Parallel and Sequential Approximation of Shortest Superstrings

Artur Czumaj[1⋆†] Leszek Gąsieniec[2⋆⋆] Marek Piotrów[1⋆⋆⋆†] Wojciech Rytter[2⋆⋆]

[1] Heinz Nixdorf Institut, Universität-GH-Paderborn,
Warburger Str. 100, 33098 Paderborn, Germany.
[2] Instytut Informatyki, Uniwersytet Warszawski, Banacha 2, 02-097 Warszawa, Poland.

Abstract. Superstrings have many applications in data compression and genetics. However the decision version of the shortest superstring problem is $\mathcal{NP}$-complete. In this paper we examine the complexity of approximating a shortest superstring. There are two basic measures of the approximations: the compression ratio and the approximation ratio. The well known and practical approximation algorithm is the sequential algorithm GREEDY. It approximates the shortest superstring with the compression ratio of $\frac{1}{2}$ and with the approximation ratio of 4. Our main results are:

(1) An $\mathcal{NC}$ algorithm which achieves the compression ratio of $\frac{1}{4+\varepsilon}$.
(2) The proof that the algorithm GREEDY is not parallelizable, the computation of its output is P-complete.
(3) An improved sequential algorithm: the approximation ratio is reduced to 2.83. Previously it was reduced by Teng and Yao from 3 to 2.89.
(4) The design of an $\mathcal{RNC}$ algorithm with constant approximation ratio and an $\mathcal{NC}$ algorithm with logarithmic approximation ratio.

1 Introduction

Let $S = \{s_1, \ldots, s_n\}$ be a set of n strings over some alphabet Σ. A *superstring* of S is a string sp over Σ such that each string $s_i \in S$ appears as a substring (a consecutive block of characters) of sp. *The shortest superstring* problem is to find for a given set S the shortest superstring $\mathrm{SS}(S)$. We use $opt(S)$ to denote the length of $\mathrm{SS}(S)$. Assume further that no string $s_i \in S$ is a substring of any other $s_j \in S$.

It is known that the shortest superstring problem is $\mathcal{NP}$-hard [5, 6]. Because of its important applications in data compression practice [13] and DNA sequencing procedure [8, 12], it is of interest to find approximation algorithms with good performance guarantees. For example, a DNA molecule can be represented as a character string over a set of nucleotides $\{A, C, G, T\}$. Although a DNA string can have up

⋆ Supported by DFG-Graduiertenkolleg "Parallele Rechnernetzwerke in der Produktionstechnik", ME 872/4-1. Email: artur@hni.uni-paderborn.de.
⋆⋆ Supported in part by the EC Cooperative Action IC 1000 Algorithms for Future Technologies "ALTEC". Email: {lechu,rytter}@mimuw.edu.pl.
⋆⋆⋆ Supported in part by Alexander von Humboldt-Stiftung and Volkswagen Stiftung. Permanent address: Instytut Informatyki, Uniwersytet Wrocławski, Przesmyckiego 20, 51-151 Wrocław, Poland. Email: marekp@uni-paderborn.de.
† Supported in part by the ESPRIT Basic Research Action No. 7141 (ALCOM II)

to 3×10^9 characters (for a human being), with current laboratory methods only small fragments of at most 500 characters can be determined at a time. Then from a huge number of these fragments, a biochemist should construct the superstring representing the whole molecule. Efficient superstring approximation algorithms are routinely used to cope with this job. In particular, good parallel algorithms would be useful in that context.

To evaluate how good is the obtained approximation, two kinds of measure are used. The first, most important in practice, is to find a superstring sp of S such that the ratio $\frac{|sp|}{opt(S)}$ is minimized. We will call this ratio to be the *approximation factor of a superstring*. The second approach is to find a superstring sp of S such that the ratio of the total compression obtained by sp and by SS(S) is maximized. That is, we want to maximize $\frac{|S|-|sp|}{|S|-opt(S)}$, where $|S| = \sum_{1 \leq i \leq n} \mid s_i \mid$. We will call this ratio to be the *compression factor of a superstring*.

The algorithm GREEDY is a simple sequential approximation of a shortest superstring and appears to do quite well. It can be presented in the following way. Given a non-empty set of strings $S = \{s_1, \ldots, s_n\}$, repeat the following steps until S contains just one string (which is a superstring of S): Select a pair of strings $s', s'' \in S$ that maximizes overlap between s' and s''; Remove s' and s'' from S replacing them with the merge of s' and s''.

It was proved by Tarhio and Ukkonen [14] and Turner [16] that GREEDY achieves the compression factor of at least 1/2. Other heuristics have been also considered by the authors, but 1/2 is still the best obtained compression factor. The approximation factor of GREEDY was unknown for a long time. The first breakthrough was made by Blum et al. [2], where they proved that GREEDY achieves an approximation factor of 4. Furthermore, they showed a modified greedy algorithm that has an approximation factor of 3, and proved that the superstring problem is MAX-SNP-hard [11]. The recent result in [1] that MAX SNP-hard problems do not have polynomial time approximation scheme unless $\mathcal{P} = \mathcal{NP}$ implies that a polynomial time approximation scheme (that is, polynomial time algorithms with approximation factor of $1 + \varepsilon$ for any fixed $\varepsilon > 0$) for this problem is unlikely. Recently Teng and Yao [15] improved the result of Blum et al. [2] and presented an algorithm that achieved an approximation factor of 2.89. Our contribution is an algorithm whose approximation factor can be bounded by 2.83.

As far as we know, no parallel approximation algorithm for the superstring problem has been presented. In this paper, we give the following results concerning the parallel complexity of the problem:

1. An $\mathcal{NC}$ algorithm which achieves the compression ratio of $\frac{1}{4+\varepsilon}$.
2. The proof that the algorithm GREEDY is not parallelizable, the computation of its output is P-complete.
3. The design of an $\mathcal{RNC}$ algorithm with constant approximation factor and an $\mathcal{NC}$ algorithm with logarithmic approximation factor.

The open problem is to construct an $\mathcal{NC}$ algorithm with a constant approximation factor.

Below we introduce some necessary definitions.

For two strings s and t let v be the longest string such that $s = uv$ and $t = vw$ for some non-empty strings u and w. The *overlap* between two strings s and t is the

length of the string v. We will denote it as $\mathrm{ov}(s,t)$. The *prefix* of a string s with respect to a string t is the length of the string u. We will denote it as $\mathrm{pref}(s,t)$. It will cause no confusion if sometimes we also call the string v to be the overlap ($\mathrm{ov}(s,t)$) and the string u to be the prefix ($\mathrm{pref}(s,t)$) of s and t. Define $s \circ t$ to be the string uvw, that is $s \circ t = \mathrm{pref}(s,t)\, t$.

For a given set of strings $S = \{s_1, \ldots, s_n\}$ define an *overlap graph of* S to be the weighted digraph $OG(S) = (V, E, \mathrm{ov})$ which has n vertices $V = \{1, \ldots, n\}$ and n^2 edges $E = \{(i,j) \ : \ 1 \leq i, j \leq n\}$. Here we take as weight function the overlap $\mathrm{ov}(,)$: edge (i,j) has weight $\mathrm{ov}(i,j) = \mathrm{ov}(s_i, s_j)$.

Algorithm GREEDY can be also restated in terms of the overlap graph $OG(S)$. Repeat until selected edges do not form a Hamiltonian path in $OG(S)$: Scan the edges of $OG(S)$ in non-increasing order of weights and select an edge (i,j) if no edge of the form (i,p) or (q,j) has been previously selected and if the collection of paths constructed so far does not include a path from j to i. Obtained Hamiltonian path $(i_1, i_2), (i_2, i_3), \cdots, (i_{n-1}, i_n)$ defines us the superstring $s_{i_1} \circ s_{i_2} \circ s_{i_3} \circ \ldots \circ s_{i_{n-1}} \circ s_{i_n}$.

A weighted digraph G is an *overlap graph* if there exists a set S of strings such that the graph obtained from $OG(S)$ after removing all zero-weighted edges is isomorphic to G.

Due to space limitations some proofs and details are omitted here and will appear in the full version of the paper.

2 $\mathcal{NC}$-Approximation of Shortest Superstrings

Define a *cycle-cover* of a graph G to be a maximal collection of cycles in G such that each vertex is in at most one cycle. Let also a *path-cycle cover* be a collection of paths and cycles in G such that each vertex is in exactly one path or cycle. We call a path-cycle cover *maximal* if it can not be extended by any other edge in G.

Let $G = (V, E, w)$ be a complete weighted digraph without selfloops, where $V = \{1, \ldots, n\}$ is the set of vertices, $E = \{(i,j) \ : \ i \neq j \in V\}$ is the set of edges and $w : E \rightarrow \mathbb{R}_+$ is the (non-negative) weight function. Assume also that the unary weights are given. The *maximum cycle-cover* problem is to find a cycle-cover with the maximum weight (ie., the total weight of the cycles is maximized). It is known that this problem is reduced to the maximum-weighted matching problem in bipartite graphs [10]. Thus it can be solved in polynomial sequential time. However there is not known any $\mathcal{NC}$ algorithm for it. In this paper we are only focused on this problem for the case when all weights are given in unary. In this case there is known an $\mathcal{RNC}$ algorithm for the maximum-weighted matching problem in bipartite graphs, thus also for the maximum cycle-cover problem [17]. In what follows we show that there is an $\mathcal{NC}$ algorithm that finds an $(\frac{1}{2+\varepsilon})$-approximation of the maximum cycle-cover (and also for the maximum-weighted matching problem in bipartite graphs).

2.1 Sequential Approximation of a Maximum Cycle-Cover

We begin with a sequential $\frac{1}{2}$-approximation. The following is a simple greedy algorithm that finds a cycle-cover.

Algorithm CC-GREEDY :
Repeat until selected edges do not form a cycle-cover of G:

Scan the edges of G in non-increasing order of weight and select an edge (i, j) if no edge of the form (i, p) or (q, j) has been previously selected.

Lemma 1. *Algorithm* CC-GREEDY *finds a cycle-cover of weight that is at least half of the weight of a maximum cycle-cover.*

Proof. The proof follows the ideas of Turner's estimation of the superstring compression factor achieved by GREEDY [16]. Algorithm CC-GREEDY selects n edges in non-increasing order and let e_i be the i-th chosen edge. Let P_i be a maximum cycle-cover that includes edges $\{e_1, \ldots, e_i\}$ and let $C_i = P_i - \{e_1, \ldots, e_i\}$. We show that for $1 \leq i \leq n$, $w(C_{i-1}) \leq w(C_i) + 2w(e_i)$. Since $w(C_0)$ is the weight of a maximum cycle-cover and $w(C_n) = 0$, this would imply the lemma.

Let $e_i = (p, q)$. Since e_i is the i-th edge chosen by CC-GREEDY , $w(e_i) \geq \max\{w(e) : e \in C_{i-1}\}$. There are at most two edges e' and e'' that are in $C_{i-1} - C_i$ and which share the head or the tail with e_i. Let $e' = (p, s)$, $e'' = (t, q)$ and when $s \neq t$, let us define $e^* = (t, s)$, Then we can obtain a cycle-cover $(P_{i-1} - \{e', e''\}) \cup \{e_i, e^*\}$ when $s \neq t$ or a cycle-cover $(P_{i-1} - \{e', e''\}) \cup \{e_i\}$ otherwise. In both cases we have:

$$\begin{aligned} w(C_i) + w(e_i) &\geq w((C_{i-1} - \{e', e''\}) \cup \{e_i\}) \\ &= w(C_{i-1}) - w(e') - w(e'') + w(e_i) \\ &\geq w(C_{i-1}) - w(e) - w(e) + w(e_i) \\ &= w(C_{i-1}) - w(e_i) \end{aligned}$$

which implies $w(C_{i-1}) \leq w(C_i) + 2w(e_i)$.

Using well-known transformations (see eg. [10]) Lemma 1 can be restated as the following Corollary.

Corollary 2. *In a weighted bipartite graph the* GREEDY *matching algorithm finds a matching of weight that is at least half of the weight of a maximum-weighted matching.*

2.2 Parallel Approximation of a Maximum Cycle-Cover

In this section we describe an approximation of algorithm CC-GREEDY . Let $c > 1$, $G = (V, E, w)$ and for each edge e if $w(e) \in (c^{k-1}, c^k]$ then define c-level of e, $\text{LEVEL}_c(e)$, to be k (ie., $\text{LEVEL}_c(e) = \lceil \log_c w(e) \rceil$); additionally if $w(e) \leq 1$ then $\text{LEVEL}_c(e) = 0$.

For a given path $v_1, v_2, \ldots, v_k$ its *contraction* is defined as follows. We remove vertices $v_2, \ldots, v_k$ together with the edges incident to them and change the edges (with their weights) outgoing from v_1 to be the edges outgoing previously from v_k. Vertex v_1 is called the *contraction* of the path $v_1, v_2, \ldots, v_k$. The *recontraction of a cycle or a path* is obtained by recursively replacing each contraction by its path.

Algorithm ACC-GREEDY

1. Let s be the maximum c-level of edges in G (ie., $s = \max_{e \in E}\{\text{LEVEL}_c(e)\}$) and let $\tilde{G} = G$ and $C = \emptyset$.
2. Repeat for $t = s$ downto $t = 0$
   (a) Let G_t be the (non-weighted) subgraph of $\tilde{G}$ induced by the edges from the t-th c-level
   (b) Find a maximal path-cycle cover of G_t
   (c) Remove all cycles in G_t from $\tilde{G}$ and after recontracting add them to C; contract all obtained paths
3. The set C contains a cycle-cover

Lemma 3. *For any $c > 1$ algorithm* ACC-GREEDY *finds a cycle-cover of weight at least $\frac{1}{2c}$ of the weight of a maximum cycle-cover.*

Proof. Let $\overline{G} = (V, E, \overline{w})$ be the graph with weights

$$\overline{w}(e) = \begin{cases} c^{\text{LEVEL}_c(e)} & \text{if ACC-GREEDY chooses } e \\ w(e) & \text{otherwise} \end{cases}$$

Note that $w(e) \leq \overline{w}(e) < c\, w(e)$ for every $e \in E$. Let $\text{MCC}(G)$ be the weight of a maximum cycle-cover in G and $w(AM(G))$ be the sum of weights of the edges chosen by algorithm ACC-GREEDY on G. Observe that algorithm ACC-GREEDY on the graph $\overline{G}$ is equivalent to CC-GREEDY assuming that both algorithms select the same edges in the case of equal weights. Thus $w(AM(\overline{G})) \geq \frac{1}{2}\text{MCC}(\overline{G}) \geq \frac{1}{2}\text{MCC}(G)$. On the other hand $w(AM(\overline{G})) < c\, w(AM(G))$. Thus finally we get $c\, w(AM(G)) \geq \frac{1}{2}\text{MCC}(G)$.

Now we want to show that this algorithm can be implemented to run in poly-logarithmic time with polynomial number of processors. The main loop is executed $\lceil \log_c(\max_{e \in E}\{w(e)\}) \rceil$ times. Since the weights of the graph are given in unary, we only have to show that there is an $\mathcal{NC}$ algorithm that finds a maximal path-cycle cover. The proof of the following lemma will appear in the full version of the paper.

Lemma 4. *There is an $\mathcal{NC}$ algorithm that finds a maximal path-cycle cover of a digraph G.*

Lemma 3 and Lemma 4 immediately imply the following theorem.

Theorem 5. *There is an $\mathcal{NC}$ algorithm that finds a cycle-cover of a weighted digraph G with the cost of at least $\frac{1}{2+\varepsilon}$ of the weight of a maximum cycle-cover. Here $\varepsilon > 0$ is any arbitrary but fixed constant, and we assume that the weights of G are given in unary.*

The running time of this algorithm is either $O(\log^2 n \log_{1+\varepsilon}(\max_{e \in E}\{w(e)\}))$ with n^4 processors or $O(\log^3 n \log_{1+\varepsilon}(\max_{e \in E}\{w(e)\}))$ with n^2 processors.

2.3 Parallel Approximation of Shortest Superstrings

In this section we develop techniques presented in the previous section to design an $\mathcal{NC}$ algorithm that finds a superstring that has the overlap (the compression measure) at least $\frac{1}{4+\varepsilon}$ that of a shortest superstring.

First we build the overlap graph $OG(S)$ for the set of strings S. We assume in this construction that there is no selfloop in $OG(S)$. Then we find a cycle-cover of OG using algorithm ACC-GREEDY from Section 2.2. We next remove from every cycle an edge with the minimum weight and join (by any edges) obtained paths to get a Hamiltonian path.

Lemma 6. *Obtained Hamiltonian path is of the weight at least $\frac{1}{4+\varepsilon}$ of the weight of a maximum weight Hamiltonian path, for any $\varepsilon > 0$.*

Proof. Any maximum cycle-cover MCC is of weight not smaller than the weight of a maximum Hamiltonian path MHP. Let C be a cycle-cover obtained by algorithm ACC-GREEDY and GS be a superstring obtained by the algorithm presented above. From Theorem 5 we get $w(C) \geq \frac{1}{2+\frac{1}{2}\varepsilon} w(\mathrm{MCC})$ for any $\varepsilon > 0$. Since we remove the least weighted edge from every cycle in C, we get $w(GS) \geq w(C)/2$. Thus

$$w(GS) \geq w(C)/2 \geq \frac{1}{4+\varepsilon} w(\mathrm{MCC}) \geq \frac{1}{4+\varepsilon} w(\mathrm{MHP})$$

Theorem 7. *There is an $\mathcal{NC}$ algorithm that achieves compression factor of $\frac{1}{4+\varepsilon}$. It runs either in $O(\log^2 n \cdot \log_{1+\varepsilon} |S|)$ time with n^4 processors or in $O(\log^3 n \cdot \log_{1+\varepsilon} |S|)$ time with n^2 processors.*

3 Algorithm GREEDY Is Not Parallelizable

Algorithm GREEDY appears to be very sequential in nature, since to select a current pair of strings with the largest overlap we need to know the results of previous merges. To formalize this observation we would like to prove that GREEDY applied to the superstring problem is $\mathcal{P}$-complete [3], what is commonly believed to mean: a hardly parallelizable algorithm.

We start with proving that the problem of finding the Hamiltonian path chosen by algorithm GREEDY is $\mathcal{P}$-complete. For a given Boolean circuit, a certain complete weighted digraph is introduced, in which a Hamiltonian path selected by GREEDY can simulate a computation of the circuit's value. Then we argue that the digraph is an overlap graph, i.e., a set of strings can be constructed whose overlap graph is isomorfic to the digraph.

Lemma 8. *The problem of finding the Hamiltonian path chosen by algorithm* GREEDY *is P-complete.*

Proof. Due to space limitations the construction of the graph is omitted here. From now on we will call the result digraph the *circuit-simulating graph*.

[3] To be more precise: a search problem of finding a superstring obtained by GREEDY is considered and proved to be P-complete

For a digraph G define its *skeleton* $\overline{G}$ to be an undirected graph with the vertex set the same as the vertex set of G and the edge set which is obtained from the edge set of G by removing directions.

In the following lemmas we would like to derive some sufficient conditions of a digraph to be an overlap graph. The first observation is that a positive-weighted edge in an overlap graph relates the beginning of one string to the end of another. Therefore, when we want to collect the related strings in a structure, we have to consider adjacent edges in alternating directions. This leads us to the following definitions:

An *alternating path* is a sequence of nodes and edges $v_1e_1v_2e_2\cdots v_{k-1}e_{k-1}v_k$ such that either $e_1 = (v_1, v_2), e_2 = (v_3, v_2), e_3 = (v_3, v_4), e_4 = (v_5, v_4)\cdots$, or $e_1 = (v_2, v_1), e_2 = (v_2, v_3), e_3 = (v_4, v_3), e_4 = (v_4, v_5)\cdots$. An *alternating tree* is a connected (ie., $\overline{\mathrm{AT}}$ is connected) digraph $\mathrm{AT} = (V_T, E_T)$ containing t nodes and $t-1$ edges such that each sequence $v_1e_1v_2e_2\cdots v_{l-1}e_{l-1}v_l$, where $v_i \in V_T$, $e_i \in E_T$ and $v_1e_1v_2e_2\cdots v_{l-1}e_{l-1}v_l$ is a path in $\overline{\mathrm{AT}}$, is an alternating path. Assume that all edges in AT have weights defined by a function w. A weighted alternating tree AT is *monotone* if there exists a vertex r, called the *root* of AT, such that for each alternating path $re_1v_1e_2v_2\cdots e_kv_k$ in AT, $w(e_1) < w(e_2) < \cdots < w(e_k)$ An *alternating cycle* in a digraph G is a cycle in $\overline{G}$ that can be transformed into an alternating path by splitting it in a node.

Lemma 9. *Every monotone alternating tree with positive, integer weights is an overlap graph.*

Lemma 10. *If a weighted digraph G with positive, integer weights can be edge-covered by a disjoint sum of monotone alternating trees in such a way that for each vertex in G all its incoming edges are in one tree and all its outgoing edges are in another tree, then G is an overlap graph.*

Proof. According to Lemma 9, each alternating tree AT in the cover C of G is an overlap graph. That is, strings can be assigned to nodes of AT to obtain the corresponding overlap graph. Let Σ_{AT} denote an alphabet of the strings. W.l.o.g. we can assume that alphabets Σ_{AT}, $\mathrm{AT} \in C$, are pairwise disjoint. Let the incoming edges of a vertex v be in AT and its outgoing edges in AT$'$. Thus we have got two strings $in_v \in \Sigma^*_{AT}$ and $out_v \in \Sigma^*_{AT'}$. The result string for v we obtain by concatenating in_v with out_v. It can be easily checked that the overlap between two such strings is non-zero if and only if the corresponding nodes are joint by an edge.

Lemma 11. *If a digraph G does not contain alternating cycles, then it can be (uniquely) edge-covered by edge-disjoint alternating trees such that each vertex of G has all its incoming edges in one tree and all its outcoming edges in another tree.*

Theorem 12. *The problem of finding a superstring chosen by algorithm* GREEDY *is $\mathcal{P}$-complete.*

Proof. With respect to Lemma 8 we have only to show that a circuit-simulating graph G is an overlap graph. The gates in G can been designed in such a way that G contains no alternating cycles. By Lemma 11, G can be uniquely edge-covered

by disjoint alternating trees. Moreover the trees are of size bounded by a constant (independent on the size of a circuit) and they are monotonic. Hence, by Lemma 10, G is an overlap graph.

4 Sequential Algorithm with 2.83 Approximation Factor

In this section we present a new sequential algorithm for the superstring problem that has an approximation factor of $2\frac{5}{6}$ and thus supersedes the algorithm of [15]. The later algorithm has the factor of $2\frac{8}{9}$ and is an improvement on algorithm TGREEDY [2] that has the factor of 3. We base on the ideas from both papers.

In this section, according to the previous papers, we will use the terms *assignment* and *cycle-cover* interchangeably. For a given set of strings $S = \{s_1, \ldots, s_n\}$ we consider two complete digraphs with S as a set of nodes: one $OG(S)$ weighted by $\mathrm{ov}(\cdot,\cdot)$, the other $PG(S)$ weighted by $\mathrm{pref}(\cdot,\cdot)$. We assume that both contain no selfloop. Let us notice that a minimum assignment in $PG(S)$ is also a maximum assignment in $OG(S)$, and vice versa. We will call such an assignment an *optimal assignment*. For a cycle c in an assignment C, let $d(c)$ denote the total $\mathrm{pref}(\cdot,\cdot)$ weight of the edges in c. We refer to $d(c)$ as the *weight* of c.

For the sake of completeness we recall two crucial lemmas from [2] and [15]. Recall that a minimum assignment is called *canonical* if each string s is assigned to a cycle whose weight is the smallest among all cycles that s fits (see [15]).

Lemma 13. [2] *Let c_1 and c_2 be two cycles in a minimum weight assignment C with $s_1 \in c_1$ and $s_2 \in c_2$. Then, the overlap between s_1 and s_2 is less than $d(c_1) + d(c_2)$.*

Lemma 14. (2-cycle Lemma, Teng and Yao [15])
Let c_1 and c_2 be two cycles in a canonical minimum assignment C with $r_1 \in c_1$ and $r_2 \in c_2$. Then $ov(r_1, r_2) + ov(r_2, r_1) < \max(|r_1|, |r_2|) + \min(d(c_1), d(c_2))$.

As the shortest superstring problem for S corresponds to the maximum Hamiltonian path problem in $OG(S)$ graph, approximation schemes start by computing an optimal assignment. Then the problem is how to join the cycles in the assignment to obtain a Hamiltonian path. Algorithm TGREEDY [2] opens each cycle by deleting the edge with the shortest overlap and joins the obtained strings by GREEDY . One can obtain the same approximation factor of 3 by the following procedure:

1. select a set R of cycles' representatives (one node s_i from each cycle c_i) and find an optimal assignment CC of R;
2. open each cycle in CC by deleting the shortest-overlap edge, and concatenate the obtained strings to form α;
3. split each cycle c_i in the selected node s_i (the result will begins and ends with s_i) and replace s_i in α by the results.

These stages form the basis of the algorithm presented by Teng and Yao [15]. The improvement on the approximation factor in [15] is obtained by making the assignment in Stage 1 canonical and by treating separately 2-cycles in Stage 2. Our further improvement is achieved by selecting "good" representatives of 2-cycles and 3-cycles in Stage 2, and by finding an optimal assignment on them.

Below there is an analog of Lemma 14 for a 3-cycle.

Lemma 15. (3-cycle Lemma)
Let c_1, c_2 and c_3 be cycles in a minimum assignment C with $r_1 \in c_1$, $r_2 \in c_2$ and $r_3 \in c_3$. Then $ov(r_1, r_2) + ov(r_2, r_3) + ov(r_3, r_1) \leq 2 \cdot \min(|r_1|, |r_2|, |r_3|) + d(c_1) + d(c_2) + d(c_3)$.

Proof. Without loss of generality we can assume that $|r_1| = \min(|r_1|, |r_2|, |r_3|)$. Then $ov(r_1, r_2) \leq |r_1|$, $ov(r_3, r_1) \leq |r_1|$. By Lemma 13, $ov(r_2, r_3) \leq d(c_2) + d(c_3)$.

The Algorithm

1. Find an optimal assignment C of S, and make C canonical.
2. Take an arbitrary string from each cycle of C to form a set R, and find an optimal assignment CC for R.
3. Select representative set RR containing one element for each 2-cycle in CC and one element for each 3-cycle in CC. From each 2-cycle take the longer string and from each 3-cycle take a string that is not in the pair with the longest overlap (i.e. when s_1, s_2 and s_3 are the strings of a 3-cycle, $ov(s_1, s_2) \geq ov(s_2, s_3)$ and $ov(s_1, s_2) \geq ov(s_3, s_1)$, then select s_3). Let $RR = \{g_1, g_2, \ldots, g_m\}$. If g_i is from a 2-cycle then let f_i be the second element of the cycle. If g_i is from a 3-cycle then let f_i be the superstring of two other strings in the cycle, ordered as on the cycle. Let $RR' = \{f_1, f_2, \ldots, f_m\}$.
4. Find an optimal assignment CCC on RR. Create two superstrings for $RR \cup RR'$ by splitting cycles in CCC and inserting strings from RR' (see Fig. 1). Cycles with even number of nodes are split exactly (i.e. every edge of a cycle is built in a superstring), in cycles with odd number of nodes the edge with a smallest overlap is left out. Let q_0 and q_1 be the result superstrings and let q be the shorter of them.

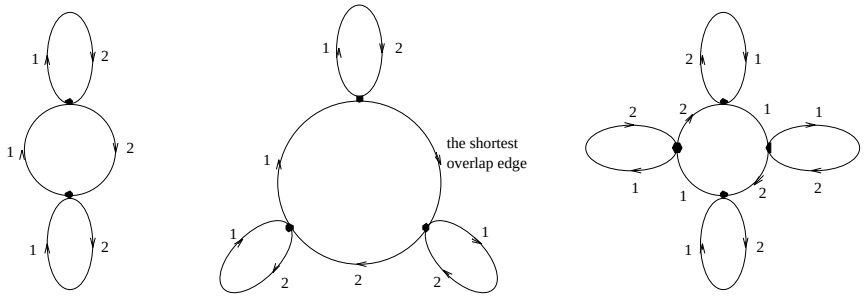

Fig. 1. Splitting CCC cycles together with 2,3-cycles from CC into two superstrings. Edges labeled with 1(2) are in the superstring g_1 (g_2).

5. Open each i-cycle, $i \geq 4$, in CC by deleting the edge with the smallest overlap; concatenate the resulting strings together with q to obtain α. Note that α is a superstring for R. Split each cycle in C in a node from R to obtain superstrings that begin and end with the strings from R. Let $\bar{\alpha}$ be the extended string of

α obtained by replacing each string of R with the superstring representing its cycle in C. Return $\bar{\alpha}$.

Analysis

Observe first that the algorithm runs in polynomial time, because an optimal assignment can be constructed in time $O(n^3)$ (see e.g. [10]), and for given a minimum assignment one can transform it into a canonical minimum one in $O(n|S|)$ time [15].

Let d_2, d_3, d_4 be, respectively, the total weight of the cycles in C that have representatives in 2-cycles, 3-cycles, and all i-cycles for $i \geq 4$, in assignment CC. Thus $d_2 + d_3 + d_4 = d(C)$. Let ov_2, ov_3 and ov_4 be, respectively, the total overlap present in the 2-cycles, 3-cycles, and all i-cycles for $i \geq 4$, in assignment CC. Then, we have: $ov(q_1) + ov(q_2) \geq ov_2 + ov_3 + ov_3/3 + \frac{2}{3}ov(CCC)$, since in each 3-cycle of CC we take twice the edge with the longest overlap and in CCC-cycles with the odd number of edges only the edge with the smallest overlap is deleted. Since $ov(CCC) \geq ov_{opt}(RR) = |RR| - opt(RR)$, we get

$$ov(q) \geq \frac{1}{2}ov_2 + \frac{2}{3}ov_3 + (|RR| - opt(RR))/3.$$

Let RR_i, $i = 2, 3$, denote the set $\{r \in RR : r \text{ is in a } i \text{ cycle in } CC\}$. Then $ov(\alpha) \geq ov(q) + \frac{3}{4}ov_4 = \frac{1}{2}ov_2 + \frac{2}{3}ov_3 + \frac{3}{4}ov_4 + (|RR_2| + |RR_3| - opt(RR))/3$.

Lemma 16.

$$|RR_2| \geq ov_2 - d_2/2,$$
$$|RR_3| \geq ov_3/2 - d_3/2.$$

Proof. The first inequality follows from Lemma 14 and the second from Lemma 15.

Combining all observations above we get,

$$\begin{aligned}
|\alpha| &= |R| - ov(\alpha) = opt(R) + ov_{opt}(R) - ov(\alpha) \leq opt(R) + ov(CC) - ov(\alpha) \\
&\leq opt(R) + opt(RR)/3 + \frac{1}{2}ov_2 + \frac{1}{3}ov_3 + \frac{1}{4}ov_4 - (|RR_2| + |RR_3|)/3 \\
&\leq opt(R) + opt(RR)/3 + \frac{1}{6}ov_2 + \frac{1}{6}ov_3 + \frac{1}{4}ov_4 + \frac{1}{6}(d_2 + d_3) \\
&\leq opt(S) + opt(S)/3 + \frac{1}{3}d_2 + \frac{1}{3}d_3 + \frac{1}{2}d_4 + \frac{1}{6}(d_2 + d_3) \\
&= \frac{4}{3}opt(S) + \frac{1}{2}(d_2 + d_3 + d_4) \leq opt(S)(\frac{4}{3} + \frac{1}{2}) = 1\frac{5}{6}opt(S)
\end{aligned}$$

Since $|\bar{\alpha}| = |\alpha| + d(C)$, we obtain $|\bar{\alpha}| \leq 2\frac{5}{6}opt(S)$.

5 Parallel Approximations of Superstring Length

In this section we present an $\mathcal{NC}$ algorithm with a logarithmic approximation ratio and an $\mathcal{RNC}$ algorithm with a constant approximation ratio.

5.1 The Weighted Set Cover Problem

Let $X = \{1, 2 \ldots, n\}$ and let $Y = \{Y_1, Y_2, \ldots, Y_m\} \subseteq 2^X$ be a family of its subsets. A *cover* of X is a subcollection $Y' \subseteq Y$ such that $\bigcup_{Y_i \in Y'} Y_i = X$. For each set $Y_i \in Y$ let $w(Y_i)$ denote its (positive) weight and for a subcollection $Y' \subseteq Y$ define its weight by $w(Y') = \sum_{Y_i \in Y'} w(Y_i)$. The *weighted set cover* problem is to find a cover Y^* of X of the minimum weight.

The weighted set cover problem is known to be $\mathcal{NP}$-hard [6]. A recent result of Lund and Yannakakis [9] shows that this problem cannot be approximated in $\mathcal{P}$ with ratio $c \log_2 n$ for any $c < 1/4$ unless $\mathcal{NP} = \mathrm{DTIME}(n^{O(1)})$. However there is known a polynomial-time algorithm that finds a logarithmic-factor approximation. The following lemma has been shown by Berger et al. [3].

Lemma 17. *For any $\varepsilon > 0$, there is an $\mathcal{NC}$ algorithm for the weighted set cover problem that runs in $O(\log^4 n \log m \log^2(nm)/\varepsilon^6)$ time, uses $O(n + \sum_{i=1}^{m} \mid Y_i \mid)$ processors, and produces a cover of weight at most $(1+\varepsilon) \log n$ times the weight of a minimum cover.*

5.2 The $\mathcal{NC}$ Algorithm with Logarithmic Approximation Factor

For any s_i, s_j and $d, 0 \leq d < \min\{\mid s_i \mid, \mid s_j \mid\}$, let u and v be strings of the length d such that $s_i = xu$ and $s_j = vy$ for some non-empty string x and y. If $u = v$ then we define $\textsc{con}(i, j, d) = xuy$; otherwise $\textsc{con}(i, j, d)$ is undefined. If $\textsc{con}(i, j, d)$ is defined, then let $\mathrm{CON}(i, j, d) = \{s_i, s_j\} \cup \{s_k \ : \ s_k \text{ is a substring of } \textsc{con}(i, j, d)\}$. In this way we have obtained a family FAM-CON of subsets of S. For each $\mathrm{CON}(i, j, d) \in$ FAM-CON define its weight $w(i, j, d) = \mid \textsc{con}(i, j, d) \mid = \mid s_i \mid + \mid s_j \mid - d$.

Let $C = \{\mathrm{CON}(i_1, j_1, d_1), \mathrm{CON}(i_2, j_2, d_2), \ldots, \mathrm{CON}(i_c, j_c, d_c)\}$ be a cover of S defined by a collection of sets from FAM-CON . We can obtain the corresponding superstring $S_C = \textsc{con}(i_1, j_1, d_1) \circ \textsc{con}(i_2, j_2, d_2) \circ \cdots \circ \textsc{con}(i_c, j_c, d_c)$. Since C is a cover of S, every string from S must be a substring of S_C. Note also that $w(C) = \sum_{1 \leq k \leq c} \mathrm{CON}(i_k, j_k, d_k) = \sum_{1 \leq k \leq c} \mid \textsc{con}(i_k, j_k, d_k) \mid = \mid S_C \mid$. The following fact can be easily derived (see eg., [8]).

Fact 18. *Let C^* be a minimum weighted set cover of S. Then $w(C^*) = \mid S_{C^*} \mid \leq 2 \cdot opt(S)$.*

Now, suppose that we have found a set cover C such that $w(C) \leq t \cdot w(C^*)$, for some t. Then clearly $\mid S_C \mid \leq t \cdot \mid S_{C^*} \mid$. Thus, the superstring S_C has length at most $2 \cdot t \cdot opt(S)$. Hence using Lemma 17 we obtain the following theorem.

Theorem 19. *There is an $\mathcal{NC}$ algorithm that for any $\varepsilon > 0$, finds a superstring whose length is at most $(2+\varepsilon) \log n$ times the length of a shortest superstring.*

A similar construction was used implicitly by Li [8] for a sequential algorithm.

5.3 The $\mathcal{RNC}$ Algorithm with Constant Approximation Factor

Blum et al. [2] presented the following sequential algorithm for the approximation of the shortest superstring. Let G_S be the overlap graph for the set of strings S. Find

a maximum weight cycle-cover C on G_S, where $C = \{c_1, \ldots, c_p\}$ is the collection of cycles. For each cycle $c_i = i_1 \rightarrow \cdots \rightarrow i_r \rightarrow i_1$, let $\tilde{s_i} = s_{i_1} \circ \cdots \circ s_{i_r}$ where i_1 is arbitrary chosen. Then the final superstring is obtained by concatenating all together the strings $\tilde{s_i}$. Blum et al. [2] proved that this algorithm always finds a superstring of length at most $4 \cdot opt(S)$. It is well known that the problem of finding a maximum weight cycle-cover is equivalent to the problem of finding a maximum weight matching in bipartite graph. In general it is not known whether it can be done either $\mathcal{NC}$ or in $\mathcal{RNC}$. However, when the weights of the graph are given in unary one can find a maximum weight matching in $\mathcal{RNC}$ [17]. Since in our case the weights of G_S are given in unary, the construction given by Blum et al. [2] can be parallelized to get an $\mathcal{RNC}$ algorithm.

Theorem 20. *There exists an $\mathcal{RNC}$ algorithm that finds a superstring of length at most $4 \cdot opt(S)$.*

References

1. A. Arora, C. Lund, R. Motwani, M. Sudan, and M. Szegedy. Proof verification and hardness of approximation problems. 33rd FOCS, pp. 14–23, 1992.
2. A. Blum, T. Jiang, M. Li, J. Tromp, and M. Yannakakis. Linear approximation of shortest superstrings. 23rd STOC, pp. 328–336, 1991.
3. B. Berger, J. Rompel, and P. W. Shor. Efficient NC algorithms for set cover with applications to learning and geometry. 30th FOCS, pp. 54–59, 1989. The full version is to appear in *J. Algorithms*.
4. V. Chvatal. A greedy heuristic for the set-covering problem. *Mathematics of Operations Research* **4**(1979), pp. 233–235.
5. J. Gallant, D. Maier, and J. Storer. On finding minimal length superstrings. *Journal of Computer and System Sciences* **20**(1980), pp. 50–58.
6. M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness*. Freeman, New York, 1979.
7. R. Greenlaw, H. J. Hoover, and W. L. Ruzzo. A compendium of problems complete for $\mathcal{P}$. Technical Report, University of Washington, 1991.
8. M. Li. Towards a DNA sequencing theory (Learning a string). 31st FOCS, 1990.
9. C. Lund and M. Yannakakis. On the hardness of approximating minimization problems. 25th STOC, 1993.
10. C. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Prentice-Hall, 1982.
11. C. Papadimitriou and M. Yannakakis. Optimization, approximation, and complexity classes. 20th STOC, pp. 229–234, 1988.
12. H. Peltola, H. Soderlund, J. Tarhio, and E. Ukkonen. Algorithms for some string matching problems arising in molecular genetics. IFIP, pp. 53–64, 1983.
13. J. Storer. *Data Compression: Methods and theory*. Computer Science Press, 1988.
14. J. Tarhio and E. Ukkonen. A Greedy approximation algorithm for constructing shortest common superstrings. *Theoretical Computer Science* **57**(1988), pp. 131–145.
15. S-H. Teng and F. Yao. Approximating shortest superstrings. 34th FOCS, 1993.
16. J-S. Turner. Approximation algorithms for the shortest common superstring problem. *Information and Computation* **83**(1989), pp. 1–20.
17. V. V. Vazirani. Parallel graph matching. In J. H. Reif, editor, *Synthesis of Parallel Algorithms*, chapter 18, pp. 783–811. Morgan Kaufmann, 1993.

This article was processed using the LaTeX macro package with LLNCS style