Approximation Algorithms for String Folding Problems

Giancarlo Mauri and Giulio Pavesi

Dept. of Computer Science, Systems and Communication
University of Milan–Bicocca
Milan, Italy
{mauri,pavesi}@disco.unimib.it

Abstract. We present polynomial–time approximation algorithms for string folding problems over any finite alphabet. Our idea is the following: describe a class of feasible solutions by means of an ambiguous context-free grammar (i.e. there is a bijection between the set of parse trees and a subset of possible embeddings of the string); give a score to every production of the grammar, so that the total score of every parse tree (the sum of the scores of the productions of the tree) equals the score of the corresponding structure; apply a parsing algorithm to find the parse tree with the highest score, corresponding to the configuration with highest score among those generated by the grammar. Furthermore, we show how the same approach can be extended in order to deal with an infinite alphabet or different goal functions. In each case, we prove that our algorithm guarantees a performance ratio that depends on the size of the alphabet or, in case of an infinite alphabet, on the length of the input string, both for the two and three–dimensional problem. Finally, we show some experimental results for the algorithm, comparing it to other performance–guaranteed approximation algorithms.

1 Introduction

We present performance–guaranteed approximation algorithms for different versions of the string folding problem. The motivation of string folding problems comes mainly from computational biology. One of the greatest challenges for computational biologists nowadays is to determine the three–dimensional native structure of a protein starting from the amino acid sequence that composes it. The problem has been studied from many different viewpoints, and many models have been proposed. Theoretical models are abstractions of the folding process that emphasize the effect of some factors while hiding other aspects. Perhaps the simplest and most studied model is the *two–dimensional hydrophobic–hydrophilic (HP) model* introduced by Dill [1]. In this model, the amino acid residues are grouped in two classes, according to their chemical properties: the hydrophobic, i.e. non–polar, and hydrophilic, i.e. polar. The protein instance can be thus reduced to a binary sequence of H's (meaning hydrophobic) and P's (meaning hydrophilic). Furthermore, to reduce the number of possible configurations, the

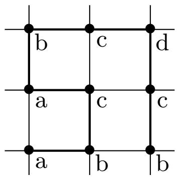


Fig. 1. Two–dimensional embedding of the string abcabcdcb. The score of the embedding is 4.

conformational space is discretized into a square lattice. Feasible structures are therefore mappings (embeddings) of the string into the grid, where adjacent symbols of the string lie on adjacent nodes, and no node is occupied by more than one symbol. It has been observed experimentally that hydrophobic amino acids tend to group inside the native structure, shielded from the environment by the hydrophilic ones. Thus, an optimal configuration for the protein is one that maximizes the number of H's that are *in contact* on the lattice, that is, lie on adjacent nodes of the lattice but are not adjacent in the input string. The study of structures generated by this and other theoretical models can provide useful insights into the dynamics of the folding process [2].

In this paper, we also deal with string folding problems of a more general type. We are given as input a string over some alphabet. The *score* of an embedding of the string is the number of equal symbols of the string that are in contact on the grid. Figure 1 shows an example. Usually, a *neutral symbol* is included in the alphabet. Contacts between neutral symbols do not contribute to the score of an embedding. For example, in the HP model P is the neutral symbol, and the score of the embeddings is determined only by the contacts between H's. The problem is to find the embedding of the string with the maximum score. The three–dimensional version of the problem is defined in the same way; in this case, strings are mapped into the three–dimensional rectangular grid.

The string folding problem over any alphabet (finite or infinite) is NP–hard both in the two and three–dimensional case [3,4,5]. Moreover, the three–dimensional version has been proved to be MAX–SNP hard [6].

The algorithms we present are suitable for more specialized discrete models of the folding of biological sequences, where the goal function does not depend only on contacts between equal symbols, or where contacts between equal symbols have different weights. For example, they could be applied to the string folding problem over an alphabet of twenty symbols, representing the twenty amino acids that build proteins, or to the RNA folding problem over an alphabet of four symbols. Although approximation algorithms for the HP model have already been proposed [7,8], to our knowledge these are the first performance–guaranteed algorithms for the generalized problems. Moreover, the same approach could be easily extended to other discrete or non–discrete models, where the goal function does not necessarily depend on the contacts between equal symbols, as long as the correspondence between parse trees and feasible structures is preserved.

2 The Algorithm

We will now show the basic algorithm, that works on any finite alphabet. Let $\mathbf{\Sigma}$ be the alphabet, with $|\mathbf{\Sigma}| = k$, let a_i, $0 \leq i \leq k-1$ be the symbols of the alphabet. Let a_0 be the neutral symbol, included in the alphabet. Our algorithm is based on the following steps:

1. Define an ambiguous context–free grammar, that generates all the possible instances of the problem (i.e., every string belonging to $\mathbf{\Sigma}^*$).
2. Define a relation between the derivations of the grammar and a subset of all the possible embeddings, where every production of a derivation recursively corresponds to a layout on the lattice of the terminal symbols generated by the production itself.
3. Assign to every production of the grammar an appropriate *score*, representing (a lower bound to) the number of contacts between equal (but not neutral) symbols generated by the spatial position of the symbols associated with the production.
4. Given an instance of the problem, apply a parsing algorithm in order to find the parse tree with the highest score (computed as the sum of the scores of the productions of the tree), that is, the tree corresponding to the embedding of maximum score among those that can be generated by the grammar.

Let us now introduce the grammar we employed in our algorithm. We defined a context–free grammar $G = \{T, N, S, P\}$, where:

1. $T = \{\mathbf{\Sigma} \cup u\}$ is the set of *terminal symbols*, where u is a dummy terminal symbol whose function will be explained later.
2. $N = \{S, L, R\}$ is the set of *nonterminal symbols*.
3. R is the *start symbol*, i.e. the root of every parse tree.
4. P is the set of the productions, composed by the following production schemes:
   (1) $S \rightarrow t_1\ S\ t_2$
   (2) $S \rightarrow t_1\ L\ t_2\ S\ t_3\ L\ t_4$
   (3) $S \rightarrow t_1\ L\ t_2\ S\ t_3 t_4$
   (4) $S \rightarrow t_1 t_2\ S\ t_3\ L\ t_4$
   (5) $S \rightarrow t_1 t_2$
   (6) $S \rightarrow t_1\ L\ t_2 t_3\ L\ t_4$
   (7) $S \rightarrow t_1 t_2 t_3\ L\ t_4$
   (8) $S \rightarrow t_1\ L\ t_2 t_3 t_4$
   (9) $L \rightarrow t_1\ L\ t_2$
   (10) $L \rightarrow t_1 t_2$
   with $t_i \in \mathbf{\Sigma}$;
   and by the following productions, that do not involve symbols from $\mathbf{\Sigma}$:
   (11) $S \rightarrow Suu$
   (12) $S \rightarrow uu$
   (13) $R \rightarrow SS$

The layout of the terminal symbols associated with each production is shown in Fig. 2. The proof that every parse tree corresponds to a feasible structure is straightforward. The score of every production is increased by one every time two equal non–neutral symbols generated by the production are in contact. For example, the production $S \rightarrow a_1La_1Sa_1La_1$ has score four, $S \rightarrow a_1La_1Sa_0La_0$ has score one, since neutral symbols do not contribute to the score, and so on. Possible contacts between equal symbols generated by different productions cannot be added to the score of the parse tree.

It could be argued that this grammar generates only sequences of *even* length. To solve this problem, and to avoid adding further productions to the grammar, in case of a sequence s of odd length the string actually parsed is $s^* = sa_0$. In fact, it can be proved that the best embedding among those that can be generated by the algorithm for the original sequence s is the structure found by the algorithm for s^*, with the final neutral symbol removed.

The algorithm builds structures in which the sequence is folded onto itself twice (see Fig. 3). The parse tree is split into two sub–trees, whose roots are the two S symbols generated by the start symbol R. The symbols generated by each sub–tree form a structure shaped like an "U", giving an overall configuration similar to a "C". If the length of the string is even, the first and last symbol are always in contact. Terminals generated by S nonterminals form the "backbone" of the structure, while symbols generated by L nonterminals form lateral branches. The introduction of the dummy terminal symbol u allows the grammar to generate a larger set of structures. The string actually parsed (after the possible addition of a neutral symbol) is $s^u = suu$. If the second sub–tree contains only the production $S \rightarrow uu$, the first sub–tree generates the whole sequence, which is again folded once to form a structure shaped like a "U". Without this extension the algorithm would not be able to generate U–shaped structures, with a significant decrease on its performance ratio (take for instance the string PHPPHP in the HP model, whose optimal structure is U–shaped).

3 The Parsing Algorithm

The parsing algorithm is based on the Earley algorithm for context–free grammars [9], and it is similar to the version that computes the Viterbi parse of a string generated by a stochastic grammar proposed by Stölcke [10]. It preserves the worst case time ($O(n^3)$) and space ($O(n^2)$) complexity of the two algorithms.

The Earley parser keeps a set of *states* for each symbol in the input, describing all pending derivations. A state has the form:

$$i \ : \ {}_kX \rightarrow \ \lambda.\mu$$

where X is a nonterminal symbol of the grammar, λ and μ are strings of terminals or nonterminals, such that $X \rightarrow \lambda\mu$ is a production of the grammar, i and k are indices into the input string. The i indicates that the state belongs to the set associated to the i-th symbol of the input string. The k indicates that the nonterminal X has been expanded starting from the k-th symbol of the input,

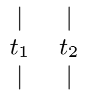
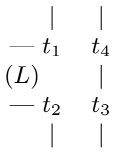
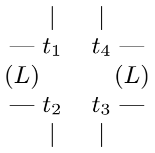
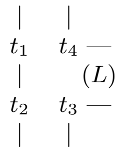
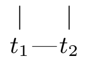
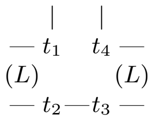
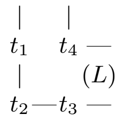
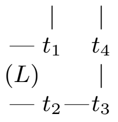
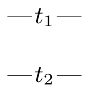
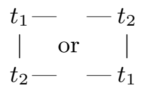
(1)	$S \to t_1\ S\ t_2$		+1 if $t_1 = t_2 \neq a_0$
(2)	$S \to t_1\ L\ t_2\ S\ t_3 t_4$		+1 if $t_1 = t_2 \neq a_0$ +1 if $t_1 = t_4 \neq a_0$ +1 if $t_2 = t_3 \neq a_0$
(3)	$S \to t_1\ L\ t_2\ S\ t_3\ L\ t_4$		+1 if $t_1 = t_4 \neq a_0$ +1 if $t_1 = t_2 \neq a_0$ +1 if $t_2 = t_3 \neq a_0$ +1 if $t_3 = t_4 \neq a_0$
(4)	$S \to t_1 t_2\ S\ t_3\ L\ t_4$		+1 if $t_1 = t_4 \neq a_0$ +1 if $t_2 = t_3 \neq a_0$ +1 if $t_3 = t_4 \neq a_0$
(5)	$S \to t_1 t_2$		always zero
(6)	$S \to t_1\ L\ t_2 t_3\ L\ t_4$		+1 if $t_1 = t_2 \neq a_0$ +1 if $t_1 = t_4 \neq a_0$ +1 if $t_3 = t_4 \neq a_0$
(7)	$S \to t_1 t_2 t_3\ L\ t_4$		+1 if $t_3 = t_4 \neq a_0$ +1 if $t_1 = t_4 \neq a_0$
(8)	$S \to t_1\ L\ t_2 t_3 t_4$		+1 if $t_1 = t_2 \neq a_0$ +1 if $t_1 = t_4 \neq a_0$
(9)	$L \to t_1\ L\ t_2$		+1 if $t_1 = t_2 \neq a_0$
(10)	$L \to t_1 t_2$		always zero

Fig. 2. Production schemes and corresponding layout of the symbols and scores. a_0 is the neutral symbol.

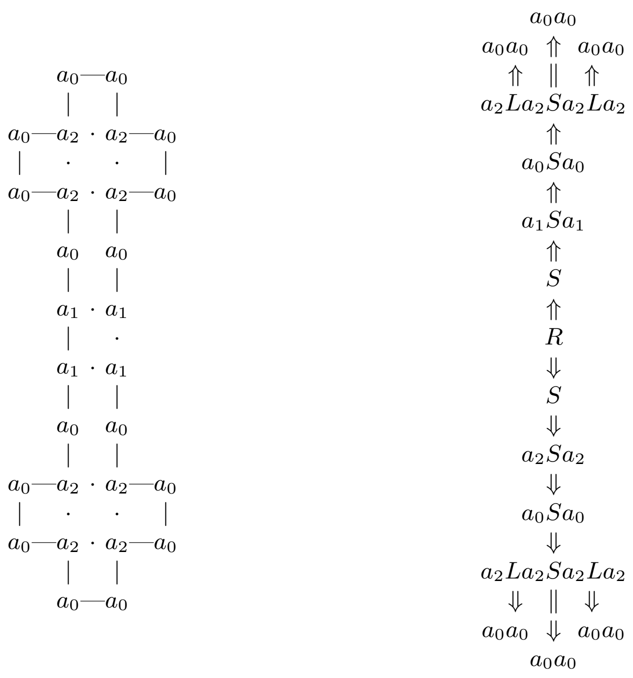


Fig. 3. Structure generated by the algorithm (score 11) for the sequence $a_1a_0a_2a_0a_0a_2a_0a_0a_2a_0a_0a_2a_0a_1a_1a_0a_2a_0a_0a_2a_0a_0a_2a_0a_0a_2a_0a_1$ and corresponding parse tree. Contacts between equal symbols are shown by dots $(\cdot)$. $\mathbf{\Sigma} = \{a_0, a_1, a_2\}$, where a_0 is the neutral symbol that does not contribute to the score of the embedding.

and the right–hand side of the production has been expanded up to the position indicated by the dot. A state with the dot at the end of the right–hand side is called a *complete* state, since the dot indicates that the left–hand nonterminal has been completely expanded.

The algorithm is based on three steps that scan the input string from left to right and build new states starting from the current set of states and the current input symbol. The three steps, given in input a string $s = s_0 \ldots s_{n-1}$, work as follows.

Prediction For each state

$$i \ : \ {}_kX \rightarrow \ \lambda.Y\mu$$

where Y is a nonterminal, and for all the productions $Y \rightarrow \nu$ of the grammar, add the state:

$$i \ : \ {}_iY \rightarrow .\nu$$

It can be seen that every prediction corresponds to a potential expansion of a nonterminal in a left–most derivation. A state generated by this step is called a *predicted* state.

Scanning For each state

$$i\ :\ {}_kX \rightarrow\ \lambda.a\mu$$

where a is a terminal symbol that matches the current input symbol s_i, add the state:

$$i+1\ :\ {}_kX \rightarrow\ \lambda a.\mu$$

that is, move the dot one position to the right in the right–hand side. This ensures that terminals generated by the productions match the input string.

Completion For each *complete* state:

$$i\ :\ {}_jY \rightarrow\ \nu.$$

and for each state in the set $j \leq i$

$$j\ :\ {}_kX \rightarrow\ \lambda.Y\mu$$

with the nonterminal Y after the dot, add the state:

$$i\ :\ {}_kX \rightarrow\ \lambda Y.\mu$$

A state generated by the completion step is called a *completed* state. A completed state corresponds to the fact that one of the nonterminal symbols in the right–hand side has been completely expanded (starting from a prediction step) and has generated a sub–string of the input string. The algorithm performs the three operations described above exhaustively, that is, until no new states can be generated.

The algorithm starts from an initial *dummy* state, whose left–hand side is empty:

$$0\ :\ {}_0 \rightarrow .R$$

(R is the start symbol of the grammar). Then, the states corresponding to R (and the possible states deriving from the productions with R on the left–hand side, and so on) are predicted, and the first scanning step examines the first symbol of the input string.

After scanning the last symbol of the string, and performing the corresponding completion step, the algorithm checks whether the state

$$n\ :\ {}_0 \rightarrow R.$$

is contained in the last set of states that has been produced, where n is the length of the input string. This means that the start symbol R has been completely expanded in order to build the input string, that is, the string belongs to the language generated by the grammar. If during the computation any set of states remains empty, the algorithm aborts, indicating that a prefix of the input string that cannot be generated by the grammar has been detected.

The only difference between our grammar and a context–free grammar is the introduction of a *score* associated with each production. We modified Earley's algorithm in order to compute the derivation that generates the input string with the highest score. Basically, we added to each state a score, as follows:

$$\begin{gathered} i \ : \ {}_kX \rightarrow \ \lambda.\mu \\ \text{Score} = p \end{gathered}$$

Intuitively, the idea is to have at the end of the parsing a state of the form:

$$\begin{gathered} n \ : \ {}_0 \rightarrow \ R. \\ \text{Score} = h \end{gathered}$$

where h is the score corresponding to the highest–score parse tree among those that can be generated for the input string. In order to obtain this result, we modified the three steps of the algorithm as follows.

Prediction For each state

$$\begin{gathered} i \ : \ {}_kX \rightarrow \ \lambda.Y\mu \\ \text{Score} = p \end{gathered}$$

where Y is a nonterminal, and for all the productions $Y \rightarrow \nu$ of the grammar, add the state:

$$\begin{gathered} i \ : \ {}_iY \rightarrow .\nu \\ \text{Score} = 0 \end{gathered}$$

that is, all predicted states have their score set to zero.

Scanning For each state

$$\begin{gathered} i \ : \ {}_kX \rightarrow \ \lambda.a\mu \\ \text{Score} = p \end{gathered}$$

where a is a terminal symbol that matches the current input symbol s_i, add the state:

$$\begin{gathered} i+1 \ : \ {}_kX \rightarrow \ \lambda a.\mu \\ \text{Score} = p \end{gathered}$$

that is, scores are left unchanged by the scanning step.

Completion The actual update of the scores takes place during the completion step. The score of the *complete* states changes as follows. For each *complete* state:

$$\begin{gathered} i \;:\; {}_jY \rightarrow \;\nu. \\ \text{Score} = p \end{gathered}$$

the score becomes:

$$\text{Score} = p + q$$

where q is the score associated to the production $Y \rightarrow \nu$. That is, we add to the score of the state (corresponding, as we will see, to the score of the structures generated by the nonterminals contained in the right–hand side of the production) the score corresponding to the layout of the terminal symbols generated by the production itself. Then, for each subset of the current set containing the states $S_1 \dots S_m$ of the form:

$$\begin{gathered} i \;:\; {}_jY \rightarrow \;\nu. \\ \text{Score} = q_l \end{gathered}$$

that is, a subset containing complete states with the same nonterminal on the left–hand side that were predicted at the same j, and for each state:

$$\begin{gathered} j \;:\; {}_kX \rightarrow \;\lambda.Y\mu \\ \text{Score} = p \end{gathered}$$

add the state:

$$\begin{gathered} i \;:\; {}_kX \rightarrow \;\lambda Y.\mu \\ \text{Score} = p + q^* \end{gathered}$$

where $q^* = \max_{1 \leq l \leq m}\{q_l\}$. That is, we add to the score of the state the score corresponding to the expansion of the nonterminal symbol Y *with the highest score*, i.e. the parse sub–tree with root Y with the highest score. It can be proved recursively that, at the end, the algorithm will associate the start symbol R with the score of the parse tree with the highest score. Moreover, the best parse tree can be reconstructed by assigning to each expanded nonterminal symbol of a completed state the corresponding complete state with the highest score.

4 Dealing with an Infinite Alphabet

In case of an infinite alphabet, the basic algorithm cannot be applied, since it would be impossible to generate a priori all the productions of the grammar and their corresponding scores. The solution we adopted is the following. We let the parser work only on the *production schemes*, and we build the productions on the fly while scanning the input string. The productions contained in the states of the parser have the terminal symbols specified only on the *left* side of the dot. That is, states generated during the prediction step of the parser contain, instead of terminal symbols, a special wildcard symbol ($*$). Wildcard symbols are replaced by terminals during the scanning step. For example, suppose we are scanning a given symbol s_i of the input string. Each state with a wildcard symbol after the dot:

$$i \ : \ {}_kX \rightarrow \lambda. * \mu$$

generates a new state:

$$i+1 \ : \ {}_kX \rightarrow \lambda s_i.\mu$$

where λ is composed by terminal or nonterminal symbols of the grammar, and μ is composed *only* by wildcard or nonterminal symbols (it does not contain symbols belonging to $\mathbf{\Sigma}$, since it still has to be expanded).

Also, the score of the productions cannot be computed in advance. As we have seen, the score of a state is set to zero, until the state is *completed* or *complete.* For complete states, the score of the corresponding production is computed according to the rules shown in Fig. 2, and then added to the score of the state itself, as in the finite case. When a state is completed, i.e. a nonterminal in its right–hand side has been completely expanded, the score is updated by adding the score of the maximum parse sub–tree generated, once again as in the finite case. Moreover, the scoring scheme for the productions can be easily changed in order to deal with different goal functions.

5 Performance Results

In this section, we prove some results concerning the performance of the algorithm when applied to the different versions of the problem.

In order to have a *performance–guaranteed approximation algorithm*, for every possible instance of the problem the ratio between the score of the structure generated by the algorithm and the score of the optimal structure must be bounded by a constant. That is, for every possible sequence s of arbitrary length we must have:

$$\mathcal{R}(s) = \frac{A(s)}{OPT(s)} \geq \mathcal{R} \tag{1}$$

where $A(s)$ is the score of the structure generated by the algorithm when given as input the sequence s, and $OPT(s)$ is the score of the optimal embedding. We will call $\mathcal{R}$ the *absolute performance ratio* of the algorithm, and denote with $\mathcal{R}_k$ the absolute performance ratio when dealing with an alphabet of size k.

5.1 Binary Alphabet

We will start from the performance ratio of the algorithm over a binary alphabet, i.e., in the HP model. Given a string $s = s_0 \dots s_n$, where $s_i \in$\{H, P\}, two symbols s_i and s_j can be in contact on the grid only if $|j-i|$ is odd. Furthermore, every symbol can be in contact with at most two other symbols, except when it is located at one of the endpoints of the sequence. In this case, it can be in contact with three other symbols.

Now, let h_e be the number of H's in even positions in a given sequence s; h_o the number of H's in odd positions; $h^* = \min\{h_e, h_o\}$. We also define $OPT(s)$ as the score (the number of contacts between H's) of the optimal embedding for a given sequence s. The above considerations yield the following:

Theorem 1.

$$OPT(s) \ \leq 2h^* + 2 \tag{2}$$

It can be observed that the upper bound $2h^* + 2$ can be reached only by sequences of odd length with two H's at the endpoints. We also can give a lower bound on the number of contacts that are generated by the algorithm.

Lemma 1. *Given a sequence s, there always exists an embedding for s, corresponding to a parse tree generated by the algorithm, that contains $\lceil \frac{h^*+1}{2} \rceil$ contacts between H's.*

The proof of this lemma is quite cumbersome. Actually, we have been able to prove that, in the set of the structures that can be generated by the algorithm, there always exists a structure with $\lceil \frac{h^*+1}{2} \rceil$ contacts, but not, for example, that this is the best structure of the set. Thus, we could give only a lower bound on the actual performance ratio of the algorithm. In fact, as shown in Section 6, the worst case that we found experimentally gave a performance ratio of 3/8. This will also affect, as we will see, the performance ratio for the more general versions. From the result of Lemma 1, however, it is straightforward to obtain the performance ratio of the algorithm.

Theorem 2.

$$\mathcal{R}_2 \geq \frac{\lceil \frac{h^*+1}{2} \rceil}{2h^* + 2} \geq \frac{1}{4} \tag{3}$$

5.2 Finite Alphabet

We will now show the results concerning the performance of the algorithm applied to the problem over any finite alphabet. Let s be a string of symbols taken from an alphabet $\mathbf{\Sigma}$, with $|\mathbf{\Sigma}| = k$. Also, let σ_i^o and σ_i^e, mbox$1 \leq i \leq k-1$ be the number of the occurrences of the symbol $a_i \in \mathbf{\Sigma}$ in the sequence s respectively in an odd and in an even position (we do not consider the number of occurrences of the neutral symbol a_0). Finally, let $\sigma_i^* = \min\{\sigma_i^o, \sigma_i^e\}$, $1 \leq i \leq k-1$, and $\sigma^* = \max_{1 \leq i \leq k}\{\sigma_i^*\}$.

Lemma 2. *For any sequence s over an alphabet $\mathbf{\Sigma}$ with $|\mathbf{\Sigma}| = k$,*

$$OPT(s) \leq 2 \sum_{i=1}^{k-1} \sigma_i^* + 2 \tag{4}$$

By considering the symbol that corresponds to σ^* the only non–neutral symbol of the alphabet, we can easily prove the following lemma.

Lemma 3. *For any sequence s over a finite alphabet, the set of structures generated by the algorithm contains a structure that generates at least $\lceil\frac{\sigma^*+1}{2}\rceil$ contacts.*

We want to point out that once again the structure of Lemma 3 does not correspond to the best structure that can be generated by the algorithm. In fact, the algorithm tries to generate contacts not only with the symbol corresponding to σ^*, but with all the non–neutral symbols, as shown in Fig. 3. Lemma 3 guarantees only three contacts between a_2 symbols, while the solution found by the algorithm contains eleven contacts, and corresponds to the optimal embedding. However, starting from the previous two lemmas, the worst case is a sequence where the number of occurrences of every non–neutral symbol is equal. Therefore, we have $OPT(s) \leq 2\sigma^*(k-1)+2$. This fact yields the following theorem:

Theorem 3.

$$\mathcal{R}_k \geq \frac{\lceil\frac{\sigma^*+1}{2}\rceil}{2\sum_{i=1}^{k-1}\sigma_i^*+2} \geq \frac{\sigma^*+1}{4[\sigma^*(k-1)+1]} \geq \frac{1}{4(k-1)} \tag{5}$$

5.3 Infinite Alphabet

For the proof of the performance ratio of the algorithm applied to the problem over an infinite alphabet, we start from the fact that, although the size of the alphabet is not bounded, the input sequence is finite. Therefore, the number of different symbols occurring in the sequence that can generate contacts is also finite. Let d be this number. The terms σ_i^* and σ^* are defined as in the previous section, but in this case we have $1 \leq i \leq d$, with $d \leq n/2$. Thus, the performance ratio of the algorithm on a given input s of length n can be defined as follows.

Theorem 4.

$$\mathcal{R}_\infty \geq \frac{\lceil\frac{\sigma^*+1}{2}\rceil}{2\sum_{i=1}^{d}\sigma_i^*+2} \geq \frac{\sigma^*+1}{4[\sigma^*(\frac{n}{2}-1)+1]} \geq \frac{1}{4(\frac{n}{2}-1)} \tag{6}$$

5.4 The 3D Case

Although structures generated by our algorithm are two–dimensional, they anyway guarantee a performance ratio also for the three-dimensional problem. In the three-dimensional lattice, each non–neutral symbol can be in contact with at most four equal symbols (five, if it is located at the endpoints of the sequence). This fact yields the following lemma.

Lemma 4. *For any sequence s over an alphabet $\mathbf{\Sigma}$ with $|\mathbf{\Sigma}| = k$,*

$$OPT(s) \leq 4\sum_{i=1}^{k-1}\sigma_i^*+2 \tag{7}$$

This, together with Lemma 3, gives the absolute performance ratio $\mathcal{R}_k^{3d}$ for the three-dimensional problem over an alphabet of size k.

Theorem 5.

$$\mathcal{R}_k^{3d} \geq \frac{\lceil \frac{\sigma^*+1}{2} \rceil}{4\sum_{i=1}^{k-1} \sigma_i^* + 2} \geq \frac{\sigma^* + 1}{8[\sigma^*(k-1)+1]} \geq \frac{1}{8(k-1)} \tag{8}$$

It is worth mentioning that in the case of a binary alphabet the absolute performance ratio of our algorithm (1/8) equals the best approximation algorithm known for the three–dimensional problem [7].

6 Experimental Evaluation

In the two–dimensional binary case, our algorithm equals the performance ratio of the best algorithms so far proposed [7]. Therefore, we have tested it on random instances of the two–dimensional problem over a binary alphabet, and compared the results to the other algorithms (see Table 1). Given $P_H = \Pr[s_i = H]$, $\forall i \in [0, n]$, for different values of P_H we have completed 1000 runs of our algorithm and of the other two with performance–guaranteed ratios of 1/4 (called $\mathcal{B}$ and $\mathcal{C}$ as in the original paper), on instances of length 63 with two H's at the endpoints (in order to reach the higher bound for the goal function).

The performance ratio of our algorithm seems to decrease as the average number of H's in the sequence is increased. The same trend, even if with lower ratios, is shown by algorithm $\mathcal{C}$, while algorithm $\mathcal{B}$ has a constant ratio. In the tests, the worst case ratio of 1/4 has been reached only by algorithm $\mathcal{B}$, while on sequences like PP(HPP)$^{4k+1}$, $k \geq 3$ (whose optimal score is $4k$), algorithm $\mathcal{C}$ produces structures with score $k + 3$. Thus, its performance ratio approaches 1/4 as k is increased [7]. It should be noted that our algorithm found the optimal

Table 1. Average performance ratios of algorithms $\mathcal{B}$ and $\mathcal{C}$ [7], and our algorithm (CFG) in the two–dimensional case of the problem over a binary alphabet, for different values of $P_H = \Pr[s_i = H], \forall i \in [0, n]$.

Algorithm	$\mathcal{B}$	$\mathcal{C}$	CFG
$P_H = .15$	0.52	0.60	0.79
$P_H = .33$	0.48	0.57	0.72
$P_H = .5$	0.48	0.55	0.68
$P_H = .66$	0.48	0.53	0.63
$P_H = .85$	0.48	0.50	0.55
Average	0.48	0.55	0.67
Worst case	0.25	0.33	0.375

solution on this set of instances. The worst case found for our algorithm is 3/8: this, as discussed in the previous sections, leaves open the issue of its performance ratio.

7 Conclusions

We presented polynomial–time approximation algorithms for the string folding problem that guarantee performance ratios both for the two– and three–dimensional case, and work over any alphabet, finite and infinite. To our knowledge, these are the first performance–guaranteed approximation algorithms that can be applied to the problem over alphabets larger than the binary one, while for the latter we equaled the performances of the best algorithms so far proposed, with better experimental results. In each case, the performance ratios that have been proved serve only as a lower bound for the actual ones. Our approach can also be easily extended to different goal functions, and also to more powerful grammars and non–discrete versions of string folding problems, as long as the correspondence between parse trees and feasible structures is preserved.

Acknowledgements

This work has been supported by the Italian Ministry of University, under the project "Bioinformatics and Genomic Research".

References

1. K.A. Dill, Dominant forces in protein folding. Biochemistry, **24**(1985), 1501.
2. B. Hayes, Prototeins. American Scientist, **3**(1998), 86.
3. M. Paterson, T. Przytycka, On the Complexity of String Folding. Discrete and Applied Maths, **71**(1996), 217–230.
4. P. Crescenzi, D. Goldman, C. Papadimitriou, A. Piccolboni, M. Yannakakis, On the Complexity of Protein Folding. Proc. of the Second Annual International Conference on Computational Biology (RECOMB '98), 61–62, New York, 1998.
5. B. Berger, T. Leighton, Protein Folding in the HP Model is NP Complete. Proc. of the Second Annual International Conference on Computational Biology (RECOMB '98), 30–39, New York, 1998.
6. A. Nayak, A. Sinclair, U. Zwick, Spatial Codes and the Hardness of String Folding Problems. Proceedings of the 9^{th} Annual ACM–SIAM Symposium on Discrete Algorithms (SODA '98), 639–648, San Francisco, 1998.
7. W.E. Hart, S.C. Istrail, Fast Protein Folding in the Hydrophobic-Hydrophilic Model. Journal of computational biology, **3**(1), 53–96, 1996.
8. G. Mauri, G. Pavesi, A. Piccolboni, Approximation Algorithms for Protein Folding Prediction. Proceedings of the 10^{th} Annual ACM–SIAM Symposium on Discrete Algorithms (SODA '99), S945–S946, Baltimore, 1999.
9. J. Earley, An Efficient Context-Free Parsing Algorithm. Communications of the ACM, **6**(1970), 451–455.
10. A. Stölcke, An Efficient Probabilistic Context–Free Parsing Algorithm That Computes Prefix Probabilities. Computational Linguistics, **21**(2), 165–201, 1995.