Protein folding by recursive backtracking*

Bjørn Kjos-Hanssen[0000−0002−6199−1755]

University of Hawaiʻi at Mānoa, Honolulu HI 96822, USA
`https://math.hawaii.edu/wordpress/bjoern/` `bjoernkh@hawaii.edu`

Abstract. The hydrophobic-polar (HP) protein folding model was introduced by Ken A. Dill in 1985. It attracted a great deal of attention, at least until the advent of Google's AlphaFold in 2018.
In this model, a binary string like 0110 is interpreted as a polymer, a sequence HPPH of amino acids of two types. The string is embedded in an ambient space, and this embedding is called a fold. Some folds are better than others and this is quantified by the score of the fold.
In this project I formalize basic definitions and facts for the HP model in such a way that Lean can automatically calculate optimal scores. To speed up the calculation I use recursive backtracking. For good measure, I prove in Lean that my implementation of recursive backtracking and its application are correct.
A critical issue when formalizing the HP model seems to be the choice of definition for the path induced by a sequence of folding moves (like up, down, left, right). I will present a couple of approaches used with varying success.

Keywords: Hydrophobic-polar protein folding model · Lean · recursive backtracking

1 Introduction

The hydrophobic-polar (HP) protein folding model was proposed by Ken A. Dill [5]. Its popularity may have peaked around 1998 when the NP-completeness of the basic problem involved was shown in the 2D and 3D settings ([4], [3]). With the advent of Google's AlphaFold this NP-completeness may no longer be viewed as a severe problem from a practical perspective. However, the model remains mathematically interesting and a suitable playground for automated computations in proof assistants.

The basic setup is that a protein is modeled as a sequence from the alphabet $\{H, P\}$. When two occurrences of H are next to each other in the lattice but not in the sequence, a point is achieved. An optimal folding is one that achieves the maximum number of points. (In this article we identify $H = 0$ and $P =$

* This work was partially supported by grants from the Simons Foundation (#704836 to Bjørn Kjos-Hanssen) and Majesco Inc. (University of Hawaiʻi Foundation Account #129-4770-4).

1.) We demonstrate using a Lean formalization that this optimal score in the hydrophobic-polar protein folding model is non-monotone under concatenation, in three common variants of the model: 2D, triangular, and hexagonal grids.

2 Nonmonotonicity

Definition 1. *Let $P_{2D}(x), P_{3D}(x), P_{\mathrm{tri}}(x), P_{\mathrm{hex}}(x)$ denote the maximum number of points achievable for a word x in the 2D rectangular, 3D rectangular, triangular, and hexagonal lattices, respectively.*

In a March 5, 2023, email message, Jack Stecher conjectured that the HP folding problem is "weakly monotone", that is, if we add a prefix or suffix to a given sequence, our point count should never decrease.

Conjecture 1. Let P be one of the function in Definition 1 and let x, y be binary words. Then $P(x) \leq P(xy)$ and $P(y) \leq P(xy)$.

We'll call this *Stecher's conjecture* in order to anchor the following discussion. What happened next is a story of the interplay between proof assistant work and traditional methods.

The same day I found a refutation of the conjecture using a Python script. A few months later I tasked students Brian Shu, Sophia Kop, and Tung Nguyen with writing a more reasonable version of the script, employing recursive backtracking. Thinking about recursive backtracking, I ended up implementing it in Lean and proving in Lean that the procedure was correct. At this point it was natural to formalize further results about protein folding in Lean. At some point while formalizing further results, I had learned enough about the area to provide a mathematical refutation of Stecher's conjecture, showing there are infinitely many counterexamples, that did not rely on computation. The key new result is Theorem 1.

Theorem 1. *There are infinitely many words w with $P_{rect}(w) = Z(w) + 1$, where $Z(w)$ is the number of zeros in w.*

Proof. There are examples of each length of the form $26 + 8k$, $k \geq 0$:

$$(011)^3 1^{10} (0011)^k 0110 (1100)^k 110$$

In terms of l, r, d, u (left, right, down, up) the intended folding has the $25 + 8k$ moves

$$urdrdldrru^4 l^4 dd(luld)^k ldr(drur)^k dru.$$

The case $k = 3$ looks like this:

$$\begin{tikzcd}
 & & & & & & & 1 \arrow[r, dash] \arrow[d, dash] & 1 \arrow[r, dash] & 1 \arrow[r, dash] & 1 \arrow[r, dash] & 1 \arrow[d, dash] \\
 & 1 \arrow[r, dash] \arrow[d, dash] & 1 \arrow[d, dash] & 1 \arrow[r, dash] \arrow[d, dash] & 1 \arrow[d, dash] & 1 \arrow[r, dash] \arrow[d, dash] & 1 \arrow[d, dash] & 1 \arrow[d, dash] & 1 \arrow[r, dash] \arrow[d, dash] & 1 \arrow[d, dash] & & 1 \arrow[d, dash] \\
1 \arrow[r, dash] \arrow[d, dash] & 0 & 0 \arrow[r, dash] & 0 & 0 \arrow[r, dash] & 0 & 0 \arrow[r, dash] & 0 & \textcircled{0} & 0 \arrow[r, dash] & 1 \arrow[d, dash] & 1 \arrow[d, dash] \\
1 \arrow[r, dash] & 0 \arrow[d, dash] & 0 \arrow[r, dash] \arrow[d, dash] & 0 \arrow[d, dash] & 0 \arrow[r, dash] \arrow[d, dash] & 0 \arrow[d, dash] & 0 \arrow[r, dash] \arrow[d, dash] & 0 \arrow[d, dash] & \textcircled{0} \arrow[d, dash] & 0 \arrow[r, dash] \arrow[d, dash] & 1 & 1 \arrow[d, dash] \\
 & 1 \arrow[r, dash] & 1 & 1 \arrow[r, dash] & 1 & 1 \arrow[r, dash] & 1 & 1 \arrow[r, dash] & 1 & 1 \arrow[r, dash] & 1 \arrow[r, dash] & 1
\end{tikzcd}$$

How does this help up resolve Stecher's conjecture? Well, the famous Handshaking Lemma can be used to prove the following.

Theorem 2 (Agarwala et al. [1, Lemma 2.1]). *For all w, $P_{rect}(1w1) \leq Z(w)$.*

Indeed, observe that for each zero in x, there are four directions, two are occupied by the previous and the next amino acids, and the handshaking lemma gives a factor of two so that it comes down to $\frac{4-2}{2} = 1$. When the word may start or end with a zero the bound becomes:

Theorem 3 ([2, Fact 1]). *For all w, $P_{rect}(w) \leq Z(w) + 1$.*

Consequently, if Stecher's conjecture had held, we would have a proof that $P_{rect}(x) \leq Z(x)$ for all x: $P(x) \leq P(1x1) \leq Z(x)$.

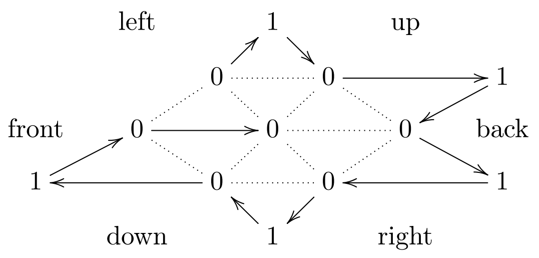
This proof does not yet work for other lattices. However, with calculations done using recursive backtracking, first in Python and then in Lean, we obtain nevertheless:

Theorem 4. *Stecher's conjecture fails in the 2D rectangular lattice, the triangular lattice, and the hexagonal lattice.*

For the 2D rectangular lattice an example looks like this:

$$\begin{tikzcd}
 & 0 \arrow[r] & 1 \arrow[d] \\
0 \arrow[r] & 0 & 0 \arrow[d] \\
1 \arrow[u] & 0 \arrow[l] & 1 \arrow[l]
\end{tikzcd} \tag{1}$$

For the triangular lattice, an example looks like this. (It also serves as an example in the 3D rectangular lattice as indicated by the labels left, right, up, down, front, back below.)



(2)

For the triangular lattice (brick wall lattice) an example looks like this:

$$\begin{tikzcd}
& 4 \arrow[r] \arrow[d] & 5 \arrow[r] & 6 \arrow[r] \arrow[d, dotted, no head] & 7 \arrow[r] & 8 \arrow[d] & \\
{} \arrow[r, dotted, no head] \arrow[d, dotted, no head] & 3 & 2 \arrow[l] \arrow[d, dotted, no head] & 1 \arrow[l] \arrow[r, dotted, no head] & A \arrow[d] & 9 \arrow[l] \arrow[r, dotted, no head] & {} \arrow[d, dotted, no head] \\
{} \arrow[rr, dotted, no head] & & D & C \arrow[l] & B \arrow[l] \arrow[rr, dotted, no head] & & {}
\end{tikzcd}$$

3 Formal verification

We verify some of our results in the proof assistant Lean 4 ([6]). The algorithmic technique of backtracking is formalized as follows. Since the number of words having a property `P` with a given suffix `w` is most easily evaluated when `w` is of full length, we need the datatype `Gap`.

```
def Gap (b L k : ℕ) : Type := Vector (Fin b) (L - k)
def Gap_cons {b n L:ℕ} (a:Fin b) (w : Gap b L.succ n.succc)
  (h: ¬ n ≥ L.succ) : Gap b L.succ n
  := ⟨a :: w.1, by rw [List.length_cons];simp;exact (Nat.succ_sub
    (Nat.not_lt.mp h)).symm⟩
def Gap_nil {k b L : ℕ} (h : k ≥ L.succ) : Gap b L.succ k
  := ⟨List.nil, by rw [Nat.sub_eq_zero_of_le h];rfl⟩
```

Mathematically, it would perhaps be even more natural to consider the number of words satisfying `P` with a given *prefix*, but the constructor of `List` in Lean adds symbols on the left, which makes using suffixes slightly more convenient.

Next `num_by_backtracking` calculates the number of words (of `List` type) that satisfy the conjunction of the predicates `P` and `Q`, where the monotonicity of `P` is used to shortcut futile searches.

```
def num_by_backtracking {k b L:ℕ}
  (P: List (Fin b) → Prop) [DecidablePred P]
  (Q: List (Fin b) → Prop) [DecidablePred Q]
  (w : Gap b L.succ k) : ℕ := by
  induction k;exact ((ite (P w.1 ∧ Q w.1) 1 0))
  exact (ite (P w.1)) (dite (n ≥ L.succ)
    (fun h ↦                              n_ih (Gap_nil         h) )
    (fun h ↦ List.sum (List.ofFn (fun a ↦ (n_ih (Gap_cons a w h)))))) 0
```

The function `num_by_backtracking` is recognized as decidable by Lean, so that we can use it for computations. To use the results in theorems we want to specify exactly what the function does, and under what conditions. For instance, if `P` is not monotone then `num_by_backtracking` is useless. Thus, we use the following type.

```
structure MonoPred (b:ℕ) where
  P : List (Fin b) → Prop
  preserved_under_suffixes (u v : List (Fin b)): u <:+ v → P v → P u
  Q (l: List (Fin b)) : Prop := True
```

Thus, `Q` is given a default value which can be overridden as desired.

In `backtracking_verification` we characterize what `num_by_backtracking` does when `P` is monotone. The current proof is several hundred lines of code. The parameter `b` is the branching rate of a tree.

```
theorem backtracking_verification {k b L:ℕ}
  (bound : k ≤ L.succ) (M:MonoPred b)
  [DecidablePred M.P] [DecidablePred M.Q]
  (w : Vector (Fin b) (L.succ-k)):
  Fintype.card {
    v : Vector (Fin b) L.succ // (M.P v.1 ∧ M.Q v.1) ∧ w.1 <:+ v.1
  }  = num_by_backtracking M.P M.Q w
```

In addition to finding the number of solutions, we are interested in finding the set of solutions,

```
   def those_with_suffix' {k b :ℕ} {L:ℕ} (P: List (Fin b) → Prop)
    [DecidablePred P]
  (Q: List (Fin b) → Prop) [DecidablePred Q] (w : Gap b L.succ k) :
    Finset (Gap b L.succ 0) :=
by
  induction k;exact ((ite (P w.1 ∧ Q w.1) {w} ∅))
  exact (ite (P w.1)) (dite (n ≥ L.succ)
        (fun h ↦ n_ih (Gap_nil h))
        (fun h ↦ biUnion (univ : Finset (Fin b)) (
            (fun a ↦ (n_ih (Gap_cons a w h)))))) ∅
```

and verifying its basic property:

```
theorem verify_those_with_suffix' {k b :ℕ} {L:ℕ} (bound : k ≤ L.succ)
{M:MonoPred b} [DecidablePred M.P] [DecidablePred M.Q] (w : Gap b L.succ
    k) :
  those_with_suffix' M.P M.Q w = filter (
    fun v : Gap b L.succ 0 ↦ M.P v.1 ∧ M.Q v.1 ∧ w.1 <:+ v.1
  ) univ
```

Now, for the application of backtracking to protein folding. We start with the four available direction in 2-dimensional rectangular folding.

```
def rectMap (a : Fin 4) : ℤ×ℤ := match a with
  | 0 => (1,0) | 1 => (-1,0) | 2 => (0,1) | 3 => (0,-1)
def rect (a : Fin 4) (x: ℤ×ℤ) : ℤ×ℤ := x + rectMap a
```

Folding of hexagons, or equivalently folding in a triangular lattice, can be defined in such a way that it is already embedded in the plane:

```
def hexMap (a : Fin 6) : ℤ×ℤ := match a with
  | 0 => (1,0) | 1 => (-1,0) | 2 => (0,1)
  | 3 => (0,-1)| 4 => (1,1)  | 5 => (-1,-1)
def hex (a : Fin 6) (x: ℤ×ℤ) : ℤ×ℤ := x + hexMap a
```

Perhaps a more graph-theoretic definition would seem more canonical, but this definition is ready for use in a theorem.

The function `points_tot` defines the protein folding score for a word `ph` (short for hydrophobic or "H") as the cardinality of a set. In this and subsequent definitions and theorems we require `A B : Type`, where `A` is the underlying space, typically $\mathbb{Z} \times \mathbb{Z}$ and `B` is the set of possible moves, typically `Fin b` for some `b`:ℕ such as $b = 4$. Further, `(go : B → A → A)` defines the moves, `l`:ℕ is a length, `(fold : Vector A l)` consists of the embedded locations, `(ph : Vector Bool l)` (or `l.succ`) is `true` for hydrophobic amino acids. The moves are given as `(moves: Vector (Fin b) l)`. In the more recent versions of the code we replace `Vector (Fin b) l` by `Fin l → Fin b`. We assume `[DecidableEq A]`, `[OfNat A 0]` (the fold is assumed to start at the "origin") and `[Fintype B]`. Moreover, we assume `open Finset` has been invoked.

```
def points_tot := card (filter (fun ik : (Fin l) × (Fin l) ↦ ((pt_loc
    go fold ik.1 ik.2 ph): Prop)) univ)
```

Here, `pt_loc` is:

```
def pt_loc (i j : Fin l) :=  ph.get i && ph.get j && i.1.succ < j.1 &&
    nearby go (fold.get i) (fold.get j)
```

The `nearby` function,

```
def nearby (p q : A) : Bool := ∃ a : B, q = go a p
```

is symmetric in most models of protein folding, but not in the *prudent* rectangular model in which the only allowed moves are right, left, down. One of the main formalized results states that the score for a word of length ℓ in a folding model with b many possible moves is at most $\frac{1}{2}\min\{b(\ell+1), \ell(\ell-1)\}$.

```
theorem pts_earned_bound (hP: Symmetric (fun x y ↦ nearby go x y))
  (path_inj : (Function.Injective fun k => (path_v go moves).get k))
  : pts_tot' go ph (path_v go moves) ≤ (min (l.succ * b) (l * l.pred)) /
    2
```

The first bound $b(\ell+1)/2$ is obtained by a Handshake Lemma type argument. The Handshake Lemma is already in Mathlib [8] and could have been used. The proof involves some manipulations of `Finset` types and their cardinalities, and a choice function of the following type:

```
def choice_ex (P : B → Fin l → Prop)
:   (filter (fun a ↦ ∃ i, P a i) (univ : Finset B))
 →   (filter (fun i ↦ ∃ a, P a i) (univ : Finset (Fin l)))
```

In the intended application, `P a i` means that there is a match in the protein folding in direction a starting from location i. With appropriate bijectivity assumptions we obtain:

```
theorem choice_ex_finset_card {P : B → Fin l.succ → Prop}
(h_unique_loc_dir : (∀ {a i₀ i₁}, P a i₀ → P a i₁ → i₀ = i₁)
                  ∧ ∀ {i a₀ a₁}, P a₀ i → P a₁ i → a₀ = a₁):
card (filter (fun a ↦ ∃ i, P a i) univ) =
card (filter (fun i ↦ ∃ a, P a i) univ)
```

Another pair of main results proved is the following inequalities: $P_{tri} \leq P_{rect} \leq P_{hex}$. These follow from a theory of embeddings of folding models:

```
def is_embedding {b₀ b₁ : ℕ} (go₀ : Fin b₀ → A → A)
(go₁ : Fin b₁ → A → A) (f : Fin b₀ → A → Fin b₁) :=
∀ i : Fin b₀, ∀ x : A, go₀ i x = go₁ (f i x) x
```

The computable function `HP` gives the optimal score for a given word:

```
def HP (ph : Vector Bool l.succ) :ℕ := Nat.find (pts_tot_bound_exists go
    ph)
```

Thus, the optimal score is the least number that serves as an upper bound on the score.

Orderliness and walks. For computational speed it is important to restrict attention to *orderly* [9] walks. This means that for two walks that are identical up to symmetry, we only consider one of them. Proving that this can be done without loss of generality seems to require a flexible definition of these walks. The best way to do so is something to be determined, although several candidates have been tried. A similar problem arises in automata theory. Mathlib has an implementation of deterministic finite automata [10], but not yet the sequence of states visited by a DFA when reading a given word. Nevertheless, some Lean exercises in this vein were included in a recent book [7].

One trick is to consider walks *implicitly* rather than *explicitly* defined. That is, a walk is simply a sequence of points in $\mathbb{Z} \times \mathbb{Z}$ such that each point is obtained from the previous by one of the permissible moves.

```
  def isPathWitness (fold : Fin l.succ → A) := (i : Fin l) →
    { a : B // fold i.succ = go a (fold (Fin.castSucc i))}
```

This is simpler than saying that a walk is defined inductively from a sequence of moves. An example of this definition in action:

```
theorem orderliness_rotate (ph : Fin l.succ.succ → Bool)
(fold : Fin l.succ.succ → ℤ×ℤ) (f : isPathWitness rect fold)
: ∃ fold′, ∃ _ : isPathWitness rect fold′,
  pts_tot′F rect ph fold = pts_tot′F rect ph fold′
  ∧  fold′ 1 = rect 0 (fold′ 0)
```

Another possibility is to restrict attention to those protein folding models where each step can be viewed as adding a vector:

```
def path {A:Type} [AddCommMonoid A] {B : Type} (goMap : B → A)
  {l : ℕ} (moves : Fin l → B) : Fin l.succ → A :=
  fun i ↦ sum univ (
    fun (j : Fin i) ↦ (goMap (moves (⟨j.1, Fin.val_lt_of_le j
    (Fin.is_le i)⟩))))
  )
```

This does not cover the case of the brick wall lattice, where "go right, then vertically" is not the same as "go vertically, then right".

Conclusion

Doing mathematics research and formalization work simultaneously can be synergetic, and protein folding models are a great target for formalization. We are still interested in finding an optimal way to work with inductively defined paths.

References

1. Agarwala, R., Batzoglou, S., Dančík, V., Decatur, S.E., Farach, M., Hannenhalli, S., Muthukrishnan, S., Skiena, S.: Local rules for protein folding on a triangular lattice and generalized hydrophobicity in the HP model. In: Proceedings of the First Annual International Conference on Computational Molecular Biology. p. 1–2. RECOMB '97, Association for Computing Machinery, New York, NY, USA (1997). https://doi.org/10.1145/267521.267522, `https://doi.org/10.1145/267521.267522`
2. Aichholzer, O., Bremner, D., Demaine, E.D., Meijer, H., Sacristán, V., Soss, M.: Long proteins with unique optimal foldings in the H-P model. Computational Geometry **25**(1), 139–159 (2003). https://doi.org/https://doi.org/10.1016/S0925-7721(02)00134-7, `https://www.sciencedirect.com/science/article/pii/S0925772102001347`, European Workshop on Computational Geometry - CG01
3. Berger, B., Leighton, T.: Protein folding in the hydrophobic-hydrophilic (hp) is np-complete. In: Proceedings of the Second Annual International Conference on Computational Molecular Biology. p. 30–39. RECOMB '98, Association for Computing Machinery, New York, NY, USA (1998). https://doi.org/10.1145/279069.279080, `https://doi.org/10.1145/279069.279080`
4. Crescenzi, P., Goldman, D., Papadimitriou, C., Piccolboni, A., Yannakakis, M.: On the complexity of protein folding. Journal of Computational Biology : a journal of computational molecular cell biology **5**, 423–65 (02 1998). https://doi.org/10.1089/cmb.1998.5.423
5. Dill, K.A.: Theory for the folding and stability of globular proteins. Biochemistry **24**(6), 1501–1509 (1985). https://doi.org/10.1021/bi00327a032, `https://doi.org/10.1021/bi00327a032`, pMID: 3986190
6. Kjos-Hanssen, B.: Lean code for "Protein folding by recursive backtracking". `https://github.com/bjoernkjoshanssen/protein`, Accessed: 2024-04-04
7. Kjos-Hanssen, B.: Automatic Complexity. De Gruyter, Berlin, Boston (2024). https://doi.org/doi:10.1515/9783110774870, `https://doi.org/10.1515/9783110774870`
8. Miller, K.: Degree-sum formula and handshaking lemma (2020), `https://github.com/leanprover-community/mathlib/blob/master/src/combinatorics/simple_graph/degree_sum.lean`
9. Shallit, J.: The logical approach to automatic sequences—exploring combinatorics on words with `Walnut`, London Mathematical Society Lecture Note Series, vol. 482. Cambridge University Press, Cambridge (2023)
10. Thomson, F.: Deterministic Finite Automata (2020), `https://leanprover-community.github.io/mathlib4_docs/Mathlib/Computability/DFA.html#DFA`