An Experimental comparison of Two Approximation Algorithms for the Shortest Common Superstring Problem

Heidi J. Romero Carlos A. Brizuela Andrei Tchernykh
Computer Science Department
CICESE Research Center
Km 107 Carr. Tijuana-Ensenada, Ensenada, B.C., México
{hjromero,cbrizuel,chernykh}@cicese.mx, +52–646–1750500

Abstract

The paper deals with an experimental comparison of a 4-approximation algorithm with a 3-approximation algorithm for the Shortest Common Superstring (SCS) problem. It has two main objectives, one is to show that even though the quotient between the two approximations is 4/3, in the worst case, the average results quotient is approximately 1, independently of the instances size. The second objective is to experimentally show that these algorithms produce high quality solutions, which are significantly lower than their guaranteed worst case bound. Most of the extensive computational experiments show that the algorithms produce an average superstring of length at most 1.4% over the optimum.

1. Introduction

Given a set of strings $P = \{s_1, s_2, ..., s_n\}$ the goal is to find the shortest string S^* such that each $s_i \in P$ is a substring of S^*. This string is known as the shortest common superstring (SCS) of P. Finding S^* arises in a variety of applications, including DNA assembling [7, 20] and data compression [14, 9].

The SCS problem is NP-hard [9], furthermore it is MAX-SNP-hard [4]. Arora [3] showed that problems in MAX-SNP-hard do not admit polynomial time approximation schemes unless $P = NP$, which implies that obtaining an algorithm with approximation factor of $1 + \epsilon$ for any $\epsilon > 0$ is very unlikely.

Many approximation algorithms for the SCS have been proposed. Tarhio and Ukkonen [17], and Turner [19] established performance guarantees for the GREEDY algorithm with respect to the overlap measure. However, this result does not imply a performance guarantee over the optimal length. Blum et al. [4], who were the first to achieve a constant factor for this problem, proved a factor 4 over the optimal length for the same algorithm. They also proposed other variants known as MGREEDY and TGREEDY with approximation factors of 4 and 3, respectively. After these results the approximation factors were reduced to 2.89 by Teng and Yao [18]. Czumaj et al. [6] gave a factor of 2.83, Armen and Stein achieved a factor of 2.6 [2], Breslauer et al. gave factors of 2.67 and 2.596 [5]. Up to date the best guarantee is of 2.5 and was achieved by Sweedyk [15].

Although a great amount of work was done in trying to find better and better bounds, less attention was paid to the performance analysis of these algorithms over a set of instances of the problem. The work proposed here goes in this direction. In this paper, we present an implementation of factor 3 and 4 approximation algorithms for the SCS problem. We also propose variants to improve them. In spite of their poor worst case approximation guarantees the experiments show that the algorithms generate extremely high quality solutions.

The remainder of the paper is organized as follows. Section 2 presents some preliminary definitions. Section 3 describes the implemented algorithms. Section 4 presents the experimental setup and results. Finally, section 5 states the conclusions and points out some future research.

2. Preliminaries

An instance of the SCS problem is represented by a set of strings $P = \{s_1, s_2, ..., s_i, ..., s_n\} \subseteq \Sigma^*$ over a finite alphabet Σ, where Σ^* is the set of all strings constructed from Σ. Without loss of generality let us assume that P is a free subset, i.e. no string s_i is a substring of s_j for all $i \neq j \in \{1, 2, ..., n\}$. A feasible solution for this problem, a superstring of P, is a string S, which contains each string $s_i \in P$ as a substring. The objective is to find a superstring S^* with the shortest length over all superstrings that can be generated. An example of this problem is given as

follows. Given a set $P = \{$ *acacg, ataga, cacgt, gtaat* $\}$, a superstring for P is $S =$*acacgtaataga* of length 12.

For any string s, $|s|$ denote the length of s. Given two strings s and t, assume that y is the longest string such that $s = xy$ and $t = yz$ for $|x|, |z| \geq 1$. That is, $|y|$ is the length of overlap existing between s and t, and it is denoted by $|over(s,t)|$. Then, x is the proper prefix of s with respect to t and is represented by $pref(s,t)$. $|pref(s,t)| = |s| - |over(s,t)|$, $|pref(s,t)|$ is denominated the distance from s to t.

Let us clarify the definitions by giving some examples. Given $s = ac\mathbf{acta}$ and $t = \mathbf{acta}ggt$, the overlap y between these two strings is *acta* with length $|over(s,t)| = 4$. The proper prefix of s is $pref(s,t) = ac$ with length $|ac| = 2$.

2.1. The distance graph and the cycle cover problem

The SCS problem can be modeled as a minimization problem with respect to the prefix distance or as a maximization problem with respect to the total overlap [18]. All implemented algorithms in this article are based on the first model. A complete digraph $G = (V, A)$ can be constructed from P as follows: a string $s_i \in P$ represents a vertex $v_i \in G$. The arc cost $c(v_i, v_j)$ is defined by $|pref(s_i, s_j)|$. This graph is denominated the distance graph. Finding the shortest common superstring from P is equivalent to finding the minimum cost Hamiltonian path in $G(V, A)$. Let $\pi^* = \pi_1^*, ..., \pi_n^*$ be the permutation of vertices of V which define this path. The shortest common superstring $S^*(P)$ is a string containing all the strings of P, therefore its length is given by $|S^*(P)| = |pref(s_{\pi_1^*}, s_{\pi_2^*})| + |pref(s_{\pi_2^*}, s_{\pi_3^*})| + \cdots + |pref(s_{\pi_{n-1}^*}, s_{\pi_n^*})| + |s_{\pi_n^*}|$. Finding a minimum weight Hamiltonian cycle in $G(V, A)$ is known as the Asymmetric Traveling Salesman Problem (ATSP); the cost of this cycle is defined as $c(ATSP^*) = |pref(s_{\pi_1^*}, s_{\pi_2^*})| + |pref(s_{\pi_2^*}, s_{\pi_3^*})| + \cdots + |pref(s_{\pi_n^*}, s_{\pi_1^*})|$. Since $|s_{\pi_n^*}| = |pref(s_{\pi_n^*}, s_{\pi_1^*})| + |over(s_{\pi_n^*}, s_{\pi_1^*})|$, $c(ATSP^*(G))$ represents a lower bound for the optimum of the SCS $S^*(P)$:

$$c(ATSP^*(G)) \leq |S^*(P)|.$$

The best approximation known for ATSP is $O(\log(N))$ times $c(ATSP^*(G))$ [8]. Blum and colleagues [4] proposed approximations based on the assignment problem in bipartite graphs, also known as cycle cover. A cycle cover is a collection of disjoint cycles, i.e. each vertex $v_i \in V$ belongs to a single cycle. The cost of a cycle cover is given by the sum of the costs of each of its cycles. Computing a cycle cover is equivalent to finding a minimum weighted perfect matching in the bipartite graph $G'(U, V, E)$ obtained from $G(V, A)$ (distance graph). This problem is reduced to finding the set of edges of G', where the cost of this set is minimum and covers all the vertices in G'. It is known that this solution can be computed in $O(n^3)$ time by using the Hungarian algorithm [13]. The optimum cycle cover cost $c(C^*(G))$ represents a lower bound for the optimum tour in the ATSP, and therefore a lower bound to the optimum of the SCS problem, i.e.,

$$c(C^*(G)) \leq c(ATSP^*(G)) \leq |S^*(P)|.$$

3. Approximation Algorithms for the SCS Problem

An algorithm is a $\rho-$approximation algorithm for the SCS if it runs in polynomial time and always finds a superstring of length at most $\rho|S^*(P)|$; ρ is the approximation factor. In this work, we implement two approximation algorithms 4_{Arb} [4] and 3_{Arb} [4]. We also propose two variants for theses algorithms and we call them 4_{All} and 3_{All}, respectively. The algorithms are detailed as follows:

3.1. A 4-approximation algorithm for the SCS

ALGORITHM 1. 4_{Arb} [4]
Input: Set of strings P.
Output: A superstring $S(P)$ of the set P.

1 Compute the distance graph $G(V, A)$ from the set P (see preliminaries).
2 Find the minimum length cycle cover of G. Let us denote this cycle cover as $C^*(G) = \{c_1, c_2, ..., c_i, , ..., c_{|C^*(G)|}\}$ where each c_i represents a cycle of $C^*(G)$ (see preliminaries).
3 For each cycle $c_i = \langle v_{i_1}, ..., v_{i_{|c_i|}}, v_{i_1} \rangle \in C^*(G)$ build the superstring $\hat{s}_{c_i} = pref(s_{i_1}, s_{i_2}) \circ ... \circ pref(s_{i_{|c_i|-1}}, s_{i_{|c_i|}}) \circ s_{i_{|c_i|}}$ ($\circ$ is concatenation)[1]. where $v_{i_1} \in c_i$ is the cycle break point vertex which is arbitrarily selected, and $s_{i_1} \in P$ is the string associated to vertex v_{i_1}.
Let $T = \{\hat{s}_{c_1}, ..., \hat{s}_{c_{|C^*(G)|}}\}$ be the set of superstrings obtained from these cycles.
4 $S(P)$ is a common superstring of P, obtained by the arbitrary concatenation of strings in T.

Clearly, $S(P)$ is a superstring of P, since each string $\hat{s}_{c_i}$ is a superstring of the strings associated to each vertex of cycle $c_i \in C^*(G)$, and $C^*(G)$ is a cycle cover. This means that each string s_k is included in some cycle c_i, therefore, it is a substring of $\hat{s}_{c_i}$ which is, at the same time, a substring of $S(P)$.

1 In the rest of the paper $\hat{s}_{c_i} = pref(s_{i_1}, s_{i_2}) \circ, ..., \circ pref(s_{i_{|c_i|-1}}, s_{i_{|c_i|}}) \circ s_{i_{|c_i|}}$, is denoted as $\hat{s}_{c_i} = s_{i_1} \circ, ..., \circ s_{i_{|c_i|}}$.

3.2. A 3-approximation algorithm for the SCS

ALGORITHM 2. 3_{Arb} [4]
Input: Set of strings P.
Output: A common superstring $S(P)$ of P.

1 Compute the distance graph $G(V, A)$ from the set P.
2 Find the minimum length cycle cover of G. Let us denote this cycle cover as $C^*(G) = \{c_1, c_2, ..., c_i, , ..., c_{|C^*(G)|}\}$ where each c_i represents a cycle of $C^*(G)$ (see preliminaries).
3 For each cycle $c_i = \langle v_{i_1}, ..., v_{i_{|c_i|}}, v_{i_1}\rangle \in C^*(G)$, build the superstring $\hat{s}_{c_i} = s_{i_1} \circ, ..., \circ s_{i_{|c_i|}}$, where $v_{i_1} \in c_i$ is the cycle break point vertex which is arbitrarily selected. Let $T = \{\hat{s}_{c_1}, ..., \hat{s}_{c_i}, ..., \hat{s}_{c_{|C^*(G)|}}\}$.
be the set of superstrings obtained from this cycles.
4 Obtain the distance graph $G'(V', A')$ from the set $T = \{\hat{s}_{c_1}, \hat{s}_{c_2}, ..., \hat{s}_{c_{|C^*(G)|}}\}$.
5 Find the non trivial cycle cover $C^*(G')$ of minimum length of G'. A non trivial cycle cover is the one where all cycles have at least two vertices. That is, selfcycles are not allowed.
6 For each cycle $c'_i = \langle v'_{i_1}, ..., v'_{i_{|c'_i|}}, v'_{i_1}\rangle \in C^*(G')$, build the superstring: $\hat{s}_{c'_i} = \hat{s}_{v'_{i_1}} \circ, ..., \circ \hat{s}_{v'_{i_{|c'_i|}}}$, where $v'_{i_1} \in c'_i$ is the cycle breaking point vertex and $\hat{s}_{v'_{i_1}} \in P$ is the string associated to vertex $v'_{i_1} \in V'$ which originates the shortest superstring that can be generated among all $|c'_i|$ vertices.
Let $T' = \{\hat{s}_{c'_1}, ..., \hat{s}_{c'_i}, ..., \hat{s}_{c'_{|C^*(G')|}}\}$ be the set of derived superstrings from these cycles.
7 $S(P)$ is a common superstring of P, obtained by arbitrarily concatenating the superstrings in T'.

Figure 1 shows an example to compare algorithms 4_{Arb} and 3_{Arb} in terms of procedures and solution qualities. Each numbered box is associated to the corresponding step of the algorithms. The first three steps are the same for both algorithms. The output of 4_{Arb} $S(P)$, is indicated on the right of box 3. The following boxes show the procedures for 3_{Arb}. Box number 5 shows how to select the cycle breaking point. For this example the cycle has only two vertices (0,1), therefore only two superstrings $\langle 0, 1\rangle$ and $\langle 1, 0\rangle$ can be constructed, the one with the shortest length is $S1$. Then it is selected as a representative cycle. Box 6 shows the output of the algorithm 3_{Arb} $S(P)$, this superstring is shorter than the one obtained by 4_{Arb}.

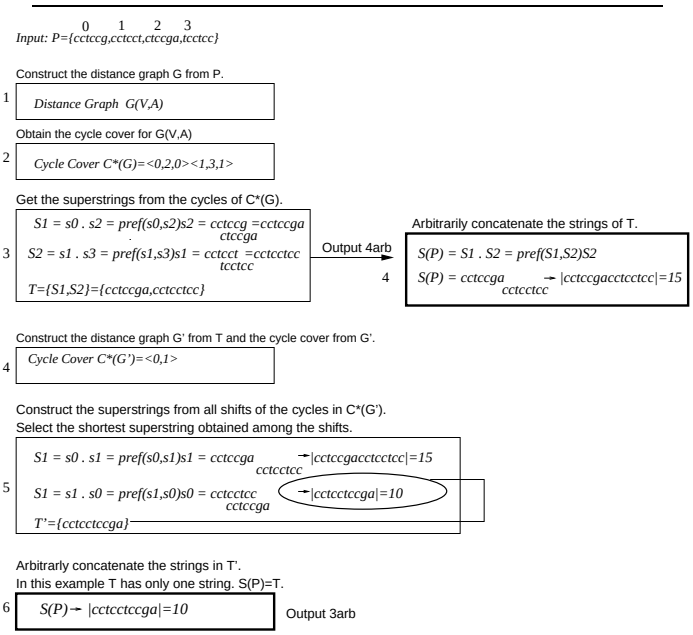

Figure 1. Comparison of 4_{Arb} with 3_{Arb}

3.3. Proposed variant for the 4-approximation algorithm

ALGORITHM 3. 4_{All}
Input: Set of strings P.
Output: A common superstring $S(P)$ of P.

We propose to modify 4_{Arb} as follows:
Steps 1, 2, and 4 remain the same as in 4_{Arb}. Replace step 3 of 4_{Arb} by:

3 For each cycle $c_i = \langle v_{i_1}, ..., v_{i_{|c_i|}}, v_{i_1}\rangle \in C^*(G)$, build the superstring $\hat{s}_{c_i} = s_{i_1} \circ, ..., \circ s_{i_{|c_i|}}$, where $v_{i_1} \in c_i$ is the cycle breaking vertex that originates the shortest superstring among all the $|c_i|$ superstrings generated starting at each vertex.
Let $T = \{s_{c_1}, ..., s_{c_t}\}$ be the set of superstrings derived from these cycles.

Notice that step 3 of algorithm 4_{Arb} decides to break each cycle c_i in an arbitrary point (vertex), while we propose to select this point so as to generate the shortest superstring $\hat{s}_{c_i}$. This will allow us to have a final superstring $S(P)$ which is a concatenation of the shortest superstrings $\hat{s}_{c_i}$ generated from each cycle c_i. Therefore, the superstring generated by our variant is shorter than the one generated by the 4_{Arb} algorithm.

This variant is similar to the MGREEDY algorithm [4]. Both algorithms find disjoint cycles and cut the cycles at optimum positions. The difference is that MGREEDY uses a greedy strategy to find the disjoint cycles and the break-

ing points, while our algorithm uses the cycle cover for finding the disjoint cycles, and the optimum position is obtained trying all possible shifts.

3.4. Proposed variant for the 3-approximation algorithm

ALGORITHM 4. 3_{All}
Input: Set of strings P.
Output: A common superstring $S(P)$ of P.

We propose to replace step 3 of 3_{Arb} by step 3 of 4_{All}. The remaining steps are as in 3_{Arb}. The main motivation for introducing this change is the same as the one for introducing the change in 4_{Arb}.

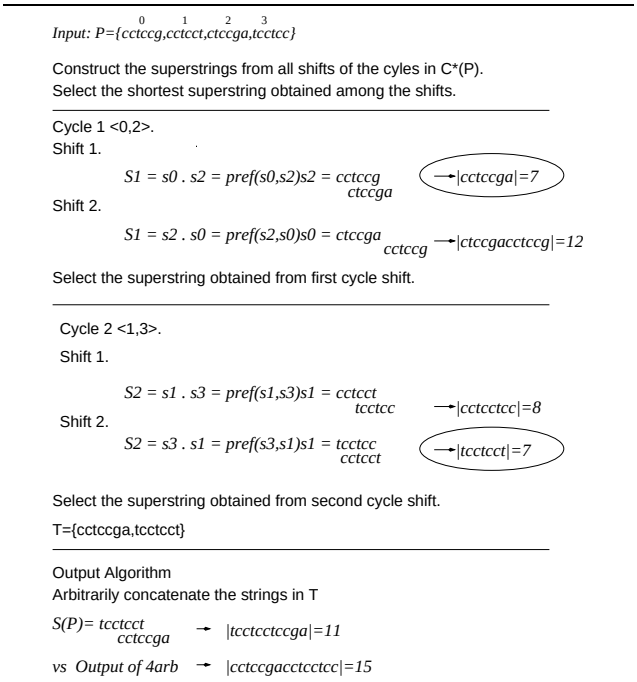

Figure 2. Proposed variants for 4_{Arb} and 3_{Arb}

The modification proposed for algorithm 4_{Arb} is shown in Figure 2. The modified part is on step 3, where all possible shifts of the cycle are performed. For the superstring constructed from cycle 1, vertex 0 is selected as the breaking point. For the superstring of cycle 2, the breaking point is vertex number 3. After concatenating superstrings 1 and 2, a superstring of length 10 is obtained, saving, in this way five characters with respect to 4_{Arb}.

4. Experimental Setup and Results

The main point of comparison for these approximation algorithms is the solution quality (superstring length). All algorithms are implemented in ANSI C using an implementation of the Hungarian algorithm available in [12].

4.1. Generation of Input Instances

Since there are not available benchmarks for the SCS problem we propose to generate them in three different ways:

1. Method D_{DNA}. Set of strings derived from a fragment of an original DNA sequence.
2. Method D_{RAND}. Set of strings derived from a randomly generated sequence with uniform distribution over a finite alphabet $\Sigma = \{A, C, G, T\}$.
3. Method I_{RAND}. Set of strings generated randomly with uniform distribution over a finite alphabet $\Sigma = \{A, C, G, T\}$.

Method D_{DNA}
The input instances are obtained as follows:

1. Obtain a DNA sequence $\mathcal{S}$ of length N^2 from GenBank (see Appendix).
2. Derive a set $\mathcal{E}$ of strings of equal length ℓ from the sequence $\mathcal{S}$ by shifting the characters one by one.
3. Modify $\mathcal{E}$ by randomly eliminating a given number of strings (percentage of error) so that some of the resulting strings do not necessarily overlap in exactly $\ell - 1$ characters. In these experiments $\mathcal{E}$ has a 20% of error. Let us denote the new set as $\mathcal{E}'$.
4. Ensure that $\mathcal{E}'$ is a free subset P which becomes the input of the algorithm.

Figure 3 shows an example for this method.

Method D_{RAND}
The instances are obtained applying the following procedures:

1. Generate randomly and uniformly a sequence $\mathcal{S}$ of length N with characters from $\Sigma = \{A, C, G, T\}$.
2. Apply steps 2-4 of D_{DNA} to the sequence $\mathcal{S}$ generated at step 1.

Method I_{RAND}
The instances for this method are generated following the procedure known as "memoryless source" [16]. A number of n strings of length ℓ are generated, where each character $w_i \in \Sigma$ of string s_i is generated independently with probability $1/|\Sigma|$. The generation of characters is modeled as a Bernoulli process.

2 $N = n/(1-\alpha) + \ell - 1$, where n is the number of strings in P, α is the percentage of error (see step 2), ℓ is the length of strings (step 2). This formula obtains the length of string $\mathcal{S}$ that assure a set P of size n.

1. DNA sequence from Genbank:

$\mathcal{S} = CACCGCA \qquad |\mathcal{S}| = 11$

2. Derive a set $\mathcal{E}$ of strings of length $\ell = 4$ from the sequence $\mathcal{S}$:

```
    CACCGCA
1 | CACC
2 |  ACCG
3 |   CCGC
4 |    CGCA
```

3. Percentage of error = 20%:

*Number of strings to erase: 0.20 * 4 = 0.8 ≈ 1*

4. Index of string to erase:

rand(1..$|\mathcal{E}|$) = rand(1..4) = 3

5. The new set $\mathcal{E}'$ is obtained by erasing: this string:

```
    CACCGCA
1 | CACC
2 |  ACCG
3 |   CCGC
4 |    CGCA
```

6. Convert $\mathcal{E}'$ into a free subset. (i.e. erase all repeated string). In this example $\mathcal{E}'$ is a free subset.

7. Let $P = \{CACC, ACCG, CGCA\}$ be the algorithm's input.

Figure 3. Illustration of the D_{DNA} method.

The main idea of using this method is to have a variation in the length of overlaps among the strings and to see whether the length of overlaps has any influence on the algorithm performance.

4.2. Solution Quality

In order to evaluate the solution quality produced by each algorithm, the Held-Karp (HK) bound also known as the *subtour elimination polytope* is computed. This is a solution to the relaxation of the integer programming formulation of the Symmetric Traveling Salesman Problem (STSP) [11, 10]. Remember that a lower bound for the shortest common superstring is the optimum cost tour in the ATSP. There exists a polynomial time transformation which maps an instance A of ATSP into an instance A' of STSP, and the cost of the optimum tour in A equals the cost of the optimum tour in A', that is, $c(ATSP^*(A)) = c(STSP^*(A'))$. Therefore HK is a lower bound for A' and for A. As it is also the case with the cycle cover cost $c(C^*(G))$, HK is a lower bound for the $ATSP^*(G)$, however, according to experimental results [10] it is closer to $ATSP^*(G)$ than to $c(C^*(G))$. Since ATSP is a lower bound for the SCS then we have that:

$$c(C^*(G)) \leq HK(G) \leq c(ATSP^*(G)) \leq |S^*(P)|$$

The HK bound is calculated based on the following procedures:

1. The distance graph $G = (V, A)$ is transformed into an undirected graph $G' = (3V, E)$. Details of this transformation can be found in [10].
2. A publically available code (the `concorde` program) [1] is run over the transformed instances.

4.3. Results

The instance sizes are in the range of 100 to 1,000 with increments of 100. The length (ℓ) of the strings are 10 and 50. All the experiments are run on 100 different instances and the mean values are taken for each of them.

Figure 4 (a) shows a comparison in the approximation ratio between the solutions of algorithms 4_{Arb} and 3_{Arb} for $\ell = 10$. The results are very close to the optimum, all of them are in an approximation range of 100% to 108% which is far away from the 400% worst case guarantee. Instances given the highest approximation ratio are those generated by the D_{DNA} method while the ones with the best approximation ratios are mainly those generated by the I_{RAND} method. In this figure, we can clearly see that the approximation percentage of algorithm 4_{Arb} is similar to that of algorithm 3_{Arb}, for all methods D_{ADN}, D_{RAND} and I_{RAND}. This behavior indicates that the quotient between them is in the range of 1.000 to 1.002 far from the worst case quotient of $4/3 = 1.333$. Note that as the instance size increases the approximation ratio approaches 100%.

The results for $\ell = 50$ are shown in Table 1, the row $D_{DNA}(600, 50)$ shows results obtained for an input instance of 600 strings all of length 50, derived from a DNA sequence (D_{DNA} Method). Our goal here is to analyze the influence of ℓ on the algorithms performance. Notice that there is an increasing difference between the algorithms Arb and All, while the first starts to produce worse approximations reaching 128.175%, the second stays low with the worst approximation of only 100.015%. The algorithms approximation quotients remains close to one.

In Figure 4(b) we can observe the high quality solution of our variant 4_{All}. For $\ell = 10$ they delivered solutions no more than 101%. When the length of ℓ increases to 50, the quality solution is better and it is not more than 100.20%. In contrast we can see that the approximation ratio of the algorithm 4_{Arb} is affected with this increase in ℓ. The solution quality of our other variant 3_{All} is very similar to that produced by 4_{All}.

We also performed experiments (no presented here) for $\ell = 100$ and $\ell = 500$. In these cases the Arb version continues with an increasing tendency in the approximation ratio (the highest reaches 128.175%). For the Arb algorithms, the worst scenario is given when the set of instances is de-

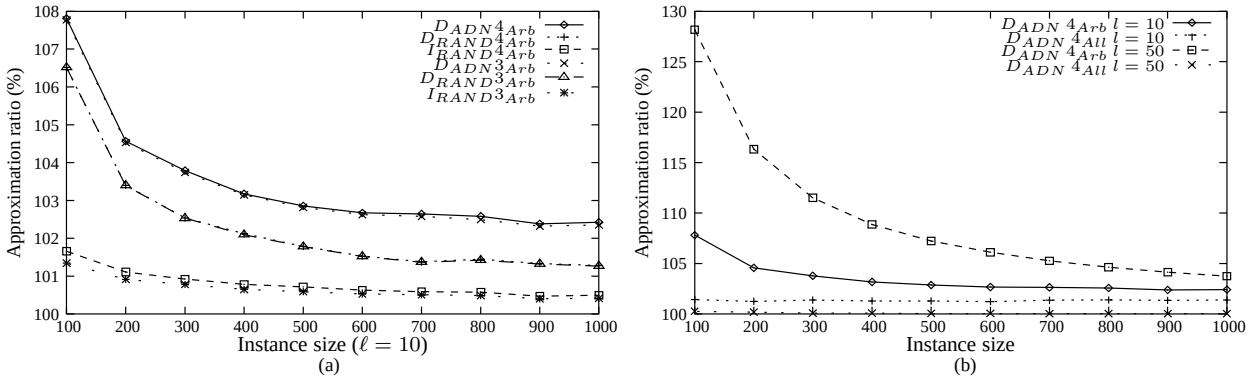

Figure 4. (a) Approximation ratio of algorithms 4_{Arb} and 3_{Arb}. Instances generated by D_{ADN}, D_{RAND} and I_{RAND} methods with $\ell = 10$. (b) Approximation ratio of algorithms 4_{Arb} and 4_{All}. Instances generated by the D_{ADN} method with $\ell = 10, 50$.

rived from a real DNA sequence or from a randomly generated sequence. However, the All variants continue to be close to the optimum. For these values of ℓ, the instances generated by methods D_{RAND} and D_{DNA} generate only one cycle in the cycle cover problem. Then the shortest cycle among all shifts (n) represents the optimum for ATSP and therefore for the SCS [10]. This results is very interesting because the solution quality is not only good, but is the optimum.

Figure 5 shows computational times in seconds for some instances for the algorithms 3_{All} and 3_{Arb}. We can see that the variant 3_{All} needs a little more computational effort to get its improved results over 3_{Arb}. We can also observed an increase in the execution time when the length of strings is increased.

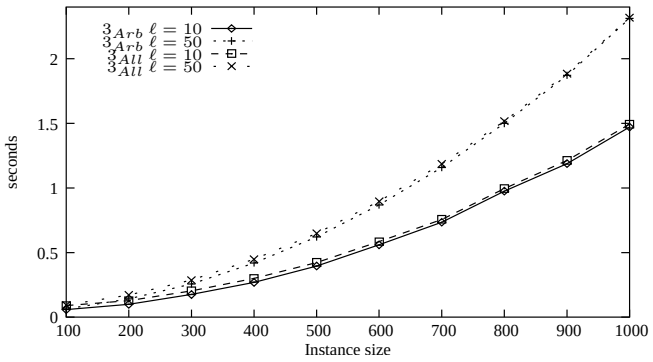

Figure 5. Computational time, in seconds, for 3_{Arb} and 3_{All} algorithms (average over one hundred instances). All the experiments were performed in a PC station with Athlon XP 2.0 Ghz , 512 MB and Linux Mandrake 9.1 operating system

5. Conclusions

An experimental comparison of two approximation algorithms for the Shortest Common Superstring Problem have been presented. The algorithms have worst case performance guarantee of three times the optimum and four times the optimum, respectively. A variant for both algorithms is presented to increase the solution quality.

Extensive experimental computations on different sets of instances show that the average case behavior of the algorithms does not follow the $4/3$ worst case quotient but it is very close to one. This allows us to think that algorithms with better bounds will not produce solutions with a significant improvement in the solution quality. The experiments also reveal that the proposed variants generate extremely high quality solutions of at most 1.4% over the optimum in a short computational time.

These results motivate a future research in the direction of finding average case approximation bounds that can better predict the algorithm behavior on real size instances. Another line of research has to do with the extension of these algorithms to deal with real DNA sequencing problems.

Acknowledgments

The authors thank to the anonymous referees for their valuable comments. This research has been partially supported by LAFMI Project REF J000-0686/2003.

6. Appendix

The accession codes of the sequences used in these experiments are:

X02160, X56088, D00723, X51841, X03350, D11428, Y00093, X03444, X13440, X14758, X14034, X13403, X03445, X02994, X02874, Y00264, X06374, X04350, Y00649, X04412, X01098, X15610, X03663, X57398, X07743, X07577, X04217, X07173, X53279, X04772,

Instance	$\%4_{Arb}$	$\%4_{All}$	$\%\,3_{Arb}$	$\%3_{All}$	$4_{Arb}/3_{Arb}$	$4_{All}/3_{All}$
$D_{ADN}(100,50)$	128.175	100.273	128.175	100.273	1.000	1.000
$D_{RAND}(100,50)$	127.886	100.204	127.886	100.204	1.000	1.000
$I_{RAND}(100,50)$	100.234	100.094	100.202	100.057	1.000	1.000
$D_{ADN}(200,50)$	116.333	100.179	116.333	100.179	1.000	1.000
$D_{RAND}(200,50)$	116.330	100.148	116.330	100.148	1.000	1.000
$I_{RAND}(200,50)$	100.164	100.078	100.136	100.044	1.000	1.000
$D_{ADN}(300,50)$	111.513	100.092	111.513	100.092	1.000	1.000
$D_{RAND}(300,50)$	111.518	100.076	111.518	100.076	1.000	1.000
$I_{RAND}(300,50)$	100.117	100.049	100.099	100.025	1.000	1.000
$D_{ADN}(400,50)$	108.892	100.091	108.892	100.091	1.000	1.000
$D_{RAND}(400,50)$	108.897	100.075	108.897	100.075	1.000	1.000
$I_{RAND}(400,50)$	100.111	100.046	100.092	100.027	1.000	1.000
$D_{ADN}(500,50)$	107.241	100.061	107.241	100.061	1.000	1.000
$D_{RAND}(500,50)$	107.232	100.067	107.232	100.067	1.000	1.000
$I_{RAND}(500,50)$	100.091	100.040	100.076	100.024	1.000	1.000
$D_{ADN}(600,50)$	106.111	100.055	106.111	100.055	1.000	1.000
$D_{RAND}(600,50)$	106.099	100.063	106.099	100.063	1.000	1.000
$I_{RAND}(600,50)$	100.077	100.032	100.064	100.018	1.000	1.000
$D_{ADN}(700,50)$	105.272	100.030	105.272	100.030	1.000	1.000
$D_{RAND}(700,50)$	105.273	100.038	105.273	100.038	1.000	1.000
$I_{RAND}(700,50)$	100.074	100.031	100.062	100.020	1.000	1.000
$D_{ADN}(800,50)$	104.643	100.035	104.643	100.035	1.000	1.000
$D_{RAND}(800,50)$	104.636	100.049	104.636	100.049	1.000	1.000
$I_{RAND}(800,50)$	100.066	100.026	100.054	100.015	1.000	1.000
$D_{ADN}(900,50)$	104.153	100.035	104.153	100.035	1.000	1.000
$D_{RAND}(900,50)$	104.153	100.029	104.153	100.029	1.000	1.000
$I_{RAND}(900,50)$	100.063	100.025	100.052	100.015	1.000	1.000
$D_{ADN}(1000,50)$	103.745	100.028	103.745	100.028	1.000	1.000
$D_{RAND}(1000,50)$	103.760	100.032	103.760	100.032	1.000	1.000
$I_{ADN}(1000,50)$	100.060	100.025	100.050	100.015	1.000	1.000

Table 1. Comparison results (average case over one hundred instances)

X58377, X52104, Y00503, X03795, X07696, X00351, X06985, X13097, X04808, NT_077402.10, NT_077911.10, NT_032968.60, NT_077912.10, NT_034471.30, NT_077913.20, NT_077914.20, NT_077915.10, NT_004350.16, NT_004321.15, NT_004547.16, NT_077919.20, NT_021937.16, NT_077382.20, NT_004873.15, NT_077920.20, NT_030584.10, NT_077921.10, NT_004610.16, NT_077383.30, NT_077922.20, NT_077384.10, NT_037485.30, NT_004538.15, NT_004511.16, NT_079482.10, NT_032977.60, NT_026943.13, NT_004686.16, NT_028050.13, NT_029860.11, NT_019273.16, NT_004754.15, NT_077387.20, NT_022052.20, NT_077988.20, NT_022071.12, NT_077930.10, NT_077389.20, NT_077931.20, NT_077932.20, NT_077933.10, NT_004434.16, NT_034398.40, NT_034400.20, NT_077936.20, NT_079483.10, NT_034401.50, NT_034403.30, NT_032962.50, NT_079484.10, NT_004668.16, NT_004487.16, NT_004671.15, NT_034410.50, NT_079485.10, NT_021877.16, NT_077939.10, NT_004559.11, NT_021973.16, NT_004433.16, NT_004836.15, NT_077941.10, NT_031730.80, NT_077390.20.

References

[1] D. Applegate, R. E. Bixby, V. Chvátal, and W. Cook. On the solution of traveling salesman problems. . volume 3, pages 645–656, 1998. The 12/15/1999 release of `Concorde` code is currently available from http://www.math.princeton.edu/tsp/concorde.html.

[2] C. Armen and C. Stein. A $2\frac{2}{3}$ approximation algorithm for the shortest superstring problem. In *Proceedings Combinatorial Pattern Matching*, volume 1075 of *Lecture Notes in Computer Science*, pages 87–101. Spring-Verlag, 1996.

[3] S. Arora, C. Lund, R. Motwani, M. Sudan, and M. Szegedy. Proof verifi cation and hardness of approximation problems.

In *33rd Annual Symposium on Foundations of Computer Science*, pages 14–23, October 24-27 1992.

[4] A. Blum, T. Jiang, M. Li, J. Tromp, and M. Yannakakis. Linear approximation of shortest superstring. *Journal of the ACM*, 41(4):630–647, 1994.

[5] D. Breslauer, T. Jiang, and Z. Jiang. Rotations of periodic strings and short superstrings. *Journal of Algorithms*, 24(2):340–353, 1997.

[6] A. Czumaj, L. Gasieniec, M. Piotrów, and W. Rytter. Sequential and Parallel Approximation of Shortest Superstrings. *Journal of Algorithms*, 23(1):74–100, 1997.

[7] A. L. (Ed.). *Computational Molecular Biology: Sources and Methods for Sequence Analysis*. Oxford University Press, 1988.

[8] A. Frieze, G. Galbiati, and F. Maffi oli. On the worst-case performance of some algorithms for the asymmetric traveling salesman problem. *Networks*, 12:23–39, 1982.

[9] J. Gallant, D. Maier, and J. Storer. On fi nding minimal length superstrings. *Journal of Computer and System Sciences*, 20(1):50–58, 1980.

[10] G. Gutin and A. Punnen. *The Traveling Salesman Problem and its Variations*. Kluwer Academic Publishers, 2000.

[11] M. Held and R. Karp. The traveling-salesman problem and minimum spanning trees: part II. *Mathematical Programming*, 1:6–25, 1971.

[12] D. Johnson, L. McGeoch, F. Glover, and C. Rego. Website for the DIMACS Implementation Challenge on the Traveling Salesman Problem. http://www.research.att.com/ dsj/chtsp/atsp.html.

[13] C. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Prentice Hall, 1982.

[14] J. Storer. *Data compression: methods and theory.* Computer Science Press, 1988.

[15] Z. Sweedyk. A 2-1/2 approximation algorithm for shortest superstring. *SIAM Journal on Computing*, 29(3):954–986, 1999.

[16] W. Szpankowski. *Average Case Analysis of Algorithms on Sequences*. Wiley-Interscience Publication, 2000.

[17] J. Tarhio and E. Ukkonen. A greedy approximation algorithm for constructiong shortest common superstrings. *Theoretical computer science*, 57(1):131–145, 1988.

[18] S. Teng and F. Yao. Approximating shortest superstrings. *SIAM Journal on Computing*, 26(2):410–417, 1997.

[19] J. Turner. Approximation algorithms for the shortest common superstring problem. *Information and computation*, 83(1):1–20, 1989.

[20] M. Waterman. *Introduction to Computational Biology: Maps, Sequences and Genomes*. CRC Press, 1995.