Combinatorial Problems on Strings with Applications to Protein Folding

Alantha Newman
MIT Laboratory for Computer Science
Cambridge, MA 02139, USA
alantha@theory.lcs.mit.edu

Matthias Ruhl
IBM Almaden
San Jose, CA 95120, USA
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 3D square 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 $\frac{3}{8} = .375$ and was given by Hart and Istrail [HI95] almost a decade ago.

In this paper, we first present another $\frac{3}{8}$-approximation algorithm for the 3D folding problem based on different geometric ideas. This algorithm improves on the absolute approximation guarantee of Hart and Istrail's 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. Building upon this result, we obtain a $(.439 - O(\delta(S)/|S|))$-approximation algorithm, where $\delta(S)$ is the number of transitions in the input string S from sequences of 1's in odd positions to sequences of 1's in even positions. Combining this with an $(.375 + \Omega(\delta(S)/|S|))$-approximation algorithm, 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 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 3D lattice and the number of adjacent pairs of 1's is maximized. Figure 1 shows an example of such 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 maximum number of H-H contacts (i.e. pairs of H's that are adjacent in the folding but not in the string) are present. 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].

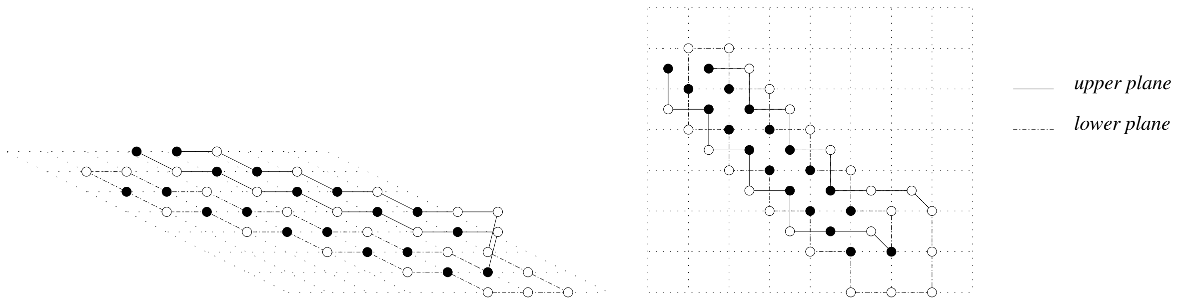

Figure 1: Two views of a folding for the string 10101010000000101010101001010101010101010101, where white circles represent 0's and black circles represent 1's. This folding yields 26 contacts and occupies two *x*-*y* planes in the 3D lattice.

Related Work. Berger and Leighton showed that this problem is NP-hard [BL98]. On the positive side, Hart and Istrail gave a simple $\frac{3}{8}$-approximation algorithm [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]. Improving on the approximation guarantee of $\frac{3}{8}$ for the 3D problem has been an open problem for almost a decade.

Our Contribution. 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. This improves on the absolute approximation guarantee of $\frac{3}{8}OPT - \Theta(\sqrt{OPT})$ given by Hart and Istrail's algorithm [HI95].

We then show that if the input string is of a certain special form, we can modify our algorithm to produce $\frac{3}{4}OPT - O(\delta(S))$ 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 Sections 2.2 and 2.3.

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 3.2, 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, but tedious, case-based algorithm producing $.375 \cdot OPT + \Omega(\delta(S))$ contacts that is described in Appendix B. We therefore 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.

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:

$$OPT \;\leq\; 4\min\{\mathcal{O}[S], \mathcal{E}[S]\} + 2. \tag{1}$$

This upper bound was introduced by Hart and Istrail [HI95]. Their algorithm for the 3D folding problem produces a folding with $\frac{3}{8}OPT - \Theta(\sqrt{OPT})$ contacts in the worst case. We will 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.

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 $z = 1$.

2.1 The Diagonal Folding Algorithm

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 $z > 1$ 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 $z < 0$ to place the strings of 0's between consecutive 1's in $S_{\mathcal{O}}$ or $S_{\mathcal{E}}$.

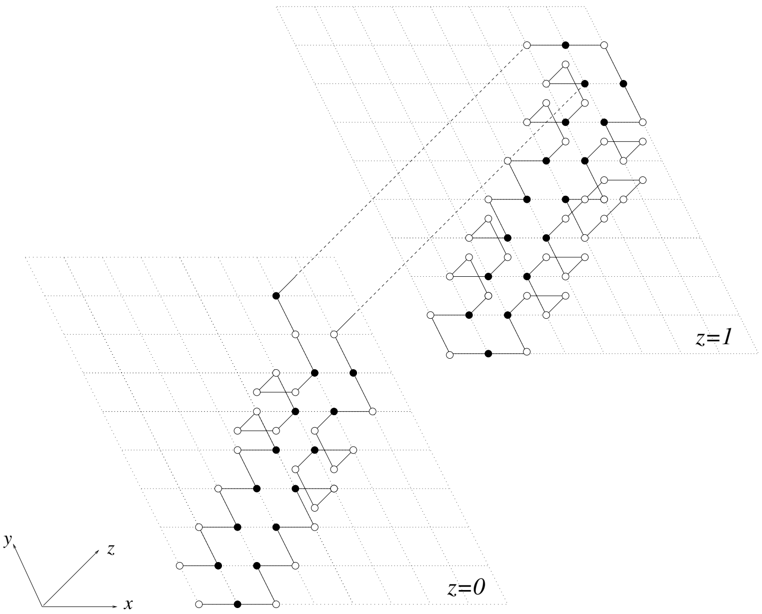

Figure 2: This figure illustrates Steps 2 and 3 of the DIAGONAL FOLDING ALGORITHM. In the folding resulting from this algorithm, all contacts are formed on or between the 2D planes $z = 0$ (lower) and $z = 1$ (upper).

Lemma 1. *The* DIAGONAL FOLDING ALGORITHM *produces a folding with at least* $\frac{3}{8}OPT - 9$ *contacts.*

Proof: Without loss of generality, we assume that $k = \mathcal{O}[S]$. Consider the i^{th} odd-1 from the first half of $S_{\mathcal{O}}$. It is placed on lattice point $(i,i,1)$. In Step 2, this odd-1 forms contacts with the even-1's on the lattice points $(i,i+1,1)$ and $(i-1,i,1)$. In Step 3, it forms a contact with the lattice point $(i,i,0)$. Thus, each odd-1 from the first half of $S_{\mathcal{O}}$ has three contacts. Now consider an odd-1 with an index $k/4+i$, where i ranges from 3 and $\frac{k}{4}$. Each such odd-1 is placed on lattice point $(k/4-i+1, k/4-i+2, 0)$. In Step 3, it forms contacts with even-1's on the lattice points $(k/4-i+1, k/4-i+1, 0)$ and $(k/4-i+2, k/4-i+2, 0)$. In Step 2, it forms a contact with the even-1 on lattice point $(k/4-i+1, k/4+i+2, 1)$. Thus, it also has 3 contacts. By (1), we see that an upper bound on the number of contacts is $OPT \leq 4\mathcal{O}[S] = 4k+2$. We obtain 3 contacts for $\frac{k}{2}-3$ of the odd-1's. Thus, the number of contacts in the resulting folding is at least $\frac{3}{8}OPT - 9$. □

2.2 Relating Folding to String Properties

As the number of 1's placed on the diagonal in the DIAGONAL FOLDING ALGORITHM (i.e. $\frac{1}{2}\min\{\mathcal{O}[S], \mathcal{E}[S]\}$) increases, the length 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 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 string 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 preceded by at least as many consecutive odd-1's. A string $S_{\mathcal{E}}$ is called* even-monotone *if every maximal sequence of* *consecutive odd-1's is preceded by at least as many consecutive even-1's.*

For example, the string 10101100011 is odd-monotone and the string 010001010110110101011 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 $S = 100\underline{1}000101011011\underline{1}01011$, $\delta(S) = 2$ since there are two transitions (underlined) from a maximal sequence of consecutive odd-1's to a sequence of even-1's.

Suppose we can divide a given string S into $S_{\mathcal{O}}$ and $S_{\mathcal{E}}$ such that $S_{\mathcal{O}}$ is odd-monotone and $S_{\mathcal{E}}$ is even-monotone and $\mathcal{O}[S_{\mathcal{O}}] = \mathcal{E}[S_{\mathcal{E}}]$ and $\mathcal{E}[S_{\mathcal{O}}] = \mathcal{O}[S_{\mathcal{E}}]$. Additionally, suppose the number of *switches* in S is $\delta(S)$. Then we have the following theorem:

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 Figures 3 and 4). 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 need some 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 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 $S_{\mathcal{E}} = E^{a_1}O^{b_1}E^{a_2}O^{b_2}\ldots E^{a_k}O^{b_k}$ 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 $S_{\mathcal{O}} = O^{c_1}E^{d_1}O^{c_2}E^{d_2}\ldots O^{c_\ell}E^{d_\ell}$ 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 even-monotone string $S_{\mathcal{E}} = E^{a_1}O^{b_1}E^{a_2}O^{b_2}\ldots E^{a_k}O^{b_k}$, the first set of $a_i - b_i$ even-1's in each block, i.e. the elements $E^{a_1-b_1}E^{a_2-b_2}\ldots E^{a_k-b_k}$, are the* main-diagonal elements *and the remaining elements $E^{b_1}O^{b_1}E^{b_2}O^{b_2}\ldots E^{b_k}O^{b_k}$ are the* off-diagonal elements *in $S_{\mathcal{E}}$.*

Definition 6. *For an odd-monotone string $S_{\mathcal{O}} = O^{c_1}E^{d_1}O^{c_2}E^{d_2}\ldots O^{c_\ell}E^{d_\ell}$, the first set of $c_i - d_i$ odd-1's in each block, i.e. the elements $O^{c_1-d_1}O^{c_2-d_2}\ldots O^{c_\ell-d_\ell}$, are the* main-diagonal elements *and the remaining elements $O^{d_1}E^{d_1}O^{d_2}E^{d_2}\ldots O^{d_\ell}E^{d_\ell}$ are the* off-diagonal elements *in $S_{\mathcal{O}}$.*

For the 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 7. *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 $1 \le i < k$.

Proof: Suppose that $S_{\mathcal{O}}$ initially is of the form

$$S_{\mathcal{O}} = O^{\alpha_1} E^{\beta_1} O^{\alpha_2} E^{\beta_2} \dots O^{\alpha_\ell}.$$

First, we convert all E^{β_i} with $\beta_i \le 8$ into 0's. This will merge some maximal sequences of odd-1's, yielding a string of the form

$$O^{a_1} E^{\gamma_1} O^{a_2} E^{\gamma_2} \dots O^{a_k}$$

with $k \le \ell$. For each i, we then convert $(\gamma_i - a_i) \bmod 8$ even-1's of E^{γ_i} into 0's, yielding a string of the desired form. □

We note that there is an analogous version of Lemma 7 for even-monotone strings. With this preparation, we can now state our folding algorithm.

2.3 A Modified Diagonal 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 7.
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 $4 \cdot \lfloor \mathcal{O}[S_{\mathcal{O}}]/8 \rfloor + 2$. See Figure 3.
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 folding along the diagonals $x = -y + 8k + 4$. See Figure 4.

Proof of Theorem 4: By the correctness of the DIAGONAL FOLDING ALGORITHM, it suffices to consider whether some off-diagonals intersect each other. The first step of the algorithm ensures that all off-diagonal branches are spread apart by multiples of 8 on the main-diagonal. Thus, neighboring branches do not intersect. Furthermore, branches off the upper ($z = 1$) plane do not intersect with branches off the lower ($z = 0$) place due to Step 3. Changing the plane when the main diagonal has a length $\equiv 2 \mod 4$ ensures that branches on the upper plane will follow diagonals $x = -y + 8k$ for some k, and branches on the lower plane follow diagonals $x = -y + 8k + 4$ for some k. Thus, branches are at least 4 lattice points apart, showing that the folding is non-intersecting.

It remains to analyze the number of contacts produced by the folding. The DIAGONAL FOLDING ALGORITHM generally produces 3 contacts for every 1. So it suffices to bound the number of 1's in S that do not receive 3 contacts. The following is an exhaustive list: (i) the up to $8\delta(S)$ 1's changed into 0's in Step 1; (ii) a constant number of 1's at the ends of the main-diagonal (cf. Lemma 1) and because we fold over at a length $\equiv 2 \mod 4$ in Step 2; (iii) in Step 3, for each of the at most $\delta(S)$ off-diagonal branches: at most 3 1's at the end of each branch (by Lemma 1), and at most 5 1's to connect the off-diagonal branch to the main-diagonal (see Figure 4). So in summary, up to $16\delta(S) + O(1)$ 1's might not receive three contacts, so that we obtain $3\mathcal{O}[S] - 16\delta(S) - O(1) \geq \frac{3}{4}OPT - 16\delta(S) - O(1)$ contacts. □

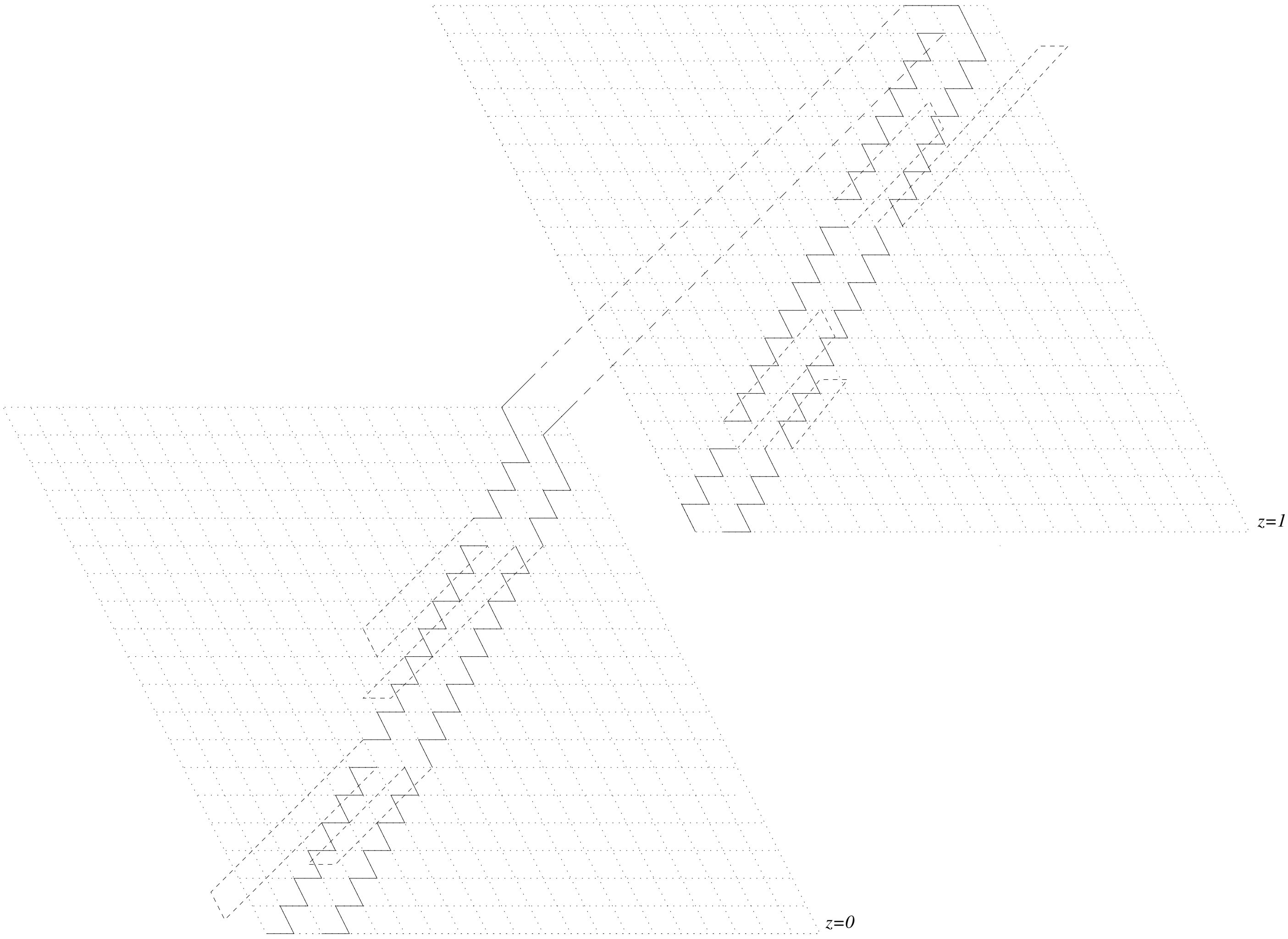

Figure 3: Folding the *main-diagonal* elements in Step 2 of the OFF-DIAGONAL FOLDING ALGORITHM. The solid lines represent the *main-diagonal* elements and the dashed lines represent the *off-diagonal* elements.

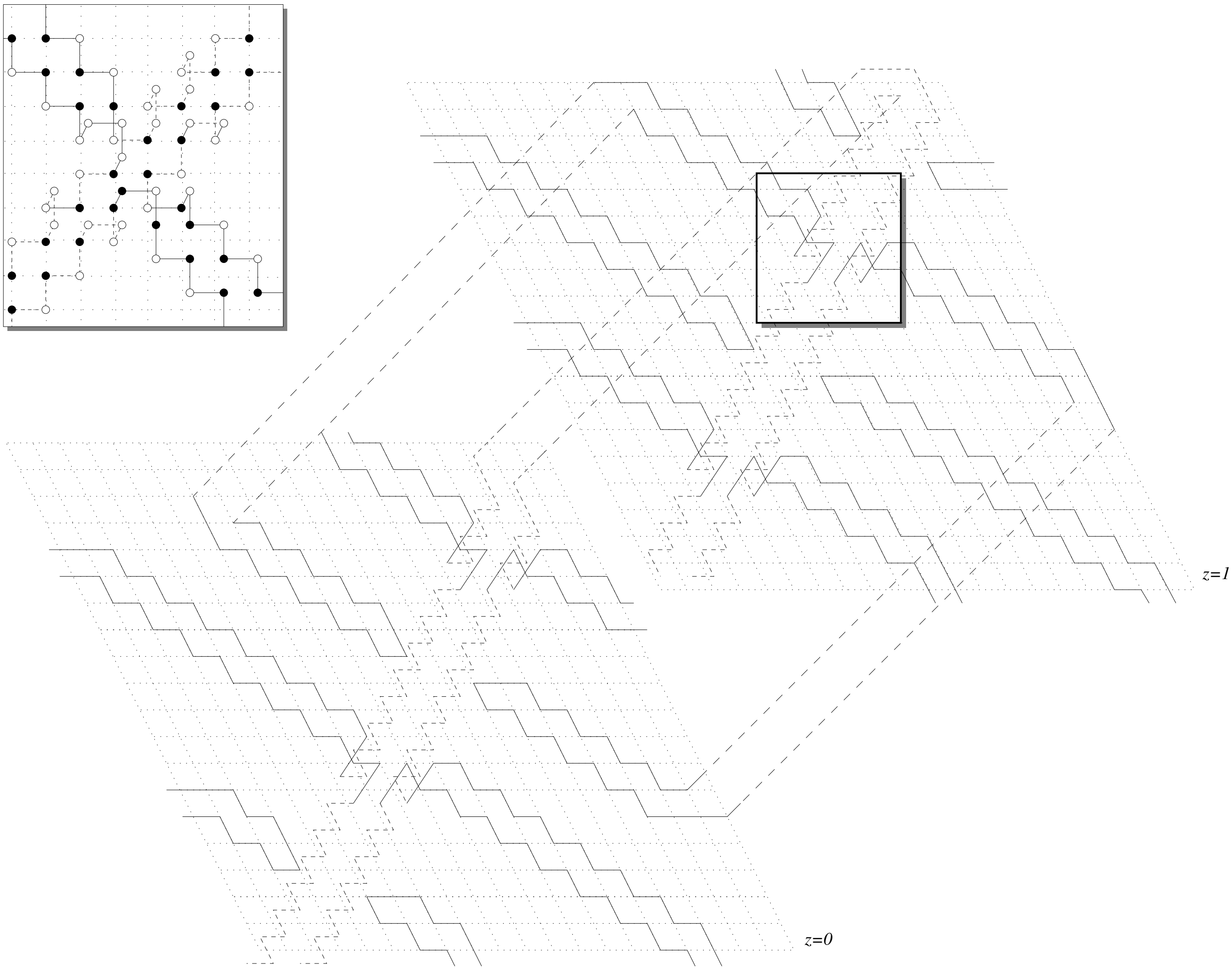

Figure 4: Folding the *off-diagonal* elements in Step 3 of the OFF-DIAGONAL FOLDING ALGORITHM. The *main-diagonal* elements are represented by the dashed lines on the main diagonal. The *off-diagonal* elements are represents by the solid lines on the off-diagonals. This figure shows how the repetitions 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

3.1 Solving the General Folding Problem

In this section, we will prove a combinatorial theorem about binary strings, which will allow us to use the algorithm from Section 2.3 to solve the general 3D string 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 folding algorithm in Section 2.3.

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 8. *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.*

This Lemma implies that every 3D folding instance can be converted into the case required by Theorem 4 by converting not too many some 1's into 0's. We get the following Corollary.

Corollary 9. *There is a linear time algorithm for the 3D folding problem that generates at least $.439 \cdot OPT - 16\delta(S) - O(1)$ contacts.*

Proof: Given an input string S, first obtain $S_{\mathcal{O}}$ and $S_{\mathcal{E}}$ with Lemma 8. Note that the number of switches does not increase from S to $S_{\mathcal{O}}S_{\mathcal{E}}$. Since the number of 1's is reduced by a factor of $(2-\sqrt{2})$, the optimal number of contacts might also have been decreased by that factor. Applying Theorem 4 to $S_{\mathcal{O}}$ and $S_{\mathcal{E}}$ therefore leads to a folding with at least $\frac{3}{4}(2-\sqrt{2})OPT - 16\delta(S) - O(1) > .439 \cdot OPT - 16\delta(S) - O(1)$ contacts. □

Proof (Lemma 8): We can generate $S_{\mathcal{O}}$ and $S_{\mathcal{E}}$ by cutting $L(S)$ in two places. First, we will use Lemma 2.2 from [Ala02]. This lemma states that given a loop $L(S)$, there is an element p in $L(S)$ such that if we start at point p and move around the loop in the clockwise direction, we see at least as many odd-1's as even-1's and if we move around the loop in the counter-clockwise direction starting at point p, we see at least as many even-1's as odd-1's.

We choose such a point p to be the first point where we cut the loop $L(S)$. We choose the second point simply by ensuring that both resulting substrings contain the same number of 1's. Now we have two substrings $S_{\mathcal{O}}$ and $S_{\mathcal{E}}$. The substring $S_{\mathcal{O}}$ has the property that every suffix (or prefix–depending on how you view the string) has at least as many odd-1's as even-1's and $S_{\mathcal{E}}$ has the property that every suffix has at least as many even-1's as odd-1's.

Now we want to change the minimum number of 1's to 0's in $S_{\mathcal{O}}$ and $S_{\mathcal{E}}$ so that the resulting substrings are odd-monotone and even-monotone, respectively, and $\mathcal{O}[S_{\mathcal{O}}] = \mathcal{E}[S_{\mathcal{E}}]$ and $\mathcal{E}[S_{\mathcal{O}}] = \mathcal{O}[S_{\mathcal{E}}]$, since these are the conditions required by Theorem 4. Consider a binary string S' corresponding to the subsequence of 1's in $S_{\mathcal{E}}$ in which each odd-1 is replaced by an a and each even-1 is replaced by a b. The problem of changing the minimum number of 1's to 0's in $S_{\mathcal{E}}$ so that the resulting string is odd-monotone is equivalent to finding the longest *block-monotone* subsequence in the string S'. A subsequence is *block-monotone* if every block of a's is immediately followed by a block of at least as many b's. (For the string $S_{\mathcal{O}}$, we have the same problem stated with a's and b's inverted: we want to find the longest subsequence in which every block of b's is immediately followed by a block of at least as many a's.)

The rest of this section is devoted to solving the following combinatorial problem: Given a binary string in $\{a,b\}^*$ in which every suffix contains at least as many b's as a's, what is the longest block-monotone subsequence? For example, suppose we are given the string *aaaaaabbbbabbbab*, some block-monotone subsequences are: *aaaabbbbabbbab* and *aaaaaabbbbbbbab*. If the number of a's and b's are equal, it is clear that we can always find a block-monotone subsequence containing at least half the elements by just choosing the subsequence of all b's. Can we always find a block-monotone subsequence of more than half the elements? We will prove in Theorem 15 that we can always find a block-monotone subsequence of with at least $(2-\sqrt{2})n$ elements where n is the number of elements in the input string and the input string contains an equal number of a's and b's.

By Lemma 17, we can furthermore choose these subsequences such that $\mathcal{O}[S_{\mathcal{O}}] = \mathcal{E}[S_{\mathcal{E}}]$ and $\mathcal{E}[S_{\mathcal{O}}] = \mathcal{O}[S_{\mathcal{E}}]$ after the transformation. This completes the proof of the Lemma. $\square$

3.2 Block-Monotone Subsequences

In this section, we will prove a combinatorial theorem about binary strings. Let S be a binary string, $S \in \{a,b\}^n$. We will use the following definitions.

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

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).

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 *baaabbbaaabbbb* is block-monotone. The string *aabbaaabb* is not block-monotone.

Definition 12. *Let $n_a(S)$ and $n_b(S)$ denote the number of a's and b's, respectively, in a string S.*

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 block-monotone subsequences.

Notation 13. $\alpha := 1 - \frac{1}{\sqrt{2}} \approx 0.2929$

Definition 14. *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. (If in addition, S also has the same number of a's and b's, then S corresponds to a string in the set of well-balanced parentheses.) We will give an algorithm to prove the following theorem.

Theorem 15. *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 15 states that we can find a block-monotone subsequence of length at least $(2-\sqrt{2}) > .5857$ the length of S. Now we will give an algorithm for finding a block-monotone subsequence of a suffix-monotone string.

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 $i:\ 1 < i \leq n$

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:

    $n_b(S_k) \geq \alpha k$

    (ii) Let $S'_\ell = s_{\ell+1} \dots s_k$ for $\ell:\ 1 \leq \ell < k$

    (iii) Find ℓ such that:

    $n_a(S_\ell) \leq n_b(S'_\ell)$

    $n_a(S_\ell) + n_b(S'_\ell)$ 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 $\overline{S}_k$

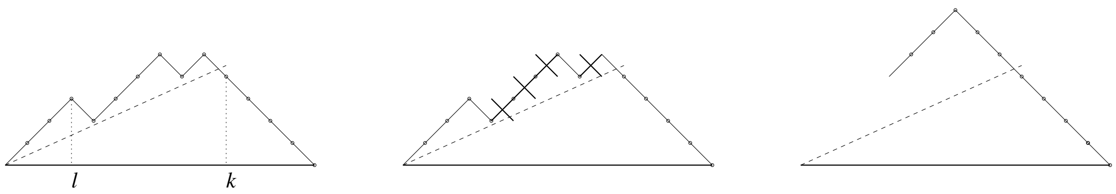

Figure 5: These three figures give a pictorial representation of a binary string S. An up edge corresponds to an a and a down edge corresponds to a b. In the first figure, k denotes the point chosen in Step 2 (i) and ℓ denotes the point chosen in Step 2 (iii). In the second figure, the crossed-out edges represent the elements that are removed from the string. The third figure shows the string after removing the crossed-out elements.

Because of space limitations, we put the proofs of Lemma 16 and Lemma 17 in Appendix A.

Lemma 16. *For a suffix-monotone string S of length n, the* BLOCK MONOTONE ALGORITHM *outputs a block-monotone subsequence of length at least $n - n_a(S)(2\sqrt{2}-2)$.*

Lemma 17. *We can modify the block-monotone subsequence S' output by the* BLOCK-MONOTONE ALGORITHM *so that*

$$n_a(S') = \left\lceil \left(1 - \frac{1}{\sqrt{2}}\right) n_a(S) \right\rceil \quad \textit{and} \quad n_b(S') = \left\lceil n - \left(\frac{3}{\sqrt{2}} - 1\right) n_a(S) \right\rceil.$$

4 Conclusion

We conclude the paper by stating an approximation guarantee independent of $\delta(S)$. 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 different cases for this algorithm are detailed in Appendix B, which contains the proof of following lemma.

Lemma 18. *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.*

Corollary 19. *There is a polynomial time algorithm for the 3D folding problem that creates a folding with* $.37501 \cdot OPT - O(1)$ *contacts for any binary string S.*

Proof: We run the algorithms referred to in Corollary 9 and Lemma 18, outputting the better of the two foldings. Their output guarantees are lowest if they are equal, i.e. $\frac{3}{8}OPT + \frac{\delta(S)}{256} = .439OPT - 16\delta(S)$, which happens for $\delta(S) \approx .04OPT$, yielding an approximation guarantee of slightly more than .3750156. □

So we have obtained 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. We also explore different approaches to this problem, i.e. foldings that mainly exploit properties of the string.

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 15. We note that we only apply Theorem 15 to binary strings in which every suffix contains at least as many b's as a's–a stronger condition that our definition of block-monotone. Theorem 15 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 2, 3, and 4.

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.

A Appendix: Proofs

Proof of Lemma 16: Note that in Step 2 (i), there always is an index k with the required property because the definition of *suffix-monotone* implies it is true for $k = n$. Similarly, $\ell = 1$ satisfies $n_a(S_\ell) = 1 \leq n_b(S'_\ell)$, thus there always is an ℓ of the required form in Step 2 (iii). Finally, the algorithm outputs a block-monotone subsequence because whenever it outputs a subsequence of a's (in Step 2 (iv)), it also outputs at least as many b's (in Step 2 (v)). This shows that the algorithm is correct.

In the algorithm, we modify the input string by removing a's and b's. However, in order to analyze the algorithm, we will first consider a version of the problem in which we can remove a fraction of each a or b. Note that the algorithm can be used for the continuous problem as well as the discrete problem. We will show that in the continuous case, the resulting string has a certain minimum length and then show that in the discrete case, the resulting string has at least this length.

Since we will cut the string fractionally, let us consider each element as a unit-length interval. For example, if $s_i = a$, then s_i is a unit-length segment labeled 'a' and if $s_i = b$, then s_i is a unit-length segment labeled 'b'. Thus, we will view the string S as a string of unit-length a- and b-segments. Suppose $s_1 = 0$ and S_k is a prefix of the input suffix-monotone string S such that $n_b(S_k) \geq \alpha k$ and $n_b(S_j) < \alpha j$ for all $j : 1 \leq j < k$ as in Step 2 (i) of the algorithm.

Let t denote the point in the string at which $n_b(S_t) = \alpha t$. The point t can be viewed as a fractional, rather than integral, index of the string S. Note that there always exists a point t at which $n_b(S_t) = \alpha t$ because the string S is suffix-monotone, which implies that at least an α fraction of S is b's. Note that t may be a non-integer real number between $k-1$ and k and that the string S_t may end with a fractional part of a b.

Let $g(t) = t - \lfloor t \rfloor$. Let y be the point in the string S_t such that $n_a(S_y) = n_b(\overline{S}_y)$. (We define $\overline{S}_y$ as the substring starting at position y up to position t). If we could keep fractional portions of the string, we could keep all the (fractions of) a-intervals in S_y and all the (fractions of) b-intervals in $\overline{S}_y$. Note that at least a $(1-\alpha)$ fraction of the elements in S_y are a's, and at least an α-fraction of the elements in $\overline{S}_y$ are b's. So for the fractional problem, the best place to cut the string is at the point $\ell = \beta t$ where:

$$\beta(1-\alpha) = (1-\beta)\alpha \implies \beta = \alpha$$

Thus, we keep a $2\alpha(1-\alpha)$ fraction of each substring considered in Step 2. Next, we are going to compute the total length of the output of our algorithm. Let T_1 represent the set of substrings (i.e. blocks of b's) that are output unmodified during the first step of the algorithm and let $|T_1|$ represent their total length. Let T_2 represent the set of substrings which are modified during the second step of the algorithm and let $|T_2|$ represent their total length. Let m be the length of the output of the algorithm. Then we have the following equations:

$$\begin{aligned} n &= |T_1| + |T_2| \\ n_a(S) &= (1-\alpha)|T_2| \\ m &= |T_1| + 2\alpha(1-\alpha)|T_2| \end{aligned}$$

Solving these three equations, we find that the total fraction of the string that remains is:

$$m = \left(2\alpha + \frac{1}{\alpha - 1}\right) n_a(S) + n.$$

This expression is maximized for $\alpha = 1 - 1/\sqrt{2}$, which is why we assigned α this value. Substituting, we get:

$$m = n - (2\sqrt{2} - 2) n_a(S). \tag{2}$$

Thus, in the case where we can remove fractions of the a's and b's, the algorithm results in a string whose length is indicated in Equation (2).

In the integral case, we will show that the algorithm results in a string whose length is at least as large as the fraction in Equation (2). If the point y in S_t is in a b-interval, then we can keep the whole b-interval. In other words, in addition to keeping the a's in S_y and the b's in $\overline{S}_y$, we are also keeping the fraction of the b-interval that lies in S_y. This will only make the block of b's from $\overline{S}_y$ longer, which does not violate the block-monotonicity of the output string.

If the point y in S_t is in an a-interval, then note that the (fractional) number of a's in S_y is equal to the (fractional) number of b's in $\overline{S}_y$. We will denote the former quantity by $e+f$ and the latter quantity by $c+d$, where e and c are integers and f and d are fractions less than 1. Note that since $e+f=c+d$, it follows that $e=c$ and $f=d$. Also, note that $d=g(t)$. Thus, $f=g(t)$. In the fractional version of the algorithm, in the next iteration, we would skip over at least $1-g(t)$ b's, applying Step 1 (i). Thus, we keep the whole b-interval in which t lies, we are keeping $c+d$ b's from $\overline{S}_y$ as well as the remaining $1-g(t)$ fraction of the last b-interval in $\overline{S}_y$. Thus, if we keep the additional $1-f=1-g(t)$ fraction of the a-interval in which y lies, this does not violate the block-monotonicity of the output string. $\square$

Proof of Lemma 17: Following the notation of the proof of Lemma 16, in the fractional case, we keep $\alpha(1-\alpha)|T_2| = \alpha n_a(S)$ a's and $|T_1|+\alpha(1-\alpha)|T_2| = n - \frac{1-\alpha+\alpha^2}{1-\alpha} n_a(S)$ b's. Since these are lower bounds on what we keep in the integral case, the subsequence output by the algorithm has at least $(1-\frac{1}{\sqrt{2}})n_a(S)$ a's and $n-(\frac{3}{\sqrt{2}}-1)n_a(S)$ b's. To keep exactly the number of symbols claimed in this Lemma, it suffices to delete the excess number of a's and b's. To do this, first delete the excess a's anywhere in the output string, the result will clearly still be block-monotone. Then we delete the excess b's. Note that at this point, the number of b's exceeds the number of a's, so there will always be a block of b's strictly greater than the preceding block of a's and we can delete b's from this block. $\square$

B Appendix: A Case-Based 3D String Folding Algorithm with Approximation Guarantee $3/8 + O(\delta(S))$

In this section, we give a case-based algorithm that has an approximation guarantee of $.375 + O(\delta(S))$. We will analyze this algorithm to conclude with a proof of Lemma 18.

Consider the substrings $S_\mathcal{O}$ and $S_\mathcal{E}$ such that $\mathcal{O}[S_\mathcal{O}] = \mathcal{E}[S_\mathcal{E}]$ and $\mathcal{E}[S_\mathcal{O}] = \mathcal{O}[S_\mathcal{E}]$. (It is shown how to divide S into such substrings in Section 3.) Furthermore, in this section, we can assume that $\mathcal{O}[S_\mathcal{O}] = \mathcal{E}[S_\mathcal{O}]$. If we have $\mathcal{O}[S_\mathcal{O}] > \mathcal{E}[S_\mathcal{O}]$, then the algorithms we describe below will have strictly better approximation ratios than what we prove.

We will consider the following modified version of the string $S_\mathcal{O}$. For every sequence of consecutive even-1's, we turn all but one of them into a 0. For example, we would transform the string 1101011 into 1100001. Slightly abusing notation, we will from now on refer to this modified string as $S_\mathcal{O}$. We will divide the even-1's in $S_\mathcal{O}$ into the following disjoint categories. Suppose each of these categories has $\delta_1 k, \delta_2 k, \delta_3 k$, and $\delta_4 k$ even-1's respectively, where $k = \mathcal{O}[S]$. Without loss of generality, we assume that $\delta_1+\delta_2+\delta_3+\delta_4 \geq \delta/2$, i.e. half the switches occur in $S_\mathcal{O}$.

Each even-1 in $S_{\mathcal{O}}$ falls in exactly one of the following categories:

1. Even-1's in blocks of 1's of length at least 10 or in a block of 1's of length 9 that begins with an odd-1.
2. Even-1's in blocks of 1's of length at least 2 and at most 9 that begin or end with an even-1.
3. Even-1's in blocks of length 1.
4. Even-1's in blocks of length at least 3 and at most 7 that begin and end with an odd-1.

For each of the four cases above, we will show how to slightly modify the DIAGONAL FOLDING ALGORITHM so that it gives an approximation guarantee of $\frac{3}{8} + c_i\delta_i$ for some constant c_i. In the DIAGONAL FOLDING ALGORITHM, one way to account for contacts is to attribute $\frac{3}{2}$ of a contact to each odd-1 on the main diagonal and $\frac{3}{2}$ of a contact to each even-1 on the main diagonal. The main idea behind the modifications of the algorithm is to fold the string so that some odd-1's may no longer be on the main diagonal (thus losing $\frac{3}{2}$ contacts per odd-1) but form more than $\frac{3}{2}$ contacts per odd-1 with neighboring even-1's (making use of the switches). In some of the modifications (such as Case 2) we do not actually remove any of the odd-1's from the main diagonal; due to the nature of the switches, we can still get $O(1)$ contacts per switch. We will first prove a lemma that we will use in several of the cases.

Lemma 20. *Suppose we delete (i.e. change 1's to 0's) i odd-1's in $S_{\mathcal{O}}$. Then we can re-divide S into substrings $S_{\mathcal{O}}$ and $S_{\mathcal{E}}$ so that we again have $\mathcal{E}[S_{\mathcal{E}}] = \mathcal{O}[S_{\mathcal{O}}]$. If we run the* DIAGONAL FOLDING ALGORITHM *on these new strings $S_{\mathcal{E}}$ and $S_{\mathcal{O}}$, we will obtain a folding with at least $\frac{3}{2}(\mathcal{O}[S] - i)$ contacts on the main diagonal.*

Proof: In Section 3, we used Lemma 2.2 from [Ala02] to choose s_1 so that $\mathcal{O}[S_i] \geq \mathcal{E}[S_i]$ for all $i = 1,\dots n$, where $S_i = s_1 \dots s_i$. If we define $\widetilde{S_i} := s_n s_{n-1} \dots s_i$, then again by Lemma 2.2 in [Ala02] we have $\mathcal{E}[\widetilde{S_i}] \geq \mathcal{O}[\widetilde{S_i}]$ for all $i = 1,\dots n$. In Lemma 2.2 of [Ala02], we found s_p so that $S_{\mathcal{O}} = s_1 \dots s_p$ and $S_{\mathcal{E}} = s_n \dots s_{p+1}$.

If we remove i odd-1's from $S_{\mathcal{O}}$, then the main diagonal fold of $S_{\mathcal{O}}$ would be much shorter than that of $S_{\mathcal{E}}$. However, if we move $s_p = s_{p+j}$ for some j so that once again $\mathcal{O}[S_{\mathcal{O}}] = \mathcal{E}[S_{\mathcal{E}}]$, then the number of odd-1's in $S_{\mathcal{O}}$ is at least $\frac{\mathcal{O}[S]-i}{2}$. Thus, we obtain at least $\frac{3}{2}(\mathcal{O}[S] - i)$ contacts on the main diagonal. □

Case 1

Lemma 21. *There is a modification of the* DIAGONAL FOLDING ALGORITHM *with approximation guarantee at least $\frac{3}{8} + \frac{\delta_1}{40}$.*

Proof: An even-1 in Case 1 occurs in a block of 1's of length at least 10 or in a block of 1's of length 9 beginning with an odd-1. Suppose we have a block of 11 1's that begins with an odd-1, which will give the worst case approximation ratio. Then we fold this block as in Figure 6 starting at the point labeled a. Note that 3 odd-1's from $S_{\mathcal{O}}$ that would be on the main diagonal in the DIAGONAL FOLDING ALGORITHM are not placed on the main diagonal. Thus, the main diagonal will be shorter – at least $\frac{3\delta_1 k}{5}$ shorter, because for every 5 even-1's in Case 1, we take at least 3 odd-1's off the main diagonal. By Lemma 20 we can then assume that the length of the main diagonal is:

$$\frac{1}{2}\left(\mathcal{O}[S] - \frac{3\delta_1\mathcal{O}[S]}{5}\right)$$

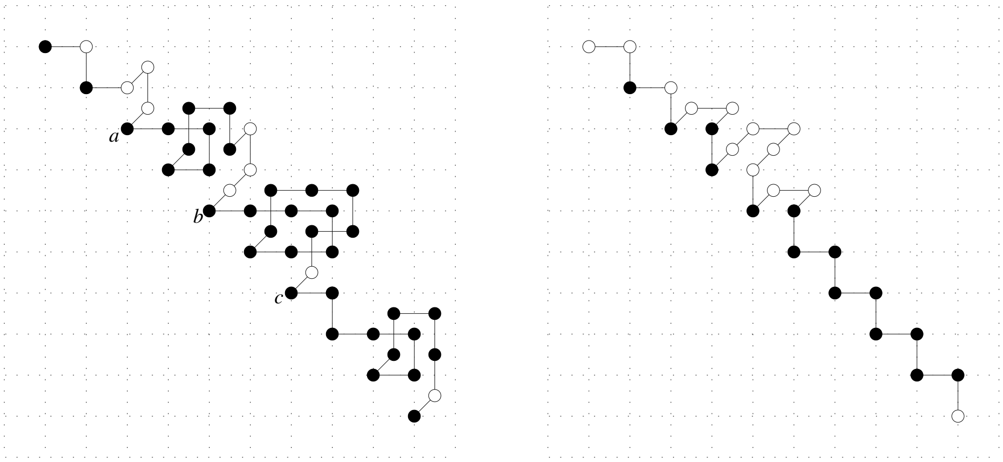

Figure 6: Cases 1 and 2. The first figure shows a folding for even-1's in Case 1. At point a begins the folding for a block of 1's of length 9 that begins with an odd-1. Note that 3 odd-1's are not placed on the main diagonal, but 5 contacts – in addition to those that will be formed on the main diagonal – are obtained. At point b, a block of 1's of length 13 is folded. Here, 5 odd-1's are not placed on the main diagonal, but 8 additional contacts are formed off the main diagonal. At point c, a block of 1's of length 11 is folded. It is basically the same folding as used for blocks of length 9. The second figure shows even-1's in Case 2. For at least half of the blocks of 1's of length at least 2 and at most 9 that begin or end with an even-1, we can get an extra contact by placing an even-1 adjacent to an odd-1 on the main diagonal.

For every odd-1 in $S_{\mathcal{O}}$ on the main diagonal, we obtain 3 contacts. For every 3 odd-1's in $S_{\mathcal{O}}$ off the diagonal (corresponding to 5 even-1's in Case 1), we obtain 5 contacts. Thus, the approximation guarantee is:

$$\left(\frac{3}{2}\left(\mathcal{O}[S]-\frac{3\delta_1\mathcal{O}[S]}{5}\right)+\frac{5\delta_1\mathcal{O}[S]}{5}\right)\frac{1}{4\mathcal{O}[S]} \;=\; \frac{3}{8}-\frac{9\delta_1}{40}+\frac{\delta_1}{4} \;=\; \frac{3}{8}+\frac{\delta_1}{40}$$

□

Case 2

Lemma 22. *There is a modification of the* DIAGONAL FOLDING ALGORITHM *with approximation guarantee at least* $\frac{3}{8}+\frac{\delta_2}{32}$.

Proof: An even-1 in Case 2 is in a block of 1's of length at least 2 and at most 9 that begins or ends with an even-1. In this case, the main diagonal will remain the same length as in the DIAGONAL FOLDING ALGORITHM. We will obtain extra contacts by placing even-1's from $S_{\mathcal{O}}$ next to odd-1's on the main diagonal. This is shown in Figure 6.

For at least half of the blocks (in $S_{\mathcal{O}}$) of 1's of length at least 2 and at most 9 that begin or end with even-1's, we can get an extra contact by placing an even-1 adjacent to an odd-1 on the main diagonal. We may only be able to do this for half of the blocks, because the folding in Figure 6 will work only for an even-1 followed immediately by an odd-1 or an odd-1 followed immediately by an even-1, but does not allow alternating between these two cases. Among these types of blocks, the worst case is a block of 8 1's that begins or ends with an even-1. Such a block uses 4 even-1's from Case 2. If all the Case 2 even-1's fell in this category, we could get an extra contact for half of them, which is one per 8 switches. This ratio is better for block lengths other than 8. In particular, note that a block of length 9 that begins with an even-1 must also end with an even-1, so we always get a contact for one of the two ends of such a block. In summary, we get the following approximation guarantee:

$$\left(\frac{3\mathcal{O}[S]}{2}+\frac{\delta_2\mathcal{O}[S]}{8}\right)\frac{1}{4\mathcal{O}[S]} = \frac{3}{8}+\frac{\delta_2}{32}$$

□

Case 3

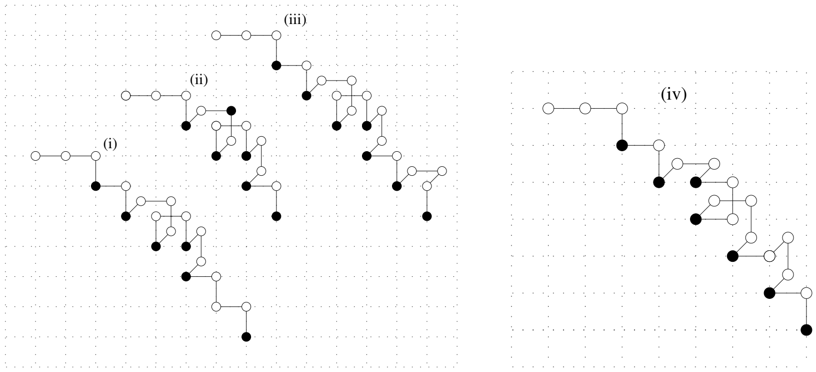

Figure 7: Case 3.

Lemma 23. *There is a modification of the* DIAGONAL FOLDING ALGORITHM *with approximation guarantee at least* $\frac{3}{8}+\frac{\delta_3}{32}$.

Proof: An even-1 in Case 3 is in a block of length 1. Thus, substrings containing such an even-1 look like: $1001001, 100001001$, etc. In other words, an even-1 in Case 3 is in a substring $10^{q_1}10^{q_2}1$ where q_1 and q_2 are both positive even integers. Consider the string $10^i10^{q_1}10^{q_2}10^j1$ where i and j are odd integers, i.e. the first two 1's and last two 1's in the string are odd-1's and the middle 1 is an even-1. (We can assume for now that there is no even-1 between the first two odd-1's or the last two odd-1's because as we will discuss later, if there are two Case 3 even-1's that share an odd-1 as a neighbor, our folding will only use one of these even-1's.) We will use four different modifications of the DIAGONAL FOLDING ALGORITHM based on the values of i and j. We name these types of even-1's as follows: (i) $i \geq 3, j \geq 3$; (ii) $i = 1, j = 1$; (iii) $i \geq 3, j = 1$; (iv) $i = 1, j \geq 3$. See Figure 7 for illustrations of the foldings for each of these types. We now distinguish two cases: first, if more than half of the Case 3 even-1's are of type (i),(ii) or (iii), and second, if more than half are of type (iv).

Suppose that more than half of the Case 3 even-1's are of types (i)-(iii). The foldings for these three types can be used consecutively (as opposed to the folding of (iv), which cannot be applied right after itself). However, we can only guarantee a contact for half of the even-1's in these three types because we may have, for example, $10^i10^{q_1}10^{q_2}1001001$, i.e. 2 even-1's that are both adjacent to the same odd-1. In this case, we can only get an extra contact for one such even-1.

We note that the approximation guarantee obtained is a linear combination of the approximation guarantees for the three types, weighted by their relative frequency. The worst case therefore occurs if half the of Case 3 even-1's are of a single type, (i),(ii) or (iii). Since they change the length of the main diagonal, types (i) and (ii) are worse than (iii).

Since types (i) and (ii) either remove an odd-1 from the main diagonal (type (ii)) or result in some even-1's from $S_{\mathcal{E}}$ not having contacts on the main diagonal (type (i)), they are worse than type (iii). Both of these types have the same approximation guarantee. We will just analyze the case when half the Case 3 even-1's are type (i). The folding modification for this type changes the length of the main diagonal to at least:

$$\frac{1}{2}\left(\mathcal{O}[S]+\frac{\delta_3\mathcal{O}[S]}{4}\right)$$

This is because we assumed that at least half of the Case 3 even-1's are of types (i)-(iii) and we can use half of these even-1's. For each even-1 in Case 3, we lose 1 odd-1 on the main diagonal and we gain 2 contacts per even-1 off the main diagonal. Therefore, the approximation guarantee is:

$$\left(3\left(\frac{1}{2}\left(\mathcal{O}[S]+\frac{\delta_3\mathcal{O}[S]}{4}\right)-\frac{\delta_3\mathcal{O}[S]}{4}\right)+\frac{2\delta_3\mathcal{O}[S]}{4}\right)\frac{1}{4\mathcal{O}[S]} \;=\; \frac{3}{8}+\delta_3\left(\frac{3}{8}-\frac{3}{4}+\frac{1}{2}\right)\frac{1}{4\mathcal{O}[S]}=\frac{3}{8}+\frac{\delta_3}{32} \tag{3}$$

In the other case, when more than half of Case 3 even-1's are of type (iv), per type (iv) even-1 we obtain 2 contacts and one odd-1 is not used on the main diagonal. Therefore, in this case the approximation guarantee is better than that in (3). □

Case 4

Lemma 24. *There is a modification of the* DIAGONAL FOLDING ALGORITHM *with approximation guarantee at least* $\frac{3}{8}+\frac{\delta_4}{24}$.

Proof: In Case 4, even-1's occur in blocks of length at least 3 and at most 7 that begin and end with an odd-1. Consider all the odd-1's that occur in blocks of length at least 3 and at most 7 and that begin and end with an odd-1. Note that the number of such odd-1's is at least $\frac{4\delta_4}{3}$ since the ratio of odd-1's to even-1's in this case is at least 4 to 3. To deal with Case 4, we will cut the loop $L(S)$ into two pieces in a particular way. Recall that in Section 3, we cut the loop $L(S)$ into two pieces to secure certain properties. Here, we will cut the loop $L(S)$ into two pieces in the following (different) way: Let s_0 be an element in $S_{\mathcal{O}}$ that divides $S_{\mathcal{O}}$ into two parts, each containing half the odd-1's of Case 4 (i.e. odd-1's that are in blocks with Case 4 even-1's). This will be one of the new points at which we cut $L(S)$. Then we find another point such that one string contains at least half the odd-1's and the other string contains at least half the even-1's. For these new strings, let us call them $S'_{\mathcal{O}}$ and $S'_{\mathcal{E}}$, note that now $S'_{\mathcal{E}}$ contains at least half of the $\mathcal{O}[S]$ odd-1's that were in blocks with the Case 4 even-1's. Thus, we can apply the Case 2 folding to $S'_{\mathcal{E}}$, i.e. $S'_{\mathcal{E}}$ now contains blocks of 1's that begin with odd-1's. This gives the following the approximation guarantee:

$$\left(\frac{3\mathcal{O}[S]}{2}+\frac{1}{4}\frac{4\delta_4\mathcal{O}[S]}{3}\frac{1}{2}\right)\frac{1}{4\mathcal{O}[S]} \;=\; \frac{3}{8}+\frac{\delta_4}{24}$$

□

Now we can prove Lemma 18.

Proof of Lemma 18: Setting all the approximation guarantees equal, we have:

$$\frac{\delta_1}{40}=\frac{\delta_2}{32}=\frac{\delta_3}{32}=\frac{\delta_4}{24}$$

Using the fact that $\delta_1+\delta_2+\delta_3+\delta_4=\frac{\delta}{2}$, we obtain that when $\delta_1\geq\frac{5\delta}{32}$, we should use the Case 1 modification. This implies that the approximation guarantee for the four cases is at least:

$$\frac{3}{8}+\frac{5\delta}{32}\frac{1}{40}=\frac{3}{8}+\frac{\delta}{256}$$

□