Approximate Protein Folding in Oz through Frequency Analysis

Sandro Bozzoli[1], Fausto Spoto[1], and Agostino Dovier[2]

[1] Dipartimento di Informatica, Univ. di Verona
Strada Le Grazie 15, 37134 Verona, Italy
sandrobozzoli@hotmail.com, fausto.spoto@univr.it
[2] Dipartimento di Matematica e Informatica, Univ. di Udine
Via delle Scienze 206, 33100 Udine, Italy
dovier@dimi.uniud.it

Abstract. The protein folding problem consists in determining the spatial shape of a protein once the linear sequence of its amino acids is known. Its solution uses mathematical models of the protein, the HP model being one of the simplest. Despite its simplicity, the problem of protein folding inside the HP model is still $\mathcal{NP}$-complete. In this paper we rephrase the HP model in a way that allows us to devise a new heuristic to burst the performance of automatic protein folding inside the HP model. Our implementation of protein folding in Mozart/Oz shows that our heuristic is very effective in practice and features a high level of precision for the solution it computes.

1 Introduction

Protein folding is among the most important problems of computational biology [5, 6, 11]. It consists in determining the spatial shape of a protein once the linear sequence of its amino acids is known. The number of possible spatial shapes grows exponentially with the length of the protein that can be made of 500 and more amino acids. Moreover, there are 20 kinds of amino acids and this makes complicated the energy functions to be computed for each spatial shape.

The shape of a protein can be experimentally determined through crystallographic techniques, NMR, or X-rays. These techniques are slow and expensive. Nevertheless, they have suggested the rules underneath the way proteins fold. Researchers have then been inspired abstract models for protein folding and algorithms based on them.

One of those models is the HP model [8], where amino acids are partitioned into two classes H (*hydrophobic*) and P (*polar*). Each amino acid is also called a *monomer*. Hydrophobic amino acids dislike water and hence gather in order to avoid contact with it. Polar amino acids, instead, like water and are henceforth distributed on the external surface of the protein. Undeniably, this model is a major simplification of the original problem. Nevertheless, it is considered faithful in practice since researchers think that the main issue behind protein folding is how to fold the protein in such a way that hydrophobic amino acids form a

hydrophobic kernel and hence avoid contact with water. A further simplification of the HP model is to allow amino acids to take only discrete spatial positions. A cubic lattice is used here, although some recent developments of the HP model use a more complex lattice, like in the FCC model [13].

Despite the apparent simplicity of the HP model, computing the optimal protein folding inside that model is an $\mathcal{NP}$-hard problem [4, 7]. Nevertheless, some interesting results have been found recently. In [2] the authors fold proteins of up to 40 amino acids, and in [3] they reach the number of 160 amino acids.

Our first contribution in this paper is a reformulation of the HP model in such a way that the position of every amino acid is specified as an offset from that of the previous one. This allows us to estimate the *frequency* of every amino acid in the cost function of the folding, thus determining those pairs of H amino acids that should be close in order to improve the optimality of the folding. Our second contribution is the implementation of that heuristic in the Mozart/Oz programming language. We first encode declaratively the problem without our heuristic and then introduce the heuristic in the code. Then we compare the execution times of the two programs and show that the second reduces dramatically the time of the folding, still yielding a result which is extremely close to the optimal one.

Although the implementation of our heuristic inside our protein folding algorithms is not efficient enough for large foldings, its application is independent from the underlying algorithm. Hence, we think that its application to the fastest folding algorithm available at the moment [3] should further improve its efficiency. Similarly, it could be applied to protein folding algorithms that use more precise models (e.g., [13, 9]).

2 The HP Model for Protein Folding

In this section we first introduce the traditional definition of the HP model for protein folding in two dimensions *i.e.,* the one based on a function that maps every amino acid of the protein into its spatial coordinates. We then rephrase it in such a way that every amino acid is mapped into its spatial offset from the coordinates of the preceding one in the protein.

Definition 1 (2D protein folding [6]). *Let $S = s_1, \ldots, s_n$ be a finite string, with $s_i \in \{\mathrm{H}, \mathrm{P}\}$ for every $i = 1, \ldots, n$. A* folding *of S is a map $\omega : \{1, \ldots, n\} \mapsto \mathbb{N} \times \mathbb{N}$ (or $\mathbb{N}^3$) such that*

1. *$\|\omega(i) - \omega(i+1)\| = 1$ for all $i = 1, \ldots, n$, where $\|\cdot\|$ is the Euclidean norm;*
2. *$\omega(i) \neq \omega(j)$ for all $i \neq j$, $1 \leq i, j \leq n$.*

The first condition forces amino acids to be contiguous. The second one avoids cycles by forcing amino acids to occupy distinct locations.

Definition 2 (Dill's HP model). *Let S be as in Definition 1. The protein folding problem consists in finding the folding ω which maximises the number of*

contacts between the H *elements of* S *which are not contiguous in* S*. Equivalently, it must minimise*

$$\sum_{\substack{1 \le i < n-2 \\ i+2 \le j \le n}} \mathit{Energy}(i,j)$$

where Energy is defined as

$$\mathit{Energy}(i,j) = \begin{cases} -1 & \text{if } s_i = \mathrm{H} = s_j \text{ and } \|\omega(i) - \omega(j)\| = 1 \\ 0 & \text{otherwise.} \end{cases}$$

We now provide our alternative definition to Definition 1. It is based on the observation that every element of the sequence S is *bound* to the previous one. Namely, the ith element is bound to be at distance 1 from the $(i+1)$th. Then we can describe a folding by providing a map from the elements of S to their offsets from the previous element in S. For the sake of simplicity, we focus on 2D models here. It is immediate to extend the definitions to 3D models.

Definition 3 (2D protein folding, offset version). *Let* $S = s_1, \ldots, s_n$ *with* $s_i \in \{\mathrm{H}, \mathrm{P}\}$ *for every* $i \in 1, \ldots, n$*. A* folding *of* S *is a map* $\theta : \{1, \ldots, n-1\} \to D \times D$ *where* $D = \{-1, 0, 1\}$*. We define the* coordinates *of the* i*th element of* S *w.r.t. the first element of* S *as*

$$P_i^1 = \sum_{1 \le j < i} \theta(j),$$

where the sum is performed component-wise. Note that $P_1^1 = (0,0)$*. We require that*

$$P_i^1 \neq P_j^1 \text{ for every } i, j = 1, \ldots, n,\ i \neq j.$$

Note that this last requirement corresponds to the two conditions of Definition 1. It forces different elements of S to take different spatial positions.

We can now provide an equivalent definition to Definition 2 which uses, however, the alternative definition of folding we have given in Definition 3. Note that our definition, being equivalent to that originally provided by Dill (Definition 2), approximates the protein folding problem with exactly the same precision as Dill's definition.

Definition 4 (HP model, offset version). *Let* S *be as in Definition 3. The protein folding problem consists in finding the folding (Definition 3)* θ *which minimises*

$$\sum_{\substack{1 \le i < n-2 \\ i+2 \le j \le n}} \mathit{Energy}(i,j)$$

where Energy is defined as

$$\mathit{Energy}(i,j) = \begin{cases} -1 & \text{if } s_i = \mathrm{H} = s_j \text{ and } \|\Gamma(i,j)\| = 1 \\ 0 & \text{otherwise} \end{cases}$$

and

$$\Gamma(i,j) = \sum_{i \leq k < j} \theta(k).$$

Both Definition 2 and Definition 4 can be used to implement a protein folding algorithm. However, the new version (Definition 4) is in general slower because it uses offsets, which requires long sum calculations. Hence the introduction of Definition 4, as an alternative to Definition 2, is justified by other considerations. Namely, it allows us to observe that some elements of the sequence S should be in contact in order to find the optimal solution for the protein folding problem. This is the subject of next section.

3 Frequency Analysis

We first define a map that stores the positions of the H elements of a sequence.

Definition 5. *Let $S = s_1, \ldots, s_n$ with $s_i \in \{\mathrm{H}, \mathrm{P}\}$. We define*

$$hpos(S) = \{i \mid s_i = \mathrm{H}\}.$$

Namely, if $S = HHPPPHHPHPHPPPH$ then $hpos(S) = \{1, 2, 6, 7, 9, 11, 15\}$.

We know that a protein folding algorithm aims at putting in contact non-contiguous H elements of the sequence S (Definitions 2 and 4). Our heuristic is based on the hypothesis that *the optimal solution is found when the distances between those* H *elements is minimal.* A well-known result for 2D lattices is that the ith element of S can actually be put in contact with the jth element of S only if i and j have different parity *i.e.,* the first is even and second is odd, or vice versa. This restricts the set of significant pairs of H elements.

Consider again the sequence $S = HHPPPHHPHPHPPPH$. We want to over-approximate the sum of the distances of every pair of H elements that can be put in contact and to identify the distance which weighs more on that sum. In Figure 1 we introduce the variables A–F to represent the distances between the H elements of S, whose positions can be computed as $hpos(S)$.

Variable	i	j
A	1	2
B	2	6

Variable	i	j
C	6	7
D	7	9

Variable	i	j
E	9	11
F	11	15

Fig. 1. Variables representing the distance between the occurrence of H at position i and that at position j in $S = HHPPPHHPHPHPPPH$.

By taking into account the previous observation on the parity of the H elements that can actually be put in contact, we identify the following candidate pairs of H elements: 1–6, 2–7, 2–9, 2–11, 2–15, 6–9, 6–11, 6–15. We must decide which of them is more resonable to put in contact. To this purpose, we overapproximate the distances between those elements as

1–6	A+B	2–15	B+C+D+E+F
2–7	B+C	6–9	C+D
2–9	B+C+D	6–11	C+D+E
2–11	B+C+D+E	6–15	C+D+E+F.

We analyse now how *frequently* each variable occurs in the sums above:

A	occurs once	D	occurs 6 times
B	occurs 5 times	E	occurs 4 times
C	occurs 7 times	F	occurs twice.

Hence, it seems reasonable to reduce the distances C and D *i.e.,* to reduce the distance between the H elements at index 6 and 9 in S, respectively. We can even force those two elements to be in contact in a solution for the protein folding problem. In this example, this constraint on the solution we are looking for allows our implementation of the protein folding algorithm (Section 6) to run in 0.581 seconds, to be compared with the 1.2 seconds required without the heuristic. Moreover, it finds the optimal solution in this case (4 contacts). Note, however, that this is not a general result. Our heuristic can sometimes lead to suboptimal solutions, although they are always very close to the optimal one in all our experiments (see section 6). Our heuristic can fail because the choice of putting closer the pair of points of higher frequency does not always lead to the optimal solution. Sometimes, that choice does not allow subsequent choices that were instead necessary to reach an optimal solution.

The efficiency of the protein folding algorithm tuned with our heuristic shows its strength also when we consider the dimension of the search tree generated by the Mozart/Oz engine while looking for a solution. Without the heuristic, our implementation of the protein folding algorithm generates 1056 choice points, 5 success leaves and 1052 failure leaves. These numbers drop to 503 choice points, 4 success leaves and 500 failure leaves with our heuristic.

Consider now the sequence $S = HPPHPHPHPHPHPPPHPPHPPH$ for which $hpos(S) = \{1, 4, 6, 8, 10, 12, 16, 19, 22\}$. Again, we name the distance between the H elements of S.

Variable	i	j
A	1	4
B	4	6

Variable	i	j
C	6	8
D	8	10

Variable	i	j
E	10	12
F	12	16

Variable	i	j
G	16	19
H	19	22

Fig. 2. Variables representing the distance between the occurrence of H at position i and that at position j in $S = HPPHPHPHPHPHPPPHPPHPPH$.

The candidate pairs of H elements that can be put in contact are now 1–4, 1–6, 1–8, 1–10, 1–12, 1–16, 1–22, 4–19, 6–19, 8–19, 10–19, 12–19, 16–19 and 19–22. Their distances can be overapproximated as in Figure 3.

Now, all variables occurs exactly seven times in these sums, with exception of H which only occurs twice. Hence, our heuristic faces many equivalent choices.

1–4	A
1–6	A+B
1–8	A+B+C
1–10	A+B+C+D
1–12	A+B+C+D+E
1–16	A+B+C+D+E+F
1–22	A+B+C+D+E+F+G+H
4–19	B+C+D+E+F+G
6–19	C+D+E+F+G
8–19	D+E+F+G
10–19	E+F+G
12–19	F+G
16–19	G
19–22	H

Fig. 3. Overapproximations of distances in $S = HPPHPHPHPHPPPHPPHPPH$.

For instance, it can choose to put in contact the H elements at index 1 and 4. This choice leads, again, to an optimal solution (5 contacts) and reduces the running time of the algorithm from 56.82 seconds to 8.41 seconds. Similar effects can be observed on the search tree of the algorithm: from 61227 choice points, 6 success leaves and 61222 failure leaves we drop down to 9518 choice points, 4 success nodes and 9515 failure nodes.

As we showed in the description above, our heuristic has been derived from reasonings based on the offset version of the protein folding algorithm (Definitions 3 and 4) since it allows us to reason about the relative distance of two H elements in an amino acid sequence. However, its application and implementation is orthogonal to the algorithm used to solve the protein folding problem. It just amounts to adding a constraint that restricts the search space of the algorithm on the basis of the choice our heuristic has made.

4 Implementation in Mozart/Oz

We describe here our implementation in Mozart/Oz [12] of a protein folding algorithm based on the HP model (Definition 2). We have chosen Mozart/Oz because

- it provides the *Oz Explorer*, which allows one to inspect the search tree. This means that we can always be aware of how the solver is doing at a given time, which let us choose the right instantiation strategy for the variables. For instance, this allowed us to choose the first-fail labeling strategy, which works by instantiating the variables with a smaller domain first, and should lead faster to failure for finite failure subtrees;
- it allows one to thoroughly rewrite its solver, which is very important if one has to use special purpose solvers. For instance, in our implementation we have used a solver that exploits symmetries to improve the efficiency of the search (symmetries originate from rotations of the protein around the

Cartesian axes). By embedding the notion of symmetry inside the solver (Section 5), we do not need to add and remove constraints here dynamically to cope with symmetries (unlike in the SES algorithm [2]).

We first describe the algorithm for bidimensional protein folding. We then discuss how we have also implemented its three-dimensional version.

Our bidimensional implementation is based on the traditional *constrain & generate* technique for logic programming [10]. This means that we first create a set of constraints to be satisfied by the solutions of the problem and then generate those solutions, requiring them to maximise the number of contacts between the non-contiguous H elements.

The kernel of our program is a function called `SearchProc`. We use two global variables `Lam` and `NElem` that hold the H and P sequence and its length, respectively. The program computes a resulting data structure `Root` which contains the variables:

- `contact`: it holds the number of contacts for this solution;
- `data`: it holds the bidimensional coordinates of the solution.

The algorithm computes those solutions by using $2 * $ `NElem` variables that represent the bidimensional coordinates of the current candidate solution. They are bound to take values between 1 and `NElem`.

In Figure 4 we report the code of `SearchProc`. Remember that a Mozart/Oz function is defined by declaring its name followed by the sequence of its parameters.

The `CreateIL` function computes the list of integers *hpos*(`Lam`) (Definition 5). The `PointDistinct` function constrains the coordinates of the solution to be distinct. It is implemented as

```
fun {PosToInt X Y}
  {FD.plus {FD.times Y 500} X} end

proc {PointDistinct Data}
  {FD.distinct {Map2 PosToInt Data}} end
```

The `Map2` function applies a binary function to the elements of its second argument, which must be a list, taken pair-wise. This amounts to hashing all the X, Y coordinates into $500Y + X$ and then requiring those values to be distinct. If we assume that proteins are up to 499 amino acids long, this is enough to constrain those amino acids to take distinct positions.

The `NextConstraint` function adds the constraint $\|\omega(i) - \omega(i+1)\| = 1$. The `Fix2Elem` function gives position to the first two amino acid of the sequence. In this way we cut away many symmetries derived from the translation and rotation of those two amino acids. Note that the algorithm that removes all the other symmetries has been embedded inside the constraint solver (Section 5).

The `FD.distribute` function does the labeling over the constraints that have just been generated. Variables are instantiated through the first-fail strategy.

While the same technique can be applied for three-dimensional protein folding, it would result in a very inefficient algorithm. Hence we have programmed

```
%% the declaration of SearchProc
proc {SearchProc Root}
  %% we declare the local variables of the function. We bound
  %% Contact to take values between 0 and MaxInt.
  %% The other variables are still unbound
  Contact = {FD.decl}
  Data
  Tmp
  IL
  TimeX
in
  %% the body of the function. It is a constraint and generate algorithm.

  %% we unify the parameter with a structure. This will be the
  %% return value of the function
  Root = r(contact: Contact
           data: Data)

  %% we create a list Data of 2*NElem elements between 1 and NElem.
  %% This list represents the spatial positions of the amino acids
  {FD.list 2*NElem 1#NElem Data}

  %% we compute hpos and we store it in IL
  {CreateIL Lam IL}

  %% distinct amino acids cannot have the same position
  {PointDistinct Data}

  %% amino acids must be put in subsequent spatial positions
  {NextConstraint Data}

  %% we fix the positions of the first two amino acids
  %% so that we avoid many symmetries
  {Fix2Elem NElem Data}

  %% we create the contact type that describes the number of contacts
  {FD.sum {CreateFuncEnergy ReifiedDistance EnergyImp Data IL} '=:' Contact}

  %% the labeling function, by using the first-fail strategy
  {FD.distribute ff Data}
end
```

Fig. 4. The Mozart code of `SearchProc`

our three-dimensional version of protein folding by following the ideas already shown in [1] and [14]. They show that the problem can be simplified by discretising the space into slices and then reasoning on the number and kind of amino acids in every slice. In this case, the problem becomes a minimisation problem on the *surface* of the protein instead of its number of contacts. The surface of a protein is related to the surface of the smallest solid containing it; the idea is that more contacts imply smaller surface. Precisely, the *Surface* $Surf_S(c)$ is defined in [14]. Let n_H^S be the number of H-monomers in S. Then for every folding c we have $6 \cdot n_H^S = 2 \cdot [Contact_S(c) + HHBounds(S)] + Surf_S(c)$, where $HHBounds(S)$ is the number of bonds between H-monomers (*i.e.,* the number of H-monomers whose successor in S is also a H-monomers) and $Contact_S(c)$ is the number of contacts. Since $HHBound(S)$ is constant for all folding of S, this implies that minimizing the surface is equivalent to maximizing the number of contacts. We refer to [1] and [14] for more details.

5 Dealing with Symmetries

Protein folding is a kind of optimality problem for which it is not possible to know whether we found the optimal solution unless the whole search space is spanned (there is no simple *optimality test*). In that situation, it becomes very important to prune the search space so that the spanning time becomes smaller. Symmetries are used to that purpose.

It can be proved that the n-dimensional protein folding problem has $n! \cdot 2^n$ symmetries, which means that the three-dimensional case has 48 symmetries [2]. Hence the great importance given to symmetry breaking algorithms for protein folding. We follow here the idea of [2], which shows how symmetries can be tamed by modifying the strategy of the solver. Namely, if the solver faces two alternatives of the search tree, the visit of the first alternative will induce a constraint that modifies the behaviour of the solver during the visit of the second alternative. That way we avoid to generate symmetrical solutions from the second alternative.

The ability to change dynamically the behaviour of the solver has been a key issue in our choice of Mozart/Oz for the implementation of our algorithms.

6 Experimental Evaluation

We have run our implementations on random sequences and compared the efficiency of the protein folding algorithms both with and without our heuristic. Our experiments show that the heuristic always leads to optimal solutions, with the exception of some examples where it leads to almost-optimal solutions.

We start from the bidimensional case. Figure 5 shows some sequences of amino acids we used to test our bidimensional algorithm and reports the results we obtained. You can see that only for L4 our heuristic has found a suboptimal solution, and in that case only one contact has been lost.

$L1$: $HHHHPHHHHHHHHPHHHHHH$

$L2$: $HHPHHPHPHHHHHPPPHHHH$

$L3$: $PHPHPHHPHHHHPHH$

$L4$: $HHPPPPHHHHHHHHPHHPH$

$L5$: $PPHHHHHHHHHHHHHHHHHH$

	without our heuristic		**with our heuristic**	
L1	12	2h42m12s	12	22m55s
L2	10	1h11m50s	10	10m08s
L3	6	31.5s	6	6.5s
L4	9	18m46s	**8**	3m45s
L5	10	3h47m57s	10	33m45s

Fig. 5. Some sequences we used for testing the bidimensional protein folding algorithm and the results we obtained.

A graphical comparison of the times required by the bidimensional protein folding algorithm both with and without our heuristic is shown in Figure 6. You can see that the heuristic dramatically improves the efficiency of the algorithm. Moreover, that improvement is bigger for longer sequences of amino acids. The same figure compares the precision of the solutions found by the protein folding algorithm with and without our heuristic. You can see that our heuristic does not lose precision significantly.

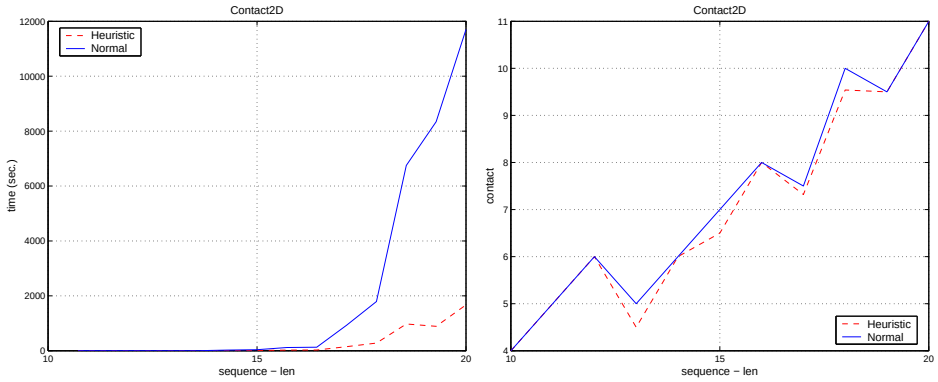

Fig. 6. An evaluation of the effect of our heuristic on the time and precision of the bidimensional protein folding algorithm.

Figure 7 shows instead the effect of our heuristic on the three-dimensional version of protein folding. You can see that it leads to (almost) optimal solutions

but halves the time required by the algorithm. Remember that the surface has now to be minimised. Although Figure 7 shows that we *only* halve the time to reach a solution, it must be said that we have used an algorithm which is based on a sequence of searchs. Our heuristic has been only applied to the last search. A more extensive use of it should lead to better results.

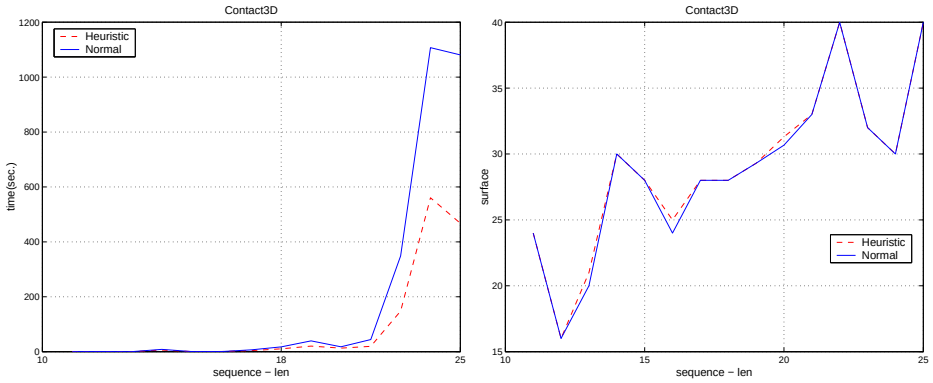

Fig. 7. An evaluation of the effect of our heuristic on the time and precision of the three-dimensional protein folding algorithm.

7 Conclusions

Our heuristic shows that almost optimal solutions to the protein folding problem can be found with a significant improvement in the time of the algorithm. We are currently implementing this technique for richer models than the HP one.

The application of our heuristic to the fastest available protein folding algorithm [3] should further burst its performance, and allow us to compute protein foldings for proteins longer than 160 amino acids.

This kind of heuristic could help also the approaches to the protein folding problems without the HP-abstraction. In these cases the energy function is based on a 20×20 table (we have 20 kinds of amino acids). Moreover, it is typically solved on more complex lattices, such as the FCC lattice. A CLP-based approach to this general problem is described in [9]. The integration of that proposal with the new heuristic is feasible and, we believe, promising.

Acknowledgments. We thank Alessandro Dal Palù for the useful discussions we had with him during the preparation of this manuscript.

References

1. R. Backofen. The Protein Structure Prediction Problem: A Constraint Optimization Approach using a New Lower Bound. *Constraints*, 6(2/3):223–255, 2001.

2. R. Backofen and S. Will. Excluding Symmetries in Constraint-based Search. In J. Jaffar, editor, *Proc. of Principles and Practice of Constraint Programming, CP'99*, volume 1713 of *Lecture Notes in Computer Science*, pages 73–81, Berlin, 1999. Springer-Verlag.
3. R. Backofen and S. Will. Fast, Constraint-based Threading of HP-Sequences to Hydrophobic Cores. In T. Walsh, editor, *Proc. of Principles and Practice of Constraint Programming, CP'01*, volume 2239 of *Lecture Notes in Computer Science*, pages 494–508, Paphos, Cyprus, 2001. Springer-Verlag.
4. B. Berger and T. Leighton. Protein Folding in the Hydrophobic-Hydrophilic (HP) Model is $\mathcal{NP}$-complete. In *Proc. of the Second Annual International Conference on Computational Molecular Biology, RECOMB'98*, pages 30–39, New York, 1998.
5. R. Bonneau and D. Baker. Ab Initio Protein Structure Prediction: Progress and Prospects. *Annual Review of Biophysics and Biomolecular Structure*, 30:173–189, 2001.
6. P. Clote and R. Backofen. *Computational Molecular Biology, An Introduction.* Wiley Series in Mathematical and Computational Biology. J. Wiley & Sons Ltd, 2000.
7. P. Crescenzi, D. Goldmain, C. Papadimitriou, A. Piccolboni, and M. Yannakakis. On the Complexity of the Protein Folding. In *Proc. of the ACM Symposium on Theory of Computing, STOC'98*, pages 596–603, Dallas, Texas, 1998.
8. K. A. Dill. Dominant Forces in Protein Folding. *Biochemistry*, 29:7133–7155, 1990.
9. A. Dovier, M. Burato, and F. Fogolari. Using Secondary Structure Information for Protein Folding in $clp(\mathcal{FD})$. In M. Comini and M. Falaschi, editors, *Electronic Notes in Theoretical Computer Science*, volume 76, 2002.
10. K. Marriott and P.J. Stuckey. *Programming with Constraints.* The MIT Press, 1998.
11. J. Skolnick and A. Kolinski. Computational Studies of Protein Folding. *Computing in Science and Engineering*, 3(5):40–50, 2001.
12. G. Smolka. The Oz Programming Model. In Jan van Leeuwen, editor, *Computer Science Today*, volume 1000, pages 324–343, Berlin, 1995.
13. S. Will. Constraint-based Hydrophobic Core Construction for Protein Structure Prediction in the Face-centered Cubic Lattice. In R.B. Altman, A.K. Dunker, L. Hunter, and T.E. Klein, editors, *Proc. of the Pacific Symposium on Biocomputing*, volume 7, pages 661–672, Lihue, Hawaii, USA, January 2002.
14. K. Yue and K. A. Dill. Sequence-Structure Relationships in Proteins and Copolymers. *Phisical Review E*, 48(3):2267–2278, 1993.