Protein Folding

Algorithms in BioInformatics 2
Mandatory Project 2

Magnus Erik Hvass Pedersen (971055)
December 2004, Daimi, University of Aarhus

1 Introduction

The purpose of this report is to verify attendance of the author to the *Algorithms in BioInformatics* course part II, at the department of computer science, University of Aarhus.

The reader is assumed to be familiar with the problem description as well as the course litterature. Following an outline of the problem, a heuristic algorithm for finding a viable solution is given, and then the actual programming is described along with experimental results.

2 Protein Folding

Consider a string $S \in \{h, p\}^*$ that is to be folded inside a lattice, in such a way that it is self-avoiding, and the number of non-connected but adjacent h's is maximized. This may be posed as a minimization problem instead, by attributing a score of -1 to each pair of such h's. Denote the total score of a folding x of the string S, as the function $f_S(x)$ which takes its values in $\mathbb{N}_-$.

3 Genetic Algorithm

As the set of possible foldings grows exponentially with the length of the protein-sequence S, a heuristic approach for minimizing f_S is needed for all but the shortest sequences. One approach is known as a *Genetic Algorithm* (GA) inspired by Darwinian evolution, in which a population of solutions is maintained, and in each iteration of the algorithm, a new population is created by combining fit agents from the parent population and altering their features at random. The three operators in play are known as *selection*, *crossover*, and *mutation*, and are detailed below for protein foldings.

The GA described here is plain vanilla without any bells or whistles. It is inspired by the GA given in [1], but is not an exact replica.

3.1 Encoding

To use a GA, we need to encode a solution x in each agent of its population. For a protein folding, the characters in S have upward, downward, leftward, or rightward connections to its neighbouring characters. However, as becomes evident below, it makes more sense to encode a folding with relative directions,

that is, whether the folding continues, turns left or turns right. The set of possible sequences of foldings is therefore given by:

$$F = \{c, l, r\}^{|S|-1}$$

where $|S| - 1$ is the number of foldings for a protein of length $|S|$.

Since it is meaningless to *continue* for the first fold, we always start a folding sequence with a right-turn, so that $x[1] = r$ for all foldings x. Because of symmetry of foldings, this is no limitation.

3.2 Selection

Selecting a better fit solution, is simply done by considering two randomly picked solutions, and choosing the best fit. This is known as *tournament selection* with a tournament-size of 2.

3.3 Crossover

Given two parent foldings $x, y \in F$ selected from the GA population as described above, and of equal length: $|x| = |y|$, we may pick an index i randomly from $\{1, \cdots, |x|\}$, and create the child z from concatenating substrings of x and y:

$$z = x[1 \cdots i]y[i + 1 \cdots |y|]$$

Since the enconding of a folding is relative, this concatenation does not require $x[i]$ and $y[i + 1]$ to be adjusted, as it would with absolute encoding, e.g. if $x[i]$ was a right-fold and $y[i + 1]$ a left-fold, then z would be invalid already.

3.4 Mutation

Given a string x, we decide randomly for each element x_i whether or not to mutate it. That is, for each x_i of x, if $(r < p_m)$ then let x_i be a randomly picked element from F, and with $r \sim U(0, 1)$ being picked at random for each x_i. Because the coding is relative, this means that we *wrap* the left- and right-hands of x angularly around the mutated point x_i.

3.5 Validity & Initialization

That the parental foldings are valid (that is, they are self-avoiding walks in the lattice), is no guarantee for the child of a crossover or the result of mutation to be valid. Therefore, the steps in the algorithm below that employ crossover and mutation, are iterated until a valid solution is created. However, much fewer iterations are required if the parental strings are valid, than if they are not. This also means that valid initial foldings should be chosen, to avoid the first iteration of the GA taking too much time.

Randomly generating the initial foldings until valid ones are found, obviously does not solve this problem, and merely becomes increasingly time-consuming

with longer protein sequences. A simple solution is therefore to let the first fold be a right-turn, and then let the rest be continues, which yields a folding having zero fitness. A more interesting way of initialization is used here, which makes a spiral for each initial folding, that usually has a decent fitness for a deterministic guess.

3.6 Algorithm

The following algorithm performs a single optimizational run of the GA:

- Initialize the agents in the population P as described in section 3.5, and compute their fitnesses.
- Repeat the following a number of times:
  - Create a temporary and initially empty population P'.
  - Repeat the following until $|P'| = |P|$: If $(r < p_c)$ then find two agents x and y from P by tournament selection from section 3.2 (with $r \sim U(0,1)$ being picked for each evaluation of the condition), and create z by crossover of x and y. Otherwise simply find z from P by tournament selection. In either case, if $(r < p_m)$ then mutate z (again with $r \sim U(0,1)$ being picked for each evaluation of the condition). Finally, if z is valid, compute its fitness and add it to P'.
  - Replace the population with the newly created: $P \leftarrow P'$.

The algorithm implemented here, furthermore ensures that the best known solution is always added to P', which is known as an *elitist* GA. Also note that each z that is invalid, still counts as an agent-step below.

4 Implementation

The above algorithms are implemented in *MS Visual C++ .NET*. The GA implementation is found in `LSwarmGA` which derives from `LSwarm`, both of which are taken from a more general numerical optimization implementation, simplified and specialized for the present work.

Agents in the GA population are instances of the `LVector`-class, which is sub-classed to `LVectorPF`, implementing crossover and mutation for strings over the folding-alphabet F.

The fitness of an agent in the GA population, is evaluated through an `LProteinFolding`-object, which implements filling of the lattice etc. This has $O(n)$ running-time for each fitness-evaluation, and has $O(n^2)$ space-usage with $n = |S|$. The lattice is implemented as an $(n+1) \times (n+1)$ double-array which uses simple modulo-arithmetics to allow the folding to work in every direction, starting at entry $(0,0)$. For very long protein sequences however, it should be possible to use a set- or map-like data-structure to achieve $O(n)$ space-usage and $O(n \cdot \log(n))$ time-usage.

Conversion between relative and absolute formats of foldings, as well as textual representations, is provided in the file `FoldingCoding.h`, and is achieved through simple arithmetic operations instead of using `switch`-constructs.

4.1 Testing

Testing was performed in steps during development. First `LProteinFolding` was tested with protein sequences and known foldings, primarily taken from [1], and then some erroneous foldings also. When no more errors were uncovered, conversion between the different formats of folding sequences was tested and checked by hand. Then `LVectorPF` was implemented, and simple iterative algorithms were created and tested on the protein sequences below. Finally, the GA algorithm was implemented and modified from a previous (real-coded) implementation, and various aspects were tested and modified, until the algorithm was suitable for protein folding.

5 Experimental Results

For these experiments, the GA population has 100 agents, and the parameters are: $p_m = 3/|S|$ and $p_c = 0.3$. For all protein sequences, 10 optimizational runs of the GA algorithm are performed, each with approximately $|S| \cdot 30000$ agent-steps. The best found solutions for the proposed protein sequences, are as follows:

1. The best found folding for the first protein sequence, is:

$$\begin{aligned} S &= hphpphhphpphphhpphph \\ x &= ruuldlululddrdldrrul \\ f_S(x) &= -9 \\ t &= 19 \text{ seconds} \end{aligned}$$

   Which is an optimal folding.

2. The best found folding for the second protein sequence, is:

$$\begin{aligned} S &= hhhpphphphpphphphpph \\ x &= ruuldlddrdrururuuldr \\ f_S(x) &= -9 \\ t &= 19 \text{ seconds} \end{aligned}$$

   Where the optimal folding would have fitness -10.

3. The best found folding for the third protein sequence, is:

$$\begin{aligned} S &= ppphhpphhpppppphhhhhhhpphhpppphhpphpp \\ x &= rrdlldrrrrddddllurulldrdlldrrrdrurru \\ f_S(x) &= -13 \\ t &= 53 \text{ seconds} \end{aligned}$$

Where the optimal folding would have fitness -14.

4. The best found folding for the fourth protein sequence, is:

$$\begin{aligned} S &= hhphphphphhhhphpppphpppphpppphpppphpppphphhhhphphphphh \\ x &= rdlluururdrurdrdllddllldllldluullururdrdruruulldru \\ f_S(x) &= -20 \\ t &= 99 \text{ seconds} \end{aligned}$$

   Where the optimal folding would have fitness -21.

5. The best found folding for the fifth protein sequence, is:

$$\begin{aligned} S &= hhhhhhhhhhhhphphpphhpphhpphpphhpphhpphpphhpphhpphphphhhhhhhhhhhh \\ x &= rdlluurrrddddllllluuuurrrurdrrrrurdrrdlddldduldluluuurrrrdllldrru \\ f_S(x) &= -34 \\ t &= 155 \text{ seconds} \end{aligned}$$

   Where the optimal folding would have fitness -42.

Which is slightly worse than the GA in [1].

References

[1] Ron Unger and John Moult. Genetic Algorithms for Protein Folding Simulations. Journal of Molecular Biology (1993) 231, 75-81.