International Journal of Engineering, Mathematical and Physical Sciences
ISSN: 2517-9934
Vol:1, No:12, 2007

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 $f=\{w_1, w_2, \dots , w_n\}$, where no w_i is an approximate substring of w_j, $i \neq j$, find a shortest string S_a, such that, every string of f is an approximate substring of S_a. 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(n2*(l2+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 $f=\{w_1, w_2, \dots , w_n\}$ where no w_i is an *approximate substring* of w_j, $i\neq j$, find a shortest string S_a, such that, every string of f is an *approximate substring* of S_a. 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 $(1\text{-}f(\varepsilon))/2$-approximation for the SACS problem, where ε is the error rate and $f(\varepsilon)\to 0$ as $\varepsilon\to 0$. Peltola

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

adaptation is a $(1\text{-}f(\varepsilon))/2$-approximation for the SACS problem, where ε 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 ν. 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 σ, insert of cost γ and delete of cost δ, that change one string w into another string w' :

$$d_{\sigma,\gamma,\delta}(w,w')=min_i\{\sigma*m_i+\gamma*n_i+\delta*l_i\} \tag{1}$$

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 ε 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 :

$$\frac{d_{\sigma,\gamma,\delta}(w,w')}{|w|} \le \varepsilon \tag{2}$$

Let w and w' be two strings and ε 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 :

$$\frac{d_{\sigma,\gamma,\delta}(x_2,x'_1)}{max(|x_2|,|x'_1|)} \le \varepsilon \tag{3}$$

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. :

$$\begin{cases} \frac{d_{\sigma,\gamma,\delta}(x_2,x'_1)}{max(|x_2|,|x'_1|)} = \min_{(x_i,x'_j)\in S(w)\times P(w')} \{\frac{d_{\sigma,\gamma,\delta}(x_i,x'_j)}{max(|x_i|,|x'_j|)}\} \\ \frac{d_{\sigma,\gamma,\delta}(x_2,x'_1)}{max(|x_2|,|x'_1|)} \le \varepsilon \end{cases} \tag{4}$$

then :

(*i*) If $|x_2|>|x'_1|$ then the length $|x_2|$ is the *Length of the Approximate Longest Overlap* (LALO), the string $x_1x'_1x'_2$, also denoted by $C_a(w,w')$, is the *Approximate Compact String* (ACS) and is of weight $\omega_a(w,w')=|x_2|$.

(*ii*) Otherwise, the length $|x'_1|$ is the LALO, the string $x_1x_2x'_2$, also denoted by $C_a(w,w')$, is the ACS and is of weight $\omega_a(w,w')=|x'_1|$.

Let $f=\{w_1, w_2, \dots , w_n\}$ 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 $\Rightarrow_a$, satisfying the following properties :

(*i*) if $w_i \Rightarrow_a w_j$ then w_i approximately overlaps with w_j.

(*ii*) if $w_i \Rightarrow_a w_j$ then for any k, $k\neq j$, we cannot have w_i- $_a w_k$

An *approximate common superstring* associated with the set f and the order relation $\rightarrow_a$ defined on f, $w_{i_1}\rightarrow_a w_{i_2}\rightarrow_a \dots \rightarrow_a w_{i_n}$, $w_{i_k}\in f$ for $1\le k\le n$, is the string $S_a=C_a(C_a(\dots C_a(C_a(w_{i_1},w_{i_2}),w_{i_3}) \dots ,w_{i_{n-1}}),w_{i_n})$. With each approximate common superstring $S_a=C_a(C_a(\dots C_a(C_a(w_{i_1},w_{i_2}),w_{i_3}) \dots ,w_{i_{n-1}}),w_{i_n})$, we associate a positive *weight*, denoted by $\Omega_a(S_a)$, that expresses the amount of compression of S_a :

$$\Omega_a(S_a) = \sum_{k=1}^{n-1} \omega_a(w_{i_k}, w_{i_{k+1}}) \tag{5}$$

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

$$\Omega_a(S_a) = (\sum_{i=1}^{n} |w_i|) - |S_a| \tag{6}$$

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 Ω_a.

By using our definition of a SACS, we can reformulate the SACS problem as follows : Given a set of strings $f=\{w_1, w_2, \dots , w_n\}$, where no w_i is an approximate substring of w_j, $i\neq j$, find an order relation $\rightarrow_a$ defined on f, $w_{i_1}\rightarrow_a w_{i_2}\rightarrow_a \dots \rightarrow_a w_{i_n}$, $w_{i_k}\in f$ for $1\le k\le n$, such that the string $S_a=C_a(C_a(\dots C_a(C_a(w_{i_1},w_{i_2}),w_{i_3}) \dots ,w_{i_{n-1}}),w_{i_n})$ maximizes Ω_a.

An algorithm A is an *α-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 $f(S)\le\alpha * f(S_{min})$, where S_{min} is a solution to P that minimizes f and $\alpha>1$. An algorithm A is an *α-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 $f(S)\ge\alpha * f(S_{max})$, where S_{max} is a solution to P that maximizes f and $0<\alpha<1$.

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 $\frac{d_{\sigma,\gamma,\delta}(x,x')}{max(|x|,|x'|)}$ 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 $d_{\sigma,\gamma,\delta}(x,x')$ 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 $d_{\sigma,\gamma,\delta}(x,x')$ is minimum. If suffix x and prefix x' are such that $\frac{d_{\sigma,\gamma,\delta}(x,x')}{max(|x|,|x'|)}$ is also minimum then they will be considered during the third step.

(*iii*) Finally, during the third step, we consider suffix x and prefix x', located during the second step. If $\frac{d_{\sigma,\gamma,\delta}(x,x')}{max(|x|,|x'|)} \le \varepsilon$, i.e., x is the longest suffix and x' is the longest prefix that 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 Sellers's one [24] to have a *string-matching with k-differences*, to Peltola *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 $\sigma_{i,j}$ of the change operation of the i^{th} character of a string w' by the j^{th} character of a string w as follows :

$$\sigma_{i,j} = \begin{cases} 0 & \text{if } w'^i = w^j \\ \sigma & \text{otherwise, } \sigma \neq 0 \end{cases} \tag{7}$$

Algorithm 1.

```
(i) (i.a) Construct a matrix $M$ of size $(|w'|+1)*(|w|+1)$; { filling }
    (i.b) for $i$:=1 to $|w'|$ do $M[i,0]:=i*\delta$ endfor;
   for $j$:=0 to $|w|$ do $M[0,j]:=0$ endfor;
   for $i$:=1 to $|w'|$ do
     for $j$:=1 to $|w|$ do
      $M[i,j]:=min\{M[i-1,j]+\delta, M[i,j-1]+\gamma, M[i-1,j-1]+\sigma_{i,j}\}$
     endfor
   endfor;
(ii) $\rho:=+\infty$; $i_\rho:=0$; $j_\rho:=0$;                { traceback }
    for $i$:=1 to $|w'|$ do
    $j:=|w|$; $i':=i$;
    (ii.a) while $i'\neq 1$ and $j\neq 0$ do
  if $M[i',j]=M[i',j-1]+\gamma$ then $j:=j-1$
   else
    if $M[i',j]=M[i'-1,j-1]+\sigma_{i',j}$ then $j:=j-1$; $i':=i'-1$
     else $i':=i'-1$
    endif
  endif
        endwhile;
    (ii.b) if $w'^1\neq w^j$ then $j:=j+1$ endif
    (ii.c) if $\rho\geq M[i,|w|]/max(|w|-j+1,i)$ then
$\rho:=M[i,|w|]/max(|w|-j+1,i)$; $i_\rho:=i$; $j_\rho:=j$
        endif
    endfor.
(iii) if $\rho\leq\varepsilon$ then                { evaluation }
      if $(|w|-j_\rho+1)<i_\rho$ then
       $i_\rho$ is the LALO;
       $ww'_{i_\rho+1,|w'|}$ is the ACS of weight $\omega_a(w,w'):=i_\rho$
      else
       $(|w|-j_\rho+1)$ is the LALO;
       $w_{1,j_\rho-1}w'$ is the ACS of weight $\omega_a(w,w'):=(|w|-j_\rho+1)$
      endif
     else
      $w$ do not approximately overlap with $w'$
     endif.
```

Proposition 1. Let w and w' be two strings and ε be an error rate, $\varepsilon>0$, 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 $(l+1)^2$ 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, $0\leq j\leq|w|$.

During step (*ii*), we locate the longest suffix x of w and the longest prefix x' of w' such that $\frac{d_{\sigma,\gamma,\delta}(x,x')}{max(|x|,|x'|)}$ is minimum : during each iteration of the "for" loop, we consider a prefix $w'_{1,i}$, $0\leq i\leq|w'|$, of w' by starting from cell $M[i,|w|]$. For this prefix, we determine the longest suffix x of w such that $d_{\sigma,\gamma,\delta}(x,w'_{1,i})$ is minimum. This can be done thanks to a traceback in the matrix M, by using the "while" loop of substep (*ii.a*) : let $M[i,j]$ be the current cell, the next cell to be visited is $M[i',j']$, where :

$M[i',j']=M[i,j-1]$, if $M[i,j]=M[i,j-1]+\gamma$ else
$M[i',j']=M[i-1,j-1]$, if $M[i,j]=M[i-1,j-1]+\sigma_{i,j}$ else
$M[i',j']=M[i-1,j]$.

Hence, at each iteration of the "while" loop, we try to go to the leftmost side of w, then, try to have a longer suffix of w. The "while" loop stops when we reach row 1. It stops, too, when we reach column 0, i.e., if the whole string w is an approximate prefix of w'. Now, let j be the column reached when we reach row 1. Suffix $w_{j,|w|}$ is the suffix of w such that $d_{\sigma,\gamma,\delta}(w_{j,|w|},w'_{1,i}) = \min_{x \in S(w)} \{d_{\sigma,\gamma,\delta}(x,w'_{1,i})\}$. During substep (*ii.c*), if prefix $w'_{1,i}$ and suffix $w_{j,|w|}$ are such that $\frac{d_{\sigma,\gamma,\delta}(w_{j,|w|},w'_{1,i})}{max(|w_{j,|w|}|,|w'_{1,i}|)} = \min_{(x,x') \in S(w)\times P(w')} \{\frac{d_{\sigma,\gamma,\delta}(x,x')}{max(|x|,|x'|)}\}$ then we set $\rho=\frac{d_{\sigma,\gamma,\delta}(w_{j,|w|},w'_{1,i})}{max(|w_{j,|w|}|,|w'_{1,i}|)}$ and locate prefix $w'_{1,i}$ and suffix $w_{j,|w|}$ by setting $i_\rho:=i$ and $j_\rho:=j$; 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 $|w_{j_\rho,|w|}|<|w'_{1,i_\rho}|$ then $|w'_{1,i_\rho}|$ is the LALO and $ww'_{i_\rho+1,|w'|}$ is the ACS of weight $\omega_a(w,w'):=|w'_{1,i_\rho}|$ otherwise $|w_{j_\rho,|w|}|$ is the LALO and $w_{1,j_\rho-1}w'$ is the ACS of weight $\omega_a(w,w'):=|w_{j_\rho,|w|}|$.

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

Proof. During step (*i*), we fill linewise matrix M of size $(l+1)^2$. So, time complexity of step (*i*) is $O(l^2)$.

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 $O(l^2)$.

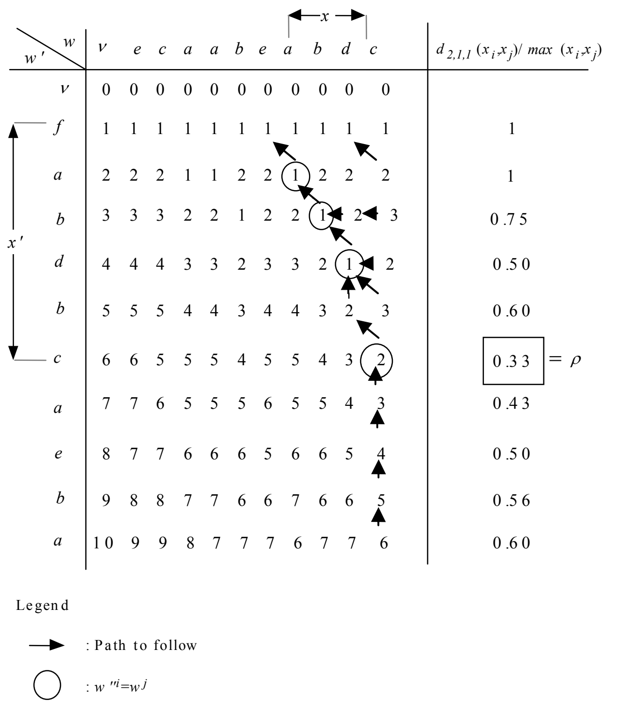
Hence, Algorithm 1 is of complexity $O(l^2)$ in computing time.

Finally, matrix M is of size $(l+1)^2$ then Algorithm 1 is of complexity $O(l^2)$ in memory space.

Example. Let us take w=*ecaabeabdc* and w'=*fabdbcaeba* and set ε=0.50, σ=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'=*fabdbc*. We have

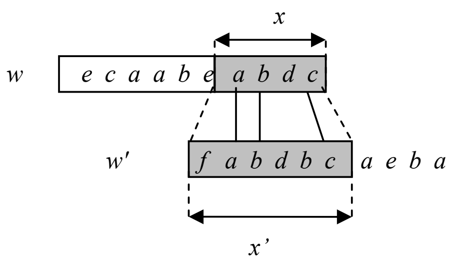
$$\begin{cases} \dfrac{d_{2,1,1}(x,x')}{max(|x|,|x'|)} = 0.33 = \underset{(x_i,x'_j)S\ (w)\times P\ (w')}{min} \left\{\dfrac{d_{2,1,1}(x_i,x'_j)}{max(|x_i|,|x'_j|)}\right\} \\ \dfrac{d_{2,1,1}(x,x')}{max(|x|,|x'|)} < 0.50 \end{cases}$$

Fig. 1 Computation the LALO



$w' \backslash w$	v	e	c	a	a	b	e	a	b	d	c	$d_{2,1,1}(x_i,x_j)/\ max\ (x_i,x_j)$
v	0	0	0	0	0	0	0	0	0	0	0	
f	1	1	1	1	1	1	1	1	1	1	1	1
a	2	2	2	1	1	2	2	1	2	2	2	1
b	3	3	3	2	2	1	2	2	1	2	3	0.75
d	4	4	4	3	3	2	3	3	2	1	2	0.50
b	5	5	5	4	4	3	4	4	3	2	3	0.60
c	6	6	5	5	5	4	5	5	4	3	2	$\boxed{0.33} = \rho$
a	7	7	6	5	5	5	6	5	5	4	3	0.43
e	8	7	7	6	6	6	5	6	6	5	4	0.50
b	9	8	8	7	7	6	6	7	6	6	5	0.56
a	10	9	9	8	7	7	7	6	7	7	6	0.60

Fig. 2 x is the longest suffix of w and x' is the longest prefix of w' that resemble the most to each other



Proposition 3. Let f be a set of strings and ε be an error rate, ε>0, by using Algorithm 1, the computation of the LALOs between all the strings of f is done in a time of the order of $O(n^2*l^2)$ and by using a memory space of the order of $O(l^2)$, where n is the number of the strings and l is the length of a string.

Proof. According to Proposition 2 :

(*i*) Algorithm 1 is of complexity $O(l^2)$ in computing time. In all, we have of the order of $O(n^2)$ couples of strings, then, we have of the order of $O(n^2)$ LALOs to be computed. Hence, the computation of the LALOs between all the strings of f is done in a time of the order of $O(n^2*l^2)$.

(*ii*) 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 $f=\{w_1, w_2, \dots, w_n\}$ 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_{max_a} be the SACS. To show that :

$$\frac{1}{2} * \Omega_a(S_{max_a}) \leq \Omega_a(S_a) \qquad (8)$$

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

$$\sum_{C_a(w_x,w_y)\in E(C_a(w_i,w_j))} \omega_a(w_x,w_y) \leq 2*\omega_a(w_i,w_j), \qquad (9)$$

where $E(C_a(w_i,w_j))$ is the set of ACSs that are portions of the SACS and that were eliminated, from the set of ACSs to be considered for during the future iterations, when constructing the ACS $C_a(w_i,w_j)$.

(*ii*) Then, we show that :

$$\sum_{C_a(w_i,w_j)\subset S_a} \left[\sum_{C_a(w_x,w_y)\in E(C_a(w_i,w_j))} \omega_a(w_x,w_y) \right] \leq \sum_{C_a(w_i,w_j)\subset S_a} 2*\omega_a(w_i,w_j). \qquad (10)$$

So, during each iteration of our algorithm, we select from f two strings w_i and w_j such that $\omega_a(w_i,w_j)$ is maximum, i.e., we construct an ACS $C_a(w_i,w_j)$ of weight $\omega_a(w_i,w_j)$ that is maximum. When constructing the ACS $C_a(w_i,w_j)$, 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 $C_a(w_i,w_r)$, that has w_i as an approximate prefix,

(ii) Or/and the ACS, let us call it $C_a(w_s,w_j)$, that has w_j as an approximate suffix.

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

In this case, we have then $E(C_a(w_i,w_j))=\emptyset$. An ACS $C_a(w_u,w_v)\in\emptyset$ implies that $C_a(w_u,w_v)=\nu$. Then we have :

$$\omega_a(w_u,w_v)=|\nu|=0$$

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

$$\sum_{C_a(w_x,w_y)\in E(C_a(w_i,w_j))} \omega_a(w_x,w_y)=\omega_a(w_u,w_v)=0 < 2*\omega_a(w_i,w_j) \tag{11}$$

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

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

$$\omega_a(w_i,w_r)\le\omega_a(w_i,w_j), \tag{12}$$

Then, we have :

$$\sum_{C_a(w_x,w_y)\in E(C_a(w_i,w_j))} \omega_a(w_x,w_y)=\omega_a(w_i,w_r) \le \omega_a(w_i,w_j)) < 2*\omega_a(w_i,w_j) \tag{13}$$

i.e. :

$$\sum_{C_a(w_x,w_y)\in E(C_a(w_i,w_j))} \omega_a(w_x,w_y) < 2*\omega_a(w_i,w_j) \tag{14}$$

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

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

In this case, we have then $E(C_a(w_i,w_j))=\{C_a(w_i,w_r), C_a(w_s,w_j)\}$. Since $\omega_a(w_i,w_j)$ is maximum, we have then :

$$\omega_a(w_i,w_r)\le\omega_a(w_i,w_j), \tag{15}$$

$$\omega_a(w_s,w_j)\le\omega_a(w_i,w_j). \tag{16}$$

Then, we have :

$$\omega_a(w_i,w_r)+\omega_a(w_s,w_j)\le 2*\omega_a(w_i,w_j), \tag{17}$$

i.e. :

$$\sum_{C_a(w_x,w_y)\in E(C_a(w_i,w_j))} \omega_a(w_x,w_y)\le 2*\omega_a(w_i,w_j) \tag{18}$$

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

$$\sum_{C_a(w_i,w_j)\subset S_a} \Big[\sum_{C_a(w_x,w_y)\in E(C_a(w_i,w_j))} \omega_a(w_x,w_y) \Big] \le \sum_{C_a(w_i,w_j)\subset S_a} 2*\omega_a(w_i,w_j), \tag{19}$$

i.e. :

$$\Omega_a(S_{max_a})\le 2*\Omega_a(S_a). \tag{20}$$

Hence :

$$\frac{1}{2}*\Omega_a(S_{max_a})\le\Omega_a(S_a). \tag{21}$$

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 $\omega_a(w_i,w_j)$, $1\le i,\ j\le n$. 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 $\omega_a(w_i,w_j)$ 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. 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 $O(n)$ 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 $O(n^2*(l^2+log(n)))$ 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). 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 α=1/2 be improved on in the worst case? Can the complexity of the proposed LALO algorithm be reduced?

References

[1] D. Maier, J. A. Storer, *A note on complexity of the superstring problem*, TR-233, Princeton University, Dept. EECS : (1977).

[2] D. Maier, *The complexity of some problems on subsequences and supersequences*, J. of ACM, Vol. 25, N°2 : (April 1978), p322-336.

[3] M. R. Garey, D. S. Johnson, *Computers and intractability*, Freeman (Ed.) : (1979).

[4] J. K. Gallant, D. Maier, J. Storer, *On finding minimal length superstrings*, J. Computer and System Sciences, Vol. 20, N°1 : (1980), p50-58.

[5] D. S. Hirschberg, *Recent results on the complexity of common-subsequence problems*, Time Warps, String Edits and Macromolecules The Theory and Practice of Sequence Comparison, Sankoff and Kruskal (Eds.), Addison-Wesley Pub. Inc. : (1983), p325-330.

[6] H. Peltola, H. Söderlund, E. Ukkonen, *SEQAID : A DNA sequence assembling program based on a mathematical model*, Nucleic Acids Research, Vol. 12, N°1 : (1984), p307-321.

[7] S. Dear, R. Staden, *A sequence assembly and editing program for efficient management of large projects*, Nucleic Acids Research, Vol. 19 : (1991), p3907-3911.

[8] X. Huang, A contig assembly program based on sensitive detection of string overlaps, Genomics, N°14 : (1992), p18-25.

[9] J. Kececioglu, E. Myers, *Combinatorial algorithms for DNA sequence assembly*, Algorithmica, Vol. 13, N°12, Springer International Eds. : (1995), p7-51.

[10] G. G. Sutton, O. White, M. D. Adams, A. R. Kerlavage, *TIGR Assembler: A new tool for assembing large shotgun sequencing projects*, Genome Science & Technology, Vol. 1, N°1, Mary Ann Liebert, Inc. : (1995), p9-19.

[11] M. Elloumi, *DNA Sequence Assembly Algorithms Based on Clustering Approaches*, The 2000 International Conference on Mathematics and Engineering Techniques in Medicine and Biological Sciences, METMBS'2000 (Las Vegas, Nevada, USA) : (June 2000).

[12] J. Cohen, *Bioinformatics-An Introduction For Computer Scientists*, ACM Computing Surveys, Vol 36, No 2, June 2004, p122-158.

[13] S. Rahmann, The Shortest Common Supersequence Problem In a Microarray Production Setting, Bioinformatics, Vol 19, (2003), p ii156-ii161.

[14] E. Ukkonen, A Linear time algorithm for finding approximate shortest common superstrings, Algorithmica, N°5: (1990), p313-323.

[15] J. Kececioglu, *Exact and approximation algorithms for DNA sequence reconstruction*, Ph.D. dissertation, Technical Report, N°91-26, Dept. of Computer Science, The University of Arisona, Tucson, AZ 85721 : (1991).

[16] J. Tarhio, E. Ukkonen, *A greedy approximation algorithm for constructing shortest common superstrings*, Theo. Comput. Sci., N°57: (1988), p131-145.

[17] S. Teng, F. Yao, *Approximation shortest superstrings*, 34th IEEE Symposium on Foundation of Computer Science : (1993).

[18] T. A. Jenkyns, *The greedy travelling salesman's problem*, Networks N°9 : (1979), p363-373.

[19] J. K. Gallant, *The complexity of the overlap method for sequencing biopolymers*, J. Theor. Biol., N°101 : (1982), p1-17.

[20] J. Van Leeuwen, *Graph algorithms*, Handbook of Theoretical Computer Science, Vol. A : Algorithms and Complexity, J. Van Leeuwen (Ed.), Elsevier Science Pub. B. V. : (1990), p527-631.

[21] R. E. Bellman, *Dynamic Programming*, Princeton University Press, New Jersey : (1957)

[22] R. E. Bellman, S. E. Dreyfus, *Applied Dynamic Programming*, Princeton University Press, New Jersey : (1962)

[23] R. A. Wagner, M. J. Fischer, *The string-to-string correction problem*, J. of ACM, Vol 21 No 1 : (1974), p168-173.

[24] P. H. Sellers, The theory and computation of Evolutionary distances : Pattern recognition, J. Algorithms, No 1 : (1980) p359-373.

[25] M. Elloumi, *An algorithm for the approximate string-matching problem*, Atlantic Symposium on Computational Biology, Genome Information Systems & Technology, CBGI'2001, (Durham, North Carolina, U.S.A.) : (March 2001).

[26] J. S. Vitter, Ph. Flajolet, *Average-case analysis of algorithms and data structures*, Handbook of Theoretical Computer Science, Vol. A : Algorithms and Complexity, J. Van Leeuwen (Ed.), Elsevier Science Pub. B. V. : (1990), p431-524.