Protein Aggregation Optimization: An Algorithmic Approach

Brandon Plost

December 4, 2009

Parkinson's, Alzheimer's, and Huntington's Diseases are primarily caused by a biological phenomenon called protein aggregation. Protein aggregation occurs when multiple proteins stick together, which renders the proteins useless and also blocks other proteins from performing their job. If we can understand how to prevent protein aggregation, we can gain insight into how to treat related diseases. Since mutations in proteins can change the probability of that protein to aggregate, we want to find the set of substitutions that gives the lowest such probability. It is infeasible to attempt to optimize the aggregation propensity with mutations in a laboratory because it could take up to six months to make one mutant, so computational optimization is the only alternative. Thus, we think of a protein as a string with a specific alphabet, and we think of the probability that aggregation occurs as a real-numbered score associated with a given string. Different character substitution in the string (mutations in the protein) can change the aggregation score. We present an algorithm that lowers aggregation score under biological constraints; the algorithm limits outputs to those which have the same biological function as the input string, minimize the Hamming distance of the input and output string, and minimize running time. Our implementation of the algorithm minimizes the aggregation score of the input string by an average of 40 percent on a set of over 30 test strings that come from the yeast organism S. cerevisiae. *To achieve this result, we experimented with optimizing pieces of the algorithm which have an effect on the output: threshold variables which determine equivalence classes, equations which choose one string over another and which determine internal state changes, and the maximum number of steps allowed. However we have not only created a procedure that can lower aggregation probability, but a procedure that can be generalized to a broader set of optimization problems beyond protein aggregation.*

1 Introduction

1.1 Background Information

Proteins are essential parts of life and participate in almost every process within cells. Protein aggregation occurs when proteins adhere to each other, thus rendering them useless. These clumps also block non-clumped proteins from doing their job within the cell. For example, protein aggregation in humans causes Alzheimer's, Parkinson's, and Huntington's Diseases. Furthermore, over 30 proteins in the yeast *Saccharomyces cerevisiae* are known to aggregate. The study of cells in yeast can provide clues to the causes and possible treatments for the diseases caused by protein aggregation in humans. There is also a basic science motivation for studying protein aggregation; changing the degree to which proteins aggregate allows us to study properties of proteins independently from aggregation. For example, some cystosolic proteins form punctate foci in yeast under starvation-like conditions; one theory states that these punctate foci contain aggregated proteins. Different mutants with increased or decreased propensity to aggregate must be realized in the laboratory and monitored under fluorescent microscopy to evaluate this theory.

The amount of aggregation a protein experiences depends upon the amino acids that comprise that protein. Each mutation in a protein can change the probability that the protein aggregates. One method to minimize the aggregation propensity of a protein would be to express numerous mutants in the laboratory and determine which aggregates the least. However, optimizing the aggregation of proteins is infeasible in a laboratory, taking up to six months to create just one mutant. Thus we want to computationally optimize proten aggregation propensity. For the computational problem, we can think of a protein as a string with an alphabet of size 20, where each character represents one of the 20 essential amino acids; we think of a mutation as a character substitution.

As a note, proteins mutate in three main ways: substitutions, insertions, and deletions. We make a simplifying assumption that proteins only mutate by substitutions. This assumption will not create invalid outputs, but rather leaves a part of the search space unexplored. Therefore the aggregation scores of our algorithm's outputs may be further optimized by looking at insertions and deletions.

1.2 The Problem

Mathematically, our goal is to find an algorithm, call it AGG-MIN, that takes three inputs: a function F that assigns aggregation scores, a function Q that determines equality of biological function, and a string p_0 that represents a protein, and outputs a string p_{out} with a lower aggregation score

than p_0. We see that

$$\text{AGG-MIN} : P \times \hat{F} \times \hat{Q} \rightarrow P$$

where P is the set of all strings that can represent proteins, $\hat{F}$ is the set of all F functions, and $\hat{Q}$ is the set of all Q functions.

The input function F will be used to determine the aggregation score for the protein strings. F takes as input a string p and outputs a positive real number r representing the aggregation score of the protein described by p. So

$$F : P \rightarrow \mathbb{R}^+$$

where P is the set of strings that can represent proteins. In our implementation we use the TANGO [2][5][7] algorithm as the F function in which lower aggregation score represents a lower propensity of the protein to aggregate. The Q input is a boolean function Q that will determine whether two proteins have the same biological function. Mathematically,

$$Q : P \times P \rightarrow \{true, false\}$$

This function is necessary for meaningful output because we want to isolate changes in aggregation propensity from changes in biological function.

Since the difficulty of expressing proteins in a laboratory depends directly upon the number of mutations in the output protein from the original, we want AGG-MIN to minimize the Hamming distance of p_0 and p_{out}. To create a well defined optimization problem, we take the minimization of Hamming distance as secondary to the minimization of aggregation score. So the minimization problem has three distinct goals:minimize F score, maintain the biological function of the input, and minimize Hamming distance.

1.3 Computational Complexity

For any given input string, p_0, we can write the length of p_0 as L_{p_0}. Since the alphabet we are working with has a size of 20 letters, the total number of possible strings with a Hamming distance of one from p_0 (single-substitutions) is $19 \cdot L_{p_0}$. Furthermore, $20^{L_{p_0}}$ strings exist with different combinations of substitutions.

We see that the search space of the problem is exponential, since we must look at $20^{L_{p_0}}$ mutant strings to find the one with the overall minimal aggregation score. Algorithms that run in exponential time are infeasible to compute on modern computers. However, a majority of strings in the search space will not preserve the biological function of p_0. We define a *mutable position* to be

an index i in p_0 in which any character substitution does not alter the biological function. Thus, if there are m mutable positions ($m \leq L_{p_0}$), then we need only consider 20^m mutants.

1.4 Contributions

In this thesis, we formalize the protein aggregation problem in order to have a well-defined optimization problem to work with. We use the simulated annealing heuristic to minimize the aggregation score of any input protein, and our implementation of the algorithm minimizes this score by an average of 40% on a set of over 30 test strings that come from the yeast organism *S. cerevisiae*. To achieve this result, we experimented with optimizing pieces of the algorithm which have an effect on the output: threshold variables which determine equivalence classes, equations which choose one string over another and which determine internal state changes, and the maximum number of steps allowed. Since the procedures for creating protein mutants in the laboratory are time and labor intensive, this algorithm provides insight that would be otherwise unattainable. Scientists can use the output from our algorithm to determine how to cure diseases genetically, to create medicines that modify the proteins so they aggregate less, or to gain general insight about protein aggregation.

2 The Algorithm AGG-MIN

2.1 The Q Function

There is currently no algorithm for taking as input two protein strings and determining whether they have the same biological function. However, well-tested proxies for the Q function do exist. We use one of these proxies along with domain knowledge to determine a set of indices in the string in which no mutation will alter the biological function.

2.2 Outline of AGG-MIN

Input: A string p_0 and a function F as described above

1. Determine the set MP of mutable positions
2. Score all single substitutions with F, mutating the input only in positions $x \in MP$
3. Using the scores from step (2), determine the set GMP of good mutable positions

4. Use optimization heuristics to determine the best combination of mutations

   Output: Among the strings with minimal F score, a string p_{out} with minimal Hamming distance from p_0 and with $Q(p_0, p_{out}) = true$

2.3 Determining Mutable Positions

We use a biology domain tool called the Basic Local Alignment Search Tool (BLAST) [1][9] to determine which indices in an arbitrary input string are mutable. The BLAST algorithm aligns p_0 with a specified number of closest matches among all known protein strings. So

$$BLAST : P \times \mathbb{N} \rightarrow P^n$$

where P^n is a list of n elements from P. An alignment that does not allow gaps in known sequences is used because we are interested only in character substitutions, not insertions or deletions. This method is valid for determining mutable positions because the BLAST databases only contain those proteins found in nature, and therefore any mutation of the input string is valid in some string that is close to the input in nature. Furthermore, there is a very low probability that the first n strings returned by BLAST are *not* related to the input string. As a note, we see that if the input is a real protein, then p_0 will be one of the n strings returned by the BLAST algorithm.

For each index i in the strings returned by BLAST, AGG-MIN counts the number that differ from p_0 in that position. Define $\text{DIFF}^n_{p_0}(i)$ as the number of strings returned by BLAST that differ from the p_0 at index i. Thus $0 \leq \text{DIFF}^n_{p_0}(i) \leq n$, and of course $\text{DIFF}^n_{p_0}(i)$ is only defined for $0 \leq i < L_{p_0}$.

Now we need a boolean function to determine whether a given index is a mutable position. We define the function MUT as:

$$\text{MUT}^{n,t}_{p_0}(i) = \begin{cases} 1 \ (true) & \frac{\text{DIFF}^n_{p_0}(i)}{n} \geq t \\ 0 \ (false) & \frac{\text{DIFF}^n_{p_0}(i)}{n} < t \end{cases}$$

where t is some threshold value chosen such that $0 < t < 1$. A threshold value is needed because as n increases, the number of indices that have $\text{DIFF}_{p_0}(i) > 0$ also increases due to the nature of the BLAST algorithm. Such an increase occurs because the more strings the BLAST algorithm is asked to return, the more dissimilar the return strings will be from the input, since BLAST ranks

based on similarity of return strings to the input.

The MUT function determines which indices in the string are mutable and acts as a proxy for the Q function needed for the optimization. By focusing the algorithmic mutation on the indices i for which $\text{MUT}_{p_0}^{n,t}(i)$ is true, AGG-MIN eliminates a large number of indices in the string as mutation candidates.

Define the set MP of mutable positions as

$$MP_{p_0}^{n,t} = \{i : \text{MUT}_{p_0}^{n,t}(i) = true\}$$

and take m to be the number of mutable positions for a given input as

$$m = |MP_{p_0}^{n,t}|$$

For each position i where $\text{MUT}_{p_0}^{n,t}(i)$ is false, 19 single mutations are eliminated, and a factor of 20 is shaved off the number of strings with multiple mutations in the search space.

In the current implementation of the algorithm, $n = 20$ and $t = \frac{3}{5}$. These are somewhat arbitrary but created to both maximize results while minimizing false-positives. Testing of these variables is discussed later. We also discuss ways to improve these values through machine learning.

2.4 Scoring Single Mutations

Using the set of mutable positions, AGG-MIN produces a $20 \times m$ matrix of strings with all of the possible single character mutations. The algorithm then assigns a score to each of these strings using the F function. AGG-MIN searches through this matrix of single mutated strings and saves the sequence with the lowest score as a quick final step to this part of the process.

2.5 Finding Good Mutable Positions

We find that a very small percentage of the strings created in the previous step of the algorithm have a significantly different F score from the input string. Therefore, mutations at certain indices affect the score of the string significantly more than other indices (figure 1). We refer to such indices as *good mutable positions* and the set of such positions as GMP.

To find the set of good mutable positions, we define a variable d that serves as a constant difference percentage and will be used as a threshold. The difference variable d could be optimized with further research, just as n and t can be. In our implementation of the algorithm, we take $d = 1.05$ which represents a five percent difference bracket.

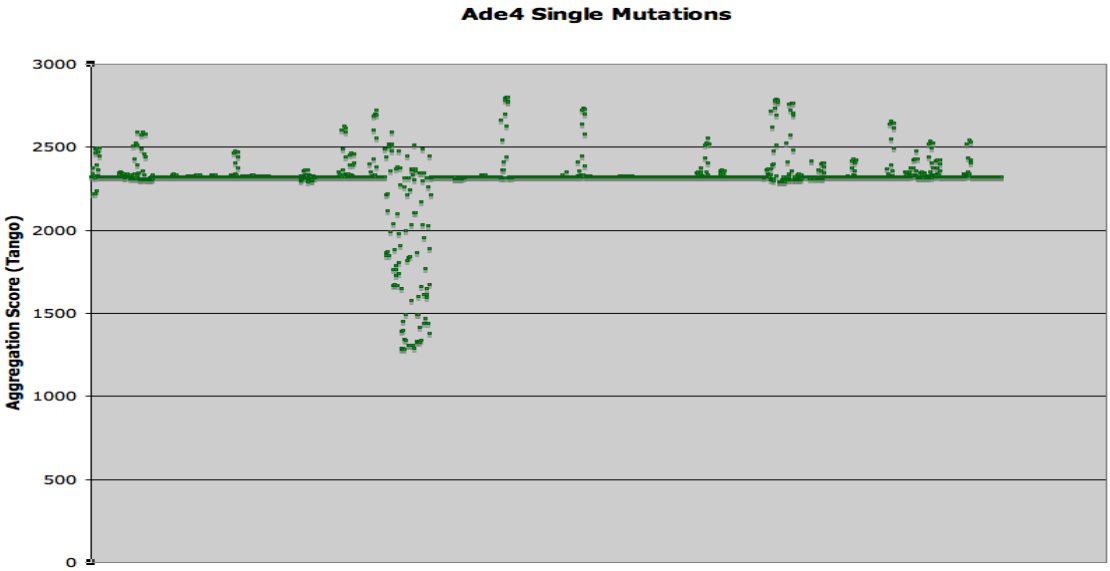


Figure 1: This plot shows the aggregation score for all single mutations of ADE4 that the algorithm looks at. Each set of nineteen consecutive points represents a given mutable position in the string. We see that mutations in certain indices affect the aggregation score of the string a large amount, and mutations in other positions do not affect the aggregation score at all. This gives us a way to eliminate a large number of positions so finding the optimal set of mutations is easier.

Using a brute force approach, AGG-MIN loops through all of the single character mutations and their scores to build the list of good mutable positions. When a mutated string has an aggregation score that is further than d from the score for the input, then the index at which that string was mutated is added to the list of good mutable positions. Formally,

$$GMP^d_{MP} = \{i : \exists p_i \; s.t. \; i \in MP \text{ AND either } d \cdot F(p_i) < F(p_0) \text{ OR } d \cdot F(p_0) < F(p_i)\}$$

where p_i is a string of length L_{p_0} with a mutation in position i.

Due to the existential in the formula, it is possible that there exist multiple strings p_i that would warrant adding a particular position to set of good mutable positions. As a precaution, the list is checked to make sure that a given position is not added more than once.

2.6 Multiple Character Mutations

Attempting to determine the combination of mutations that yields the lowest aggregation score is a complex computational problem. Even though the number of indices that need to be mutated in this step has been reduced, the problem still has a search space of size $20^{|GMP^d_{MP}|}$. We must devise a clever method to find the optimal value since this search space is exponentially large. We must also remember the main goal of the algorithm: among the strings with minimal F score and with the same biological function as the input, output the mutant with minimal Hamming distance from p_0.

Since we have eliminated indices that would change the biological function of the string, we are left with mutants with the same biological function as the input. Another characteristic of an optimal mutant is that it has minimal Hamming distance from the input. There are specific substitution matrices for calculating a protein-specific Hamming distance that accounts for the likelihood of each character substitution as well as the number of mutations in the string. We use the BLOSUM [4] substitution matrix to create our Hamming distance function B where

$$B_{p_0} : P \to \mathbb{Z}$$

This Hamming distance function outputs higher numbers for more likely sets of substitutions and lower numbers for less likely sets of substitutions. We know that

$$B_{p_0}(p_0) \geq B_{p_0}(p)$$

for any protein p (of length L_{p_0}) due to the nature of the substitution matrix.

Since AGG-MIN optimizes multiple variables, it needs a separate comparator method for determining if a given string is better than another in the context of an input string. Now define a selecting method, SEL where

$$\text{SEL}_{p_0} : P^2 \to P$$

as $\text{SEL}_{p_0}(a, b) =$

$$\begin{cases} a & \text{if } F(a) < \frac{F(b)}{d} \\ b & \text{otherwise if } F(a) > d \cdot F(b) \\ a & \text{otherwise if } F(a) < d \cdot F(b) \ \& \ B_{p_0}(a) \geq B_{p_0}(b) \\ b & \text{otherwise if } F(a) > \frac{F(b)}{d} \quad \& \ B_{p_0}(a) \leq B_{p_0}(b) \\ a & \text{otherwise if } F(a) \leq F(b) \\ b & \text{otherwise} \end{cases}$$

We note two important mathematical properties of the function SEL. First, SEL is reflexive, meaning that

$$\text{SEL}_{p_0}(a, a) = a$$

Second, the function SEL is symmetric, which means

$$\text{SEL}_{p_0}(a, b) = \text{SEL}_{p_0}(b, a)$$

If the selecting function does not have these properties then the results of the algorithm would not make sense. We must now define a method that uses this selecting function to find the mutant string that minimizes the aggregation score.

Simulated annealing [8] is an optimization heuristic used for computing a good approximation for the optimal value in a large, discrete search space. We can describe the simulated annealing process with a state machine; the initial state is the the input string. Each step of the algorithm moves the state machine to a random neighbor state (string with one mutation from the current state) with some probability that depends on the values of F and B and how many steps the algorithm has completed. The design of the algorithm, an adaptation of the Metropolis-Hastings algorithm[3][6], tends toward more random state changes during the early stages so that it does not become stuck at local minima.

SIMULATED-ANNEALING$(s_0, n_{max}, e_{thresh})$

```
1  s ← s_0; e ← F(s)                          ▷ Initial state & energy
2  s_best ← s; e_best ← e
3  n ← 0                                      ▷ Count iterations
4  while n < n_max & e_thresh < e
5      do                                     ▷ While time remains
6          s_new ← NEIGHBOR(s)
7          e_new ← F(s_new)                   ▷ New Best?
8          if SEL_{s_0}(s_new, s_best) = s_new
9            then                             ▷ Save new state
10                 s_best ← s_new
11                 e_best ← e_new
12         if M(s_0, s, s_new, e, e_new, n/n_max)
13           then                             ▷ Change state
14                 s ← s_new; e ← e_new
15         n ← n + 1
16 return s_best
```

The general simulated annealing heuristic is as follows: starting from state s_0 and continuing to a maximum of n_{max} steps, or until a state with aggregation score (energy) of e_{thresh} or better is achieved. The method call to NEIGHBOR(s) generates a random neighbor of a state s, and the method call to RANDOM generates a random floating point number in the range $[0, 1]$. The call to $M(s_0, s, s_{new}, e, e_{new}, \frac{n}{n_{max}})$ represents an acceptance probability for moving to the new state, s_{new}.

The above simulated annealing process above provides a good framework for the multiple mutation optimization. However, there are a few implementation details that need to be worked out. First, the process starts with s_0 as the input string p_0. Next the value of n_{max} must be determined; this bounds the amount of time spent in the simulated annealing process. AGG-MIN takes

$$n_{max} = 3 \cdot |GMP|^2$$

because it serves to maintain the worst-case asymptotic analysis of the algorithm while being large enough to produce good results in practice. We define the NEIGHBOR of a string $p \in P$ as a string $p' \in P$ reachable by exactly one character substitution in a good mutable position. At each step, the simulated annealing method considers some neighbor (s_{new}) of the current state (s) and probabilistically determines whether to move to the new state. For minimization, we specify $e_{thresh} = 0$ to circumvent getting a "good enough" answer since we already have a tight asymptotic bound using n_{max}.

It is difficult to define a method M that decides whether to move to the new state. We take a lesson from the original Metropolis approach to the Monte Carlo method; if $\text{SEL}_{s_0}(s, s_{new}) = s_{new}$ then the algorithm moves to the new state ($M = 1$, or $true$). In the case where $\text{SEL}_{s_0}(s, s_{new}) = s$ the algorithm moves to the new state with some probability. We formally define M as

$M(s_0, s, s_{new}, e, e_{new}, \frac{n}{n_{max}}) =$

$$\begin{cases} 1 & \text{if } \text{SEL}_{s_0}(s, s_{new}) = s_{new} \\ 1 & \text{if } \text{RANDOM} < \frac{1}{2} - \frac{3}{10} \cdot \frac{n}{n_{max}} - \frac{3}{5} \cdot \frac{e_{new} - e}{e} \\ 0 & \text{otherwise} \end{cases}$$

The definition of M follows the requirements specified by the simulated annealing process; M is non-zero when $e < e_{new}$ and in general the number of random state changes decreases as n increases. Furthermore, it follows convention that the probability of accepting a state change decreases when the difference between e_{new} and e increases. Therefore small uphill moves, or bad

state changes, are more tolerable than large uphill moves.

2.7 The Final Step - Putting Everything Together

The simulated annealing process produces a string $s_{best} \in P$ on which we can perform few quick checks to maximize the B score (minimize Hamming distance) of the final output. The first check takes

$$p_{best} = \text{SEL}_{p_0}(s_{best}, p_{best\ single})$$

where $p_{best\ single}$ is the single substitution with the best score. The AGG-MIN algorithm then takes p_{best} and attempts to replace each of the substitutions with the character from p_0 at that index. This replacement guarantees an increase in the B score and is only kept if the F score is within d of $F(p_{best})$. Therefore, the final F score will be within d of the aggregation score for p_{best}. After completing this relatively quick step of the process, AGG-MIN returns this optimal string (p_{out}).

2.8 Implementation Details

AGG-MIN is implemented in Perl 5.8 and uses the `Scalar::Util` library. There is a copy of the BLOSUM62 substitution matrix hard-coded into the algorithm for use as the B function. `Netblast` version 2.2.18 (Windows, Macintosh, or Linux), used for finding mutable positions, is available from the NCBI website at `http://www.ncbi.nlm.nih.gov/BLAST/download.shtml`. The algorithm we present uses the Tango algorithm, version 2.3 (Windows or Linux), available from the CRG website at `http://tango.crg.es`, as the F scoring function.

3 A Walkthrough for Gln-1

Here we give a walkthrough of AGG-MIN for one of the test inputs, the yeast protein gln-1. The first step of the algorithm finds that 62 of the 300 indices within gln-1 are mutable. The algorithm then calculates the F scores for the input string and the 1178 single substitution strings. The F score for gln-1 is 932.43, and the lowest single substitution's aggregation score is 515.36, which represents a maximum of a 44.7% change in F score by a single mutation. Of the 62 mutable positions found in the first step of the process, only nine of these are deemed good mutable positions (figure 2). Concretely,

$$GMP_{MP}^{1.05} = \{20, 22, 23, 24, 92, 96, 98, 185, 222\}$$

This set represents fewer than 3% of the positions in the input string. As specified by the algorithm we present,

$$n_{max} = 3 \cdot 9^2 = 243$$

After completing these 243 iterations, the algorithm eliminates those mutations it deems unnecessary, and outputs a string and corresponding score to a file. Due to its stochastic nature, each time the algorithm runs it produces a slightly different result, but over ten trials, the minimal aggregation scores were between 38 and 43, with an average of five substitutions. This represents an impressive aggregation score drop of over 95%. The best string found over these ten trial runs has an aggregation score of 40.15, with five mutations, and a B score of 21 (calculated only in the good mutable positions). It is clear that the algorithm is highly successful in minimizing the aggregation score as compared to the score of the input and even far below the score for the best single mutation string.

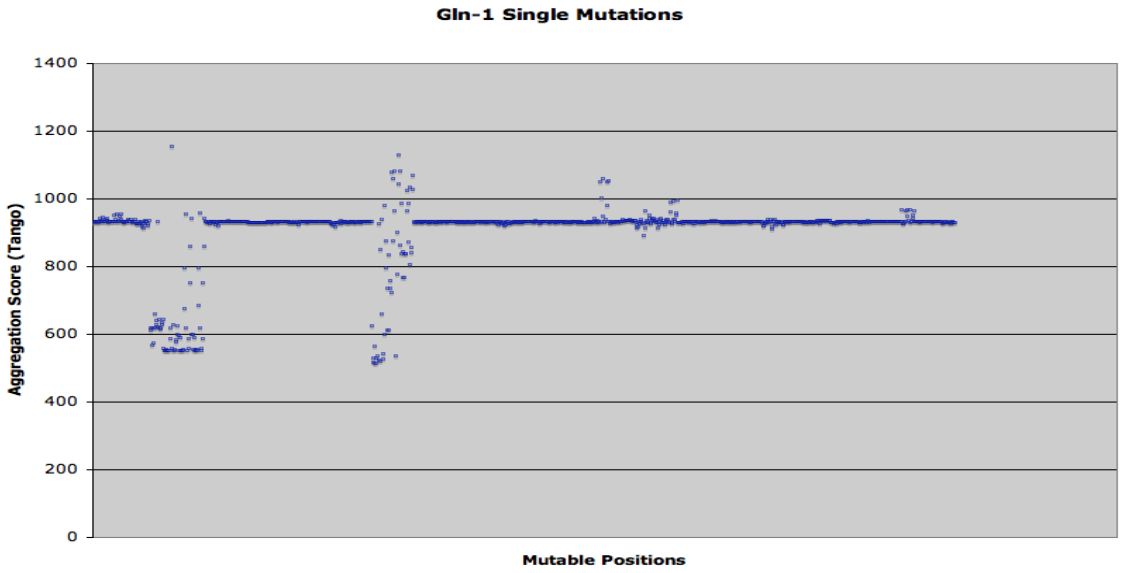


Figure 2: This plot shows the aggregation score for all single mutations of GLN1. Nine of the 62 mutable positions are determined to be good mutable positions, an 85 % drop of indices that the algorithm needs to mutate to find the optimal set of mutations. The best single mutant reduces the aggregation score by 45% and the output of AGG-MIN reduces the aggregation score by 95%.

4 Asymptotic Runtime Analysis

The worst-case runtime analysis of AGG-MIN as described and implemented is $\tau \cdot \Theta(n^2)$, where τ is the worst-case runtime analysis of the F algorithm. Therefore, the runtime of the algorithm we present is dependent upon the input algorithm. This dependency is one of the motivating factors for determining good mutable positions; to decrease the number of calls to the aggregation prediction algorithm. Although determining good mutable positions cannot change the worst-case analysis of the algorithm, it decreases the expected runtime of the algorithm. This asymptotic analysis also shows that the runtime of AGG-MIN is proportional to the square of the length of the input string. Without prior knowledge of the F algorithm, a better analysis than $T(n) = \tau \cdot \Theta(n^2)$ cannot be provided.

5 On the Performance of AGG-MIN

5.1 Objectives

Although we cannot formally prove that AGG-MIN finds the minimal value in the search space, we can comment how the algorithm satisfies each of its objectives:

1. Do not change biological function
2. Minimize F score
3. Maximize B score (minimize Hamming distance)

5.2 Changing Biological Function

Since no algorithm exists to predict the function of a protein, we cannot prove that the biological function of p_{out} is the same as that of p_0. However we use the most recognized way for determining indices that are important to the function by looking at which indices are conserved over the evolution of the protein. Thus the algorithm only mutates in positions which have mutated many times in nature. By using this well-known method we can be very certain that the function of the output and the input of AGG-MIN have the same biological function.

5.3 Optimization of F Score

We can easily prove that the output p_{out} of the algorithm has F score less than or equal to that of the input string p_0 because in the worst case the algorithm returns p_0. In the cases where the set of good mutable positions is non-empty, we should be able to say that the final score will be at least $d\%$ better than the input, or $F(p_{out}) \leq d \cdot F(p_0)$. Since we are also optimizing for B score, this property does not hold, as the F score is allowed to decrease by d %, which means that in the worst case (excluding no change), there will be a d^2 % change, which is far less since $1 < d < 2$ ($0\% < d < 100\%$). We know that the F score never grows by running the algorithm, but we cannot mathematically prove how well the optimization does due to the unknown nature of the F algorithm.

5.4 Maximization of B Score

Upfront we know that $B_{p_0}(p_0) \geq B_{p_0}(p_{out})$ because of the values in the BLOSUM substitution matrix. So if the algorithm guarantees the maximal B score, then no work would need to be done since the string with maximal B score is the input p_0. However, we output the string with the maximal B score among the set of strings with minimal F score. Strings with score within d of the one with the absolute minimal F score comprise this set. This reasoning inspired the logic in the selection function that consults the B score only when the two F scores are close. We can therefore conclude that the B score is maximal within the range minimal F score.

5.5 Minimization of Mutations

Minimizing the number of mutations is the last objective for the algorithm because the number of mutations for an optimal string is unknown. Minimization of mutations is favored since it makes realizing proteins in the laboratory easier and increases B score. Since the algorithm checks which mutations it can eliminate (without worsening the F score too much) as a finalization step, we conclude that the algorithm outputs a string with minimal Hamming distance from the input. Furthermore, any other method for minimizing mutations would interfere with the optimization process by favoring strings with fewer mutations, and inputs for which an optimal string has many mutations would never be found.

Sequence Name	|MP|	|GMP|	Single	Combinatorial	Total	Search Space Size
ADE4	74	14	1406	588	1994	$1.64 \cdot 10^{18}$
ADE5,7	157	32	2983	3072	6055	$4.29 \cdot 10^{41}$
ADE12	63	17	1197	867	2064	$1.31 \cdot 10^{22}$
ADE17	75	14	1425	588	2013	$1.64 \cdot 10^{18}$
ADH2	27	7	513	147	660	$1.28 \cdot 10^{9}$
ALA1	128	10	2432	300	2732	$1.02 \cdot 10^{13}$
ARC1	18	7	342	147	489	$1.28 \cdot 10^{9}$
CDC19	70	26	1330	2028	3358	$6.71 \cdot 10^{33}$
CDC60	197	21	3743	1323	5066	$2.10 \cdot 10^{27}$
CPR6	39	16	741	768	1509	$6.55 \cdot 10^{20}$
GLN1	45	16	855	768	1623	$6.55 \cdot 10^{20}$
GLN4	147	21	2793	1323	4116	$2.10 \cdot 10^{27}$
HSC82	12	2	228	40	268	400
HSP82	12	1	228	20	248	20
HTS1	26	4	494	48	542	$1.60 \cdot 10^{5}$
ILS1	182	26	3458	2028	5486	$6.71 \cdot 10^{33}$
IRA1	84	7	1596	147	1743	$1.28 \cdot 10^{9}$
KIC1	41	10	779	300	1079	$1.02 \cdot 10^{13}$
PAB1	243	50	4617	7500	12117	$1.13 \cdot 10^{65}$
RPL4B	43	8	817	192	1009	$2.56 \cdot 10^{10}$
RPS11B	15	2	285	40	325	400
SBP1	4	2	76	40	116	400
SSB1	40	24	760	1728	2488	$1.68 \cdot 10^{31}$
SSB2	41	19	779	1083	1862	$5.24 \cdot 10^{24}$
STI1	201	12	3819	432	4251	$4.10 \cdot 10^{15}$
THS1	35	4	665	48	713	$1.60 \cdot 10^{5}$
TPS2	67	11	1273	363	1636	$2.05 \cdot 10^{14}$
UGA1	111	21	2109	1323	3432	$2.10 \cdot 10^{27}$
UGP1	44	11	836	363	1199	$2.05 \cdot 10^{14}$
UGT51	245	93	4655	25947	30602	$9.90 \cdot 10^{120}$
VAS1	128	19	2432	1083	3515	$5.24 \cdot 10^{24}$
AVERAGE:	84.32	17	1602.13	1762.71	3364.84	$3.19 \cdot 10^{119}$

Table 1: This table shows the number of mutable positions and good mutable positions found for our test set of inputs when $n = 20$ and $t = 0.6$. The table also displays the number of single and combinatorial mutations run by the algorithm and the size of the combinatorial search space with substitutions only in the good mutable positions. The size of the search space is far too large in which to compute every value, so we use a heuristic to find a near optimal value in a feasible amount of time.

6 Testing AGG-MIN

Unless otherwise specified, all testing was run on an implementation of the algorithm that uses values $n = 20$, $t = 0.6$, and $d = 1.05$.

6.1 Difficulty of Testing AGG-MIN

It is hard to test AGG-MIN because there are no methods for determining the function of a protein without synthesizing it in the laboratory. One possible way to test the algorithm would be to synthesize some of the outputs in the lab, but this is both time and labor intensive, as creating one mutant in the lab can take up to six months. So this is not a feasible way to test the algorithm on a large scale, although it should happen at some point for some inputs. We provide different methods for systematically testing the different parts of the algorithm and the algorithm as a whole using a test set of over thirty inputs which are found in yeast. These inputs have all been shown to aggregate under different conditions and have large variations in biological function. We take biologically dissimilar inputs so that we can test if the algorithm does well on all inputs, not just those with certain properties.

6.2 Total Number of Mutations and Actual Running Times

Table 1 displays the number of single and combinatorial mutations that AGG-MIN scores at runtime, along with the total number of combinatorial mutations possible when performing substitutions only in good mutable positions. The algorithm minimizes the score of the inputs well despite only looking at a small fraction of the total search space. This reduction in search space directly relates to a reduction in the runtime of the algorithm, and we get the running time of the algorithm to be

$$T(m, g) = (19m + 3g^2) \cdot \tau$$

where m is the number of mutable positions, $|GMP| = g$, and τ is the running time of the F scoring algorithm. We compare this running time to the exponential size of the search space and find

$$\begin{aligned} 19m + 3g^2 &\ll 20^g \\ (19m + 3g^2) \cdot \tau &\ll \tau \cdot 20^g \\ T(m, g) &\ll \tau \cdot 20^g \end{aligned}$$

Sequence	Original Score	Best Single Score	Score Reduction
ADE4	2321.12	1286.64	44.57%
ADE5,7	3446.90	2842.85	17.52%
ADE12	1700.18	1493.44	12.16%
ADE17	2468.47	1563.94	36.64%
ADH2	1710.38	1113.61	34.89%
ALA1	4364.86	3882.86	11.04%
ARC1	1312.31	922.99	29.67%
CDC19	1818.91	1437.74	20.96%
CDC60	3292.16	2868.34	12.87%
CPR6	307.16	272.11	11.41%
GLN1	1203.32	617.27	48.70%
GLN4	1642.93	1176.31	28.40%
HSC82	3058.57	2885.66	5.65%
HSP82	3262.96	3118.80	4.42%
HTS1	1612.41	1370.09	15.03%
ILS1	4663.93	3908.94	16.19%
IRA1	30882.53	29503.69	4.46%
KIC1	1479.82	1170.23	20.92%
PAB1	626.33	393.88	37.11%
RPL4B	1629.31	1458.74	10.47%
RPS11B	99.94	78.21	21.74%
SBP1	381.82	360.48	5.59%
SSB1	477.27	307.43	35.59%
SSB2	716.42	356.42	50.25%
STI1	684.52	338.82	50.50%
THS1	1850.66	1467.03	20.73%
TPS2	3358.36	3003.28	10.57%
UGA1	1757.66	1127.00	35.88%
UGP1	1669.59	1235.13	26.02%
UGT51	5419.57	4902.44	9.54%
VAS1	3876.68	3339.69	13.85%
AVG:			22.69%

Table 2: This table shows the aggregation scores for the test inputs, along with the best single mutation scores and percent reductions from the original scores. AGG-MIN achieves slightly better than a twenty percent optimization by looking only at strings with a Hamming distance of one from the input, and we see that the best single mutation scores range anywhere from 4 to 50 percent. By looking at combinatorial mutations, we will find that we can reduce the aggregation score almost two times as much as with single mutations.

for all $g > 1$, since $m \geq g$ by definition (since $MP \subseteq GMP$). Therefore the algorithm reduces the problem to a manageable size; the algorithm is a heuristic with polynomial worst-case running time.

The average number of F scorings performed by the algorithm is less than 3400 over the test set of inputs, and the average search space is over $3 \cdot 10^{119}$. We find that on average about half of the computation time is spent in each the single mutation and multiple mutation phases. Since the computation time is evenly split, we would achieve much better performance *if* we could eliminate all or most of the combinatorial mutation step.

6.3 Single Mutations Only

We must run all of the single substitutions in the mutable positions, but what happens if we skip the combinatorial evaluation? We can see (table 2) that single mutations minimize the aggregation score by an average of about 22 percent, and the percent of score reduction ranges anywhere from four to fifty percent. We want to ensure that the combinatorial mutation can reduce the aggregation score more.

We observe that if two single mutations each give a good score, the string with both of those mutations will not necessarily have a better score, let alone a good score at all. Therefore the optimization is much harder because we cannot just find the best mutation in each position and combine them all to get the optimal global string. Furthermore, the single mutations alone do not optimize the inputs enough to just stop after this step in the algorithm, so we go on to examine combinatorial mutations of the input string.

6.4 The Simple Test

To test the AGG-MIN algorithm, we use a set of over thirty proteins found yeast which are known to aggregate. The main test we perform is the obvious test of the algorithm: run it over many different inputs and examine the results (table 3). Assuming correctness of our proxy to the Q function input, we judge these results based on our metrics for an optimal value. The percent of reduction in the aggregation score ranges between 4 to 95 percent, which tells us that either the algorithm does not find a good minimum for all inputs or that a good minimum does not exist for some inputs. Whatever the case for the individual inputs, it is clear that with an almost $40\ \%$ average aggregation score reduction, the AGG-MIN does its job well. Furthermore, with an average Hamming distance (number of mutations in each output) of slightly over four, these outputs can be more easily tested and implemented in the laboratory than mutants with a higher number of mutations.

Sequence Name	|MP|	|GMP|	% GMP	#Mutations	In Score	Out Score	Reduction
ADE4	74	14	18.9	3	2321.12	1148.67	50.51%
ADE5,7	157	32	20.4	11	3446.90	818.56	76.25%
ADE12	63	17	27.0	3	1700.18	1331.48	21.69%
ADE17	75	14	18.7	6	2468.47	747.31	69.73%
ADH2	27	7	25.9	2	1710.38	888.47	48.05%
ALA1	128	10	7.8	3	4364.86	3454.05	20.87%
ARC1	18	7	38.9	1	1312.31	922.99	29.67%
CDC19	70	26	37.1	8	1818.91	740.44	59.29%
CDC60	197	21	10.7	4	3292.16	2291.63	30.39%
CPR6	39	16	41.0	1	307.16	272.11	11.41%
GLN1	45	16	35.6	5	1203.32	62.19	94.83%
GLN4	147	21	14.3	7	1642.93	722.96	56.00%
HSC82	12	2	16.7	1	3058.57	2885.66	5.65%
HSP82	12	1	8.3	1	3262.96	3118.80	4.42%
HTS1	26	4	15.4	1	1612.41	1370.09	15.03%
ILS1	182	26	14.3	9	4663.93	2613.82	43.96%
IRA1	84	7	8.3	1	30882.53	29503.69	4.46%
KIC1	41	10	24.4	4	1479.82	893.79	39.60%
PAB1	243	50	20.6	11	626.33	118.75	81.04%
RPL4B	43	8	18.6	1	1629.31	1458.74	10.47%
RPS11B	15	2	13.3	1	99.94	78.21	21.74%
SBP1	4	2	50.0	1	381.82	360.48	5.59%
SSB1	40	24	60.0	3	477.27	233.74	51.03%
SSB2	41	19	46.3	4	716.42	248.49	65.32%
STI1	201	12	6.0	2	684.52	335.71	50.96%
THS1	35	4	11.4	1	1850.66	1467.03	20.73%
TPS2	67	11	16.4	5	3358.36	2627.29	21.77%
UGA1	111	21	18.9	7	1757.66	595.04	66.15%
UGP1	44	11	25.0	3	1669.59	1050.42	37.09%
UGT51	7245	93	38.0	14	5419.57	2331.10	56.99%
VAS1	128	19	14.8	7	3876.68	2156.11	44.38%
AVERAGE:	84.32	17	23.3	4.23			39.20%

Table 3: This table shows the size of sets within the algorithm for the test inputs. From this table we learn that with $n = 20$ and $t = 0.6$ we get an average aggregation score reduction of over 39 percent. Furthermore, the average Hamming distance between the input and output is slightly over four. Thus, the proteins will be easier to realize in a laboratory than if there were a mutation in each good mutable position, an average of 17 per output.

With combinatorial mutations, AGG-MIN minimizes the F score for the test set of proteins almost twice as much as with single mutations alone. This is impressive since we have seen that the single and combinatorial mutation phases of the algorithm each take about the same number of steps in practice. This larger score reduction also answers our question about whether we can eliminate the part of the algorithm that scores combinatorial mutations.

6.5 Determining the t Value

Determining a value for the threshold t that performs well requires a large amount of experimentation. Finding a value for t that is "just right" would be difficult since a value too low or too high would break certain aspects of the algorithm. If t is too small, then the set of mutable positions will be larger, meaning a larger set of good mutable positions and a longer running time. Furthermore, by allowing for a larger set of mutations to be valid, it is more likely that the output will be biologically dissimilar to the input, which would break our proxy to the Q function. If t is too large, the opposite problem occurs; the set of mutable and good mutable positions is smaller, resulting in a short computation time, which may not be enough to find an optimal solution in the undoubtedly large search space.

Since the t value, as used in AGG-MIN, only directly affects the set MP, the entire algorithm must be run to test a particular value of t on a particular input. As with many other parts of the algorithm, it is not known what a good value for t should realistically be from a biological standpoint. The only hint that we can take from biology into determining the correct value for t is that we would rather t be too small and produce an output with a less optimal F score than if t were large, producing an output that is less likely to have the same biological function as the input. Thus, we start from the high end of the $0 < t < 1$ range for the threshold and work from one to zero to determine which value gives the best overall results. Determining this value requires running an entirely new set of tests to determine the largest value of t that gives good results on average. We run the simple test again, with $t = 0.7,\ 0.8$, and 0.9. In figure 3 we can see the results for the average score reduction percentage in these tests.

We can now examine the graph in figure 3 to determine the best value for the threshold variable. The similar patterns in the lines which represent our different optimization criteria makes the interpretation much easier. For number of GMPs, mutations in the output, and percent score reduction, the lines each have a linear slope over the values $t = 0.7$ to 0.9. However the points where $t = 0.6$ change the slope of these lines drastically, which indicates that this is the largest t value that displays significantly better output than the value above it. The pattern of linearity between $t = 0.7$

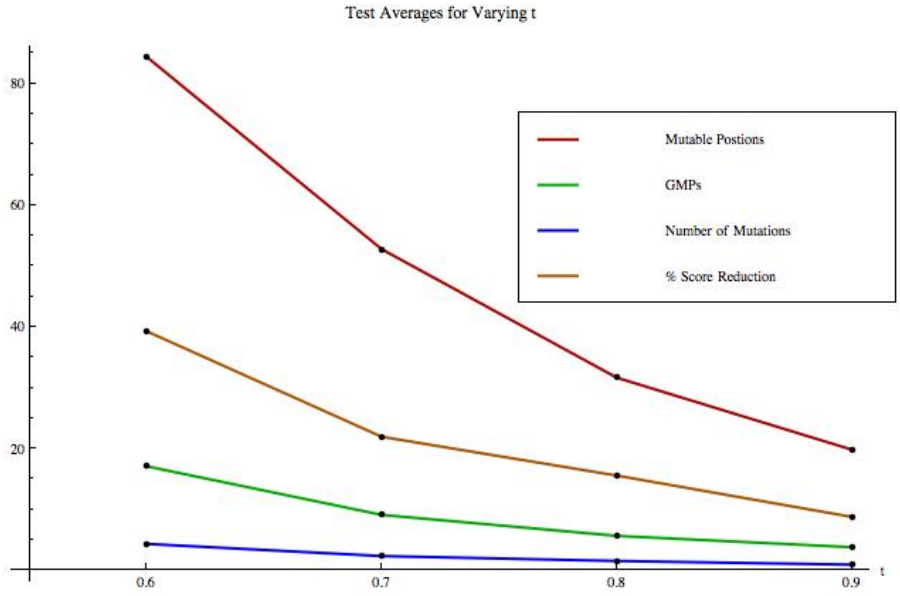
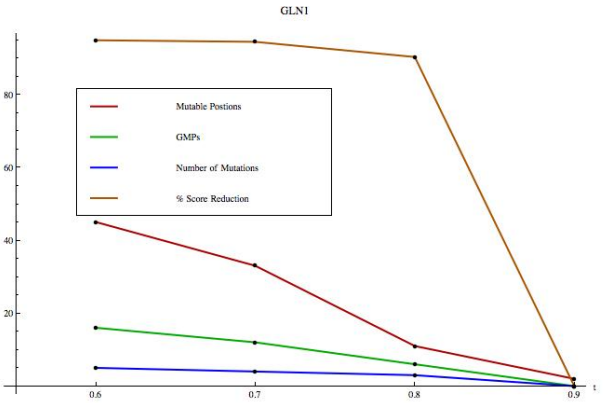


Figure 3: This graph shows the average values of pieces of the algorithm for varying t thresholds. We see that the values for everything except the number of mutable positions are close to linear for the values $t = 0.7$ through $t = 0.9$, meaning the score reduction follows the number of good mutable positions. $t = 0.6$ is the best threshold because of the change in this pattern at this value shows that each mutations has a larger effect on the aggregation score.

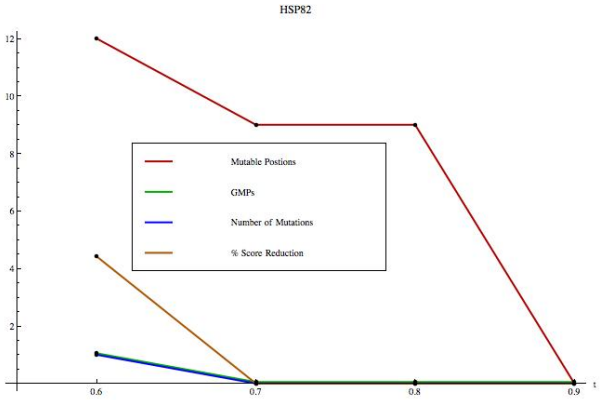
and $t = 0.9$ does not emerge in the line representing the number of mutable positions, but this metric still shows a slope increase when moving from $t = 0.7$ to $t = 0.6$. We have now found the largest value of t that shows a considerable difference in the average number of mutable positions, good mutable positions, and aggregation score.

Because the graph in figure 3 and corresponding values are based on averages from the test set of proteins, they may not accurately represent each individual input. Therefore we look at a select few inputs, each dissimilar in biological property, so we get a spread of inputs with varied characteristics. We pick the inputs that give the best, the worst, and an average optimization. These inputs are GLN1 (best), HSP82 (worst), and ADE4 (average).

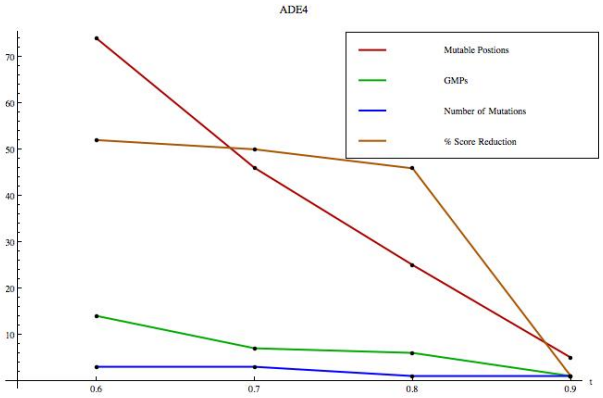
We first examine the graph for GLN1 (figure 4(a)), the input with the best optimization output from AGG-MIN. The number of good mutable positions and number of mutations in the output are both fairly linear for the varying values of t, consistent with the trend of the average. The number of mutable positions also follows the pattern of linearity with a slight jump at the low value of t. The percent of aggregation score reduction is relatively constant for $t \leq 0.8$. We see that with $t = 0.9$, AGG-MIN finds no good mutable positions and having $|GMP = 0|$ means that the value



(a) AGG-MIN values for GLN1 with varying t



(b) AGG-MIN values for HSP82 with varying t



(c) AGG-MIN values for ADE4 with varying t

Figure 4: These graphs show varying t values for particular test inputs: the input which gets the best optimization,(a), the input which gets the worst optimization, (b), and a random input, (c). We see that a threshold that is too large will not yield a large enough search space for some inputs. We also notice a plateau of the score reduction in (a) and (c), which could mean that we have found a minimum for these inputs based on our optimization definition.

of t is too large. However we see that all other values of t in the range tested yield approximately equivalent aggregation score reductions. So instead of letting t range between zero and one, we should let t range from zero to the largest value that yields a non-zero number of good mutable positions. We do see a jump in the number of mutable and good mutable positions with $t = 0.6$ which leads to an increased B score of the output and validation of $t = 0.6$ as a choice of the final threshold value.

The graph for the worst optimization (figure 4(b)) displays some peculiar patterns. The patterns do not match those of the averages or GLN1. The number of mutable and good mutable positions does not increase between the $t = 0.8$ and $t = 0.7$. Furthermore, the values of all the metrics except the number of mutable positions are zero for $t > 0.6$. Since HSP82 is the worst-optimized input from the test set, a spike in score reduction at $t = 0.6$ validates this choice as the final threshold for AGG-MIN.

We can see in figure 4(c) that ADE4 follows the same linear with a jump pattern for decreasing t in all categories except score reduction. ADE4 matches the pattern in the other three categories, which means the choice of $t = 0.6$ looks good. However, we need to determine why the percent reduction does not follow the pattern of the average. We see that the percent reduction does increase each time t decreases, however the large jump occurs in between $t = 0.9$ and $t = 0.8$. Since the algorithm finds no good mutable positions with $t = 0.9$, we again invalidate this value on the graph because the algorithm could not mutate the input string and terminated early. We see that the aggregation score and number of mutations in the output are minimal when $t = 0.6$, again validating this value as the final choice for use in AGG-MIN.

As a note on the performance of the algorithm, based on the value of the threshold, we see that all of the metrics in these all of these examples are non-decreasing as t decreases. This behavior is expected, even taking into account the randomness of the algorithm. The non-decreasing property means that if the t value chosen for the average is slightly low for a particular input, the algorithm will still find a good set of mutations. Therefore, choosing the t value for the average does not sacrifice performance in minimization of F score for any input.

6.6 Determining the M Function; the Metropolis Approach

The M function that determines whether to move to a new state is another customizable piece of the simulated annealing process. The Metropolis approach [6] accepts a state change that reduces the aggregation score and accepts a state change that increases the aggregation score with some probability. This approach to accepting state changes has been tested and proven over many opti-

mization problems in different fields [3]. The only modification that we make is establishing the threshold for the probability of accepting a random state change.

6.7 Determining the Value of n_{max}

To find the best value for n_{max}, we assign a large value and look at the aggregation scores during the simulated annealing process. One such example of this process can be seen in figure 5. We use these graphs to determine how many steps it takes for the process to find a mutant with a minimal aggregation score. Because the number of good mutable positions is small, we did not want to use a constant larger than the average number of good mutable positions, so we get that $n_{max} = 3 \cdot |GMP|^2$.

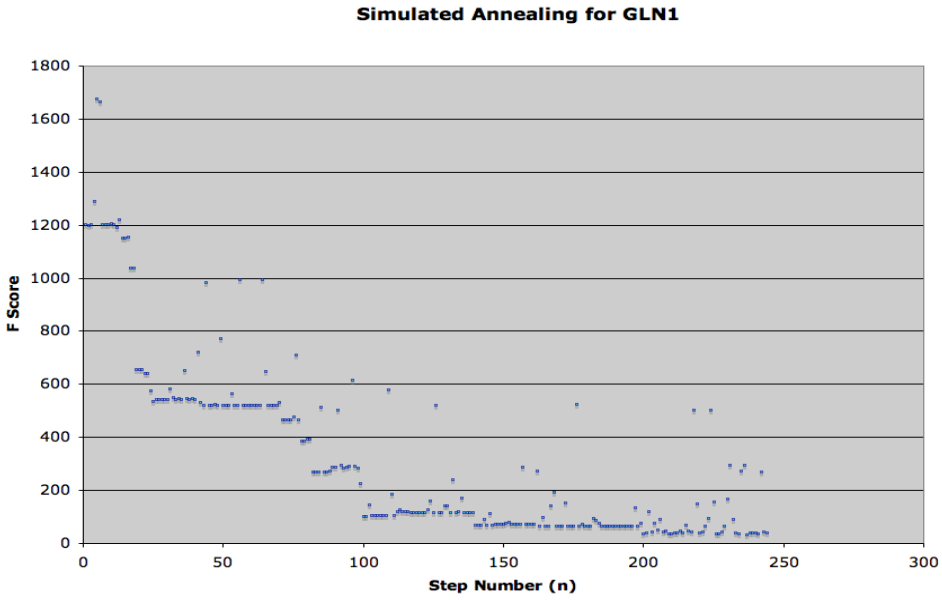


Figure 5: This plot shows the aggregation scores for each mutant during the simulated annealing process for GLN1. We see that we cannot lower the number of steps in this step of the algorithm because of the number of steps that it takes to reach a mutant with an optimal score.

6.8 The GMP Fallacy

Looking at the percentage of reduction of aggregation score in the tables above, we see that in a few cases the algorithm produces an output with less than a five percent (d) optimization when $|GMP| > 0$. We note that the algorithm takes a position as a good mutable position if there is

some mutation in that position where the score is d or more percent away, whether the new score is higher or lower than the original score. The algorithm does this for two reasons; first, we know that certain positions affect the F score more than others, and in a multiple mutation setting these positions may be helpful in attaining a string with lower aggregation score. Second, using this definition of a good mutable position makes implementing a solution which can find the string with either the minimum or the maximum aggregation score much easier. Thus, there might be no single mutation that decreases the F score, which could lead to no minimization of the F score with a non-zero amount of good mutable positions. Furthermore, the last step of the algorithm tries to decrease the Hamming distance of the output from the input, which may lead to a string with a slightly larger aggregation score for the output. This fallacy about the good mutable positions needs to be understood to correctly test the AGG-MIN algorithm.

6.9 Other Performance Metrics

We have mainly discussed minimization of aggregation score as a metric for how well AGG-MIN optimizes a particular input. We have also touched briefly on the substitution B score, the Hamming distance, and the number of F scorings done throughout the procedure as possible secondary metrics to aggregation score optimization. Other metrics may include actual runtime of the algorithm and

6.10 AGG-MIN Variations

Throughout the implementation and testing process of the AGG-MIN algorithm, we find many different ways to change the algorithm in small ways that have an impact on the performance and what we can learn from the outputs. Here we explore some variations of the AGG-MIN implementation and what can be learned through each of the exercises.

6.10.1 Stricter Mutations

One change to the algorithm which affects all of the stages of mutation is to look only at mutations that occur in nature. Instead of finding mutable positions, we look at the single substitutions for all possible mutations returned by the BLAST procedure. Using the aggregation scores for the single substitution strings, we can find good mutable positions in the same manner as before. However, in the combinatorial mutation we have a restricted list of substitutions that could occur at each index. This greatly increases the probability that a given output of the algorithm will have the same

biological function as the input (a stricter proxy for the Q function). Furthermore, we have less of a need to rely on the B substitution score because we know that the mutation occurs in nature, which makes implementing the algorithm somewhat simpler. The one caveat to this variation of AGG-MIN is that the set of possible algorithmic mutation would be smaller, leading to outputs with higher F scores.

We implemented this version of the algorithm and used the same set of protein inputs to test it. Our results are not surprising; in almost all cases the output string has an F score between the best single mutation and the output of the normal version of AGG-MIN. However we find that most scores are closer to those of the best single substitutions than to those of the combinatorial mutations in the regular version; this means that the extra computational time to find multiple mutations in this new method is not worthwhile since we could just use the single substitutions from the regular version. We also observe more mutations in the string (larger Hamming distance) since each position has fewer options for mutation, which goes against one of the objectives of the algorithm - to minimize the number of mutations in the output string. Because of these drawbacks, this design is less viable than the regular algorithm, but works well when the user is extremely cautious about mutations that do not occur in nature.

6.10.2 Using a Different n

We decided that n would have a value of 20 to both get a sizable F score reduction and to limit false positives, however varying n in the algorithm leads to differing results. Before exploring the results of this change, let us explore what changing n means to the algorithmic process. The n value matters when using the BLAST algorithm for finding proteins which exist in nature that are close to the input. So a smaller n value returns fewer sequences, and fewer positions vary from the input in each string. Because there is less variance between the strings, a constant t threshold value will lead to fewer good mutable positions, so the t value has to be adjusted for each different n used. Similarly, for large n values there is more variation between the strings returned by BLAST and the input string, so a constant threshold value leads to a larger set of mutable positions.

After experimenting with a different values of n (taking $n = 10,\ 20,\ 40,\ 80$), we find that a value of twenty gave consistent results with a medium value of t. In tests with $n = 10$, the algorithm does not find enough good mutable positions for a good combinatorial mutation. In tests with $n = 40$ and $n = 80$, BLAST returns too many strings that are not biologically related to the input. Having a value for n that is too large breaks our proxy for the Q function. Therefore we chose the value of $n = 20$ so that the average number of mutable positions is enough to optimize

the score without taking too much computational time and avoiding false positives. However, changing the value of n gives another dimension of control, and with the correct t value may yield better results than the algorithm we present.

7 Conclusion

We have solved a biological problem that is unsolvable in a laboratory setting. We have formalized the protein aggregation problem as a well-defined mathematical optimization and have created a heuristic for solving this optimization. We have also implemented the algorithm that we present and find that it minimizes the aggregation score of our test inputs by an average of forty percent. Computationally, our algorithm could be further evaluated by running it over a larger set of inputs. To truly confirm the results of the algorithm, the different outputs are being created in the laboratory and compared to the inputs by testing the amount of time it takes for the proteins to fully aggregate.

The performance of the algorithm that we have presented certainly improves with improved techniques for predicting the aggregation of proteins because such functions are inputs. The performance could also be improved by further research on the threshold values t and d, especially in determining how to have dynamic instead of static values. By creating techniques to fine-tune these values for each input, the optimization should give better results. Furthermore, we could test the value of n_{max} in order to reduce the actual running time of the algorithm; we want to find the smallest value that gives outputs will minimal aggregation scores.

Because there are many different properties of proteins that depend on the amino acid sequence, we can generalize the algorithm that we have presented to solve a large variety of biological problems such as optimizing protein electrostatics.

References

[1] S F Altschul, W Gish, W Miller, E W Myers, and D J Lipman, *Basic local alignment search tool.*, J Mol Biol **215** (1990), no. 3, 403–410 (eng).

[2] Ana-Maria Fernandez-Escamilla, Frederic Rousseau, Joost Schymkowitz, and Luis Serrano, *Prediction of sequence-dependent and mutational effects on the aggregation of peptides and proteins.*, Nat Biotechnol **22** (2004), no. 10, 1302–1306 (eng).

[3] W.K. Hastings, *Monte carlo sampling methods using markov chains and their applications*, Biometrika **57** (1970), 97–109.

[4] S Henikoff and J G Henikoff, *Amino acid substitution matrices from protein blocks.*, Proc Natl Acad Sci U S A **89** (1992), no. 22, 10915–10919 (eng).

[5] Rune Linding, Joost Schymkowitz, Frederic Rousseau, Francesca Diella, and Luis Serrano, *A comparative study of the relationship between protein structure and beta-aggregation in globular and intrinsically disordered proteins.*, J Mol Biol **342** (2004), no. 1, 345–353 (eng).

[6] M.N. Rosenbluth A.H. Teller E. Teller N. Metropolis, A.W. Rosenbluth, *Equations of state calculations by fast computing machines*, Journal of Chemical Physics **21** (1953), 1087–1092.

[7] Frederic Rousseau, Joost Schymkowitz, and Luis Serrano, *Protein aggregation and amyloidosis: confusion of the kinds?*, Curr Opin Struct Biol **16** (2006), no. 1, 118–126 (eng).

[8] M.P. Vecchi S. Kirkpatrick, C.D. Gelatt Jr., *Optimization by simulated annealing*, Science **220** (1983), no. 4598, 671–680.

[9] Yi-Kuo Yu, E Michael Gertz, Richa Agarwala, Alejandro A Schaffer, and Stephen F Altschul, *Retrieval accuracy, statistical significance and compositional similarity in protein sequence database searches.*, Nucleic Acids Res **34** (2006), no. 20, 5966–5973 (eng).