Practical lower and upper bounds for the Shortest Linear Superstring

Bastien Cazaux, Samuel Juhel, Eric Rivals

Practical lower and upper bounds for the Shortest Linear Superstring

Bastien Cazaux
Department of Computer Science, University of Helsinki, Helsinki, Finland;
L.I.R.M.M., CNRS, Université Montpellier, Montpellier, France
Institute of Computational Biology, Montpellier, France
bastien.cazaux@cs.helsinki.fi

Samuel Juhel
L.I.R.M.M., CNRS, Université Montpellier, Montpellier, France
Institute of Computational Biology, Montpellier, France
samuel.juhel@zaclys.net

Eric Rivals
L.I.R.M.M., CNRS, Université Montpellier, Montpellier, France
Institute of Computational Biology, Montpellier, France
rivals@lirmm.fr
https://orcid.org/0000-0003-3791-3973

Abstract

Given a set P of words, the **Shortest Linear Superstring** (SLS) problem is an optimisation problem that asks for a superstring of P of minimal length. SLS has applications in data compression, where a superstring is a compact representation of P, and in bioinformatics where it models the first step of genome assembly. Unfortunately SLS is hard to solve (**NP-hard**) and to closely approximate (**MAX-SNP-hard**). If numerous polynomial time approximation algorithms have been devised, few articles report on their practical performance. We lack knowledge about how closely an approximate superstring can be from an optimal one in practice. Here, we exhibit a linear time algorithm that reports an upper and a lower bound on the length of an optimal superstring. The upper bound is the length of an approximate superstring. This algorithm can be used to evaluate beforehand whether one can get an approximate superstring whose length is close to the optimum for a given instance. Experimental results suggest that its approximation performance is orders of magnitude better than previously reported practical values. Moreover, the proposed algorithm remainso efficient even on large instances and can serve to explore in practice the approximability of SLS.

2012 ACM Subject Classification F.2.2

Keywords and phrases greedy — approximation — overlap — Concat-Cycles — cyclic cover — linear time — text compression

Digital Object Identifier 10.4230/LIPIcs.SEA.2018.18

1 Introduction

Let $P := \{s_1, \ldots, s_{|P|}\}$ be a set of input words, whose sum of lengths is denoted by $||P||$. A superstring of P is a string that contains each of the input words as substrings. Without loss of generality, we assume that P is factor free, *i.e.*, that no word of P is substring of another word of P. The *Shortest Linear Superstring* (SLS) problem – also known as Shortest Common Superstring –, asks for a superstring of P of minimal length.

A recent survey gives an idea of the variety of applications of SLS: from the most known ones, DNA assembly or text compression, to job scheduling or viral genomes compression

[10]. Several variations of SLS have also been investigated in theory, *e.g.*, with reversals [13, 9], with strings of DNA [14, 4], with multiplicities [8, 7]. SLS, which is studied since the 80's, has been proven NP-hard even for instances containing only words of length 3, and difficult to approximate (MAX-SNP-hard) [10]. Several polynomial time approximation algorithms with constant ratios have been designed for SLS, and among them, the **Greedy** algorithm, which, unlike most other approximation algorithms for SLS, admits a linear time implementation [21]. Currently, the approximation ratio of the **Greedy** algorithm is proven to be 3.5 [23] (see Algorithm **Greedy** in Appendix). The 28 years old, so called, **Greedy** conjecture states that the **Greedy** algorithm achieves an approximation ratio of 2, which is better than the best known approximation ratio of $2 + 11/30$ [16, 17], the latter being achieved by a polynomial, but not linear time algorithm. Another example of approximation algorithm is **Concat-Cycles**, which linearises and concatenates the cyclic words obtained by solving the Shortest Cyclic Cover of Strings problem (SCCS) on the instance; **Concat-Cycles** has an approximation ratio of 4 [2].

Importantly, algorithm **Greedy** for SLS breaks ties randomly, and is thus not deterministic. Example 1 illustrates the consequences of this non determinism in terms of approximation ratio.

▶ **Example 1.** On the classical instance (with $k > 0$) $P := \{ab^k, b^{k+1}, b^kc\}$, **Greedy** can output either $w_b := ab^kcb^{k+1}$ or $w_g := ab^{k+1}c$ as a superstring of P. The second one is optimal, while the first is the worst greedy superstring. This instance is the one used in [20] to bound the approximation ratio of **Greedy** by 2 (which tends to 2 when $k \to \infty$).

Some recent works have developed theoretical arguments suggesting that the **Greedy** algorithm achieves good approximation in general [15]. Experimental assessments on instances up to $1,000$ words of length up to 50 have shown that two approximation algorithms for SLS with ratio 3 and 4 return solutions within 1.28 times the optimal superstring length [19]. To our knowledge, this article gives the only experimental results published so far, and clearly emphasises the gap between lower and upper bounds, as well as between theory and practice. Although the algorithms used in [19] ran in short time on relatively small instances, their running times seem to increase non linearly with the instance size [19, Figure 5], indicating their limited scalability.

It would be useful to be able to determine rapidly, and before hand, whether an approximation algorithm would return a good approximate solution for a given instance. Obviously, such an algorithm should have a reasonable worst case approximation ratio, the best possible approximation in practice, should take linear time and be efficient enough to process large instances.

We propose an algorithm to compute a lower and an upper-bound on the size of an optimal solution for SLS. These two bounds, denoted respectively $\ell_{\min}$ and $\ell_{\max}$, are defined in Section 3.

We shall obtain the following theorem.

▶ **Theorem 2.** *Let P be a set of strings and let w_{opt} denote an optimal solution of SLS of P. We can compute in linear time in $||P||$ the values ℓ_{min} and ℓ_{max} such that:*

$$\ell_{min} \leq |w_{opt}| \leq \ell_{max} \qquad \text{and} \qquad \frac{\ell_{max}}{\ell_{min}} \leq 4.$$

Contributions. Here, we exhibit a linear time algorithm to compute a lower and an upper bound, respectively $\ell_{\min}$ and $\ell_{\max}$, on the size of a shortest superstring of P. Then we present experimental results of this algorithm on a series of instances of increasing sizes.

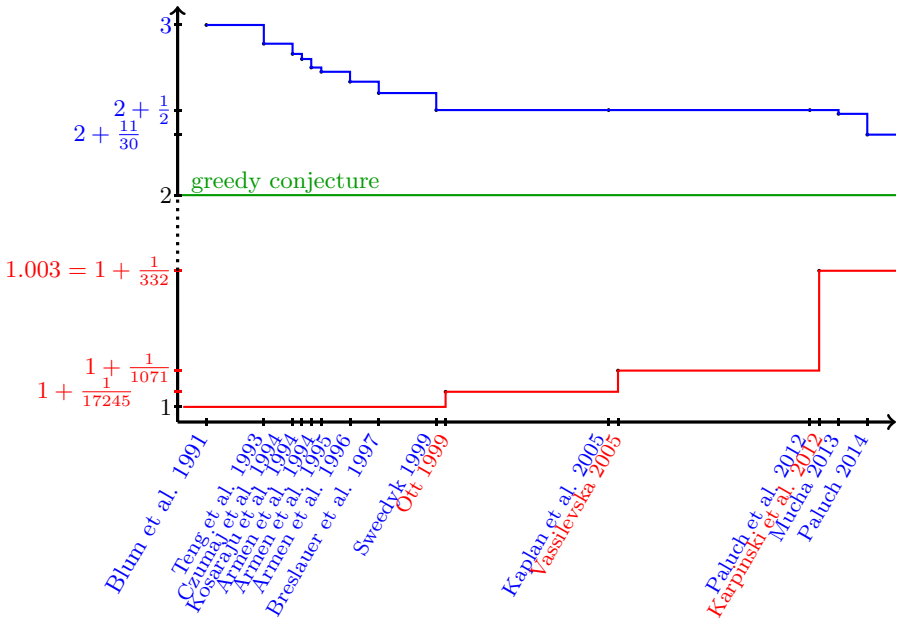

Figure 1 All the ratios of approximation (in blue) and inapproximation (in red) for the problem SLS by year.

These results show that $\ell_{\min}$ and $\ell_{\max}$ are extremely close to each other in practice. For more details, please see the web appendix at http://www.lirmm.fr/~rivals/res/superstring.

Notation We consider finite words over a finite alphabet Σ. The set of all finite words over Σ is denoted by $\Sigma^\star$, and ϵ denotes the empty word. For a word x, $|x|$ denotes the *length* of x. Given two words x and y, we denote by xy the *concatenation* of x and y.

Let s, t, u be three strings of $\Sigma^\star$. We say that s overlaps t if and only if a suffix of s also is a prefix of t. We denote by $ov(s,t)$ the longest overlap from s over t (also termed *maximum overlap*); let $pr(s,t)$ be the prefix of s such that $s = pr(s,t)ov(s,t)$, and let $su(s,t)$ be suffix of t such that $t = ov(s,t)su(s,t)$. The *merge of s over t* is the word $pr(s,t)t$. Note that neither the overlap nor the agglomeration are symmetrical.

▶ **Example 3.** Consider two strings $S := actgct$ and $T := tgcttac$. Then the longest overlap $ov(S,T) = tgct$, but the substring t also is an overlap from S over T. Then $pr(S,T) = ac$ and $su(S,T) = tac$. Moreover, we see that $ov(T,S) = ac$, which differs from $ov(S,T)$.

Throughout the article, the input is $P := \{s_1, \ldots, s_{|P|}\}$ a set of input words, and without loss of generality, we assume that P is substring free, *i.e.*, no word of P is substring of another word of P.

2 Related Works

Significant research effort has been dedicated to designing approximation algorithms for SLS and to finding the best theoretical approximation ratios (see [11] for a list of algorithms). Both upper and lower bounds of approximation ratios have been studied [22] (see Figure 1).

A crucial result regarding the design approximation algorithms for SLS is that a variant of SLS called, Shortest Cyclic Cover of Strings (SCCS), can be solved exactly and

returns a set of cycling strings covering the words of P. This set of cyclic strings can in turn be linearised and combined in various ways to form good linear superstrings [2]. A *cover* C is a set of strings such that any s_i is a substring of at least one string of C. An optimal cover can be obtained by computing a cyclic cover on the distance graph, a complete digraph representing the words of P and their maximum overlaps, using the Hungarian algorithm in $O(||P|| + |P|^3)$ time once the graph is built [18]. Blum *et al.* also state in their seminal article that a greedy algorithm computes a minimal cover of strings of P [2]. Recently, it was shown how to implement this greedy algorithm for `SCCS` in linear time in $||P||$ [5, Theorem 6]; see Algorithm 1. Algorithm 1, called `CGreedy`, minimises the norm of the Cyclic Cover of Strings, but also its cardinality, that is its number of cyclic words [5, Theorem 7].

Algorithm 1: Algorithm `CGreedy` .We denote any cyclic string w by $\langle w \rangle$.

```
1 Input: a set of strings P; Output: C, a Cyclic Cover of Strings of P;
2 C := ∅;
3 while |P| > 0 do
4 |   u and v in P (not necessarily distinct) such that ov(u, v) is maximised;
5 |   if u = v then C := C ∪ {⟨pr(u, v)⟩};
6 |   else P := P \ {u, v} ∪ {pr(u, v)ov(u, v)su(u, v)};
7 return C
```

Cyclic cover based approximation algorithms The first approximation algorithm based on a shortest cyclic cover is `Concat-Cycles` from [2]. `Concat-Cycles` computes C a Shortest Cyclic Cover of P. For $1 \leq i \leq |C|$, each cyclic string c_i of C covers a subset of words of P; let us denote this subset $P_i := \{s_{j_1}, \ldots, s_{j_{|c_i|}}\}$. For each c_i, it derives a linear string w_i, which is a partial superstring of P_i, by breaking c_i between two words of P_i, say s_{j_k} and $s_{j_{k+1}}$, by concatenating $pr(s_{j_k}, s_{j_{k+1}})$. Hence, $|w_i| \leq |c_i| + \left|s_{j_{k+1}}\right|$. Then, `Concat-Cycles` concatenates the words w_i for $1 \leq i \leq |C|$ in an arbitrary order, which yields a superstring of P. `Concat-Cycles` achieves an approximation ratio of 4 for `SLS` [2, Theorem 8].

Blum *et al.* also proposes an improvement of this strategy: each cycle can be broken at an optimal point so as to create the shortest w_i for c_i. As the cycle word c_i defines an order of occurrence for each word of P_i in c_i, this only requires to test any pair of successive words which is linear in $||P_i||$. They show that a variant of the greedy algorithm for `SLS`, which they call `MGreedy`, does exactly that [2]. In fact, we view `MGreedy` (see the web appendix) as an application of Algorithm `LCGreedy`, followed by a concatenation. In other words, `MGreedy` builds a linear cover of P (which is made of linear, rather than cyclic, strings), and concatenate those linear strings arbitrarily into a single linear superstring of P. Blum *et al.* show that this linear superstring is shorter than the one output by `Concat-Cycles` [2].

In these two algorithms `Concat-Cycles` and `MGreedy`, each cycle contributes to adding some symbols to the final superstring. We propose to optimise such procedure by minimising the number of cycles in the Shortest Cyclic Cover obtained by a greedy algorithm for `SCCS`.

Remark on non-determinism As indicated in introduction, all mentioned greedy algorithms – `Greedy`, `SCGreedy`, `LCGreedy` or `MGreedy` – break ties randomly when choosing the next overlap to use. Hence, none of these algorithms are deterministic, implying that two distinct executions may produce superstrings of different lengths or cyclic covers with different number of cycles. To our knowledge, most approximation algorithms designed to

date use at least a greedy solution for SCCS to start with, and inherit from non-determinism.

Lower and upper bounds. Among others, Vassilevska has proven new lower bounds for the approximation ratio of SLS. She noticed the huge gap separating the best upper bounds and lower bounds [22].

3 Algorithm LCGreedyMin

Overview Compared to Concat-Cycles or MGreedy [2], our algorithm builds a superstring based on a Shortest Cyclic Cover of P having a minimal number of cycles. Our algorithm proceeds as follows. First, it builds the *Extended Hierarchical Overlap Graph* (EHOG), a graph that encodes all overlaps between words of P but takes linear space. Embedded in the EHOG, it computes the Superstring Graph of P, which encodes the paths of all greedy solutions for SCCS. By finding an Eulerian path on each connected component of the Superstring Graph, it determines the node of minimal word depth of the component, and the shortest linearisation of each cyclic string. Moreover, this set of Eulerian paths constitutes an optimal Shortest Cyclic Cover of P; more precisely, we get the permutation indicating in which order the words of P are merged in each component to form the cyclic strings. Then, we then compute $\ell_{\min}$ and $\ell_{\max}$. We call our algorithm **LCGreedyMin**.

Below, we describe the graphs needed by **LCGreedyMin** and the algorithm.

3.1 EHOG

We denote by $\mathcal{O}v^+(P)$ the set of all overlaps between two (not necessarily distinct) strings of P, *i.e.* $\mathcal{O}v^+(P) := \{w \mid \exists\ u \text{ and } v \in P \text{ such that } w \text{ is a prefix of } u \text{ and } w \text{ is a suffix of } v\}$.

▶ **Definition 4.** The *Extended Hierarchical Overlap Graph* of P, denoted by $EHOG(P)$, is the directed graph $(V_E, P_E \cup S_E)$ where $V_E = P \cup \mathcal{O}v^+(P)$, while P_E is the set: $\{(x, y) \in (P \cup \mathcal{O}v^+(P))^2$ such that x is the longest proper **prefix** of $y\}$ and S_E is the set: $\{(x, y) \in (P \cup \mathcal{O}v^+(P))^2$ such that y is the longest proper **suffix** of $x\}$.

The EHOG has a node for each word of P and a node for any string that is an overlap between words of P. It can be seen that both types of nodes are also nodes of the Generalised Suffix Tree of P [12] – a Suffix Tree is a data structure that indexes all substrings of a text, while the Generalised Suffix Tree is the version that indexes several texts concatenated. Additionally, there are two types of arcs: one for recording the longest suffix relationship between nodes of V_E, the other for the longest prefix relationship. The first type can be seen as the arcs of the generalised suffix tree, while the second type corresponds to its Suffix Links. It follows that the EHOG occupies less space than the Generalised Suffix Tree of P. Examples of EHOG can be viewed in [6].

Rationale of the EHOG. The words of P and all their overlaps (*i.e.*, $\mathcal{O}v^+(P)$) are nodes of the EHOG. Consider u, v two words of P. Following arcs of S_E from u, one visits all its right overlaps in order of decreasing length. The first of such nodes that is an ancestor of v represents $ov(u, v,)$. Hence, the merge of (u, v) is (bijectively) associated to the shortest path from u to v through $ov(u, v,)$ in $EHOG(P)$. Call this the *merging path* from u to v. As any superstring (that does not waste any symbol) is determined by the order in which words of P are merged (solely using maximum overlaps between successive words), we see that it corresponds to a unique succession of merging paths in the EHOG. Similarly, any cyclic cover of strings of P is uniquely associated with a collection of merging cycles that

visit all nodes of P once in the EHOG. In fact, $EHOG(P)$ encode all possible, interesting superstrings and cyclic covers of P.

3.2 Superstring Graph

Consider a shortest cyclic cover of strings of P found by algorithm `CGreedy`. Its cyclic strings induce merging cycles in $EHOG(P)$, and hence a permutation of P representing the order in which words are merged. The Superstring Graph is **the subgraph** of $EHOG(P)$ visited by such a shortest cyclic cover of strings of P (since it is shown in [5, Proposition 3] that all greedy shortest cyclic covers of P visit the same subgraph). This is the intuitive rationale of the Superstring Graph, for which now we provide a formal definition.

▶ **Definition 5.** The *Superstring Graph* of a set of strings P is the sub-graph of $\texttt{EHOG}(P) = (V_E, P_E, S_E)$ represented by the weight functions n and d on the nodes of V_E such that:

$$\big(n(u), d(u)\big) \quad = \quad \begin{cases} \big(1, 1\big) & \text{If } u \in P, \\ \big(0, -\texttt{dif}_{n,d}(u)\big) & \text{If } u \notin P \text{ and } \texttt{dif}_{n,d}(u) \leq 0, \\ \big(\texttt{dif}_{n,d}(u), 0\big) & \text{If } u \notin P \text{ and } \texttt{dif}_{n,d}(u) > 0, \end{cases}$$

where

$$\texttt{dif}_{n,d}(u) = \sum_{(v,u)\in S_E} n(v) - \sum_{(u,v)\in P_E} d(v).$$

Among all overlaps stored in the EHOG, a shortest cyclic cover of P will use some overlaps to merge words, eventually more than once. An overlap is used if the cycles traversed the corresponding EHOG node. While building the SG, we compute a function Ov_{SG} that indicates how many times a shortest cyclic cover use an overlap. Precisely, we define Ov_{SG} as the function from the set of nodes of $\texttt{EHOG}(P) = (V_E, P_E, S_E)$ to $\mathbb{N}$, such that

$$Ov_{SG}(u) = \min\Big(\sum_{(v,u)\in S_E} n(v), \sum_{(u,v)\in P_E} d(v)\Big).$$

Algorithm 2 computes the superstring graph as well as the function Ov_{SG}. The idea of the algorithm is to traverse the EHOG in reverse depth order and to compute the different weight functions (n, d, and Ov_{SG}). Indeed, the weights (n, d, and Ov_{SG}) of a node only depends on the weights of deeper nodes in the EHOG. Each node represents a string: the substring built by concatenating the labels from the root to that node. With *deeper*, we refer to the string depth of a node.

In [5], we gave a proof that the Superstring Graph is a graph that represents all greedy solutions of `SCCS`. Because the Superstring Graph is Eulerian, it has the following property:

▶ Proposition 3.1 ([5]). Let P be a set of strings. One can compute in $O(||P||)$ time a greedy solution of `SCCS` with the least number of cyclic strings by computing an Eulerian path on each connected component of the Superstring Graph.

Indeed, taking a single cyclic path to cover each of its connected component is possible (a component could be covered by combining several cycles instead of only one); finding those paths takes a time that is linear in the number of nodes of the Superstring Graph.

3.3 Linearisation of cycles and computation of the bounds

Algorithm **MGreedy** [2] first computes an optimal cycle cover of P, linearises each cycle optimally, and then concatenates the resulting linear strings. As above mentioned, it is not deterministic and instances like the one given in Example 1 shows that the resulting superstring may vary a lot. Indeed, the linearisation of each cycle increases the size of the final superstring. We introduce a variant of **MGreedy**, called **MGreedyMin**, which chooses a greedy (and thus optimal) solution of SCCS with the least number of cycles. We compute the bounds of Theorem 2 ($\ell_{\min}$ and $\ell_{\max}$) based on such a cyclic cover of minimal cardinality.

Computation of $\ell_{\min}$ The norm of a set Z of cyclic strings, denoted $||Z||$, is the sum of the length of strings in Z.

▶ Proposition 3.2. Let C be a solution of the greedy algorithm for SCCS on P:

$$||C|| = ||P|| - \sum_{u \in V_E} Ov_{SG}(u) \times |u|\,.$$

Proof. Given a string v of P, we denote by $next_C(v)$ the string of P which follows directly v in the cyclic cover of strings C. As each greedy solution of SCCS is embedded in the Superstring Graph, we have

$$\begin{array}{rcl} ||C|| & = & \sum_{v\in P} |v| - |ov(v, next_C(v))| \\ & = & \sum_{v\in P} |v| - \sum_{v\in P} |ov(v, next_C(v))| \\ & = & ||P|| - \sum_{u\in V_E} |u| \times |\{v \in P \mid u = ov(v, next_C(v))\}| \\ & = & ||P|| - \sum_{u\in V_E} |u| \times Ov_{SG}(u). \end{array}$$

◀

By nature, the norm of C is smaller than an optimal shortest superstring of P. But for some instances, their difference can be as large as desired (can tend to infinity when the norm of the input tends to infinity). Thus defining $\ell_{\min}$ as the norm of C would not guarantee that $\ell_{\min}$ and $\ell_{\max}$ are close. We define $\ell_{\min}$ as the maximum between 1/4 of $\ell_{\max}$ and the norm of C, which is an optimal cyclic cover for P.

Computation of $\ell_{\max}$ By definition, the Superstring Graph is a sub-graph of the EHOG. Denoting by $G_1, \ldots, G_m$ the different connected components of the Superstring Graph, we get that $G_1, \ldots, G_m$ partition the node set of the Superstring Graph. We define $Cut(P)$ as the sum of the string depths (*i.e.*, the length of the string represented by a node) of the smallest node of each connected component, *i.e.*, $Cut(P) = \sum_{i=1}^{m} \min_{u \in G_i} |u|$.

▶ Proposition 3.3. Let w a solution of **MGreedyMin**. We have that :

$$|w| = ||P|| - \sum_{u \in V_E} Ov_{SG}(u) \times |u| + Cut(P).$$

Proof. Let w_g be a solution of **MGreedyMin** given by a greedy solution c_{min} of SCCS with the least number of cycles. By the property of the Superstring Graph, $c_{min} = \{c_1, \ldots, c_m\}$, where for all i between 1 and m, c_i is the cyclic string representing a Eulerian cycle in G_i and c_i is a cyclic superstring of a subset of P_i of P. By the definition of **MGreedyMin**, we take w_i the minimal linearisation of c_i, *i.e.*

$$w_i \in \underset{(s_{j_k}, s_{j_{k+1}}) \in P_i \times P_i}{\textbf{Arg min}} \left| Linearisation(c_i, s_{j_k}, s_{j_{k+1}}) \right|$$

where $Linearisation(c_i, s_{j_k}, s_{j_{k+1}})$ is the string obtain by breaking c_i between s_{j_k} and $s_{j_{k+1}}$ where s_{j_k} and $s_{j_{k+1}}$ are successive in c_i.

Hence, we have

$$\begin{aligned}
|w_g| &= \sum_{i=1}^{m} |w_i| \\
&= \sum_{i=1}^{m} \min_{(s_{j_k}, s_{j_{k+1}}) \in P_i \times P_i} \left| Linearisation(c_i, s_{j_k}, s_{j_{k+1}}) \right| \\
&= \sum_{i=1}^{m} \min_{(s_{j_k}, s_{j_{k+1}}) \in P_i \times P_i} \left(|c_i| + \left| ov(s_{j_k}, s_{j_{k+1}}) \right| \right) \\
&= \sum_{i=1}^{m} |c_i| + \min_{(s_{j_k}, s_{j_{k+1}}) \in P_i \times P_i} \left| ov(s_{j_k}, s_{j_{k+1}}) \right| \\
&= \sum_{i=1}^{m} |c_i| + \min_{u \in G_i} |u| \\
&= \sum_{i=1}^{m} |c_i| + \sum_{i=1}^{m} \min_{u \in G_i} |u| \\
&= ||c_{min}|| + Cut(P) \\
&= ||P|| - \sum_{u \in V_E} Ov_{SG}(u) \times |u| + Cut(P).
\end{aligned}$$

Indeed, by Proposition 3.2, we have $||c_{min}|| = ||P|| - \sum_{u \in V_E} Ov_{SG}(u) \times |u|$. ◀

By Proposition 3.3, we get that all solutions of `MGreedyMin` have the same length; we denote this length by $\ell_{\max}$.

Clearly, as a solution of `MGreedyMin` is also a solution of `MGreedy`, it follows that $|w_{opt}| \leq \ell_{\max} \leq 4 \times |w_{opt}|$, where w_{opt} denotes any optimal solution of `SLS`. This yields Theorem 2.

Difference between $\ell_{\min}$ and $\ell_{\max}$ We have defined $\ell_{\max}$ as the length of a solution of the algorithm `MGreedyMin`, *i.e.*

$$\ell_{\max} = ||P|| - \sum_{u \in V_E} Ov_{SG}(u) \times |u| + Cut(P).$$

The value of $\ell_{\min}$ is the maximum between the norm of an optimal solution of `SCCS` and $\ell_{\max}/4$, *i.e.*

$$\ell_{\min} = \max\left(\frac{\ell_{\max}}{4}, ||P|| - \sum_{u \in V_E} Ov_{SG}(u) \times |u|\right).$$

With these definitions, we obtain the following proposition.

▶ Proposition 3.4. Let P be a set of strings. The bounds $\ell_{\min}$ and $\ell_{\max}$ are invariant and $\ell_{\max} - \ell_{\min} \leq Cut(P)$.

By invariant, we mean that their computation is deterministic. Hence, although $\ell_{\min}$ and $\ell_{\max}$ depend on the instance P, their values do not vary upon the execution of `MGreedyMin`, unlike the solutions computed by `Greedy`, `MGreedy`, `Concat-Cycles`, and other approximation algorithms.

4 Implementation and experimental results

Here, we explain how each step of Algorithm `MGreedyMin` is implemented. First it builds the EHOG of P in memory: for this, we rely on the data structure named `COvI`, a compact implementation of the EHOG that can be used as an indexing and supports queries on overlaps [3]. The algorithm that builds `COvI`, first builds a compact version of the Aho-Corasick automaton of P [1], then prunes its set of states (or nodes in the tree) to keep only nodes that represent overlaps between words of P. When visiting a node of the EHOG, we need to know the length of the substring it represents. In `COvI`, for each node this length is accessible in constant time. For a node u of the EHOG, we can also access in constant time

$p_{pr}(u)$ (resp. $p_{su}(u)$), which denotes the node of the EHOG corresponding to the longest prefix (resp. suffix) of u.

Then, computing the Superstring Graph from the EHOG is done with Algorithm 2.

▶ Proposition 4.1. Algorithm 2 builds a superstring graph in time linear in $||P||$.

Algorithm 2: Computing the Superstring Graph

```
1  Input: EHOG(P); Output: V_SG, n(V_E), d(V_E) and Ov_SG(V_E);
2  V_SG ← ∅;
3  ∀u ∈ V_E : Ov_SG'(u) ← 0; n'(u) ← 0; d'(u) ← 0;
4  Q ← a reverse depth order on the nodes of EHOG(P);
5  for u ∈ Q do
6      (s, p) ← (p_su(u), p_pr(u));
7      if u is a leaf then
8          (n'(u), d'(u)) ← (1, 1);
9      else
10         Ov_SG'(u) ← min(n'(u), d'(u));
11         if n'(u) > d'(u) then (n'(u), d'(u)) ← (n'(u) − d'(u), 0) ;
12         else (n'(u), d'(u)) ← (0, d'(u) − n'(u)) ;
13     n'(s) ← n'(s) + n'(u);
14     d'(p) ← d'(p) + d'(u);
15     if d'(u) ≠ 0 or n'(u) ≠ 0 then V_SG ← V_SG ∪ {u} ;
16 return V_SG, n', d' and Ov_SG'
```

Proof. *Complexity* : Finding a reverse depth order on the nodes of $\texttt{EHOG}(P)$ may be done in linear time. The **for** loop is executed once for each node of $\texttt{EHOG}(P)$, and there are at most $||P||$ nodes. All operations inside the loop are assignations or comparisons of integers. *Correctness* : Since when starting the **for** loop (line 5), we have $n'(u) = \sum_{(v,u)\in S_E} n(v)$ and $d'(u) = \sum_{(u,v)\in P_E} d(v)$, at the end of the loop (line 15), we get $n'(u) = n(u)$ and $d'(u) = d(u)$. ◀

Let `Comp` be the table of size $|V_E|$ that maps each node of the superstring graph to its connected component, and all other nodes to 0.

▶ Proposition 4.2. Algorithm 3 computes `Comp` in time linear in $||P||$.

Algorithm 3: Algorithm to build the table `Comp`.

```
1  Input: EHOG(P), V_SG, n(V_E) and d(V_E); Output: Comp;
2  Comp ← Table of size |V_E| initialised to 0;
3  nb ← 1;
4  for u ∈ V_E do
5      Update_Component_Table(u, Comp, nb);
6      nb ← nb + 1;
7  return Comp
```

Algorithm 4: Algorithm *Update_Component_Table*.

```
1  Input: T: an integer table, u: element of T, k: an integer; Output: T updated;
2  if T[u] = 0 then
3      if n(u) ≠ 0 then
4          T[u] ← k;
5          v ← Parent of u in (V_E, S_E);
6          Update_Component_Table(v, T, k);
7      for all children v of u in (V_E, P_E) do
8          if d(v) ≠ 0 then
9              T[u] ← k;
10             Update_Component_Table(v, T, k);
```

Proof. The superstring graph being Eulerian, if there is a path q from a node u to a node v, there is another path from v to u that do not share any edge with q. Using this property, it is possible to recursively follow all paths in the superstring graph from a node to itself while marking all traversed nodes. Applying this process on every node of graph allows to discover all its connected components. The number of arcs of the superstring graph is linear in $||P||$, and during the whole process each arc of the superstring graph is visited once and only once, implying that Algorithm 4 takes linear time. ◀

▶ Proposition 4.3. Let P be a set of strings. We can compute $Cut(P)$ in linear time in $||P||$.

Proof. By Proposition 4.2, we can compute the table `Comp` in linear time. Using `Comp`, we can easily obtain the node with the least string depth of each connected component. ◀

▶ Proposition 4.4. Let P be a set of strings. We can compute $\ell_{\min}$ and $\ell_{\max}$ in linear time in $||P||$.

Proof. By Proposition 3.3, we have that $\ell_{\max} = ||P|| - \sum_{u \in V_E} Ov_{SG}(u) \times |u| + Cut(P)$. By Proposition 4.1, Algorithm 2 computes $Ov_{SG}(V_E)$ in linear time in $||P||$. By Proposition 4.3, we can compute $Cut(P)$ in linear time in $||P||$. Hence, it follows that we can compute $\ell_{\max}$ in linear time in $||P||$.

By Proposition 3.2, we have that $\ell_{\min} := \max\left(\frac{\ell_{\max}}{4}, ||P|| - \sum_{u \in V_E} Ov_{SG}(u) \times |u|\right)$ can be computed in linear time in $||P||$. ◀

4.1 Empirical results

We performed experimental tests to check how close the bounds $\ell_{\min}$ and $\ell_{\max}$ are from an optimal superstring length. We used one synthetic dataset and one real dataset from a genomic experiment on the *E. coli* genome (Strain K-12 substrain MG1655); the data is available at https://github.com/PacificBiosciences/DevNet/wiki/EcoliK12MG1655HybridAssembly.

Results on synthetic data We randomly generated large sets of words of length 100 for a DNA alphabet (4 symbols). The model for random words is an unbiased Bernoulli model. The instances have an increasing number of words.

1. From $200,000$ to $4,000,000$ words with a step of $200,000$ words. The norm of such instances goes from 20 to 400 million symbols. For each size of instances, we ran 10 generations and executions, and took the average times, and memory usages.

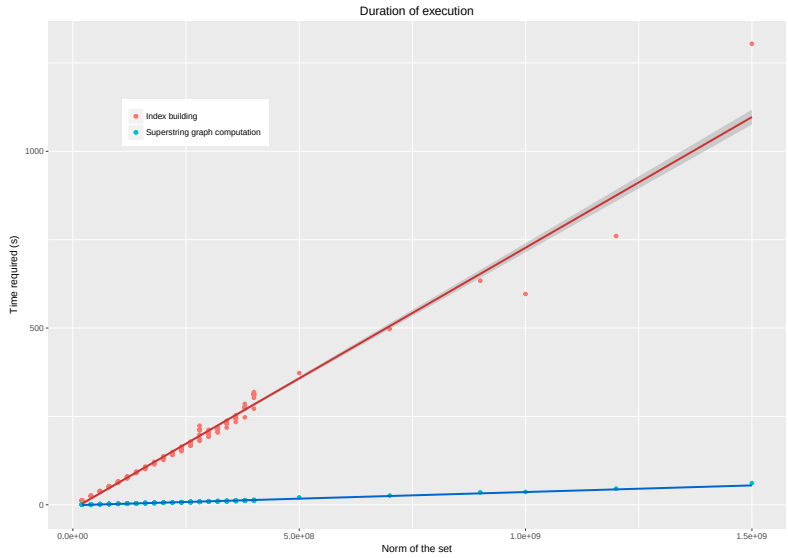

Figure 2 Execution times in function of the norm of the input set for i/ building the index (red dots) and for ii/ computing the Superstring graph and the solution (blue dots).

2. Then instances of 500, 700, 900, 1,000, 1,200 and 1,500 million words of length 100. Test were run using a single core on a desktop machine (x86_64 processor) running Linux 4.13.0-26-generic with 32 gigabytes of RAM.

Running times are displayed in Figure 2, and for each run, the largest memory consumption over the entire execution is shown in Figure 3. Most of the time is spent, and most of the memory used, while building the EHOG with `COvI`. Comparatively, the computation of $\ell_{\min}$ and $\ell_{\max}$ becomes rapidly at least an order of magnitude faster than `COvI`construction. The peak of memory for the largest instance, with 1.5 billion words, reached 22 gigabytes. In turn, `COvI`construction spends most of its time and space while building the Aho-Corasick automaton [3]. Thus, it would be advantageous to build the EHOG from a compressible and more compact index than `COvI`. However, the linear increases of running time and memory usage with the norm of the instances suggest that **`LCGreedyMin`** is very efficient and scalable.

Our algorithm computes the length of an approximate superstring. However, with some modifications, it could also output the computed superstring rather than only its length. Surprisingly, for 67% of the instances $\ell_{\min}$ and $\ell_{\max}$ are equal. In the remaining instances their difference (*i.e.*, $\ell_{\max} - \ell_{\min}$) is at most 0.0001% of the norm of the instance. This shows that most instances are entirely or almost "solved" with **`LCGreedyMin`**. This is coherent with theoretical results [15].

Results on real data We used a publicly available set of genomic reads obtained from an Illumina sequencing machine. The reads of length 100 make up a coverage on the *E. coli* genome of 50x, meaning that every position appears on average in 50 reads. Such data are designed for genome assembly purposes and thus contain a huge number of overlaps between the reads. The set contains 4,503,422 reads for a norm of 454,845,622 symbols.

Our algorithm ran on a simple core of a standard laptop equipped with 8 gigabytes of RAM; it took 272 seconds and used less than 5.5 gigabytes of memory. The EHOG had 46,566,901 nodes. The Shortest Cyclic Cover had length 187,250,434, $\ell_{\min}$ was equal to 187,250,434, while $\ell_{\max}$ was 187,250,672 symbols long (41% of $||P||$), making a difference

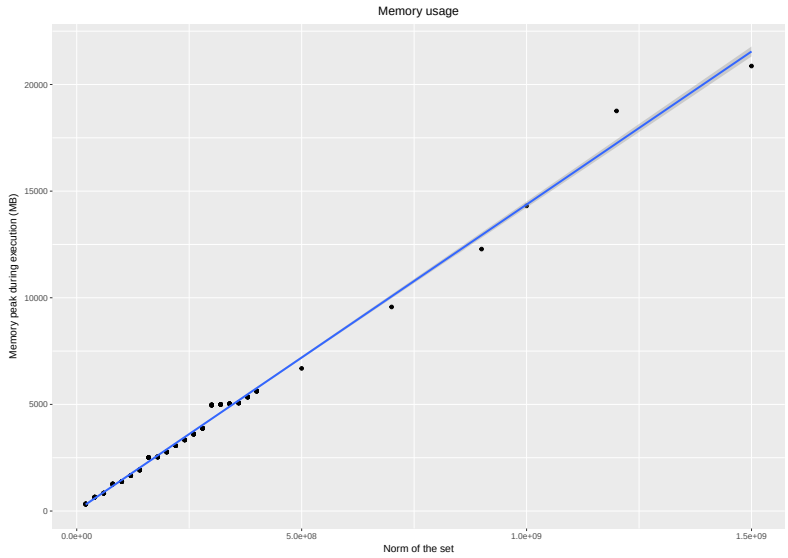

Figure 3 Memory peak (black dots) during execution in function of the norm of the input set.

of 710 symbols. Hence, the results on real data confirm our observations on synthetic data.

5 Conclusions

Here, we provide an algorithm to compute practical lower and upper bounds on the length of an optimal superstring. Importantly, those bounds are computed in a deterministic way. They appear to be very tight in practice on synthetic and genomic data (although there is little to compare to due to the lack of published experiments on approximation of known algorithms). Theorem 2 gives an upper bound of 4 for the ratio between $\ell_{\max}$ and $\ell_{\min}$. Empirically, this ratio is several orders of magnitude lower, meaning that the superstring is very close to the optimum. This result does not contradict the existence of a lower bound for `SLS` approximation ratio (see Figure 1 or [22]). For SLS, improving **`MGreedy`** into **`TGreedy`** algorithm led to an approximation ratio of 3. The same improvement is possible with our algorithm **`MGreedyMin`** and also would lead to the same ratio. This is left for future work.

Unfortunately, it is complex to understand why **`MGreedyMin`** yields an empirical ratio so close to the optimum. Several factors come into play. First, it turns out that the Shortest Cyclic Cover of P often contains a single cyclic word. In that case, this optimal cyclic cover also is an optimal cyclic superstring, which is necessarily shorter than the optimal linear superstring. Second, the cyclic superstring often corresponds to a path that uses the empty word as an overlap. In that case, the cyclic superstring can be cut between the two corresponding words and makes up a shortest linear superstring of exactly the same length, which is then optimal [5]. Another fact is important: if a cyclic string c_k of the cyclic cover merges at least two words of P, say s_i and s_j, then the difference between a shortest superstring of these words and c_k is smaller than the shortest word occurring in c_k.

The fact that **`MGreedyMin`** concatenates in an arbitrary order the linear strings to form a superstring makes no sense in DNA assembly or in genomics applications. The order of strings obtained by merging reads (which are called *contigs*) are determined *a posteriori* by a subsequent step of assembly pipelines named *scaffolding* using additional data like optical or genomic maps, long reads, or chromosomal capture data (Hi-C).

Acknowledgements: This work is supported by the Institut de Biologie Computationnelle (ANR-11-BINF-0002) and Défi MASTODONS C3G from CNRS.

References

1. A. V. Aho and M. J. Corasick. Efficient string matching: An aid to bibliographic search. *Communincations of ACM*, 18(6):333–340, June 1975.
2. A. Blum, T. Jiang, M. Li, J. Tromp, and M. Yannakakis. Linear approximation of shortest superstrings. *Journal of the ACM*, 41(4):630–647, 1994.
3. R. Cánovas, B. Cazaux, and E. Rivals. The Compressed Overlap Index. *ArXiv e-prints*, abs/1707.05613, 2017.
4. 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.
5. B. Cazaux and E. Rivals. A linear time algorithm for Shortest Cyclic Cover of Strings. *J. of Discrete Algorithms*, 37:56–67, 2016. http://dx.doi.org/10.1016/j.jda.2016.05.001.
6. B. Cazaux and E. Rivals. Hierarchical Overlap Graph. *ArXiv e-prints*, 1802.04632v2, Feb. 2018.
7. B. Cazaux and E. Rivals. Superstrings with multiplicities. In D. S. Gonzalo Navarro and B. Zhu, editors, *29th Annual Symposium on Combinatorial Pattern Matching, CPM 2018, July 2–4, 2018, Qingdao, China*, volume 105 of *LIPIcs*, page in press, 2018.
8. M. Crochemore, M. Cygan, C. S. Iliopoulos, M. Kubica, J. Radoszewski, W. Rytter, and T. Walen. Algorithms for three versions of the shortest common superstring problem. In *Proc. of 21st Annual Symp. Combinatorial Pattern Matching (CPM)*, volume 6129 of *Lecture Notes in Computer Science*, pages 299–309. Springer, 2010.
9. G. Fici, T. Kociumaka, J. Radoszewski, W. Rytter, and T. Walen. On the greedy algorithm for the Shortest Common Superstring problem with reversals. *Information Processing Letters*, 116(3):245–251, 2016.
10. T. P. Gevezes and L. S. Pitsoulis. *Optimization in Science and Engineering: In Honor of the 60th Birthday of Panos M. Pardalos*, chapter The Shortest Superstring Problem, pages 189–227. Springer New York, New York, NY, 2014.
11. A. Golovnev, A. Kulikov, and I. Mihajlin. Approximating Shortest Superstring Problem Using de Bruijn Graphs. In *Combinatorial Pattern Matching*, volume 7922 of *Lecture Notes in Computer Science*, pages 120–129. Springer Verlag, 2013.
12. D. Gusfield. *Algorithms on strings, trees, and sequences: computer science and computational biology*. Cambridge University Press, 1997.
13. T. Jiang, M. Li, and D. Du. A note on shortest superstrings with flipping. *Information Processing Letters*, 44(4):195–199, 1992.
14. J. D. Kececioglu and E. W. Myers. Combinatiorial algorithms for DNA sequence assembly. *Algorithmica*, 13(1/2):7–51, 1995.
15. B. Ma. Why greed works for shortest common superstring problem. *Theoretical Computer Science*, 410(51):5374–5381, 2009.
16. M. Mucha. Lyndon words and short superstrings. In *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA*, pages 958–972, 2013.
17. K. E. Paluch. Better Approximation Algorithms for Maximum Asymmetric Traveling Salesman and Shortest Superstring. *CoRR*, abs/1401.3670, 2014.
18. C. H. Papadimitriou and K. Steiglitz. *Combinatorial optimization : algorithms and complexity*. Dover Publications, Inc., 2nd edition, 1998. 496 p.
19. H. J. Romero, C. A. Brizuela, and A. Tchernykh. An Experimental Comparison of Approximation Algorithms for the Shortest Common Superstring Problem. In *5th Mexican International Conference on Computer Science*, pages 27–34, 2004.

20 J. Tarhio and E. Ukkonen. A greedy approximation algorithm for constructing shortest common superstrings. *Theoretical Computer Science*, 57:131–145, 1988.

21 E. Ukkonen. A Linear-Time Algorithm for Finding Approximate Shortest Common Superstrings. *Algorithmica*, 5:313–323, 1990.

22 V. Vassilevska. Explicit inapproximability bounds for the shortest superstring problem. In *30th Int. Symp. on Mathematical Foundations of Computer Science (MFCS)*, volume 3618 of *Lecture Notes in Computer Science*, pages 793–800. Springer, 2005.

23 M. Weinard and G. Schnitger. On the greedy superstring conjecture. *SIAM Journal on Discrete Mathematics*, 20(2):502–522, 2006.