Approximation Algorithm for the Shortest Approximate Common Superstring Problem

A.S. Rebaï, and M. Elloumi

Abstract—The Shortest Approximate Common Superstring (SACS) problem is: Given a set of strings $f=\{w_1, w_2, \ldots, w_n\}$, where no $w_i$ is an approximate substring of $w_j$, $i \neq j$, find a shortest string $S_f$, such that, every string of $f$ is an approximate substring of $S_f$. When the number of the strings $n>2$, the SACS problem becomes NP-complete. In this paper, we present a greedy approximation SACS algorithm. Our algorithm is a 1/2-approximation for the SACS problem. It is of complexity $O(n^2*(l^2+\log(n)))$ in computing time, where $n$ is the number of the strings and $l$ is the length of a string. Our SACS algorithm is based on computation of the Length of the Approximate Longest Overlap (LALO).

Keywords—Shortest approximate common superstring, approximation algorithms, strings overlaps, complexities.

I. INTRODUCTION

The Shortest Approximate Common Superstring (SACS) problem is: Given a set of strings $f=\{w_1, w_2, \ldots, w_n\}$, where no $w_i$ is an approximate substring of $w_j$, $i \neq j$, find a shortest string $S_f$, such that, every string of $f$ is an approximate substring of $S_f$. When the number of the strings $n>2$, the SACS problem becomes NP-complete [1, 2, 3, 4, 5].

Motivation: DNA Sequence Assembly [6, 7, 8, 9, 10, 11, 12]: The SACS problem is actually a reduction of the DNA Sequence Assembly (DSA) one, since the strings of $f$ code fragments of, only, one strand of a DNA macromolecule.

Microarray Production [13]: During microarray production, several thousands of oligonucleotides (short DNA sequences) are synthesized in parallel, one nucleotide at a time. We are interested in finding the shortest possible nucleotide deposition sequence to synthesize all oligos in order to reduce production time and increase oligo quality. Thus we study the shortest common superstring problem of several thousand short strings over a four-letter alphabet.

Previous works: Among the approximation algorithms that deal with the SACS problem, we mention Peltola et al.'s one [6], Ukkonen's one [14], Kececioglu's one [15], that is an adaptation of Tarhio and Ukkonen's greedy one [16], Teng and Yao's one [17]. Kececioglu conjectures that his adaptation is a $(1-f(\varepsilon))/2$-approximation for the SACS problem, where $\varepsilon$ is the error rate and $f(\varepsilon)\to 0$ as $\varepsilon\to 0$. Peltola et al. give no guarantee on the performance of their algorithm.

Our result: In this paper, we present a greedy approximation SACS algorithm. Our greedy algorithm is comparable to the greedy one, described in [18, 19, 16], to construct the longest hamiltonian path [20]. Our greedy algorithm is a 1/2-approximation for the SACS problem. Our greedy algorithm is of complexity $O(n^2*(l^2+\log(n)))$ in computing time, where $n$ is the number of the strings and $l$ is the length of a string. Our SACS algorithm is based on the computation of the Length of the Approximate Longest Overlap (LALO).

In the first section of this paper, we present some definitions and notations.

In the second section, we present our algorithm of computation of the LALO.

In the third section, we present our greedy approximation SACS algorithm.

Finally, in the last section, we present our conclusion and pose open problems.

II. DEFINITIONS AND NOTATIONS

Let $A$ be a finite alphabet, a string is an element of $A^*$, it is a concatenation of elements of $A$. The length of a string $w$, denoted by $|w|$, is the number of the characters that constitute this string. The null length string will be denoted by $\epsilon$. The $i^{th}$ character of $w$ will be denoted by $w_i$. A portion of a string $w$ that begins at the position $i$ and ends at the position $j$, $1 \leq i \leq j \leq n$, is called substring of $w$ and will be denoted by $w_{i,j}$. When $i=1$ and $1 \leq j \leq n$ then the substring $w_{1,j}$ is called prefix of $w$ and when $1 \leq i \leq n$ and $j=n$ then the substring $w_{i,n}$ is called suffix of $w$. The set of the suffixes of $w$ will be denoted by $S(w)$ and the set of the prefixes of $w$ will be denoted by $P(w)$.

The Levenshtein distance, denoted by $d_{\sigma,\gamma,\delta}$ is the minimum cost of a sequence of edit operations, i.e., change of cost $\sigma$, insert of cost $\gamma$ and delete of cost $\delta$, that change one string $w$ into another string $w'$:

$$d_{\sigma,\gamma,\delta}(w,w')=\min\{d^\sigma m^\gamma n^\delta l^\delta\}$$

(1)

with $m_i$, $n_i$ and $l_i$ are, respectively, the numbers of changes, inserts and deletes necessary to change $w$ into $w'$.

Let $S$ and $w$ be two strings, $|S|>|w|$, and $\varepsilon$ be an error rate, $\varepsilon>0$, we say that $w$ is an approximate substring of $S$, if and only if, there exists a substring $w'$ of $S$ such that:

A.S. Rebaï is with Computer Science Doctoral School, Faculty of Sciences of Tunis, El Manar 2092 Tunis, Tunisia (e-mail: sanirebaï@webmails.com).

M. Elloumi is with Computer Science Department, Faculty of Economic Sciences and Management of Tunis, Faculty of Economic Sciences and Management of Tunis, El Manar 2092 Tunis, Tunisia (e-mail: Mourad.Elloumi@fsegt.rnu.tn).
Let \( w \) and \( w' \) be two strings and \( \varepsilon \) be an error rate, \( \varepsilon > 0 \), we say that \( w \) approximately overlaps with \( w' \), if and only if, there exist \( x_1 \) and \( x_2 \), respectively, a prefix and a suffix of \( w \) and there exist \( x'_1 \) and \( x'_2 \), respectively, a prefix and a suffix of \( w' \) such that:

\[
\frac{d_{\sigma,\varepsilon}(x_2,x'_2)}{\max|x_2|,|x'_2|} \leq \varepsilon
\]

If \( x_2 \) is the longest suffix of \( w \) and \( x'_1 \) is the longest prefix of \( w' \) that resemble the most to each other, i.e.:

\[
\begin{align*}
\left\{ \begin{array}{ll}
d_{\sigma,\varepsilon}(x_2,x'_2) \\ \max(x_2,|x'_2|) \to& \min \left( \frac{d_{\sigma,\varepsilon}(x_1,x'_1)}{\max|x_1|,|x'_1|} \right) \end{array} \right. \\
\end{align*}
\]

then:

(i) If \( |x_2|x'_1| \) then the length \( |x_2| \) is the Length of the Approximate Longest Overlap (ALO), the string \( x_1x'_2 \), also denoted by \( C_d(w,w') \), is the Approximate Compact String (ACS) and is of weight \( \omega_d(w,w')=|x_2| \).

(ii) Otherwise, the length \( |x'_1| \) is the LALO, the string \( x_1x'_2 \), also denoted by \( C_d(w,w') \), is the ACS and is of weight \( \omega_d(w,w')=|x'_1| \).

Let \( f=\{w_1, w_2, \ldots, w_n\} \) be a set of strings, where no \( w_i \) is an approximate substring of \( w_j \), \( i \neq j \), we define on \( f \) an order relation, denoted by \( \Rightarrow \), satisfying the following properties:

(i) if \( w_i \Rightarrow w_j \) then \( w_i \) approximately overlaps with \( w_j \).

(ii) if \( w_i \Rightarrow w_j \) then for any \( k, k \neq j \), we cannot have \( w_i \Rightarrow w_k \).

An approximate common superstring associated with the set \( f \) and the order relation \( \Rightarrow \), defined on \( f \), \( w_1 \Rightarrow w_2 \Rightarrow \ldots \Rightarrow w_n \), for \( 1 \leq k \leq n \), is the string \( S_a=C_d(\ldots C_d(w_{i_{n-1}},w_{i_n}),w_{i_n}) \). With each approximate common superstring \( S_a=C_d(\ldots C_d(w_{i_{n-1}},w_{i_n}),w_{i_n}) \), we associate a positive weight, denoted by \( \Omega_d(S_a) \), that expresses the amount of compression of \( S_a \):

\[
\Omega_d(S_a) = \sum_{k=1}^{n-1} \omega_d(w_{i_k},w_{i_{k+1}})
\]

The weight \( \Omega_d(S_a) \) can also be expressed by the following equation:

\[
\Omega_d(S_a) = \sum_{i=1}^{n} |w_i| - |S_a|
\]

Hence, since \( \sum_{i=1}^{n} |w_i| \) is a constant for a given family \( f \), we can define, by using equation (6), the SACS to \( f \) as the one that maximizes \( \Omega_d \).

By using our definition of a SACS, we can reformulate the SACS problem as follows: Given a set of strings \( f=\{w_1, w_2, \ldots, w_n\} \), where no \( w_i \) is an approximate substring of \( w_j \), \( i \neq j \), find an order relation \( \Rightarrow \) defined on \( f \), \( w_1 \Rightarrow w_2 \Rightarrow \ldots \Rightarrow w_n \), \( w_i \Rightarrow w_j \) for \( 1 \leq k \leq n \), such that the string \( S_a=C_d(\ldots C_d(w_{i_{n-1}},w_{i_n}),w_{i_n}) \ldots w_{i_{n+1}},w_{i_n}) \) maximizes \( \Omega_d \).

An algorithm \( A \) is an \( \alpha \)-approximation for a minimization problem \( P \) with respect to a function \( f \), if and only if, it gives in a polynomial time a solution \( S \) for \( P \) such that \( f(S) \leq \alpha f(S_{\text{min}}) \), where \( S_{\text{min}} \) is a solution to \( P \) that minimizes \( f \) and \( \alpha < 1 \). An algorithm \( A \) is an \( \alpha \)-approximation for a maximization problem \( P \) with respect to a function \( f \), if and only if, it gives in a polynomial time a solution \( S \) for \( P \) such that \( f(S) \geq \alpha f(S_{\text{max}}) \), where \( S_{\text{max}} \) is a solution to \( P \) that maximizes \( f \) and \( \alpha > 1 \).

III. COMPUTATION OF THE LALO

The computation of the LALO between two strings boils down to find the longest suffix of the first string and the longest prefix of the second one that resemble the most to each other. Our algorithm of computation of the LALO, Algorithm 1, is a dynamic programming one [21, 22]. By using this algorithm, we proceed within three steps:

(i) During the first step, we compute the Levenshtein distances between the different suffixes of the first string and the different prefixes of the second one: the computation of the distances between the longer prefixes and the shorter suffixes is done by using the results of the computations of the distances between the shorter prefixes and the longer suffixes. We reiterate this process, until the distances between the different suffixes and prefixes are computed.

(ii) During the second step, we locate the pairings that generate the longest suffix \( x \) of the first string and the longest prefix \( x' \) of the second one such that \( \frac{d_{\sigma,\varepsilon}(x,x')}{\max(|x|,|x'|)} \) is minimum: during each iteration, we consider a prefix \( x' \) of the second string. For this prefix, we determine the pairings that generate the longest suffix \( x \) of the first string such that \( d_{\sigma,\varepsilon}(x,x') \) is minimum. The pairings between the longer suffixes of the first string and the shorter prefixes of \( x' \) are located according to the pairings between the shorter suffixes of the first string and the longer prefixes of \( x' \). We reiterate this process, until we locate all the pairings that generate the longest suffix \( x \) of the first string such that \( d_{\sigma,\varepsilon}(x,x') \) is minimum. If suffix \( x \) and prefix \( x' \) are such that \( \frac{d_{\sigma,\varepsilon}(x,x')}{\max(|x|,|x'|)} = \varepsilon \), i.e., \( x \) is the longest suffix and \( x' \) is the longest prefix that resembles the most to each other.

(iii) Finally, during the third step, we consider suffix \( x \) and prefix \( x' \), located during the second step. If \( \frac{d_{\sigma,\varepsilon}(x,x')}{\max(|x|,|x'|)} = \varepsilon \), then:

\[
\sum_{i=1}^{n} |w_i| = \text{constant}
\]

and:

\[
\sum_{i=1}^{n} |w_i| - |S_a| = \text{constant}
\]
resemble the most to each other, then from \(|x|\) and \(|x'|\) we compute the LALO and construct the ACS.

Algorithm 1 is comparable to Wagner and Fischer’s algorithm [23] to compute the Levenshtein distance between two strings, to Sellars’s one [24] to have a string-matching with \(k\)-differences, to Peliolela et al.’s one [6] to compute an overlap between two strings and to Elloumi’s one [25] to have an approximate string-matching.

We define the cost \(c_{ij}\) of the change operation of the \(i^{th}\) character of a string \(w\) by the \(j^{th}\) character of a string \(w'\) as follows:

\[
\sigma_{i,j} = \begin{cases} 
0 & \text{if } w^i = w'^j \\
\sigma & \text{otherwise, } \sigma \neq 0
\end{cases}
\quad(7)
\]

**Algorithm 1.**

(i) (a) Construct a matrix \(M\) of size \((|w'|+1)\times(|w|+1)\); filling

\[
(i.b) \text{for } i=1 \to |w| \text{ do } M[i,0]:=i \cdot \delta \text{ endfor;}
\]

\[
(i.j) \text{for } j=1 \to |w| \text{ do } M[0,j]:=0 \text{ endfor;}
\]

\[
(ii.a) \text{while } i \neq 1 \text{ and } j \neq 0 \text{ do}
\]

\[
\text{if } M[i^*,j]:=M[i^*,j-1]+\delta, M[i^*,j-1]+\gamma, M[i^*,j-1]+\sigma_j \text{ endfor;}
\]

\[
(ii.b) \text{while } j \neq 1 \text{ and } i \neq 0 \text{ do}
\]

\[
\text{if } M[i,j^*]:=M[i-1,j^*]+\gamma, M[i-1,j^*]+\sigma_i \text{ then...}
\]

\[
(iii) \text{if } \rho \leq \varepsilon \text{ then}
\]

\[
\text{if } (|w|-|w'|+1) < i < j \text{ then...}
\]

\[
\text{endfor.}
\]

(iii) (a) Construct matrix \(M\) of size \((|w'|+1)\times(|w|+1)\); filling

\[
(ii.a) \text{while } i \neq 1 \text{ and } j \neq 0 \text{ do}
\]

\[
\text{if } M[i,j]:=\min(0, |w|+1, \max(|w|+1, i)), M[i,j-1]+\sigma_j \text{ endfor;}
\]

\[
(ii.b) \text{while } j \neq 1 \text{ and } i \neq 0 \text{ do}
\]

\[
\text{if } M[i,j]:=\min(0, |w|+1, \max(|w|+1, i)), M[i-1,j]+\sigma_j \text{ endfor;}
\]

\[
(iii) \text{if } \rho \leq \varepsilon \text{ then}
\]

\[
\text{if } (|w|-|w'|+1) < i < j \text{ then...}
\]

\[
\text{endfor.}
\]

**Proposition 1.** Let \(w\) and \(w'\) be two strings and \(\varepsilon\) be an error rate, \(\varepsilon \leq 0\), Algorithm 1 tests if \(w\) approximately overlaps with \(w'\) and, if so, computes the LALO and constructs the ACS.

**Proof.** During step (i), we compute the Levenshtein distances between the different suffixes of \(w\) and the different prefixes of \(w'\); we construct a matrix \(M\) of size \((|w|+1)^2\) and fill it in the same way as Wagner and Fischer’s dynamic programming algorithm [23] but we set \(M[0,j]=0,\) for any \(j, 0 \leq |w'|\).

During step (ii), we locate the longest suffix \(x\) of \(w\) and the longest prefix \(y\) of \(w'\) such that \(\rho_{\sigma,\delta}^{(x,y)}=\min\) for each iteration of the “for” loop, we consider a prefix \(w'[j], 0 \leq |w'|\), of \(w'\) by starting from cell \(M[1,|w'|]\). For this prefix, we determine the longest suffix \(x\) of \(w\) such that \(d_{\sigma,\delta}^{(x,y)}\) is minimum. This can be done thanks to a traceback in the matrix \(M\), by using the “while” loop of the ACS algorithm [23].

During step (iii), if \(w[j]\) and \(w'[j]\) are such that \(d_{\sigma,\delta}^{(w[j],w'[j])}\) is the ACS of weight \(w'\) is the ACS of weight \(\sigma_{w,\sigma,\delta}(w,w')\) then we set \(\rho_{\sigma,\delta}^{(w[j],w'[j])}\) and locate prefix \(w'_{i1}\) and suffix \(w_{i1}\) by setting \(i_{1}=i\) and \(j_{1}=j\), to consider them during step (iii). Finally, during step (iii), we check-up if \(w\) approximately overlaps with \(w'\) and, if so, we compute the LALO and construct the ACS : if \(\rho \leq \varepsilon\) then, if \(|w'|\leq |w|\), \(|w'_{i1}|\) is the LALO and the ACS of weight \(\sigma_{w,\sigma,\delta}(w,w')\) otherwise \(|w|_{j1}\) is the LALO and \(w_{i1}\) is the ACS of weight \(\sigma_{w,\sigma,\delta}(w,w')\).

**Proposition 2.** Algorithm 1 is of complexity \(O(\varepsilon^2)\) in computing time and in memory space, where \(\varepsilon\) is the length of a string.

**Proof.** During step (i), we fill line-wise matrix \(M\) of size \((|w|+1)^2\). So, time complexity of step (i) is \(O(\varepsilon^2)\).

During step (ii), for each prefix of \(w\), we do a traceback in matrix \(M\). This traceback is done in a time of the order of \(O(|w|)\). In all, we have \(|w'|\) prefixes in \(w\), so step (ii) is achieved in a time of the order of \(O(|w'||w|)\), i.e., of the order of \(O(\varepsilon^2)\).

Hence, Algorithm 1 is of complexity \(O(\varepsilon^2)\) in computing time.

Finally, matrix \(M\) is of size \((|w|+1)^2\) then Algorithm 1 is of complexity \(O(\varepsilon^2)\) in memory space.

**Example.** Let us take \(w=ecaaabeabdc\) and \(w'=fabdcaebda\) and set \(\varepsilon=0.50, \sigma=2\) and \(\delta=\gamma=1\). The longest suffix of \(w\) and the longest prefix of \(w'\) that resemble the most to each other are,
respectively, \( x=abdc \) and \( x=fadbec \). We have

\[
\begin{align*}
\frac{d_{2,1}(x,x')}{\max(|x|,|x'|)} &= 0.33 = \min_{(x_i,x'_j) \in S} \left( \frac{d_{2,1}(x_i,x'_j)}{\max(|x_i|,|x'_j|)} \right), \\
\frac{d_{2,1}(x,x')}{\max(|x|,|x'|)} &< 0.50
\end{align*}
\]

Fig. 1 Computation the LALO

![Fig. 1 Computation the LALO](image)

Let \( \epsilon \geq 0 \), by using Algorithm 1, the computation of the LALO between two strings \( w \) and \( w' \) is done by using a matrix \( M \) of size \((|w'|+1)(|w|+1)\), i.e., of size \((l+1)^2\). The same matrix is used to compute all the LALOs between all the strings of \( f \). Hence, the computation of the LALOs between all the strings of \( f \) is done by using a memory space of the order of \( O(l^2) \).

IV. CONSTRUCTION OF A SACS

Let \( f=\{w_1, w_2, \ldots, w_n\} \) be a set of strings, where no \( w_i \) is an approximate substring of \( w_j, i \neq j \), and \( S_a \) be a SACS to \( f \). Our SACS algorithms are based on following observation: the greater \( S_a \)'s weight is the shorter \( S_a \)'s length is.

Our Approximation Algorithm

Our approximation SACS algorithm is a greedy one, it operates as follows:

(i) First, we compute all the weights \( \omega_a(w_i,w_j), 1 \leq i, j \leq n \), eliminate from \( f \) all the strings that are approximate prefix/suffix of others, eliminate the weights related to these strings from the set of the computed weights, and sort this set of weights.

(ii) Then, during each iteration, we select from \( f \) two strings \( w_i \) and \( w_j \) such that \( \omega_a(w_i,w_j) \) is maximum, remove from \( f \) the strings \( w_i \) and \( w_j \), and add to \( f \) the ACS \( C_a(w_i,w_j) \). We repeat this process until \( f \) contains only one string. This string is considered to be a solution to the SACS problem.

**Proposition 4.** Our greedy SACS algorithm is a 1/2-approximation for the SACS problem.

**Proof.** Let \( S_a \) be the approximate common superstring constructed thanks to our greedy SACS algorithm and \( S_{\text{max}a} \) be the SACS. To show that:

\[
\frac{1}{2} \cdot \Omega_a (S_{\text{max}a}) \leq \Omega_a (S_a)
\]

\[\text{(8)}\]

(i) First, we show that for every constructed ACS \( C_a(w_i,w_j) \), we have:

\[
\sum_{a \in \Sigma^* \text{ such that } C_a(w_i,w_j)} \omega_a(w_i,w_j) \leq 2^* \omega_a(w_i,w_j).
\]

\[\text{(9)}\]

(ii) Then, we show that:

\[
\sum_{a \in \Sigma^*} \left[ \sum_{a \in \Sigma^*} \omega_a(w_i,w_j) \right] C((w_i,w_j) \in S) \leq \sum_{a \in \Sigma^*} 2^* \omega_a(w_i,w_j).
\]

\[\text{(10)}\]
So, during each iteration of our algorithm, we select from \( f \) two strings \( w_i \) and \( w_j \) such that \( \omega_d(w_i, w_j) \) is maximum, i.e., we construct an ACS \( C_d(w_i, w_j) \) of weight \( \omega_d(w_i, w_j) \) that is maximum. When constructing the ACS \( C_d(w_i, w_j) \), we eliminate from the set of ACSs to be considered for during the future iterations, at most, two ACSs that are portions of the SACS:

(i) The ACS, let us call it \( C_d(w_i, w_j) \), that has \( w_i \) as an approximate prefix,

(ii) Or/and the ACS, let us call it \( C_d(w_i, w_j) \), that has \( w_j \) as an approximate suffix.

Case 1 : \( |E(C_a(w_i, w_j))|=0 \).

In this case, we have then \( E(C_a(w_i, w_j))=\varnothing \). An ACS \( C_d(w_i, w_j) \in \varnothing \) implies that \( C_d(w_i, w_j) \) is \( v \). Then we have :

\[
\omega_d(w_i, w_j)=|v|=0
\]

Since \( \omega_d(w_i, w_j) \) is positive, we have then :

\[
\sum_{C_a(w_x, w_y) \in E(C_a(w_i, w_j))} \omega_d(w_x, w_y) - \omega_d(w_i, w_j) - 0
\]

\[
< 2 * \omega_d(w_i, w_j)
\]

\[
(11)
\]

Case 2 : \( |E(C_a(w_i, w_j))|=1 \).

In this case, we have then \( E(C_a(w_i, w_j))=\{C_d(w_i, w_j)\} \) or \( E(C_a(w_i, w_j))=\{C_d(w_j, w_i)\} \). Let us consider the subcase where \( E(C_a(w_i, w_j))=\{C_d(w_i, w_j)\} \). Since \( \omega_d(w_i, w_j) \) is maximum, we have then :

\[
\omega_d(w_i, w_j)\leq \omega_d(w_i, w_j).
\]

Then, we have :

\[
\sum_{C_a(w_x, w_y) \in E(C_a(w_i, w_j))} \omega_d(w_x, w_y) - \omega_d(w_i, w_j)
\]

\[
< 2 * \omega_d(w_i, w_j)
\]

\[
(13)
\]

\[
\omega_d(w_i, w_j)\leq 2 * \omega_d(w_i, w_j)
\]

\[i.e.:
\]

\[
\sum_{C_a(w_x, w_y) \in E(C_a(w_i, w_j))} \omega_d(w_x, w_y) < 2 * \omega_d(w_i, w_j)
\]

\[
(14)
\]

We process in the same way the subcase where \( E(C_a(w_i, w_j))=\{C_d(w_j, w_i)\} \).

Case 3 : \( |E(C_a(w_i, w_j))|=2 \).

In this case, we have then \( E(C_a(w_i, w_j))=\{C_d(w_i, w_j), C_d(w_j, w_i)\} \). Since \( \omega_d(w_i, w_j) \) is maximum, we have then :

\[
\omega_d(w_i, w_j)\leq \omega_d(w_i, w_j).
\]

\[
(15)
\]

\[
\omega_d(w_i, w_j)\leq \omega_d(w_i, w_j).
\]

Then, we have : 

\[
\omega_d(w_i, w_j)\leq \omega_d(w_i, w_j)\leq \omega_d(w_i, w_j)
\]

\[
(16)
\]

Now, if we consider all the ACSs \( C_d(w_i, w_j) \), \( 1 \leq i, j \leq n \), constructed thanks to our algorithm, we have then :

\[
\sum_{a} \sum_{i \neq j} \omega_d(w_i, w_j)
\]

\[i.e.:
\]

\[
\Omega_d(S_{max})\leq 2 * \Omega_d(S_a).
\]

\[Hence:
\]

\[
\frac{1}{2} * \Omega_d(S_{max}) \leq \Omega_d(S_a).
\]

\[Proposition 5. Our greedy SACS algorithm is of complexity \( O(n^2*(l^2+log(n))) \) in computing time.\]

**Proof.** By using our greedy SACS algorithm, we operate as follows:

(i) First, we compute all the weights \( \omega_d(w_i, w_j) \), \( 1 \leq i, j \leq n \). That is, we compute all the LALOs. According to Proposition 3, this phase is of complexity \( O(n^2*l^2) \) in computing time. Then, we sort the computed weights. It is well known that the sorting of \( k \) integers can be done in a time of the order of \( O(k*log(k)) \) [26]. We have of the order of \( O(n^2) \) weights to be sorted, so the sorting phase of our algorithm can be achieved in a time of the order of \( O(n^2*log(n)) \). Hence, the first step of our greedy SACS algorithm is of complexity \( O(n^2*(l^2+log(n))) \) in computing time.

(ii) Then, during each iteration, we select from \( f \) two strings \( w_i \) and \( w_j \) such that \( \omega_d(w_i, w_j) \) is maximum, remove from \( f \) the strings \( w_i \) and \( w_j \) and add to \( f \) the ACS \( C_d(w_i, w_j) \). We repeat this process until \( f \) contains only one string. Then, each iteration is achieved in a constant time. We have of the order of \( O(n) \) iterations, then the second step of our greedy SACS algorithm is of complexity \( O(n^2) \) in computing time.

Hence, our greedy SACS algorithm is of complexity \( O(n^2*(l^2+log(n))) \) in computing time.

V. CONCLUSION AND OPEN PROBLEMS

We have presented a SACS greedy approximation algorithm. Our algorithm is comparable to the greedy one,
described in [18, 19, 16], to construct the longest hamiltonian path [20]. Our greedy algorithm is a 1/2-approximation for the SACS problem. Our greedy SACS algorithm is of complexity $O(n^2 (1/2 + \log(n)))$ in computing time, where $n$ is the number of the strings and $l$ is the length of a string. Our SACS algorithm is based on computation of the Length of the Approximate Longest Overlap (LALO). We have presented an algorithm of computation of the LALO. This algorithm is of complexity $O(l^2)$ in computing time and in memory space.

Finally, to conclude we pose the following open problems:

Can the factor $\alpha=1/2$ be improved on in the worst case? Can the complexity of the proposed LALO algorithm be reduced?

REFERENCES