Combinatorial Problems on Strings with Applications to Protein Folding

Alantha Newman[1] and Matthias Ruhl[2]

[1] MIT Laboratory for Computer Science
Cambridge, MA 02139
alantha@theory.lcs.mit.edu
[2] IBM Almaden Research Center
San Jose, CA 95120
ruhl@almaden.ibm.com

Abstract. We consider the problem of protein folding in the HP model on the 3D square lattice. This problem is combinatorially equivalent to folding a string of 0's and 1's so that the string forms a self-avoiding walk on the lattice and the number of adjacent pairs of 1's is maximized. The previously best-known approximation algorithm for this problem has a guarantee of $\frac{3}{8} = .375$ [HI95]. In this paper, we first present a new $\frac{3}{8}$-approximation algorithm for the 3D folding problem that improves on the absolute approximation guarantee of the previous algorithm. We then show a connection between the 3D folding problem and a basic combinatorial problem on binary strings, which may be of independent interest. Given a binary string in $\{a, b\}^*$, we want to find a long subsequence of the string in which every sequence of consecutive a's is followed by at least as many consecutive b's. We show a non-trivial lower-bound on the existence of such subsequences. Using this result, we obtain an algorithm with a slightly improved approximation ratio of at least .37501 for the 3D folding problem. All of our algorithms run in linear time.

1 Introduction

We consider the problem of protein folding in the HP model on the three-dimensional (3D) square lattice. This optimization problem is combinatorially equivalent to folding a string of 0's and 1's, i.e. placing adjacent elements of the string on adjacent lattice points, so that the string forms a self-avoiding walk on the lattice and the number of adjacent pairs of 1's is maximized. Figure 1 shows an example of a 3D folding of a binary string.

Background. The widely-studied HP model was introduced by Dill [Dil85,Dil90]. A protein is a chain of amino acid residues. In the HP model, each amino acid residue is classified as an H (hydrophobic or non-polar) or a P (hydrophilic or polar). An optimal configuration for a string of amino acids in this model is one that has the lowest energy, which is achieved when the number of H-H contacts (i.e. pairs of H's that are adjacent in the folding but not in the string) is maximized. The *protein folding* problem in the hydrophobic-hydrophilic (HP) model on the 3D square lattice is combinatorially equivalent to the problem we just described: we are given a string of P's and H's (instead of 0's and 1's) and we wish to maximize the number of adjacent pairs of H's (instead of 1's). An informative discussion on the HP model and its applicability to protein folding is given by Hart and Istrail [HI95].

Related Work. Berger and Leighton proved that this problem is NP-hard [BL98]. On the positive side, Hart and Istrail gave a simple algorithm with an approximation guarantee of $\frac{3}{8}OPT - \Theta(\sqrt{OPT})$ [HI95]. Folding in the HP model has also been studied for the 2D square lattice. This variant is also NP-hard [CGP$^+$98]. Hart and Istrail gave a $\frac{1}{4}$-approximation algorithm for this problem [HI95], which was recently improved to a $\frac{1}{3}$-approximation algorithm [Ala02].

Our Contribution. Improving on the approximation guarantee of $\frac{3}{8}$ for the 3D folding problem has been an open problem for almost a decade. In this paper, we first present a new 3D folding algorithm (Section 2.1). Our algorithm produces a folding with $\frac{3}{8}OPT - \Theta(1)$ contacts, improving the absolute approximation guarantee. We then show that if the input string is of a certain special form, we can modify our algorithm to yield $\frac{3}{4}OPT - O(\delta(S))$ contacts, where $\delta(S)$ is the number of transitions in the input string S from sequences of 1's in odd positions in the string to sequences of 1's in even positions. This is described in Section 2.2.

In Section 3, we reduce the general 3D folding problem to the special case above, yielding a folding algorithm producing $.439 \cdot OPT - O(\delta(S))$ contacts. This reduction is based on a simple combinatorial problem for strings, which may be of independent interest.

We call a binary string from $\{a, b\}^*$ *block-monotone* if every maximal sequence of consecutive a's is immediately followed by a block of at least as many b's. Suppose we are given a binary string with the following property: every suffix of the string (i.e. every sequence of consecutive elements that ends with the last element of the string) contains at least as many b's as a's. What is the longest block-monotone subsequence of the string? It is easy to see that we can find a block-monotone subsequence with length at least half the length of the string by removing all the a's. In Section 3.1, we show that there always is a block-monotone subsequence containing at least a $(2 - \sqrt{2}) \approx .5857$ fraction of the string's elements.

Finally, we combine our folding algorithm with a simple, case-based algorithm that achieves $.375 \cdot OPT + \Omega(\delta(S))$ contacts, which is described in the full version of this paper. We thereby remove the dependence on $\delta(S)$ in the approximation guarantee and obtain an algorithm with a slightly improved approximation guarantee of .37501 for the 3D folding problem. Due to space restrictions, all proofs are omitted and can be found in the full version of this paper.

2 A New 3D Folding Algorithm

Let $S \in \{0, 1\}^n$ represent the string we want to fold. We refer to each 0 or 1 as an *element*. We let s_i represent the i^{th} element of S, i.e. $S = s_1 s_2 \ldots s_n$. We refer to a 1 in an odd position (i.e. $s_i = 1$ with odd index i) as an *odd-1* and a 1 in an even position (i.e. $s_i = 1$ with even index i) as an *even-1*. An *odd* or *even* label is determined by an element's position in the input string and does not change at any stage of the algorithm. We will use $\mathcal{O}[S]$ and $\mathcal{E}[S]$ to denote the number of odd-1's and even-1's, respectively, in a string S. For example, for $S = 10111100101101$, we have $\mathcal{O}[S] = 5$ and $\mathcal{E}[S] = 4$.

Note that because the square lattice is bipartite, the odd/even label determines the set of lattice points on which an element can be placed. For example, suppose we divide the lattice points into two bipartite sets, one red and one blue. If the first element of the string is placed on a red lattice point, then all the elements in odd positions in the string will be placed on red lattice points and all the elements in even positions in the string will be placed on blue lattice points.

A contact between two elements placed on the square lattice can therefore only occur between an odd-1 and an even-1. Each lattice point is adjacent to six neighboring lattice points. In any folding, if an odd-1 is placed on a particular lattice point, two neighboring lattice points will be occupied by preceding and succeeding (even) elements of the string unless the element is one of the two endpoints of the string. Therefore, there are four remaining adjacent lattice points with which contacts can be formed. Thus, an upper bound on the size of an optimal solution is $OPT \ \leq \ 4\min\{\mathcal{O}[S], \mathcal{E}[S]\} + 2$.

2.1 The Diagonal Folding Algorithm

We now present an algorithm that produces a folding with at least $\frac{3}{8}OPT - \Theta(1)$ contacts in the worst case, thereby improving the *absolute* approximation guarantee of the algorithm of Hart and Istrail [HI95]. Our algorithm is based on *diagonal folds*. The algorithm guarantees that contacts form on and between two adjacent 2D planes. Each point in the 3D lattice has an (x, y, z)-coordinate, where x, y, and z are integers. We will fold the string so that all contacts occur on or between the planes $z = 0$ and $z = 1$. The Diagonal Folding Algorithm is described on the next page and illustrated in Figure 1.

Lemma 1. *The* Diagonal Folding Algorithm *produces a folding with at least $\frac{3}{8}OPT - O(1)$ contacts.*

2.2 Relating Folding to String Properties

As the number of 1's placed on the diagonal in the Diagonal Folding Algorithm increases, the length (i.e. $\frac{1}{2}\min\{\mathcal{O}[S], \mathcal{E}[S]\}$) of the resulting folding increases in a direction parallel to the line $x = y$. The height of the folding may also increase depending on the maximum distance between consecutive odd-1's in $S_{\mathcal{O}}$ or consecutive even-1's in $S_{\mathcal{E}}$. However, regardless of the input string, the resulting folding has the same constant width in the direction parallel to the line $x = -y$. In other words, although the algorithm produces a three-dimensional folding, with increasing k and n, the folding may increase in length and height

Diagonal Folding Algorithm

Input: a binary string S.
Output: a folding of the string S.

1. Let $k = \min\{\mathcal{O}[S], \mathcal{E}[S]\}$.
2. Divide S into two strings such that $S_{\mathcal{O}}$ contains at least half the odd-1's and $S_{\mathcal{E}}$ contains at least half the even-1's. We can do this by finding a point on the string such that half of the odd-1's are on one side of this point and half the odd-1's are on the other side. One of these sides contains at least half of the even-1's. We call this side $S_{\mathcal{E}}$ and the remaining side $S_{\mathcal{O}}$. Then we replace all the even-1's in $S_{\mathcal{O}}$ with 0's and replace all the odd-1's in $S_{\mathcal{E}}$ with 0's.
3. Place the first odd-1 in $S_{\mathcal{O}}$ on lattice point $(1, 1, 1)$ and the next odd-1 in $S_{\mathcal{O}}$ on lattice point $(2, 2, 1)$ and so on. For the first $\frac{k}{4}$ of the odd-1's in $S_{\mathcal{O}}$, place the i^{th} odd-1 on lattice point $(i, i, 1)$. Then place the $(k/4+1)$ odd-1 on lattice point $(k/4-1, k/4+1, 1)$. For the first $\frac{k}{4}-1$ of the even-1's in $S_{\mathcal{E}}$, place the i^{th} even-1 on lattice point $(i, i+1, 1)$. Use the dimensions $z \geq 1$ to place the strings of 0's between consecutive odd-1's in $S_{\mathcal{O}}$ and the strings of 0's between consecutive even-1's in $S_{\mathcal{E}}$.
4. Place the $(k/4+2)$ odd-1 in $S_{\mathcal{O}}$ on lattice point $(k/4-2, k/4+1, 0)$. Then place the $(k/4+i)$ odd-1 in $S_{\mathcal{O}}$ on lattice point $(k/4-i+1, k/4-i+2, 0)$. Place the $(k/4)$ even-1 in $S_{\mathcal{E}}$ on lattice point $(k/4-1, k/4-1, 0)$. Place the $(k/4+i)$ even-1 in $S_{\mathcal{E}}$ on lattice point $(k/4-i-1, k/4-i-1, 0)$. Use the dimensions $z \leq 0$ to place the strings of 0's between consecutive 1's in $S_{\mathcal{O}}$ or $S_{\mathcal{E}}$.

but not in width. We will explain how we can use this unused space to improve the algorithm for a special class of strings.

By *consecutive odd-1's* we mean odd-1's that are not separated by even-1's and similarly for consecutive even-1's. For example, in the string 1010001100011, there is a sequence of 3 consecutive odd-1's followed by two consecutive even-1's followed by an odd-1.

Definition 2. *A string $S_{\mathcal{O}}$ is called* odd-monotone *if every maximal sequence of consecutive even-1's is immediately preceded by at least as many consecutive odd-1's.*

An *even-monotone* string is defined analogously. For example, the string 10101100011 is odd-monotone and the string 0100010101101101011 is even-monotone. We define a *switch* as follows:

Definition 3. *A* switch *is an odd-1 followed by an even-1 (separated only by 0's). We denote the number of switches in S by $\delta(S)$.*

For example, for the string $S = 100\underline{1}00010101101\underline{1}01011$, $\delta(S) = 2$ since there are two transitions (underlined) from a maximal sequence of consecutive odd-1's to a sequence of even-1's. We use these definitions in the following theorem.

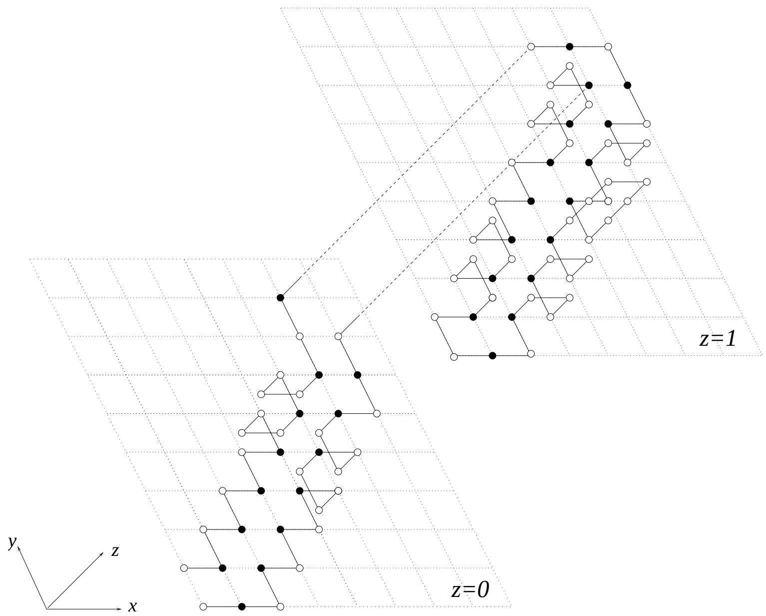

Fig. 1. This figure illustrates Steps 3 and 4 of the DIAGONAL FOLDING ALGORITHM. In the folding resulting from this algorithm, all contacts are formed on or between the 2D planes $z = 0$ (lower) and $z = 1$ (upper). Black dots represent 1's and white dots represent 0's.

Theorem 4. *Let $S = S_{\mathcal{O}}S_{\mathcal{E}}$ and let $S_{\mathcal{O}}$ be an odd-monotone string and $S_{\mathcal{E}}$ be an even-monotone string such that $\mathcal{O}[S_{\mathcal{O}}] = \mathcal{E}[S_{\mathcal{E}}]$ and $\mathcal{E}[S_{\mathcal{O}}] = \mathcal{O}[S_{\mathcal{E}}]$. Then there is a linear time algorithm that folds these two strings achieving $\frac{3}{4}OPT - 16\delta(S) - O(1)$ contacts.*

The main idea behind the proof of Theorem 4 is that we partition the elements in $S_{\mathcal{O}}$ and $S_{\mathcal{E}}$ into *main-diagonal elements* and *off-diagonal elements*. We then use the DIAGONAL FOLDING ALGORITHM to fold the main-diagonal elements along the direction $x = y$ and the off-diagonal elements into branches along the direction $x = -y$ (see Figure 2). All 1's will receive 3 contacts except for a constant number of 1's for each off-diagonal branch, which correspond to switches in the strings $S_{\mathcal{O}}$ and $S_{\mathcal{E}}$, and a constant number at the ends of the main diagonal. This yields the claimed number of $\frac{3}{4}OPT - O(\delta(S)) - O(1)$ contacts.

To precisely define *main-diagonal* and *off-diagonal* elements, we use additional notation. We use 0^k and 1^k (for some integer $k \geq 0$) to refer to the strings consisting of k 0's and k 1's, respectively. By writing $S = E^k$ for some integer k, we mean that S is of the form $S = 0^{2i_0+1}10^{2i_1+1}10^{2i_2+1}10^{2i_3+1}\ldots 0^{2i_{k-1}+1}10^{i_k}$ for integers $i_j \geq 0$, and all the 1's in S are even-1's. Likewise, we write $S = O^k$ to refer to a string of the same form where all 1's are odd-1's, i.e. $S = 10^{2i_1+1}10^{2i_2+1}\ 10^{2i_3+1}\ldots 0^{2i_{k-1}+1}10^{i_k}$. So we can express any string $S_{\mathcal{E}}$ as $S_{\mathcal{E}} =$

$E^{a_1}O^{b_1}E^{a_2}O^{b_2}\dots E^{a_k}O^{b_k}$ for $k = \delta(S_\mathcal{E})$ and integers a_i and b_i. If $S_\mathcal{E}$ is even-monotone, then $a_i \geq b_i$ for all i. We can express any string $S_\mathcal{O}$ as $S_\mathcal{O} = O^{c_1}E^{d_1}O^{c_2}E^{d_2}\dots O^{c_\ell}E^{d_\ell}$ for $\ell = \delta(S_\mathcal{O})$ and integers c_i and d_i. If $S_\mathcal{O}$ is even-monotone, then $c_i \geq d_i$ for all i.

Definition 5. *For an odd-monotone string $S_\mathcal{O} = O^{c_1}E^{d_1}O^{c_2}E^{d_2}\dots O^{c_\ell}E^{d_\ell}$, the first set of $c_i - d_i$ odd-1's in each block, i.e. the elements $O^{c_1-d_1}O^{c_2-d_2}\dots O^{c_\ell-d_\ell}$, are the* main-diagonal elements *and the remaining elements $O^{d_1}E^{d_1}O^{d_2}E^{d_2}\dots O^{d_\ell}E^{d_\ell}$ are the* off-diagonal elements *in $S_\mathcal{O}$.*

For even-monotone strings, we define main-diagonal and off-diagonal elements analogously. In our modified algorithm, it will be useful to have $S_\mathcal{E}$ and $S_\mathcal{O}$ in a special form. Two sets of off-diagonal elements in $S_\mathcal{O}$, $O^{d_i}E^{d_i}$ and $O^{d_{i+1}}E^{d_{i+1}}$, are separated by $c_{i+1} - d_{i+1}$ odd-1's that are main-diagonal elements. We want them to be separated by a number of main-diagonal elements that is a multiple of 8. This will guarantee that the off-diagonals used to fold the off-diagonal elements are regularly spaced so that none of the off-diagonal folds interfere with each other. We will use the following simple lemma.

Lemma 6. *For any odd-monotone string $S_\mathcal{O}$ it is possible to change at most $8\delta(S_\mathcal{O})$ 1's to 0's so that the resulting string S' is of the form $S' = O^{a_1}E^{b_1}O^{a_2}E^{b_2}\dots O^{a_k}$, where $a_i - b_i$ is a positive multiple of 8 for $1 \leq i < k$.*

We note that there is an analogous version of Lemma 6 for even-monotone strings. With this preparation, we can now state our folding algorithm.

Off-Diagonal Folding Algorithm

Input: A binary string $S = S_\mathcal{O}S_\mathcal{E}$, such that $S_\mathcal{O}$ is odd-monotone, $S_\mathcal{E}$ is even-monotone, $\mathcal{O}[S_\mathcal{O}] = \mathcal{E}[S_\mathcal{E}]$ and $\mathcal{E}[S_\mathcal{O}] = \mathcal{O}[S_\mathcal{E}]$.
Output: A folding of the string S.

1. Change at most $8\delta(S)$ 1's to 0's in $S_\mathcal{O}$ and $S_\mathcal{E}$ to yield the form specified in Lemma 6.
2. Run Diagonal Folding Algorithm on *main-diagonal* elements along the direction $x = y$ and change from plane $z = 0$ to $z = 1$ when the length of the main diagonal equals $4 \cdot \lfloor \mathcal{O}[S_\mathcal{O}]/8 \rfloor + 2$.
3. Run Diagonal Folding Algorithm on the *off-diagonal* elements along the direction $x = -y$. The *off-diagonal* elements attached to the *main-diagonal* elements on the plane $z = 1$ are folded along the diagonals $x = -y + 8k$. The *off-diagonal* elements attached to the *main-diagonal* elements on the plane $z = 0$ are folded along the diagonals $x = -y + 8k + 4$. (See Figure 2.)

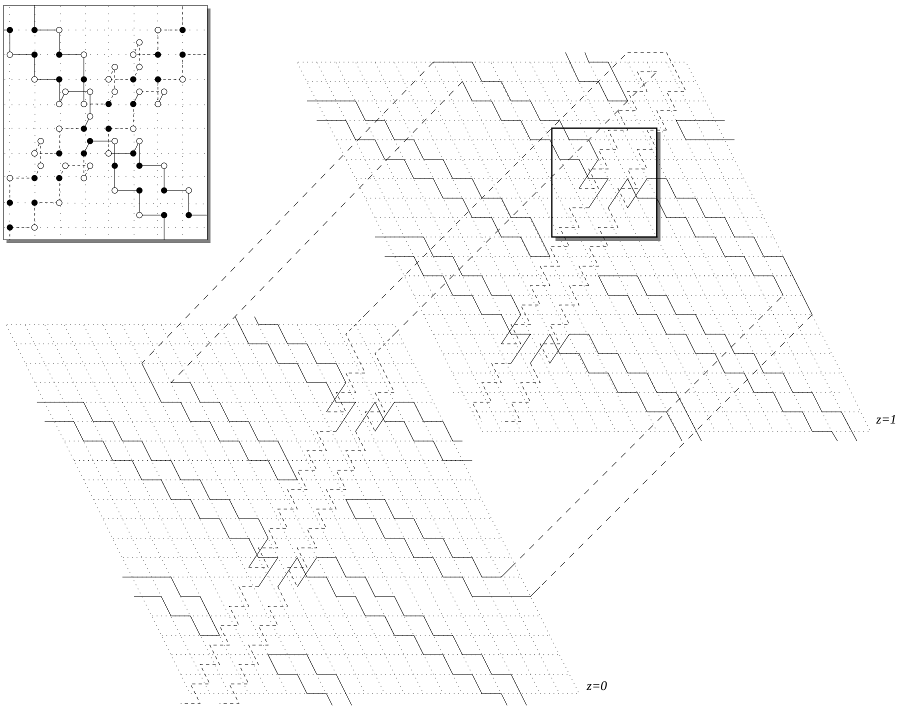

Fig. 2. Folding the *off-diagonal* elements in Step 3 of the OFF-DIAGONAL FOLDING ALGORITHM. The *main-diagonal* elements are represented by the dashed lines on the main diagonal. The *off-diagonal* elements are represents by the solid lines on the off-diagonals. This figure shows how the repetitions of the DIAGONAL FOLDING ALGORITHM on the off-diagonals interleave and thus so not interfere with each other. The closeup gives an example of how the off-diagonal folds are connected to the main diagonal.

3 Combinatorial Problems on Strings

In this section, we present a combinatorial theorem about binary strings that allows us to use the algorithm from Section 2.2 for the general 3D folding problem. The binary strings that we consider in this section are from the set $\{a, b\}^*$. Given a string to fold in $\{0, 1\}^*$, we map it to a corresponding string in $\{a, b\}^*$ by representing each odd-1 by an a and each even-1 by a b. For example, the string 10100101 would be mapped to the string $aabb$. We will use theorems about the strings in $\{a, b\}^*$ to prove theorems about *subsequences* of the strings in $\{0, 1\}^*$ that we want to fold.

The combinatorial problem that we want to solve is the following: given a string $S \in \{0, 1\}^*$ such that $\mathcal{E}[S] = \mathcal{O}[S]$, we want to divide the string into two substrings such that one contains an even-monotone subsequence and the other contains an odd-monotone subsequence and the number of 1's contained in these monotone subsequences is as large as possible, since the 1's in these subsequences are the 1's that will have contacts in the Off-Diagonal Folding Algorithm.

Given a string $S \in \{0,1\}^*$, we will treat it as a loop $L(S)$ by attaching its endpoints. In other words, we are only going to consider foldings of the string that place the first and last element of S on adjacent lattice points. (If S has odd length, we can add a 0 to the end of the string and fold this string instead of S; a folding of this augmented string will yield a valid folding of the original string.)

Lemma 7. *Let $L(S) \in \{0,1\}^*$ be a loop, and $k = \min\{\mathcal{O}[S], \mathcal{E}[S]\}$. Then it is possible to change some 1's of $L(S)$ to 0's such that there is a partition $L(S) = S_{\mathcal{O}}S_{\mathcal{E}}$ with $S_{\mathcal{O}}$ and $S_{\mathcal{E}}$ odd- and even-monotone, respectively, $\mathcal{O}[S_{\mathcal{O}}] = \mathcal{E}[S_{\mathcal{E}}]$, $\mathcal{E}[S_{\mathcal{O}}] = \mathcal{O}[S_{\mathcal{E}}]$, and $\mathcal{O}[S_{\mathcal{O}}] + \mathcal{O}[S_{\mathcal{E}}] \geq (2-\sqrt{2})k$. Furthermore, this partition can be constructed in linear time.*

To prove this lemma, we first apply Lemma 2.2 from [Ala02] to cut the string into two substrings and then apply Theorem 13 to each substring. Lemma 7 implies that every 3D folding instance can be converted into the case required by Theorem 4 by converting not too many 1's into 0's. We obtain the following corollary of Lemma 7 and Theorem 4.

Corollary 8. *There is a linear time algorithm for the 3D folding problem that generates at least $.439 \cdot OPT - 16\delta(S) - O(1)$ contacts.*

3.1 Block-Monotone Subsequences

Let S be a binary string, $S \in \{a,b\}^n$. We will use the following definitions.

Definition 9. *Let $n_a(S)$ and $n_b(S)$ denote the number of a's and b's, respectively, in a string S.*

Definition 10. *A* block *is a maximal substring of consecutive a's or b's in a binary string.*

Definition 11. *A binary string is* block-monotone *if every block of a's is immediately followed by a block of at least as many b's.*

For example, the string *bbbbaaabb* has two blocks of b's (of length four and two) and one block of a's (of length three). An example of a block-monotone string is *baaabbbaaabbbb*. The string *aabbaaabb* is not block-monotone.

Given a binary string S, our goal is to find a long block-monotone subsequence. It is easy to see that S contains a block-monotone subsequence of length at least $n_b(S)$ since the subsequence of b's is trivially block-monotone. It is also easy to see that there are strings for which we cannot do better than this. For example, consider the string $b^i a^i$. In this string, there is no block monotone subsequence that contains any of the a's. Thus, we will put a stronger condition on the binary strings in which we want to find long block-monotone subsequences.

Notation. $\alpha := 1 - \frac{1}{\sqrt{2}} \approx 0.2929$

Definition 12. *A binary string $S = s_1 \dots s_n$ is* suffix-monotone *if for every suffix $\overline{S}_k = s_{k+1} \dots s_n$, $0 \le k < n$, we have $n_b(\overline{S}_k) \ge \alpha \cdot (n-k)$.*

For example if every suffix of S has at least as many b's as a's, the string is suffix-monotone. We will give an algorithm to prove the following theorem.

Theorem 13. *Suppose S is a suffix-monotone string of length n. Then there is a block-monotone subsequence of S with length at least $n - n_a(S)(2\sqrt{2}-2)$. Furthermore, such a subsequence can be found in linear time.*

If $n_a(S) \le \frac{1}{2}n$ and S is suffix-monotone, then Theorem 13 states that we can find a block-monotone subsequence of length at least $(2-\sqrt{2}) > .5857$ the length of S. This is accomplished by the following algorithm.

Block-Monotone Algorithm

Input: a suffix-monotone string $S = s_1 \dots s_n$
Output: a block-monotone subsequence of S
Let $S_i = s_1 \dots s_i$, $\overline{S}_i = s_{i+1} \dots s_n$ for $i: \ 1 < i \le n$
1. If $s_1 = b$:
   (i) Find the largest index k such that S_k is a block of b's and output S_k
2. If $s_1 = a$:
   (i) Find the smallest index k such that:
      $n_b(S_k) \ge \alpha k$
   (ii) Let $S'_\ell = s_{\ell+1} \dots s_k$ for $\ell: \ 1 \le \ell < k$
   (iii) Find ℓ such that:
      $n_a(S_\ell) \le n_b(S'_\ell)$
      $n_a(S_\ell) + n_b(S'_\ell)$ is maximized
   (iv) Remove all the b's from S_ℓ and output S_ℓ
   (v) Remove all the a's from S'_ℓ and output S'_ℓ
3. Repeat algorithm on string $\overline{S}_k$

4 Conclusion

We conclude by stating an approximation guarantee independent of $\delta(S)$. In the full version of this paper, we give a case-based algorithm whose approximation guarantee is $\frac{3}{8}OPT + O(\delta(S))$. This algorithm is based on the following idea: Suppose $S_{\mathcal{O}}$ and $S_{\mathcal{E}}$ contain half the odd-1's and half the even-1's, respectively. We use the Diagonal Folding Algorithm, but for each switch in $S_{\mathcal{O}}$, we use different local foldings to obtain an additional (constant) number of contacts, e.g. we use an even-1 in the switch to obtain another contact with an odd-1 placed on the main diagonal. The performance of this algorithm is summarized in Lemma 14, which in combination with Corollary 8 yields Lemma 15.

Lemma 14. *We can modify the* DIAGONAL FOLDING ALGORITHM *to create a folding with* $\frac{3}{8}OPT + \frac{\delta(S)}{256} - O(1)$ *contacts for any binary string* S*.*

Lemma 15. *There is a linear time algorithm for the 3D folding problem that creates a folding with* $.37501 \cdot OPT - O(1)$ *contacts for any binary string* S*.*

We have described an algorithm for protein folding in the HP model on the 3D square lattice that slightly improves on the previously best-known algorithm to yield an approximation guarantee of .37501. The contribution of this paper is not so much the actual gain in the approximation ratio, but the demonstration that the previously best-known algorithm is not optimal, even though there have been no improvements for almost a decade.

In closing, we discuss the problem of finding block-monotone subsequences of binary strings. One way to improve the approximation ratio of our algorithm is to improve the guarantee given by Theorem 13. We note that we only apply Theorem 13 to binary strings in which every suffix contains at least as many b's as a's—a stronger condition than our definition of block-monotone. Theorem 13 implies that such strings contain block-monotone subsequences of at least .5857 their length. We conjecture that the real lower bound is actually $\frac{2}{3}$ their length. Currently, the best upper bound we are aware of is the string:

$$aaaaabaaaabaaabaababbbaaabaaabababaababbbbbbbbbbbbb$$

whose longest block-monotone subsequence is $a^{18}b^{19}$, which is $\frac{37}{52} \approx 71.15\%$ of the length of the original string.

Acknowledgments. We thank Santosh Vempala for many helpful discussions and suggestions and comments on the presentation. We thank Edith Newman for drawing Figures 1 and 2.

References

[Ala02] Alantha Newman. A New Algorithm for Protein Folding in the HP Model. In *Proceedings of the 13th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2002.

[BL98] Bonnie Berger and Tom Leighton. Protein Folding in the Hydrophobic-Hydrophilic (HP) Model is NP-Complete. In *Proceedings of the 2nd Conference on Computational Molecular Biology (RECOMB)*, 1998.

[CGP+98] P. Crescenzi, D. Goldman, C. Papadimitriou, A. Piccolboni, and M. Yannakakis. On the Complexity of Protein Folding. In *Proceedings of the 2nd Conference on Computational Molecular Biology (RECOMB)*, 1998.

[Dil85] K. A. Dill. Theory for the Folding and Stability of Globular Proteins. *Biochemistry*, 24:1501, 1985.

[Dil90] K. A. Dill. Dominant Forces in Protein Folding. *Biochemistry*, 29:7133–7155, 1990.

[HI95] William E. Hart and Sorin Istrail. Fast Protein Folding in the Hydrophobic-hydrophilic Model within Three-eighths of Optimal. In *Proceedings of the 27th ACM Symposium on the Theory of Computing (STOC)*, 1995.