Protein structure prediction by search among combinations of reoccurring structural components

Johan Alexander Källberg Zvrskovec, February 2008.

Supervisor: Torgeir R. Hvidsten, The Linnaeus Centre for Bioinformatics, Uppsala University.

Introduction

Functional properties of large macromolecules, such as proteins, largely depend on their structural conformation. With the growing number of known and hypothetical amino-acid sequences from, among other sources, more than 100 sequenced prokaryotic and eukaryotic genomes, there also is a following challenge of determining respective protein function.[1] For many fields of science dealing with proteins, there thus is an interest in determining 3-dimensional protein structure.

To determine protein structure, there are both traditional structural biology methods, such as X-ray crystallography and NMR, and computational modelling methods. In Structural Genomics, an initiative by various research centres, both approaches are used in a systematic way to increase efficiency and decrease cost.[2,3] To further enhance the idea of Structural Genomics, there is a need of improved computational methods that take advantage of existing structural information. One newly proposed method to this end is a fragment based method that utilizes reoccurring local substructures called Local Descriptors to create a library of 3-dimensional building blocks.[3] By using Hidden Markov Models (HMM's) to model the relationship between amino-acid sequence and local 3D-structure coded by Local Descriptor Groups, the Local Descriptor Groups can be prioritized based on the probability of the corresponding HMM generating a set of local amino-acid sequence fragments.[4]

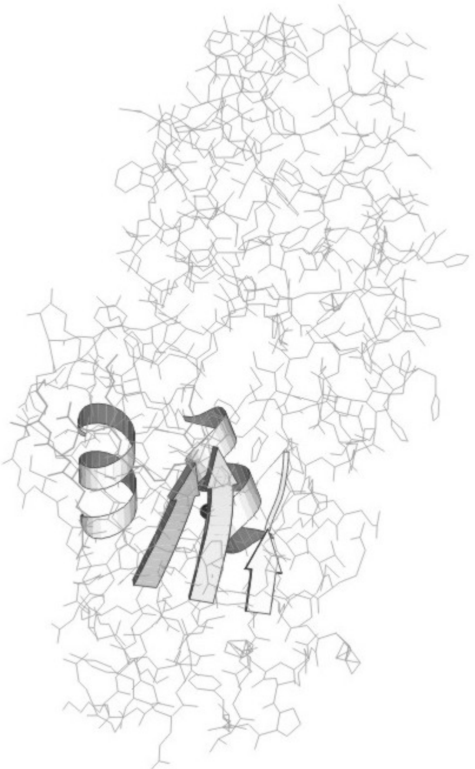
In an attempt to explore the possibilities using Local Descriptors as a basis for structural building blocks in protein structure prediction, we have in this project created a search algorithm for searching among combinations of fragmented structural data, implemented it and tested it on data obtained using Local Descriptors and HMM's.

Background

Traditionally, two classes of computational methods for protein structure prediction exist: de novo modelling and comparative homology modelling (CHM), with the difference that comparative homology modelling uses already known 3-dimensional template data from an alignment between a target and the template, and de novo modelling techniques does not. De novo modelling methods can in turn be divided into categories based on the use of statistical data from known structures to be used for various variables in the modelling process. Such "knowledge based" de novo methods are discerned from so called ab initio methods, which solely rely on physical models for interactions between parts in the protein modelling process. Within CHM there are currently two main directions: The first is where coordinates in one template, or averages of coordinates in a collection of templates, in some way are used for modelling the target structure. The other direction is where template data, such as distances and torsion angles, is used for setting up modelling restraints and using these to deduce structure. When alignments are made based on structure rather than sequence, this is generally called "fold recognition", both when dealing with CHM-methods and "knowledge-based" de novo –methods.[13]

Our method combines the two main directions in CHM, while we use both Local Descriptors and also modelling by satisfaction of spatial restraints using MODELLER (see Materials and Methods). Since our method uses statistically determined structural fragments as building blocks, it has some similarity to "knowledge based" de novo methods. To what category it should belong to is probably a matter of interpretation. Fragment based modelling techniques have become somewhat of a trend lately since they have been proved to produce relatively accurate models. The main difference between these and our approach is the use of Local Descriptors for identifying and storing the structural data. This provides us with fragmented building blocks which also are able to store interactions between parts of the protein not necessarily close in terms of sequence, something that other contemporary fragment based methods seem not to do. Other studies on methods based on assembly of fragments have been done with similar choices of implementation. Variants of genetic algorithms are frequently used among these for search, and MODELLER is a recurring choice for assembling structural fragments into molecular models. No earlier studies have been made on using the Local Descriptor concept for structure assembly.[3,13]

Data



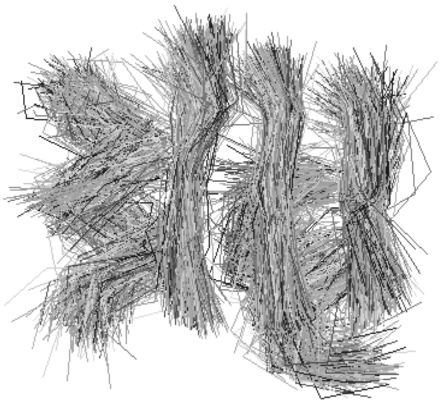


Figure 1. Figure 2.

Figure 1. An example of a Local Descriptor. A central amino acid has an environment consisting of five fragments. The structural information of this environment constitutes the amino acid's Local Descriptor.
Figure 2. The geometry of Local Descriptors form the same Local Descriptor Group superimposed on each other.

Protein data from ASTRAL 1.63 with less than 40% sequence identity was used as training data. Local Descriptors were created and HMM's were trained using this training data.[5]

The Geobacillus stearothermophilus Carboxylesterase protein, with the PDB callsign 1tqh, has been our test data for all tests.[6]

Local Descriptors have been given names according to:

{protein name}{chain}{domain}#{residue number}

A descriptor centred at residue number 68, domain 1, chain a, in the protein 1ll7, would be given the name 1ll7a1#68. A descriptor with n fragments is generally stored in the following format:

{name} {range 1}{sequence 1} {range 2}{sequence 2} … {range n}{sequence n}

This, combined with the structural data from respective fragment, represent the Local Descriptor. Local Descriptors were ordered into Local Descriptor Groups based on similarity. Similarity is measured by comparing the parameters.[3,4]
:

- Length and number of segments,
- Shape of segments,
- Number of geometrically similar segments,
- Fit quality by RMSD score after superpositioning descriptors

Local Descriptor Groups were named by the central Local Descriptor in that group. As an example, the group characterized by 1hr6a1#61 could be stored like:

```
GROUP: 1hr6a1#61 : 5
1bccb1#66  51-60 IKAGSRYENS    64-68 GTSHL    98-102   VESTR   103-107  ENMAY   192-200  HDFVQNHFT
1ezva1#69  54-63 FGSGAANENP    67-71 GVSNL    94-98     SNISR  99-103    DFQSY  193-201  ESFANNHFL
1ezvb1#48  35-44 VHGGSRYATK    46-50 GVAHL    80-84     STLDR  85-89      EYITL 173-181  KDFADKVYT
1hr6a1#61  46-55 IDAGSRFEGR    59-63 GCTHI    93-97     CTSSR  98-102    ENLMY  188-196  LDYRNKFYT
1hr6b1#69  54-63 VDAGSRAENV    67-71 GTAHF    101-105  AYTSR   106-110  ENTVY   196-204  KDYITKNYK
```

For the Hidden Markov Modelling, the amino-acid alphabet was converted into a modified variant (see Appendix 1). The test protein was used as query sequence for an alignment search in the library of HMM's trained on the training data. This yielded a list of HMM's, each corresponding to a Local Descriptor Group, sorted by score and E-value, and the alignments between the query sequence and HMM's upon which the scores and E-values had been calculated. As HMM implementation we used HMMER [7]

The scores resulting from the search are determined by:

$$S = \log_2 \frac{P(seq \mid HMM)}{P(seq \mid null)}$$

where $P(seq \mid HMM)$ is the probability of the query sequence given the HMM, and $P(seq \mid null)$ is the probability of the query sequence given the null-model. The null model is represented by a one-state HMM that treats all residues as independent and identically distributed.[7]

E-values are the expected number of false positive hits with scores equal or above the currently calculated.[7]

We chose a number of top-scoring Local Descriptor Groups for the search algorithm to combine. The structural data used for protein modelling was taken from the Local Descriptor that represented the group with its name (i.e. the central descriptor in terms of structure).

Materials and Methods

Fitness Calculation

Since a genotype can undergo processes that will change its fitness in our algorithm, i.e. ageing, we have decided to represent fitness with two stored variables in each genotype. Calculated Fitness (Fc) is a constant unmodified value which only depend on the genotype’s gene configuration, and Modelled Fitness (Fm) is a value that might change through the artificial life of the genotype but is initially set equal to Fc for newly produced offspring. Both of these values are represented as floating point variables in the range between 0 and 1. To calculate Fc the application relies on an external program called MODELLER. MODELLER is a program used for homology modelling or comparative modelling of proteins by satisfaction of spatial restraints.[8-12] When a new genotype is created in the implementation of the search algorithm, a new MODELLER-process is initialized to create a protein model based on the structural information of the newly created genotype. In the modelling process, MODELLER evaluates the model and presents the evaluation in the form of a value called PDF-value. This is the value of the objective function that MODELLER minimizes in the modelling process.[8] The PDF-value is retrieved from the resulting PDB-file after the modelling is completed, and is used to determine the Fc of the new genotype (see *Search Algorithm*).

The information given to MODELLER to create a protein model is:

- Amino acid sequence of the protein being modelled.
- Structural information for (parts of) the amino-acid sequence. This data is retrieved from PDB-files with the coordinates stored in the Local Descriptor data structure.

We have used MODELLER 9v2 in both test runs.

Genetic Search Algorithm

Genetic search algorithms, or shortly genetic algorithms (GA), are a general collection of search algorithms inspired by the natural evolutionary mechanisms based on genetics. A multitude of interpretations of what such mechanisms are, and what ideas among such mechanisms might prove useful, have resulted in equally different approaches to design of GAs. Generally, a GA somehow maps perturbations of variables, traditionally called genes (or something else that might illustrate some sort of connection to genetics or biological evolutionary theory, for example loci), to locations in search space. The choice of variables varies from simple bits, Boolean variables to floating point variables for example, all depending on preference and exactly what a gene is supposed to represent in the design. Different sets of perturbations can then be stored in entities, also these usually named according to genetic-evolutionary tradition. Search is performed by applying certain operations upon the entities holding the perturbation of genes and upon the perturbations of genes themselves, to change their representations of points in search space. These operations are often translated directly from genetic-evolutionary theories according to the design of the GA, and regularly includes concepts such as reproduction, death, fitness, point mutation, crossover, insertion, deletion et.c., or any other function applicable to the GA. While this design of a search algorithm by creating an evolutionary model can possibly be extended to unknown limits, one must examine the biological effects of the corresponding functions and their effects they have on the search to evaluate them. As an example it might be tempting to include population mechanics and the concept of species in a GA, but the use of this must be examined. This is not done in our project.

Our search algorithm is a kind of GA that mimics natural evolution. The purpose is to optimize a set of Boolean variables, each representing a building block of stored structural data from a Local Descriptor Group, to create the best possible combination of structural data. A true Boolean variable means that the corresponding structural data is used, and a false variable means that it is not used. As the algorithm is a GA, we have chosen to call the Boolean variables – genes. A set of genes is stored in a data structure that we call a genotype. Currently the algorithm is written so that the number of genes in a genotype is determined in the beginning and remains constant throughout the whole search – no additional genes can be created, and also no genes can be deleted in a genotype. Each genotype stores two fitness-values; one constant fitness value (Fc) that has been calculated based on the combination of genes (i.e. the genotype), and one variable modelled fitness value (Fm) depending on the evolutionary model that does not have to correspond to the Fc-value, but is initially set equal to Fc for all newly produced offspring (not the initial genotypes). At the beginning of each search, a set of initial genotypes are created based on the start-variables *nrInitialGenotypes* and *initialFitness*, which determines the number of initial genotypes and their corresponding initial Fm-fitness. The Fc is set to -1 to mark it as uncalculated. Each initial genotype will have the full set of possible genes with each gene having a starting value randomized with equal possibility for the value true and false (i.e. 0.5).

Depending on the starting parameters *iter* and *genIter*, a number of iterations are performed, in which a number of actions are taken. Because of the bottleneck functionality (see explanation below and *Implementation*), there are two types of iterations. The iteration where all of the algorithm specific actions take place is the type of iteration that is represented by *genIter* – general iterations. The number of such general iterations is taken place for each iteration represented by *iter*, so the total number of general iterations will be *iter* * *genIter*.

A general iteration is meant to represent some sort of timeframe in the artificial life of every genotype. The general iteration can be divided into the following actions:

- *Depending on the environmental model, the number of genotypes and each genotype's individual Fm – decide which genotypes die before they can reproduce in this iteration.* In this step the algorithm loops through all existing genotypes and decide if they die or not based on a survival probability. The formula for survival probability is:

  $$P_{survive} = F_{m,i} \frac{0.5L}{N}$$

  where $F_{m,i}$ is the Fm-value of the genotype i, L is a limit value determined by the starting parameter *popSizeBefore* (and also *popSizeAfter* if the bottleneck functionality is used) and N is the number of genotypes in this iteration before any has been removed from the cause of dying – to make this number equal for all genotypes. The limit *popSizeBefore* is used in the first general iteration (*genIter*) in each normal iteration, and if *genIter* >1, *popSizeAfter* is used as limit for all general iterations after the first one.

  This probability helps to control the population size by making the survival probability proportional to both the Fm and half the population size limit, as well as keeping it inverted proportional to the number of genotypes. Genotypes that are decided to die are removed from the genotype storage.

- *Depending on the parameters for reproduction – decide which genotypes are reproducing and with which genotype. Also create the genotypes of the resulting offspring and add them to a specific storage.* Here the algorithm loops through all genotypes and, depending on the probability of reproduction which is the same for all genotypes, performs the reproduction steps. The probability of reproduction is decided by the *pOffspring* starting parameter. A genotype with a higher Fc would generally live longer than a genotype with lower Fc and thus have higher chance of producing offspring, but at each general iteration the probability of reproduction is equal for all genotypes. When a genotype is reproducing, it does so in combination with a randomly chosen currently existing genotype. A genotype can reproduce with itself. The offspring of the two genotypes is created and given genes from either one of the two parents. The probability to receive a gene from one of the parents is 0.5. For each gene that is created in the offspring in this way, there also is a chance of point mutation depending on the variable set by *pMutation*. If a mutation occurs, there is an equal chance for a gene of mutating into true and false, which is 0.5. Genotypes of offspring are stored separately until the end of the iteration.

- *Alter all genotypes' Fm with respect to the ageing model.* In our search algorithm, ageing is an alteration of the genotypes' Fm depending on their artificial life. A genotype incurs a penalty in Fm for every iteration. This penalty is decided by the parameter *ageingDecayFactor* and should be positive to be a penalty. The resulting Fm of every genotype is:

  $$F_{m,i,j+1} = F_{m,i,j} - a$$

where $F_{m,i,j}$ is the Fm of genotype i at iteration j and a is the decay factor. Note that when a genotype is produced through reproduction, its Fm is set equal to its recently calculated Fc (see the next step).

- *Determine fitness for the newly created offspring and add them to the general pool of genotypes.* Last in the general iteration, all created offspring have both their types of fitness determined. First is the Fc value calculated from the combination of genes in the genotype. Second is the Fm set equal to the calculated fitness as an initial value. To determine Fc for a genotype, a 3-dimensional protein model is built based on the structural information stored in the respective genotype's genes with a true value. The quality of this model is represented by a function value, the pdf-function, which is calculated in the building process (more on this under *Fitness Calculation*). Depending on this function value and how this number is valued, Fc is calculated by:

$$F_c = \frac{1}{\left(1 + \frac{\phi_{calculated}}{\phi_{normal}}\right)}$$

where $\phi_{calculated}$ is the calculated pdf-function, and ϕ_{normal} is a variable to control how to value $\phi_{calculated}$, and is determined by the *normalpdf* start variable.

- *Add all offspring to the same storage as the other genotypes.*

After each general iteration, a comparison is made between the best genotype in the set of genotypes and a stored globally best genotype. If the best genotype in the set is better than the globally best, it will replace the former as the globally best genotype. Here the calculated fitness is used for comparison, since it is the unmodified fitness directly depending upon the genetic configuration of the genotype.

Implementation

Our GA was implemented in the programming language Java using JDK 1.5.0_10.

A couple of useful features, apart from the search algorithm itself, were also implemented into the program:

- Configuration-file – Most important parameters for the search algorithm and the program itself can be altered through a configuration-file, which is read at the startup of the program.
- HMMER-output file reading function – a function to read a type of output-file from the program HMMER, and also the type of file where Local Descriptor Group data is stored, and puts everything in an internal data structure.
- Ability to store both Local Descriptor information and current search algorithm information to file – this is to make it possible to pause the search algorithm and continue the search at a later time and on another machine.
- Log file – after each iteration the program writes a vector with information about the state of the search algorithm for the current iteration.

Since there has been a rather iterative and experimental design process where new implementation solutions have been introduced on top of old ones, there are a couple of implementation artefacts such as the bottleneck functionality. While the bottleneck functionality fully works, it functions badly together with the Log file functionality, which only writes to the Log file for each larger iteration – the type of iteration controlled by *iter* (see Search Algorithm above).

Hardware

All search tests were run on an AMD Thurion 64 Mobile Techology ML-32 with the following capacity:

792 MHz Processor
896 MB RAM

Time Approximation

To approximate running times, we measured the time it took for MODELLER to complete a model using an arbitrary set of genes. From this, a time value per gene was obtained and the following crude formula was used to approximate the running times:

$$T = \frac{tIGmp_m}{2}$$

where t is the time in seconds per true gene to calculate the arbitrary model, I is the number of iterations, G is the number of genes in each genotype, m is the mean number of genotypes during the run, and p_m is the offspring probability for each genotype. While we have not bothered with calculating the real mean number of true genes in an offspring, the estimate of half the number of genes was used, with the resulting division by two in the formula.

Results

We have performed two test runs of the implemented genetic search algorithm. The first run was more of a tryout run to discover bugs and to test new features that was gradually implemented. No significant changes were made to the search algorithm part during this time. The resulting data from this first run can still be interesting and is thus also presented here.

Run times are only approximated values since the real run time was not measured. (see Materials and Methods).

Data from the 212 best Local Descriptor Groups with regards to HMM-alignment scores were used in both runs. We have chosen to initially examine a selection of the 212 genes and their development throughout the search. These genes are called Selected Genes and it is those that are plotted under respective run below. The Selected Genes are:

1m33a_#202
1wht.1#A155
1auoa_#112
1m33a_#29
1jfra_#133

1llfa_#215
1hlga_#99
1ispa_#102
1n1ma2#653
1m33a_#210

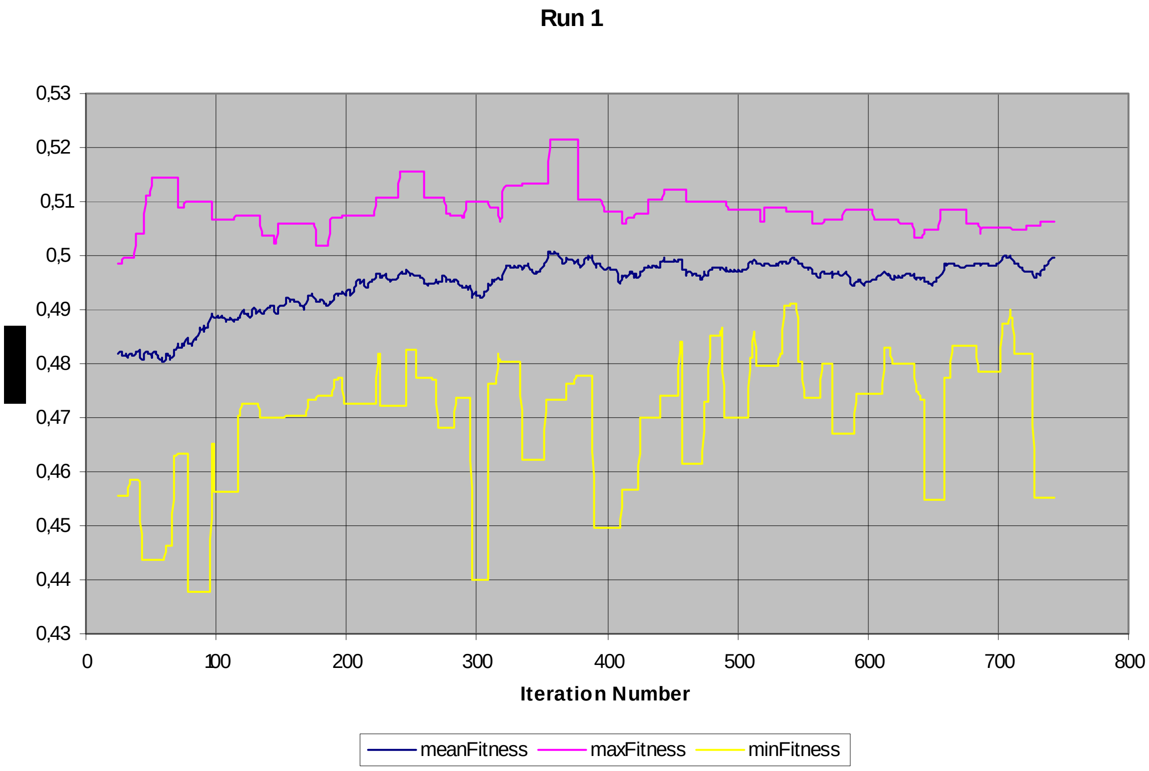
1l7aa_#188
1jkma_#229
1jkma_#232
1ju3a2#268
1k4ya_#347

1iupa_#212
1mtza_#273
1ju3a2#254

Run 1

Run-specific parameters
(in the format they are set in the config-file)
[genIter] 1;
[normalpdf] 8000;
[pOffspring] 0.05;
[pMutation] 0.0005;
[popSizeBefore] 300;
[popSizeAfter] 300;
[ageingDecayFactor] 0.015;
[nrInitialGenotypes] 40;

For Run 1 we let our program iterate for a total of 743 iterations and was ended intentionally to start Run 2 instead. The total run time for Run 1 was approximately 472548 seconds, or 131.26 hours. Mean number of genotypes used for the approximation is 40.



[redacted]

Figure 3. Plot of fitness development against iterations for Run 1. *meanFitness* is a mean value of all genotypes' Fc during an iteration. *maxFitness* is the maximum Fc in the population and *minFitness* is the minimum. Data before iteration 25 is not shown in the plot since there still existed surviving initial genotypes then with non-representative fitness values (i.e. -1). The full plot can be seen in Appendix 2.

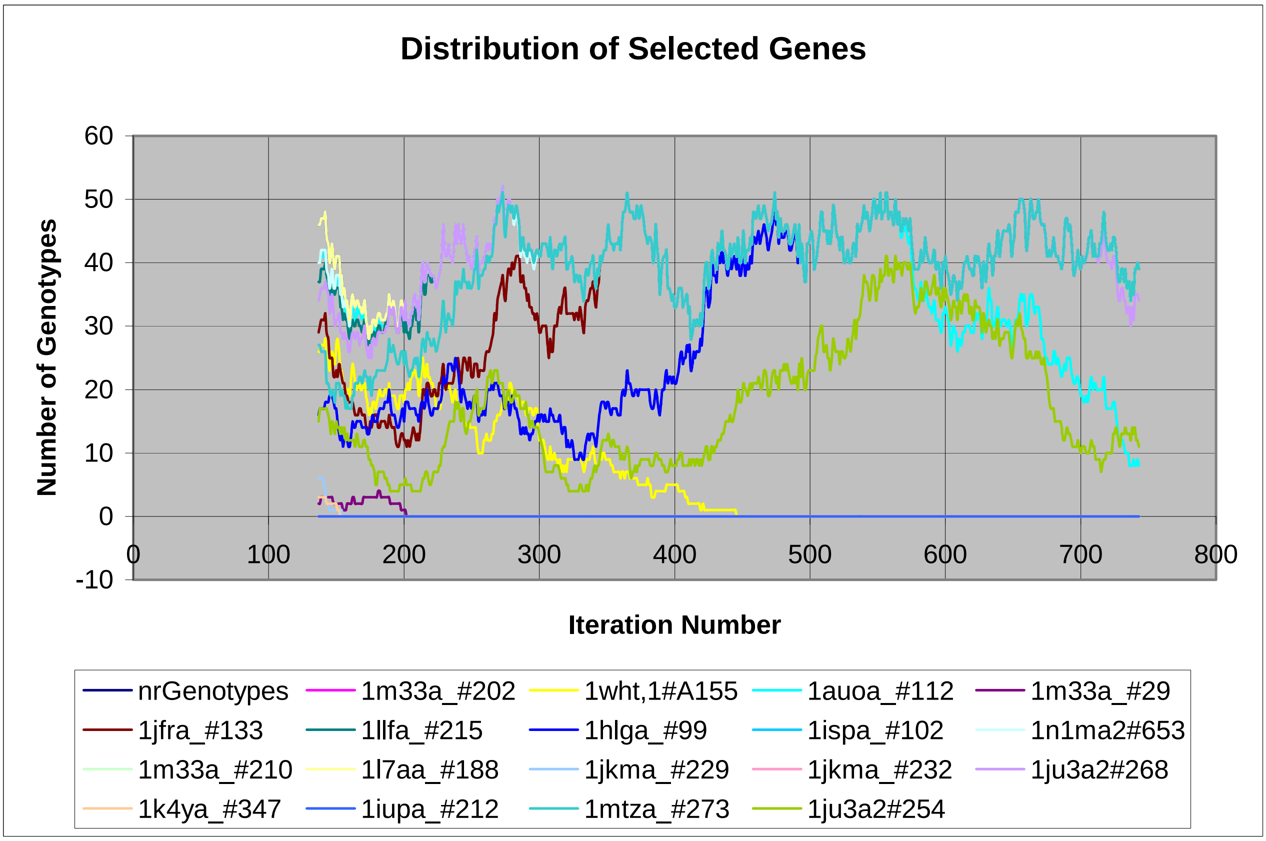


Figure 4. Plot of gene distribution against iterations for Run 1. Each curve shows in how many genotypes the gene can be found, in which the gene is TRUE. As comparison, the number of genotypes is also presented. The counting functionality was introduced at iteration 137, so genes before that iteration has not been counted for this run.

Run 2

Run-specific parameters

(in the format they are set in the config-file)
[genIter] 1;
[normalpdf] 10000;
[pOffspring] 0.1;
[pMutation] 0.001;
[popSizeBefore] 300;
[popSizeAfter] 300;
[ageingDecayFactor] 0.02;
[nrInitialGenotypes] 40;

For Run 2 we let the program iterate for a total of 266 iterations. This Run happened to be interrupted due to a power failure while writing to file and consequently aborted. The total run time for Run 2 was approximately 253764 seconds or 70.49 hours. Mean number of genotypes used for the approximation is 60.

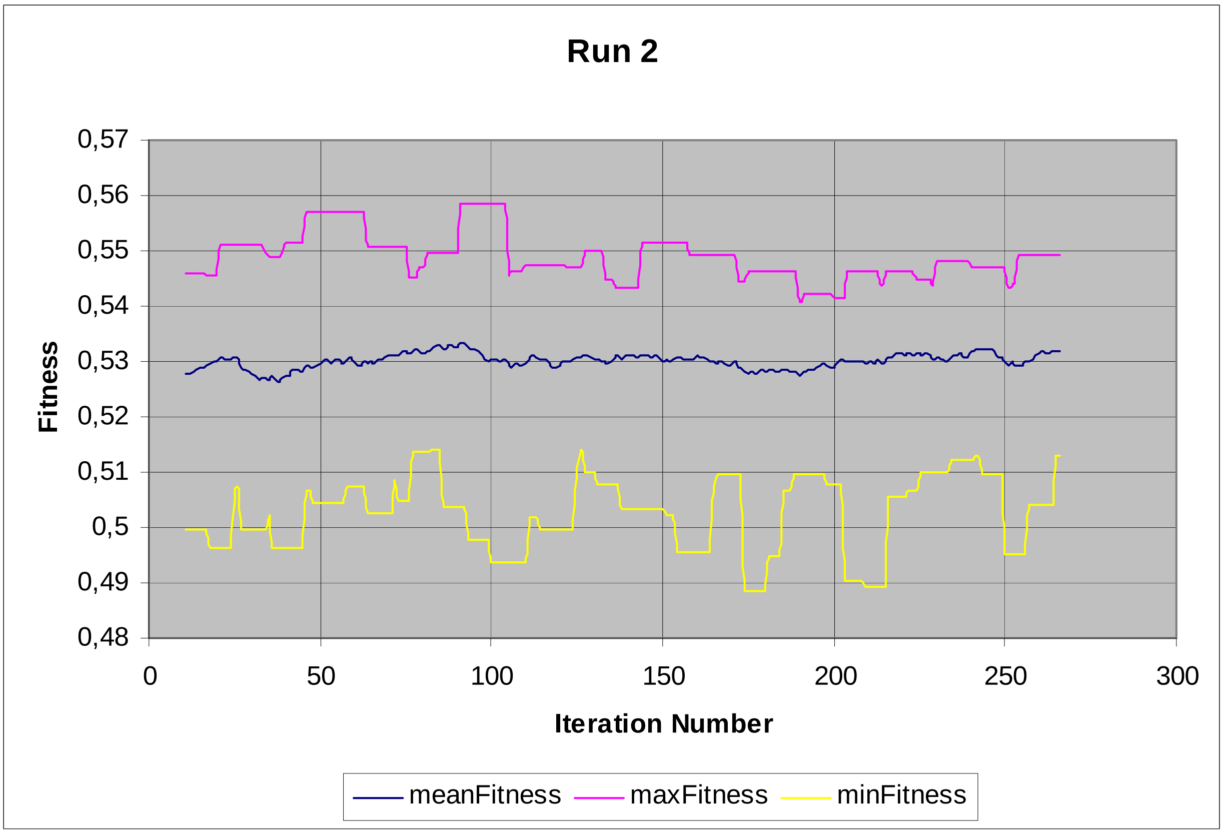


Figure 5. Plot of fitness development against iterations for Run 2. In the same way as for Run 1, *meanFitness* is a mean valule of all genotypes' Fc during an iteration. *maxFitness* is the maximum Fc in the population and *minFitness* is the minimum. Data before iteration 11 is not shown in the plot since there still existed surviving initial genotypes then with non-representative fitness values (i.e. -1). The full plot can be seen in Appendix 2.

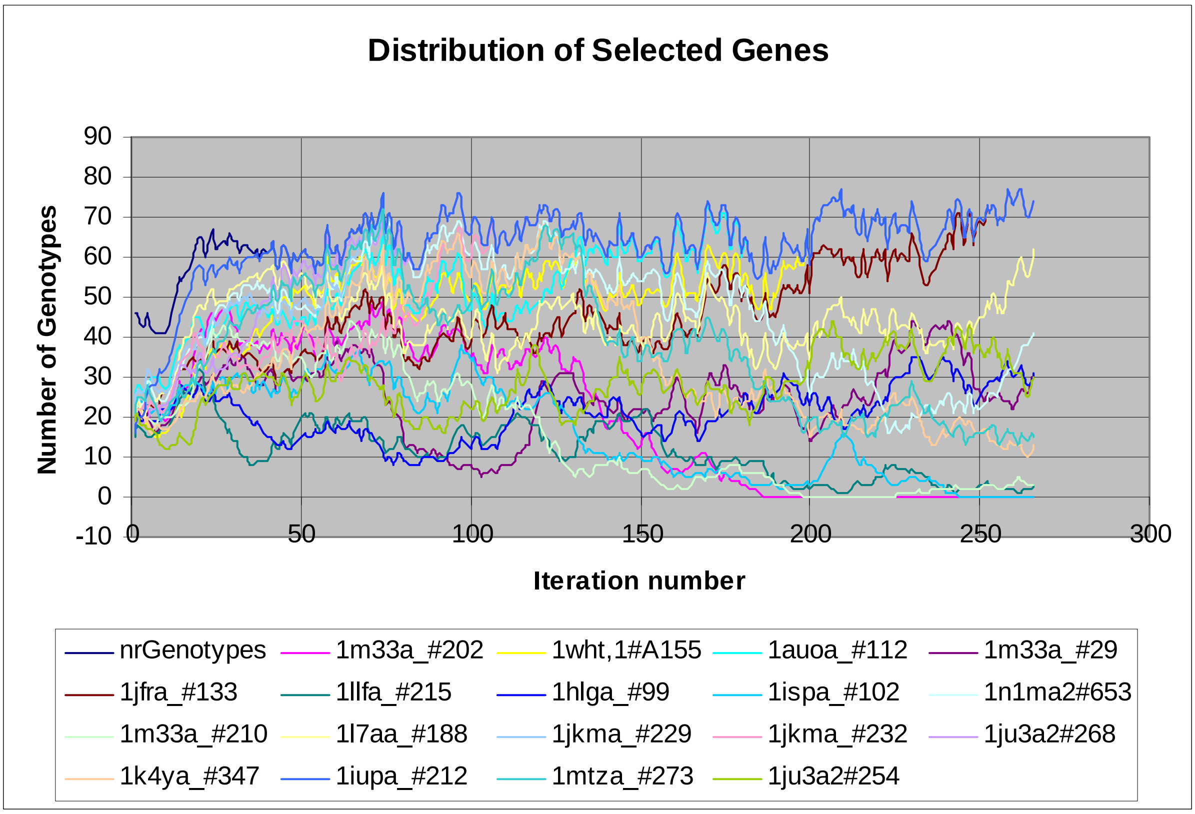


Figure 6. Plot of gene distribution against iterations for Run 2. in the same way as for Run 1, each curve shows

in how many genotypes the gene can be found, in which the gene is TRUE. As comparison, the number of genotypes is presented.

Run 3

Run-specific parameters
(in the format they are set in the config-file)
[genIter] 1;
[normalpdf] 10000;
[pOffspring] 0.2;
[pMutation] 0.00001;
[popSizeBefore] 200;
[popSizeAfter] 200;
[ageingDecayFactor] 0.001;
[nrInitialGenotypes] 30;

For Run 3 we let the program iterate for a total of 79 iterations. In Run 3 we wanted to try out a more greedy approach to the search and thus used a low ageingDecayFactor coupled with a high offspring probability and a low probability of mutation. The total run time for Run 3 was approximately 301464 seconds or 83.74 hours. Mean number of genotypes used for the approximation is 60.

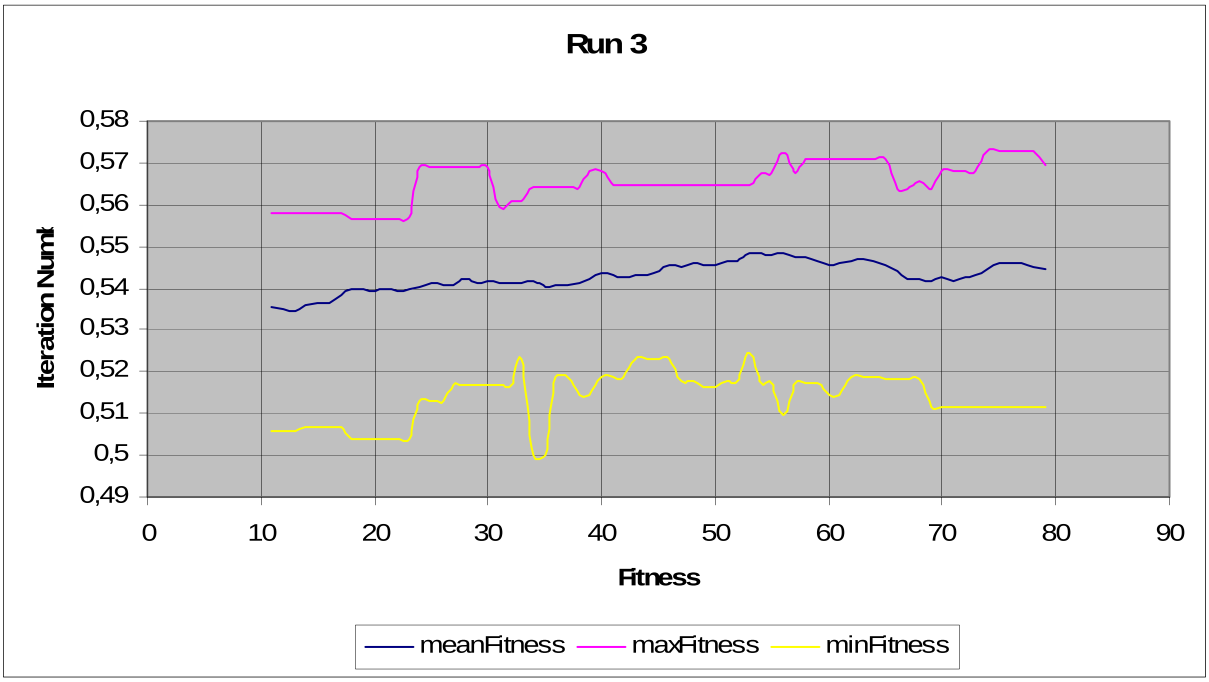


Figure 7. Plot of fitness development against iterations for Run 3. *meanFitness* is a mean valule of all genotypes' Fc during an iteration. *maxFitness* is the maximum Fc in the population and *minFitness* is the minimum. Data before iteration 11 is not shown in the plot since there still existed surviving initial genotypes then with non-representative fitness values (i.e. -1).

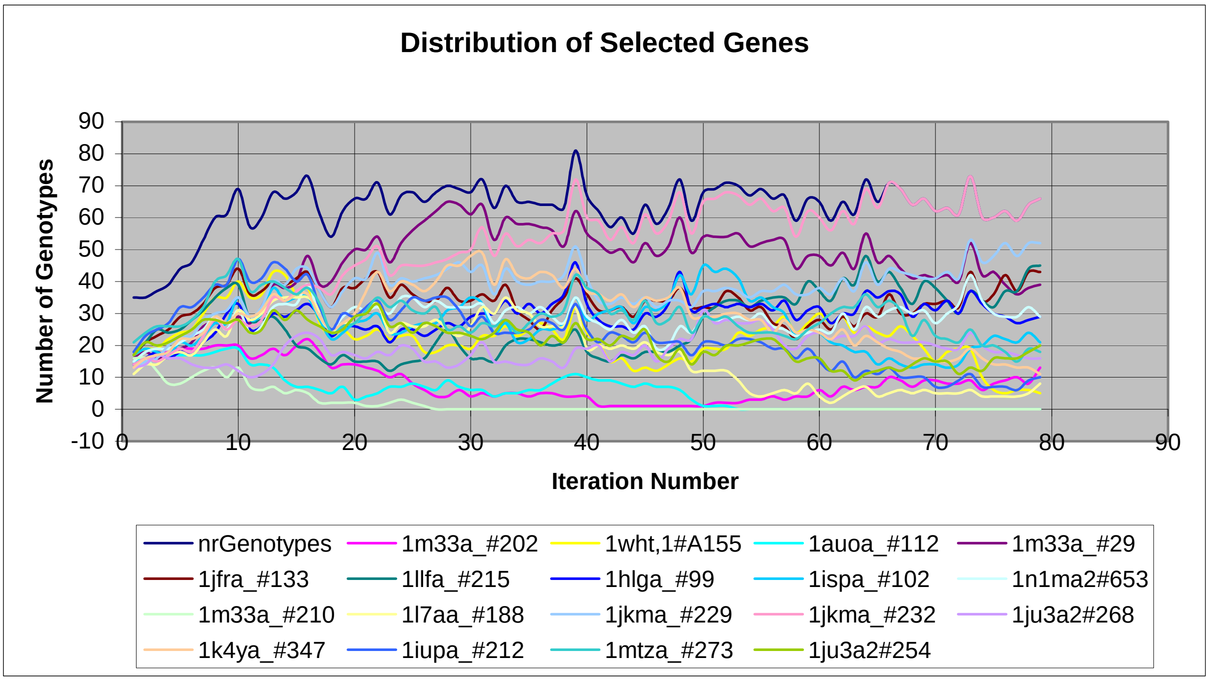


Figure 8. Plot of gene distribution against iterations for Run 3. Each curve shows in how many genotypes the gene can be found, in which the gene is TRUE. As comparison, the number of genotypes is presented.

Discussion

This project has focussed on designing, developing, and implementing a genetic search algorithm for protein modelling. For the application to be tested in a satisfactory way, the need of other functionalities required that some effort had to be diverted towards finding implementation solutions to these as well. Over all it has been a good practice in application development and planning. Implementation decisions had to be made for general application architecture, data structures, file handling and process handling among other things. Sometimes decisions were made in a hasty fashion to be able to quickly test the search algorithm, and multiple solutions for practically the same tasks can also be found; everything which contributed to some insight in general development procedure.

One of the first things that is to say about the results from this project is that the search algorithm has not been tested enough. What search algorithm parameters that yield the shortest search time to the optimal solution is an optimization problem in itself. Although we have not done it here, the search algorithm could be tested on a function that is quicker to evaluate to discern its capabilities, before applying it to the initial problem of protein modelling.

The search space in both test runs where we have applied the search algorithm can be seen as a space composed of 212 room vectors. Each vector corresponds to a gene and can have two discreet values – either it's used or not used in the modelling. This makes the search space both finite and discreet with 2^{212} coordinates in space, each coordinate with its own combination of genes and thus also a corresponding molecule with a PDF-value.

Search time is a big uncertainty in the evaluation of the search algorithm since it’s difficult to visualize the search landscape and thus how difficult it is for a search algorithm to navigate through this landscape. Because of the high time requirement for each iteration, which is a result of the relatively high time requirement to calculate a molecule’s PDF-value, it seems very probable that only a small part of the search space has been examined during our tests. Depending on the topology of the search landscape, more time might be needed for the search algorithm to run, to evaluate the performance of the algorithm on this particular search landscape. An estimation of our application’s time dependency is:

$$T(i,l,g,t) = \mathrm{O}\left(i\left(gt + tT_{MODELLER}(l,g)\right)\right)$$

where i is the number of iterations, l is the length of the amino-acid query sequence, g is the number of genes and t is the number of genotypes. $T_{MODELLER}(l,g)$ is the time dependency of MODELLER which is unknown to us. In practice this seems to be the main time consuming contribution of the complete application, for larger l,g and t, or at least during our tests.

From the obtained results, a possible interpretation would be that our genetic search algorithm is indeed working towards better solutions in terms of measured PDF-score. The algorithm’s behaviour seems to be sensitive to changes in its search parameters and there is a following challenge to find suitable parameter combinations.

The ability to thoroughly examine a specific place in the search space has of course to be compared to the ability of moving between different interesting locations. This judgement has to be made based on an estimation of the search landscape, if possible. Our approach to this was rather intuitive, and the first test was supposed to be a tryout of some more or less arbitrary search variables to improve this in the second run. It would seem like our algorithm might have had problems during our tests with spending too much time at already explored places in the search space, i.e. getting stuck in local minima. This seemed to be a problem especially during the first run with a somewhat lower mean number of genotypes compared to the second run. If examining the graphs from both runs, an explanation for this might be the difficulty the algorithm had to recover from gene fixation. Mean fitness increases, but after a while it seems to converge towards a limit. Maximum fitness can be seen to converge towards mean fitness and only a few genes escape from fixation by the means of mutation. If examining the differences between the first run and the second run, it seems like the increased *normalpdf* and *pOffpring* parameters’ effect on the mean number of genotypes in the second run, increased the number on non-fixated genes during the test and made it less likely for the algorithm to get stuck through the effects of gene fixation. The doubled probability of mutation in the second run compared to the first might also have contributed to this. That the mean fitness does not seem to increase in the second run is probably because of this decreased sensitivity of the search algorithm. Which parameter settings are too sensitive or too insensitive is hard to tell. In our case when a combination of Local Descriptor information is already known to be able to be used in modelling a molecule with a somewhat good PDF-value (see the discussion below), it would be interesting to see if the search algorithm could find a similar result. However, from both tests, a conclusion can be made that a search with a parameter setting that is too sensitive to local minima, might linger around solutions that are far from as good as already known combinations, as happened in Run 1. If on the other hand a parameter setting that is less sensitive is used, as in the second run, the run time might grow too large for the search to be of interest. Run 3 was made as a reaction on the comparisons between Run 1 and 2. The idea was to test how the algorithm would behave with greedy parameter settings – low ageingDecayFactor, high probability of producing offspring and low mutation probability. In run 3 the mean fitness development is very similar to that of Run 1, but is made in shorter time and in a fewer number of iterations. Max fitness does not seem to converge towards mean fitness in the same way as Run 1, which might be explained by that only few genes have reached fixation in the relatively few number of iterations of Run 3 compared to Run 1. This hints that a similar of even better result might have been obtained with a lower probability of offspring to not waste offspring with calculated Fc on large population growth between iterations. If one considers the actual search time, the increased probability of producing offspring may not have made the algorithm "greedier" in terms of gene fixation, rather the opposite. This is further strengthened by the comparison between the distributions of Selected Genes between Run 1 and Run 3, where in Run 1 a low probability of producing offspring was used compared to Run 3.

To say something about the building block capabilities of Local Descriptor Groups might still be early considering the results of this project. Before the test runs of the search algorithm were performed, we made MODELLER model our test protein using the same amino acid sequence as in the test runs, but also providing the already known structure. This produced a molecule with a PDF-value of 1326. The known structure of 1tqh does not include the 5 first amino acid residues, which we have included in the provided amino acid sequence. Hopefully this part does not have a too large impact on the PDF-score, but still needs to be taken into account when the PDF-value is considered. As comparison, we let MODELLER generate a model using the information in the set of Local Descriptors called "selected" in the Results-part, which were known to align well to the 1tqh -protein. The result was a molecule with a PDF-value of around 5312.

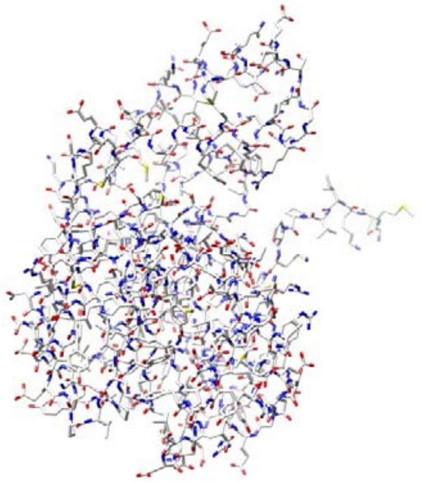


Figure 9. The modelled 1tqh –protein using known structural data with a PDF-score of ~1326. What is seen as a tail sticking out of the molecule is an unmodelled part of the protein that the known structural information didn't cover.

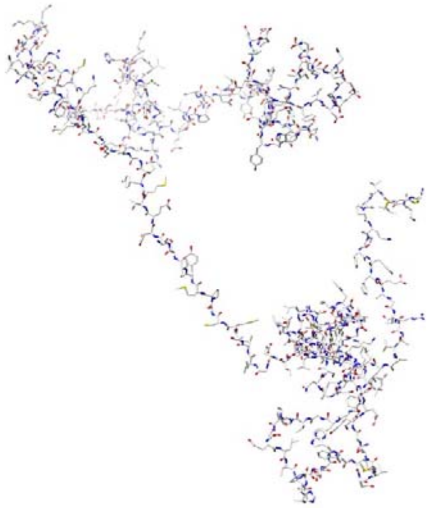


Figure 10. The modelled 1tqh –protein using structural data from the selected Local Descriptors. PDF-score is ~5312.

The resulting best molecule from Run 1 had a PDF-value of ~7343 and the best from Run 2 a value of ~7909. The best molecule from Run 3 had a PDF-value of ~7454 and is by far the most compact molecule among all results. When observing these values, one needs to consider that it is not at all certain that our search algorithm is able to produce results with PDF-values on level with the model of the known structure. What might be expected of our search algorithm are values equal or better than the score of the model made from our selected genes, which is a model created upon a combination of genes that really exists within the search space.

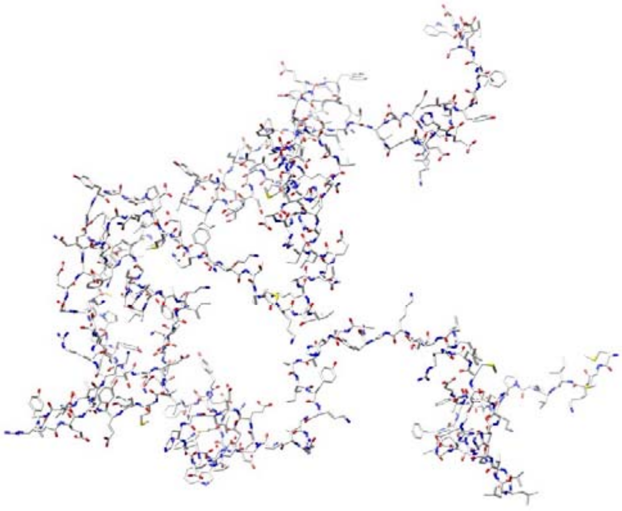


Figure 11.The best resulting molecule from Run 1. PDF-score is ~5312.

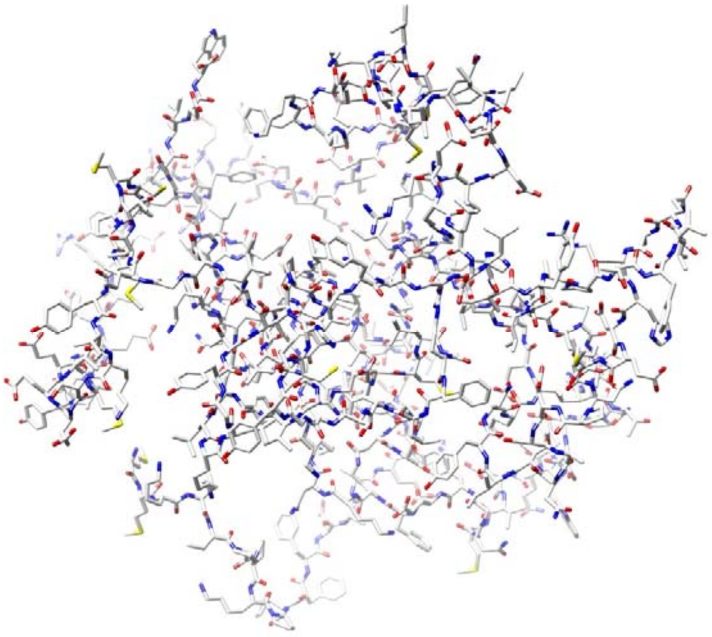


Figure 12. The best resulting molecule from Run 3. PDF-score is ~7454.

What has been concluded during this test is that our modelling method with building blocks from Local Descriptors and HMM's can be used during these specific circumstances to generate these specific results. If the method is useful in a more general way is difficult to say from our tests. One can speculate that for the method to give interesting results, a greater number of Local Descriptor Groups might have to be used for the search. Also improvements might have to be made to the search algorithm, both to make it take advantage of more structure combinations by, for example, using each backbone fragment of a Local Descriptor as an individual building block, and to make it more time efficient in general. A discussion might have to be made about how to create a model from the fragmented structural data in the Local Descriptors and if the structural restraints –method that MODELLER uses really takes advantage of the long range interaction information within. In general, further examination of different modelling techniques and improvements to this part of the method is needed. Since the structural restraints evaluation sometimes seem to value "uncompact" models, it would further be interesting to try and complement our method with some sort of energy measurement. An interesting question is also if results obtained by our method can be helpful as starting points in further modelling, for example with molecular dynamics or other ab initio -methods.

References

1. Chao Zhang, Sung-Hou Kim (2003)
   Overview of structural genomics: from structure to function.
   Curr Opin Chem Biol 7, 28-32.

2. John-Marc Chandonia, Steven E. Brenner (2006)
   The impact of structural genomics: expectations and outcomes.
   Science 311, 347-51.

3. Torgeir R. Hvidsten, Andriy Kryshtafovych, Jan Komorowski and Krzysztof Fidelis
   Local descriptors of protein structure: A systematical analysis of the sequence-structure relationship in proteins using short- and long-range interactions

4. Minyan Hong
   Fold recognition using local descriptors of protein structure and Hidden Markov Models

5. Brenner, S.E., Koehl, P. and Levitt, M. (2000)
   The astral compendium for sequence and structure analysis.
   Nucleic Acids Research., Vol. 28, No. 1 , 254–256.

6. Liu, P., Wang, Y.F., Ewis, H.E., Abdelal, A.T., Lu, C.D., Harrison, R.W., Weber, I.T. (2004)
   Covalent reaction intermediate revealed in crystal structure of the Geobacillus stearothermophilus carboxylesterase Est30.
   J.Mol.Biol. v342 pp. 551-61.

7. HMMER
   http://hmmer.janelia.org/

8. MODELLER
   http://salilab.org/modeller/

9. N. Eswar, M. A. Marti-Renom, B. Webb, M. S. Madhusudhan, D. Eramian, M. Shen, U. Pieper, A. Sali.
   Comparative Protein Structure Modeling With MODELLER.
   Current Protocols in Bioinformatics, John Wiley & Sons, Inc., Supplement 15, 5.6.1-5.6.30, 200.

10. M.A. Marti-Renom, A. Stuart, A. Fiser, R. Sánchez, F. Melo, A. Sali (2000)
    Comparative protein structure modeling of genes and genomes.
    Annu. Rev. Biophys. Biomol. Struct. 29, 291-325, 2000

11. A. Sali & T.L. Blundell (1993)
    Comparative protein modelling by satisfaction of spatial restraints.
    J. Mol. Biol. 234, 779-815, 1993

12. A. Fiser, R.K. Do, & A. Sali (2000)

Modeling of loops in protein structures
Protein Science 9. 1753-1773, 2000.

13. J.M. Bujnicki
Protein-Structure Prediction by Recombination of Fragments
ChemBioChem 7,19 – 27, 2006

Appendix 1 – Amino-Acid Alphabet

```
2:nd letter explanation
H: Helix
C: Coil
E: Other


P H -> A
P C -> B
P E -> C
Q H -> J
Q C -> K
Q E -> L
A H -> A
A C -> B
A E -> C
R H -> G
R C -> H
R E -> I
S H -> J
S C -> K
S E -> L
C H -> A
C C -> B
C E -> C
T H -> J
T C -> K
T E -> L
D H -> D
D C -> E
D E -> F
E H -> D
E C -> E
E E -> F
V H -> A
V C -> B
V E -> C
F H -> A
F C -> B
F E -> C
W H -> A
W C -> B
W E -> C
G H -> A
G C -> B
G E -> C
H H -> G
H C -> H
H E -> I
Y H -> J
Y C -> K
Y E -> L
I H -> A
I C -> B
I E -> C
K H -> G
K C -> H
K E -> I
L H -> A
L C -> B
```

```
L E -> C
M H -> A
M C -> B
M E -> C
N H -> J
N C -> K
N E -> L
```

Appendix 2 – Additional Figures

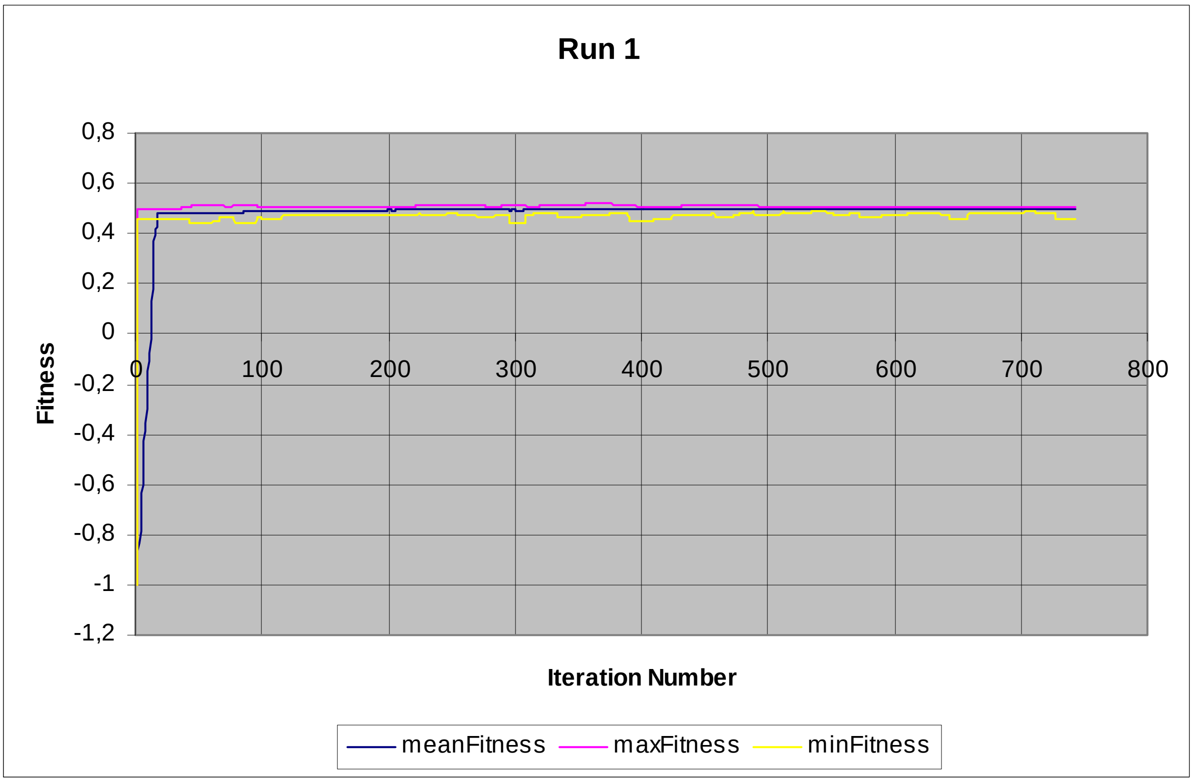


Figure 7. Complete plot of fitness development against iterations for Run 1. This plot also includes the first 24 iterations. See Figure 1.

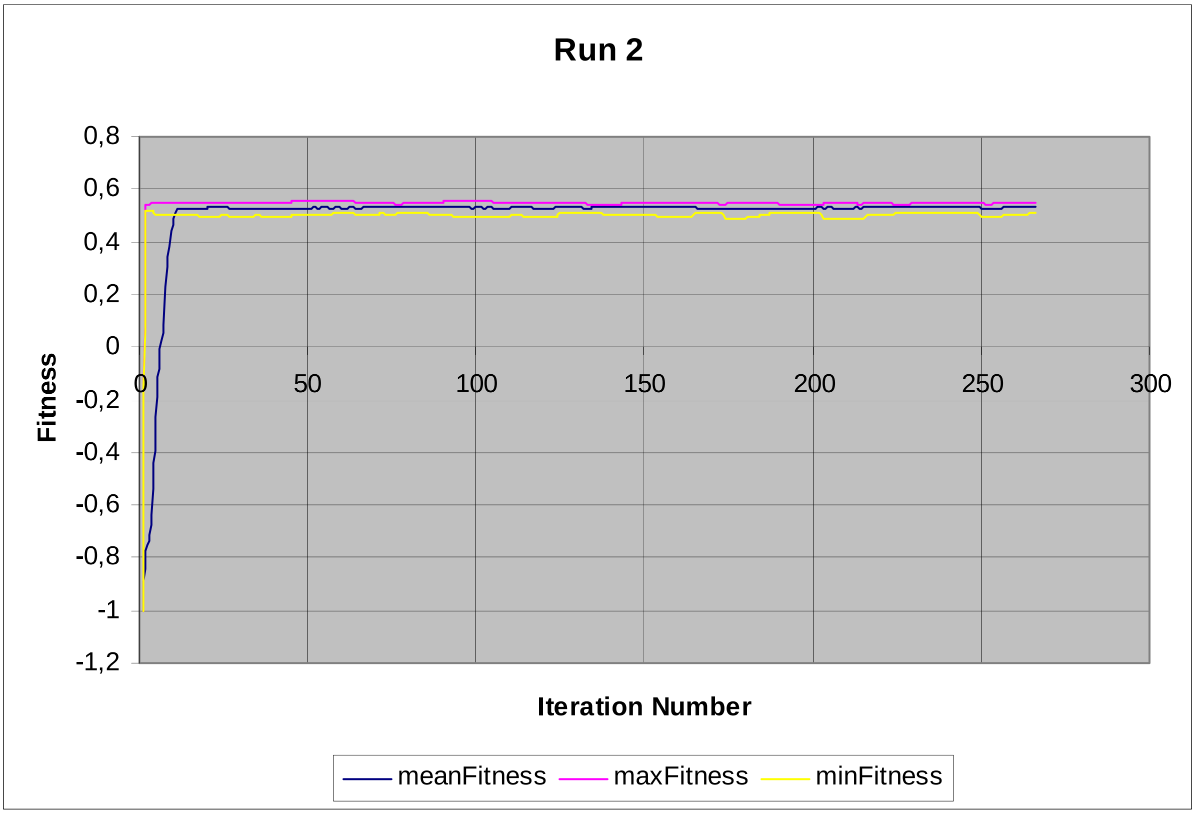


Figure 8. Complete plot of fitness development against iterations for Run 2. This plot also includes the first 10 iterations. See Figure 2.