Computational Biology
Lecture 21: Protein folding and threading
Saad Mneimneh

As was the case with RNA folding, the goal is to determine the three-dimensional structure of a protein based on its amino acid sequence. There is a strong belief that the amino acid sequence completely and uniquely determines the folding. This is reinforced by the following experiment: Unfold a protein (by exerting a certain temperature) and then release it such that no other substances are present. The protein immediately folds back to the three-dimensional structure it had before, thus called its *native* structure. The folding process takes less than a second. Therefore, it appears that all information necessary for the protein to achieve its native structure is contained in the amino acid sequence (this is not true for all proteins because there are some that need auxiliary molecules to fold). Before discussing the protein folding prediction, we describe the main components of the protein three-dimensional structure, known as protein secondary structures. These are α-helices, β-sheets, and loops.

Protein secondary structures

There are three main secondary structures that we observe in proteins:

α-helices

An α-helix is a simple helix having on average 10 residues (3 turns of the helix). Some helices can have as many as 40 residues. Some amino acids appear more frequently on helices than other amino acids, but this is not a strong enough fact to allow accurate prediction.

β-sheets

A β-sheet consists of bindings between several sections of the amino acid sequence. Each participating section is called a β-strand and has generally 5 to 10 residues. The strands become adjacent to each other forming a kind of twisted sheet, thus the name. Certain amino acids show a preference for being in a β-sheet, but again these preferences are not so positive to allow accurate prediction.

Loops

A loop is neither a helix not a sheet. It is a section of the sequence that connects the other two kinds of secondary structure (α-helices and β-sheets). Loops do not form regular structures, both in shape and size. In general, loops are outside a folded protein, whereas the other structures form the protein **core**. The core is going to be important for the threading problem that we discuss later.

Here's an illustration of a folded protein (better seen in color):

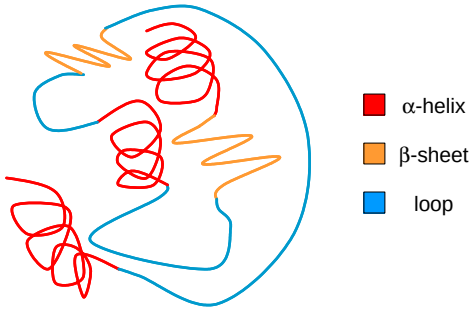

Figure 1: A folded protein

Motifs and Domains

A motif is a simple combination of a few secondary structures that appears in several different proteins. An example of a motif is the helix-loop-helix. Research as shown that this motif is used as a binding site for calcium atoms, so this structure has a clear function. Other motifs appear to play no role at all in protein function.

A domain is a more complex combination of secondary structures that by definition has a very specific function, and therefore contains an *active site*, which is the a section of the protein where some binding to an external molecule can take place. A protein may have only one domain, or may contain several. All of them taken together form the protein's *tertiary structure*. We will study the problem of motif (domain) finding and identification later in the course.

Protein folding problem

The protein folding problem is the following: Given the amino acid sequence of a protein, determine:

- where exactly all of its α-helices, β-sheets, and loops are, and
- how they arrange themselves in motifs and domains

There are several approaches to this problem (non of them is fully satisfactory). Most of these approaches are based on finding a folding with minimum free energy and, therefore, they assume that such a folding (or conformation) **is** the protein's native structure (not yet a proved fact).

Greedy approach

Recall from the early lectures that a protein fold thanks in large to the angles ϕ and ψ between the carbon atom of a residue and the neighboring atoms (N and (CO)) in the peptide bond $-N-C-(CO)-$. Experiments have shown that these angles can assume only a few values independently of each other. Therefore each residue will be in a *configuration* given by a pair of values for ϕ and ψ.

Suppose both ϕ and ψ can assume 3 values and the protein is 100 residues long, then we have to examine 9^{100} foldings because each reside can be in any of the $3 \times 3 = 9$ configurations. This is of course not feasible. Even if we forget about this *tiny* detail and we agree that the protein's native structure is in deed the one with minimum free energey, there is no agreement on how to compute the free energy of a folding. Too many factors are involves such as shape, size, polarity of molecules, strength of interactons between molecules, etc... From current knowledge it is possible to formulate some general rules, such as the requirement that hydrophobic (H) amino acids (they are afraid of water) stay "inside" the protein and hydrophilic or polar (P) amino acids staty "outside" the protein, in the case of proteins in solutions where water is the solvent found both inside and outside cells. We will see this HP model of protein folding again in the combinatorial approach.

Simulation approach

With this approach we attempt to simulate the real combined physical/chemical/biological process of protein folding. There are many methods for carrying out the simulation and they highly depend on the physical assumptions and and the underlying mathematical models.

One type of simulation aims at understanding the bevaior of the protein by simulating the folding of known structures. In such simulation, the minimum energy conformation is known (or explicitly constructed). Then starting from the sequence of amino acids, the simulation guides the protein to reach the native conformation, and the behavior of the protein during this process is observed. The protein was observed to exhibit some intermediate conformation, then unfold back, and repeat this behavior until the native conformation (the one with the minimum energy) is reached. This behavior was verified experimentally. We will describe very briefly how such a simulation might proceed.

Given a conformation, if two amino acids are neighbors, they are said to be in "contact", and hence they contribute to the free energy. Therefore, given a sequence $a = a_1...a_n$, for two given amino acids a_i and a_i we define $d_c(i,j)$ to be the distance between a_i and a_j in a conformation c. If $d_c(i,j)$ is less than a threshold, we let $\Delta_c(i,j) = 1$ (a_i and a_j are in contact); otherwise, $\Delta_c(i,j)$=0.

Let $H_c = \sum_{i,j} B(i,j)\Delta_c(i,j)$ where $B(i,j)$ represents the interaction between a_i and a_j if they are in contact. H desribes the short-range pairwise interactions, and hence it is a sort of energy, but it is not exactly the free energy of the comformation.

A good measure of how close a conformation c is to the native conformation nc is given by $Q_c = \sum_{i,j} \Delta_c(i,j)\Delta_{nc}(i,j)$. Therefore, the simulation should guide the protein to maximize Q. The dynamics of the protein are modeled as a random

process where local movements of amino acids are considered, and always accepted if the new conformation has higher Q, but only accepted with probability $e^{-k(Q_{new}-Q_{old})}$ for some constant k.

Let $Q(t) = Q_c$ where conformation c is the current conformation at time t. The simulation revieled that for the early times of the folding process, $Q(t)/Q_{nc}$ is a low constant suggesting that the protein finds some native contact. But non of these constacts were found to be fixed, i.e. $n_{fix}(t)/n \approx 0$ where $n_{fix}(t)$ is the number of proteins with low positional variance at time t. After some time however, persistent contacts suddenly form and $n_{fix}(t)/n$ raises rapidly to become comparable to $Q(t)/Q_{nc}$. There is a gradual increase in $Q(t)/Q_{nc}$ and then both measures remain nearly constant for some time. This could then lead to folding to the native conformation or unfolding, and the process repeats until the native conformation is reached.

By running the simulation several times, statistical data regarding the intermediate conformations can be obtained; for instance, one could calculate the probability that a given intermediate conformation will lead to a native conformation.

Combinatorial approach

The combinatorial approach abstracts the problem as follows: given a string s and an infinite grid, fold the string by finding a walk on the grid, i.e. a non self intersecting path, such that some objective function is maximized. For instance, the objective might be to maximize the number of adjacent identical characters (amino acids). More generally, the objective is to maximize $f = \sum_{i,j} w(i,j)\Delta(i,j)$ where $w(i,j)$ is the amount of interaction between s_i and s_j, and $\Delta(i,j) = 1$ iff s_i and s_j are adjacent on the grid. For example, let $s = bacbbcacba$ and the grid be the two dimensional square grid. Let $w(i,j) = 1$ iff $s_i = s_j$. This means that f is the number of adjacent identical characters. Then the following fold maximizes f to be 5 (by creating 4 bonds in addition to b-b that is already part of the string).

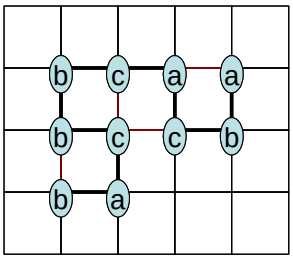

Figure 2: Folding *bacbbcacba* on a 2D square grid

In general, the grid needs not be a square nor two dimensional. The three dimensional square grid (cube) is the most popular. But triangluar grids (both 2D and 3D) have been considered as well.

We have different variations of the problem depending on the objective function f. Here are some popular models:

- contact model: $w(i,j) = 1$ iff $s_i = s_j$
- HP model: $w(i,j) = 1$ iff $s_i = s_j$ and both represent a hydrophobic amino acid

The motivation behind the first model is the maximize bonds among identical amino acids. The motivation behind the second model is to maximize bonds among hydrophobic amino acids, since these are thought to be the ones that fold quickly into the core of the protein to be protected from water.

The problem is NP hard under the various models. Paterson and Przytucka showed that the contact model is NP hard on two and three dimensional square grids when the string alphabet is not finite. Atkins and Hart showed that the contact model is NP hard on three dimensional square grids with a finite alphabet size (13 amino acids). Berger and Leighton showed that the HP model is NP hard on three dimensional square grids (here the alphabet size is two since we distinguish among H and P amino acids only).

Because of the difficulties associated with the above approaches, other techniques have been developed, e.g. Protein Threading.

Protein threading problem

An early method used for secondary structure prediction was based on the idea that similar sequences should have similar structures. If we know the structure of protein A (from x-ray crystallography, say) and protein B is very similar to it at the sequence level, it seems reasonable to assume that B's structure will be the same or nearly the same as

A's. Unfortunately, this is not true in general. Similar proteins at the sequence level may have very different secondary structures. On the other hand it has been observed that certain proteins that are very different at the sequence level are structurally related in the following sense: although they have different kinds of loops, they have very similar cores. These observations have led researches to propose the *protein threading* problem: instead of trying to predict the structure from the sequence, we try to fit the core of a known structure to a sequence.

Therefore, given a protein whose structure is known, we obtain a structural model by replacing amino acids by place holdres, but keeping with each place holder some basic properties of the original amino acid. These properties can vary depending on the model adopted, but they should retain the fact that the original amino acid was in an α-helix or a β-sheet, or in a loop, and reflect spacial constraints of the structure such as distances to other amino acids, how much inside or outside the whole structure the place holder is, and so on...

Given a structural model we can now get a protein with unknown structure and try to align it to the model. This type of alignment is called protein threading.

We can formalize the problem as follows:

We are given:

- A protein sequence A with n amino acids $a_1...a_n$
- A core s̄tructural model C, with m core segments, we have:
  - the length c_i of each core segment i
  - core segments i and $i+1$ are connected by loop for which we know the maximum l_i^{max} and minimum l_i^{min} lengths
  - properties of each amino acid in C
- A score function f to evaluate a threading

We want to find the set $T = \{t_1, ..., t_m\}$ of integers that will maximize f such that each t_i indicates what amino acid from A occupies the first position from core segment i.

The following figure provides an illustration:

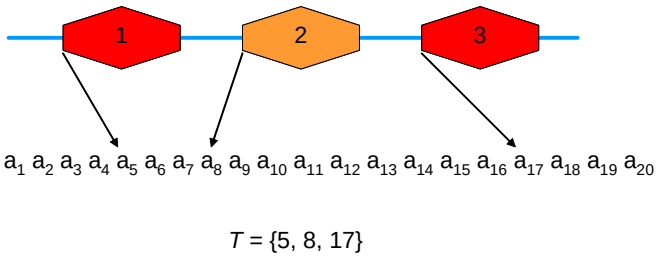

Figure 3: Threading: alignment of sequence to a core structural model

If we ignore interactions between amino acids, while still allowing variable-length loop regions, the threading problem can be solved by a standard dynamic programming technique. The most difficult part is how to model the loops. Each loop i will be modeled as a string $xxx...xyyy...y$ of length l_i^{max} with l_i^{min} occurences of x followed by y's. The scoring function will be designed such that an alignment of an x to a gap will contribute $-\infty$. This will force loop i to have at least l_i^{min} characters from A. Similarly, to make sure that no gaps are present in the core segments, a gap in a segment will contribute $-\infty$. The rest is easy and we will not focus much on it here since we have seen many dynamic programming algorithms.

If we allow pairwise interactions between amino acids, the threading problem becomes NP hard. We will allow pairwise interactions and show how to solve the problem exactly with a standard technique for handling NP hard problems. This technique is known as branch-and-bound.

Branch-and-Bound

In a combinatorial optimization problem we want to find the optimum solution among may possibilities in the solution space. The optimum refers the maximum (or minimum) value of some function f that can be computed for each candidate solution. For most of the problems the solution space is exponentially large. This does not necessarily mean that we have to spend an exponential amount of time to find the optimum (we do not necessarily examine all the candidate solutions in a greedy way). However, for some problems, no algorithm is known that can search the solution space and find the optimum in polynomial time. These are typically the NP hard problems. Therefore, the only alternative to obtain the optimum solution for sure is to enumerate and evaluate each candidate solution one by one and pick the best. We can try to speedup the process by applying the branch-and-bound technique. Assume we are after maximizing $f(s)$.

[branch]
First, it should be possible to divide the solution space into subspaces according to some constraints. For instance, given a solution space X, we could partition it into X_1 (all solution that have a certain property) and X_2 (all solution that do not). Note that the partitioning should be implicit, i.e. we do not explicitly enumerate the solutions.

[bound]
Second, for every partition X obtain an upper bound on the value of $f(s)$ for every solution $s \in X$

Now how can this technique help speedup the process? Assume we have computed $f(s)$ for a given s. Now consider a subset X of the solution space with an upper bound u. If $f(s) > u$, then we can we discard all candidate solutions in X because we already have a solution, s, that scores better than all solutions in X.

Of course branch-and-bound is just a high level technique. The actual implementation of this technique will differ from one problem to another depending on how the solution space can be divided and how the bounds can be computed.

Before we proceed to our specific case, the threading problem, we list some remarks concerning branch-and-bound. The upper bound (or lower bound in other cases) should be as close to the actual function value as possible, so that we can find the maximum faster than with a weaker upper bound. The upper bound should also be efficient to compute; otherwise, we defeat the purpose of speed up. Finally, even if we speedup the process by discarding some subsets of the solution space, the worst case running time will still be exponential (NP hard problem).

Finding the solution space

Every solution of the threading problem has to satisfy the following two conditions for every i:

- $1 + \sum_{j<i}(c_j + l_j^{min}) \leq t_i \leq n + 1 - \sum_{j \geq i}(c_j + l_j^{min})$
- $t_i + c_i + l_i^{min} \leq t_{i+1} \leq t_i + c_i + l_i^{max}$

The first condition says that the sequence A has to fit in the structure. So it is a combination of the two conditions: $t_i \geq 1 + \sum_{j<i}(c_j + l_j^{min})$ and $t_i + \sum_{j \geq i}(c_j + l_j^{min}) - 1 \leq n$. The second condition enforces the minimum and maximum constraints on the loop sizes. So it is a combination of the two conditions: $t_{i+1} \geq t_i + c_i + l_i^{min}$ and $t_{i+1} \leq t_i + c_i + l_i^{max}$.

This implies that each t_i must lie in some interval $[b_i, e_i]$. Therefore, a set of candidate solutions can be expressed as a collection of intervals, one for each of the m t_i's as illustrated in the figure below:

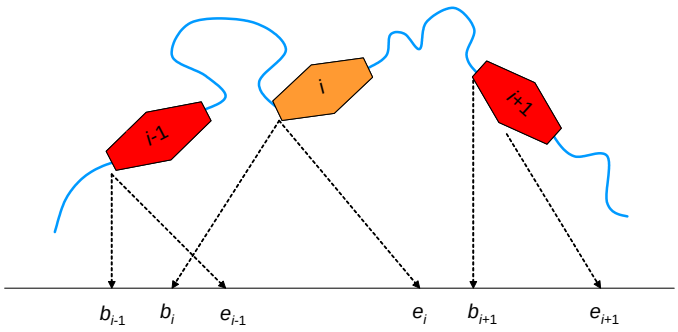

Figure 4: A set of solutions is a collection of intervals

Branching

Given a set of solutions, we can divide it as follows:

- choose a segment, say i
- choose a position u_i inside the interval $[b_i, e_i]$
- split the set into three sets in which the intervals for t_i will be:
  - $[b_i, u_i - 1]$
  - $[u_i + 1, e_i]$
  - $[u_i]$

We have not specified which core segment to choose nor which position inside the segments's interval. These choices can be made based on considerations such as interval size. Other approaches for dividing the set are possible and the one outlined above is just an example. However, this approach guarantess to generate, upon a division, a set of solutions in which some t_i is forced to be equal to u_i. Therefore, this approach helps in quickly obtaining a set X with a single solution s (where t_i is fixed for all i). This solution might not be optimal, but the upper bound on its set X is exactly the value $f(s)$. Therefore, $f(s)$ can be used to discard all sets with an upper bound greater or equal to $f(s)$.

Bounding

Given a candidate solution s, $f(s)$ can be expressed as:

$$f(s) = \sum_i g_1(i, t_i) + \sum_i \sum_{j>i} g_2(i, j, t_i, t_j)$$

where $g_1(i, t_i)$ gives the score pertaining to segment i and $g_2(i, j, t_i, t_j)$ gives the score pertaining to the pairwise interactions of segments i and j.

Given a set X of solutions, a simple upper bound is:

$$\begin{aligned} \max_{s \in X} f(s) = &\quad \max_{s \in X} \textstyle\sum_i [g_1(i, t_i) + \sum_{j>i} g_2(i, j, t_i, t_j)] \\ \leq &\quad \textstyle\sum_i [\max_{b_i \leq x \leq e_i} g_1(i, x) + \sum_{j>i} \max_{b_i \leq y, z \leq e_i} g_2(i, j, y, z)] \end{aligned}$$

This simple upper bound is obtained by maximizing every term in the summation while disregarding any constraints among the terms. Therefore, it might not correspond to any $f(s)$ where $s \in X$, but it is definitely an upper bound on any $f(s)$ for $s \in X$.

Given the above branch-and-bound technique, here's a simple algorithm for searching for the optimum solution.

Algorithm

$X \leftarrow$ all possible threadings (implicit representation by intervals)
$ub \leftarrow$ upper bound for X

[*use a max priority queue Q with keys being the upper bounds*]
ENQUEUE$(Q, (ub, X))$

while (true)
  do $(ub, X) \leftarrow DEQUEUE(Q)$
    if $|X| = 1$ [*the only remaining threading in the set*]
      then return X
      else split(X)
        for each subset X_i from X
          do $ub_i \leftarrow$ upper bound for X_i
            ENQUEUE$(Q, (ub_i, X_i))$

Note that once a set with a single solution s is enqueued, it blocks all subsets with an upper bound smaller than $f(s)$. Therefore, if such a set is at the head of the queue, it definitely contains a solution s with maximum $f(s)$.

Remark: The splitting might generate an invalid solution set; for instance, $b_{i+1} \leq e_i$. Such splitting must not be considered.

References

Setubal J., Meidanis J., Introduction to Computational Molecular Biology, Chapter 8.