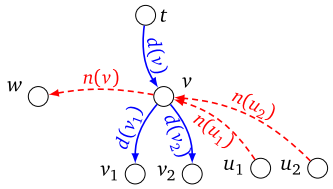


Fig. 6. General scheme of the in- and out-going arcs of a node v in GST. The red arcs belong to R and the blue arcs to B. If a weight is zero, the arc is not included in the Superstring Graph. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

$$B = \{(t, v)^{d(v)} \mid v \in V_T,\ t \text{ the parent of } v \text{ in } T\}$$

and $U = \{v \in V_T \mid v \text{ is not an extremity of an arc of } R \cup B\}$ where $n()$ and $d()$ are the weights on arcs as defined above.

Now that we can build the Superstring Graph for an input P by traversing the generalised suffix tree of P, we provide an algorithm (Algorithm 2) that uses this graph to solve SCCS for P.

Theorem 6. *Algorithm 2 solves SCCS in linear time and space.*

Proof. *Correctness*. By Proposition 4, the set w of cyclic words computed by Algorithm 2 is a solution of `greedy` for SCCS. Hence, each word of P appears as a substring of a cyclic word in w, and by Theorem 1, the norm of w is minimal.

Complexity. The Superstring Graph of P is embedded in the generalised suffix tree of P, and thus built in a time linear in $\|P\|$.

First the cyclic concatenation of the words in P is longer than the SCCS obtained through any cyclic permutation of P. By Proposition 2, the length of a solution of `greedy` is bounded by $\|P\|$; thus, the length of a multi-cycle of G_P is bounded by $2 \times \|P\|$. Consequently, one can find a multi-cycle also in $O(\|P\|)$ time and space. Finally, traversing all its cycles takes a time proportional in the size of the Superstring Graph. □

Note that with Algorithm 2, one obtains a cyclic cover.

Algorithm 2: The Superstring Graph algorithm for SCCS.

```
1 Input: P a set of linear words. Output: w a greedy solution for SCCS;
2 build G_P the Superstring Graph of P;
3 compute a cyclic cover c = (c_1, ..., c_n) of G_P;
4 for i ∈ [1, n] do
5     traverse c_i: list the words of P whose node are in c_i and create w_i by concatenating cyclically pr(s_j, s_k) whenever s_k is the successor of s_j;
6     insert the cyclic string w_i in w;
7 return w
```

4. Implications of the Superstring Graph

4.1. SCCS with a cardinality constraint on greedy solutions

We can split up G_P into connected components: $G_P = \{C_1, \ldots, C_g\}$. By Proposition 4, each multi-cycle c of G_P can be partitioned in a subset of cycles $c = S_1 \cup \ldots \cup S_g$, where each set of cycles S_i covers a connected component C_i of G_P.

Algorithm 2, on its third line (in blue),[1] computes a cyclic cover of G_P. This requires to perform a random choice when facing two alternative out-going arcs. Clearly, the combination of all possible successive choices allows to produce all possible cyclic covers of G_P. Certain choices will generate a solution containing several cycles to cover one connected component (see Fig. 7b), although it is always possible to cover one connected component with a single cycle (see Fig. 7c). Thus, any two cyclic covers may differ in their number of cycles, *i.e.* in their cardinality. This gives rise naturally to a new question, to a new variant of SCCS, which we define below.

Problem 2 *(SCCS with a cardinality constraint on `greedy` solutions).*

Input: A set of linear strings P.
Output: A set of cyclic strings c, with the least number of elements, that is a solution of the greedy algorithm on Shortest Cyclic Cover of Strings.

[1] For interpretation of the colours, the reader is referred to the web version of this article.

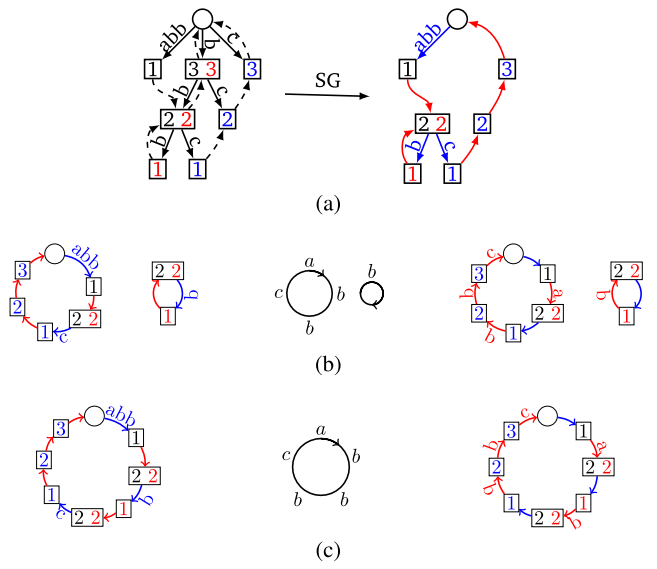


Fig. 7. A Superstring Graph and SCCS solutions. (a) The generalised suffix tree and the Superstring Graph (SG) for $P := \{abb, bbb, bbc\}$. Square nodes of T represent strings that are suffixes of some word of P: the integer in the square node is a starting position of its suffix. A word of P is identified by its colour. Figure (b) and (c) show the two multi-cycles that cover the SG and their respective sets of cyclic strings. This example is well known for showing the lower bound of the approximation ratio of the greedy algorithm for SLiS. The SG for this instance is connected and thus it also gives an optimal cyclic superstring (a solution of SCyS). As the Eulerian circuit visits the root of the GST, this optimal cyclic superstring can be cut at the root to obtain an optimal linear superstring.

Algorithm 3: The Superstring Graph algorithm for SCCS with a cardinality constraint.

```
1 Input: P a set of linear words. Output: w a greedy solution for SCCS;
2 build G_P the Superstring Graph of P;
3 compute a Eulerian multi-cycle c = (c_1, ..., c_n) of G_P;
4 for i ∈ [1, n] do
5     traverse c_i: list the words of P whose node are in c_i and create w_i by concatenating cyclically pr(s_j, s_k) whenever s_k is the successor of s_j;
6     insert the cyclic string w_i in w;
7 return w
```

Algorithm 3 differs from Algorithm 2 only on the third line: instead of computing any cyclic cover of G_P, it computes a Eulerian multi-cycle of G_P. This is possible since each connected component of the Superstring Graph is Eulerian and it can be done in linear time in the norm of P. A Eulerian multi-cycle of G_P covers each connected component with a single cycle, hence it minimises the cardinality of the cyclic cover, and as it is a greedy solution it solves Problem 2. We obtain Theorem 7. The minimisation of the number of cyclic strings by Algorithm 3 is illustrated in Fig. 7.

Theorem 7. *Algorithm 3 solves Problem 2 in linear time.*

Proof. As the connected components of the Superstring Graph are Eulerian, this is done in linear time and space in the size of G_P. By Proposition 4, a label of a Eulerian multi-cycle of G_P is an optimal solution for the Problem 2. □

Remark. Although for SCCS, all solutions of `greedy` are optimal, some optimal solutions are not output by `greedy`. Thus, the question of finding a shortest cyclic cover with a minimum number of elements is not solved by the Algorithm 3 (see a counter example in Fig. 8).

4.2. Consequences of the topology of the Superstring Graph

The characterisation of the SG can also be applied to solve or approximate the "classical" shortest superstring problem (SLiS – for Shortest Linear Superstring), in which the solution is a linear superstring, as well as its variant where the solution is a single cyclic word (SCyS -= for Shortest Cyclic Superstring). Let $w_{OPT(SLiS)}$, respectively $w_{OPT(SCyS)}$, denote an optimal solution for SLiS, resp. for SCyS. Let w a greedy solution for SCCS, we have: $|w| \leq \left|w_{OPT(SCyS)}\right| \leq \left|w_{OPT(SLiS)}\right|$. If the SG is connected, then w is a cyclic superstring of P, *i.e.* a solution of SCyS. By the above relation, it is an optimal solution for SCyS.

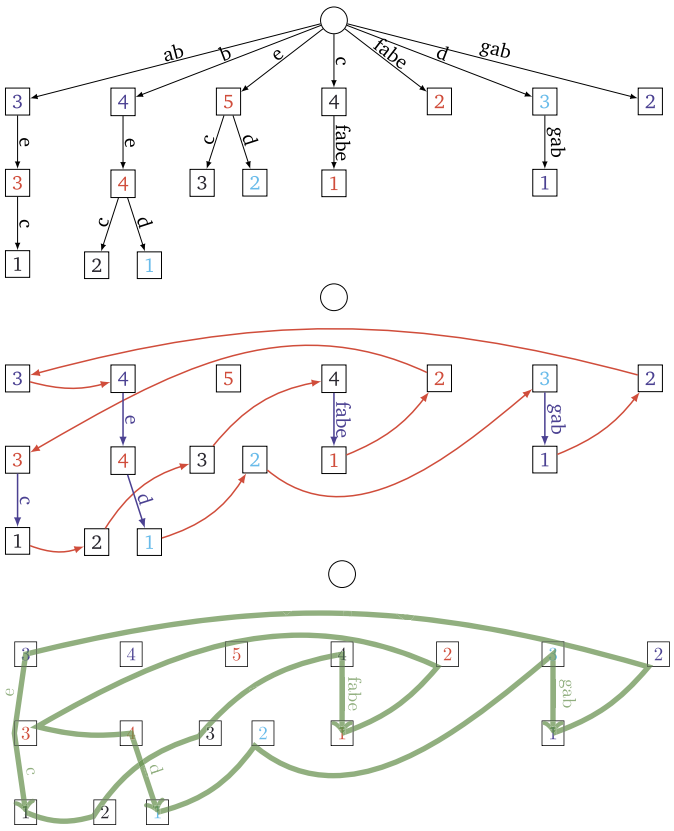


Fig. 8. An instance with an optimal solution that is not a solution of `greedy` for SCCS (however, the solution of the SG is also optimal). Let $P = \{abec, bed, cfabe, dgab\}$. (Top) the generalised suffix tree, (middle) the SG, and (bottom) an optimal solution of SCCS that consists in single cycle, while the SG has two components.

Now, from this cyclic superstring, we can easily derive a linear superstring of P. Basically, we cut the cyclic string at the start of a suffix of a word of P and pre-catenate the corresponding prefix. Let v a node of the SG such that $n(v) = d(v) = 0$, then v is a prefix of some word of P. Then vw makes up a linear superstring of P. We obtain $|w| \leq \left|w_{OPT(SLiS)}\right| \leq |v| + |w|$. As $|v| \leq \left|w_{OPT(SLiS)}\right|$, we get $|v| + |w| \leq 2\left|w_{OPT(SLiS)}\right|$. In other words, for instances P that have a connected SG, we obtain a 2-approximation for the shortest linear superstring. Furthermore, if the Eulerian cycle visits the root of the suffix tree, then v is the empty word and then $|w| \leq \left|w_{OPT(SLiS)}\right| \leq |w|$, meaning that w is an optimal linear superstring.

Note that there exist non-trivial instances, with an arbitrary large norm, whose SG is connected: any instance in which all words are cyclic shifts of another belong to this class. As the SG can be built and explored in linear time in $\|P\|$, one can use this procedure as a predicate to check whether SCyS or SLiS are solvable in linear time. If not, one can launch a more complex approximation algorithm [15].

5. Conclusion

We consider the problem Shortest Cyclic Cover of Strings, which is solved by Algorithm `greedy`. The set of solutions of `greedy` can be exponential. Nevertheless, with the SG, we exhibit a characterisation of this set in a graph that is Eulerian and linear in the input size. This enables us to solve SCCS in linear time, but also to find a `greedy` solution with the least number of cyclic strings. Moreover, this approach can solve or approximate some non-trivial instances of the shortest superstring problems. Our approach can be adapted to a variant of SCCS where the multiplicity of each input word is given as input.

Currently, approximation algorithms for shortest superstring problems use a procedure for solving SCCS that takes cubic time (*i.e.* $O(\|P\| + |P|^3)$ time) and quadratic space in the number of strings. This step represents their time complexity bottleneck since other steps can be done in $O(\|P\|)$ time. Using our algorithm makes the overall time complexity linear in the input size.

Acknowledgements

This work is supported by ANR Colib'read (ANR-12-BS02-0008), by ANR Institut de Biologie Computationnelle (ANR-11-BINF-0002), and by CNRS (Défi MASTODONS).

References

[1] A. Blum, T. Jiang, M. Li, J. Tromp, M. Yannakakis, Linear approximation of shortest superstrings, in: ACM Symposium on the Theory of Computing, 1991, pp. 328–336.
[2] D. Breslauer, G.F. Italiano, On suffix extensions in suffix trees, Theor. Comput. Sci. 457 (2012) 27–34.
[3] D. Breslauer, T. Jiang, Z. Jiang, Rotations of periodic strings and short superstrings, J. Algorithms 24 (2) (1997) 340–353.
[4] B. Cazaux, E. Rivals, Approximation of greedy algorithms for Max-ATSP, Maximal Compression, Maximal Cycle Cover, and Shortest Cyclic Cover of Strings, in: Proc. of Prague Stringology Conference, PSC, Czech Technical Univ. Prague, 2014, pp. 148–161, http://www.stringology.org/event/2014/p14.html.
[5] B. Cazaux, E. Rivals, Reverse engineering of compact suffix trees and links: a novel algorithm, J. Discret. Algorithms 28 (2014) 9–22, http://dx.doi.org/10.1016/j.jda.2014.07.002.
[6] B. Cazaux, E. Rivals, The power of greedy algorithms for approximating Max-ATSP, Cyclic Cover, and superstrings, Discrete Appl. Math. (2015), http://dx.doi.org/10.1016/j.dam.2015.06.003.
[7] J. Gallant, D. Maier, J.A. Storer, On finding minimal length superstrings, J. Comput. Syst. Sci. 20 (1980) 50–58.
[8] A. Golovnev, A. Kulikov, I. Mihajlin, Approximating shortest superstring problem using de Bruijn Graphs, in: Combinatorial Pattern Matching, in: Lect. Notes Comput. Sci., vol. 7922, Springer Verlag, 2013, pp. 120–129.
[9] D. Gusfield, Faster implementation of a shortest superstring approximation, Inf. Process. Lett. 51 (5) (1994) 271–274.
[10] D. Gusfield, Algorithms on Strings, Trees and Sequences, Cambridge Univ. Press, 1997.
[11] D. Gusfield, G.M. Landau, B. Schieber, An efficient algorithm for the all pairs suffix–prefix problem, Inf. Process. Lett. 41 (4) (1992) 181–185.
[12] S.R. Kosaraju, A.L. Delcher, Large-scale assembly of DNA strings and space-efficient construction of suffix trees, in: Proceedings of the Twenty-Seventh Annual ACM Symposium on Theory of Computing, STOC '95, 1995, pp. 169–177.
[13] S.R. Kosaraju, A.L. Delcher, Large-scale assembly of DNA strings and space-efficient construction of suffix trees, in: Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing, STOC '96, ACM, New York, NY, USA, 1996, p. 659.
[14] M. Mucha, Lyndon words and short superstrings, in: Proceedings of the Twenty-Fourth Annual ACM–SIAM Symposium on Discrete Algorithms, SODA, 2013, pp. 958–972.
[15] K.E. Paluch, Better approximation algorithms for maximum asymmetric traveling salesman and shortest superstring, CoRR arXiv:1401.3670 [abs], 2014.
[16] C.H. Papadimitriou, K. Steiglitz, Combinatorial Optimization: Algorithms and Complexity, 2nd edition, Dover Publications, Inc., 1998, 496 pp.
[17] J. Tarhio, E. Ukkonen, A greedy approximation algorithm for constructing shortest common superstrings, Theor. Comp. Sci. 57 (1988) 131–145.
[18] S. Teng, F.F. Yao, Approximating shortest superstrings, in: 34th Annual Symposium on Foundations of Computer Science, 1993, pp. 158–165.
[19] E. Ukkonen, A linear-time algorithm for finding approximate shortest common superstrings, Algorithmica 5 (1–4) (June 1990) 313–323.
[20] V. Vassilevska, Explicit inapproximability bounds for the shortest superstring problem, in: Mathematical Foundations of Computer Science 2005, 30th International Symposium, Proceedings, MFCS 2005, Gdansk, Poland, August 29–September 2, 2005, in: Lect. Notes Comput. Sci., vol. 3618, Springer, 2005, pp. 793–800.