

Detection of approximate tandem repeats in protein sequences

Master Thesis
Computer Science and Engineering

R.J. Frehen

Eindhoven University of Technology
Department of Mathematics and Computer Science
&
Wageningen University and Research Centrum
Department of Bioinformatics

August 2008

This report was written as the final graduation report for the project “Detection of approximate tandem repeats in protein sequences”. This project was performed as the master project for the master study Computer Science and Engineering at Eindhoven University of Technology by Roger Frehen in 2007/2008, supervised by prof. dr. P.A.J. Hilbers.

The master project was performed as part of the Ph.D. project of Luo Hong, at the department of Bioinformatics at Wageningen University and Research Center, supervised by prof. dr. J.A.M. Leunissen.

Contents

1	Abstract	1
2	Introduction	2
3	Approach	4
4	Definition of an approximate tandem repeat	6
4.1	Tandem repeat	6
4.2	Perfect tandem repeat	6
4.3	Approximate tandem repeat	8
4.4	Amino acids	10
4.5	Amino acid distance matrices	11
4.5.1	PAM	11
4.5.2	BLOSUM	12
4.5.3	GONNET	12
5	Existing algorithms	13
5.1	Internal Repeats Finder	14
5.2	TRUST	15
5.3	REPPER	16
5.4	HHrep	17
5.5	Fire μ Sat	18
5.6	LocRepeat	19

6	Choosing an algorithm	21
6.1	Performance of LocRepeat	21
6.2	Problems of LocRepeat	22
7	LocRepeat algorithm	24
7.1	Example matrix	27
7.2	Calculating the highest scoring region from the matrix	28
7.3	Additional analysis of $w(i,j)$	30
7.3.1	Component 1: $c \cdot i$	31
7.3.2	Component 2: $w(i, j - 1) + \mu(s[j - i], s[j])$	32
7.3.3	Component 3: $w(i - 1, j - 1) + \mu(\Delta, s[j]) + c/2$	32
7.3.4	Component 4: $w(i + 1, j) + \mu(s[j - i], \Delta) - c/2$	33
8	Algorithm improvements	35
8.1	Improvement suggested by the author	35
8.2	Other improvements	36
8.3	From a path in the matrix to a self alignment	36
8.3.1	Example sequence	37
8.4	Storage space	41
8.5	Removing part of the matrix	43
8.6	Removing double sequences	44
8.7	Making distance matrices optional	44
8.8	Custom input settings	44
8.8.1	Explanation of the standard values	45
8.8.2	Changing the input settings	46
9	Performance	48
9.1	Expected running time	48
9.2	Observed running time	48
10	Conclusions	50
	References	50

Appendix I: Extrapolated running times for all sequences in the Swiss-Prot database	53
Appendix II: Standard amino acid abbreviations	55

Chapter 1

Abstract

The department of Bioinformatics in Wageningen maintains a database in which all known perfect tandem repeats (PTRs) in protein sequences are stored. The aim is to extend this database, so it will contain approximate tandem repeats (ATRs) as well. One of the reasons is completeness, but this is not the only one. Another reason is to be able to study the evolutionary causes and the ultimate purpose of such repeats. Several algorithms are known already to find ATRs, but none of them captures the needs of the department of Bioinformatics in Wageningen. The goal of this project is to find an algorithm or a combination of algorithms which finds as many ATRs as possible.

Chapter 2

Introduction

Tandem repeats are presumed to occur frequently in genomic sequences, comprising perhaps 10% or more of the human genome [1]. These tandem repeats might be indications of certain diseases, like Huntington's disease. To be able to study these repeats more thoroughly, possible tandem repeats will be stored in a database. Currently, a database with perfect tandem repeats already exists. These are repeating patterns without any errors. The challenge for this research was to find or create algorithms to populate a database that contains approximate tandem repeats. These are patterns where the repeated pattern is not always exactly the same, but contains some modifications.

The input for our algorithm will be the Swiss-Prot database, a database containing protein sequences. These protein sequences are basically strings over an alphabet of 20 different amino acids, which are denoted by a letter of our normal alphabet. The length of these sequences varies a lot. Some have a length of less than 5 and the longest one currently in the database has a length of over 35000. A very big part of the sequences has a length between 50 and 500 though. The Swiss-Prot database is also used as input for the perfect tandem repeat algorithm. The algorithm used to find perfect tandem repeats is based on suffix trees. For perfect matches this is a very fast and efficient algorithm. For approximate tandem repeats we cannot use

suffix trees, so we needed to look for other algorithms. Several algorithms are known already that try to find repeats with errors in sequences. None of the currently known algorithms matches the need of the department of Bioinformatics though.

Chapter 3

Approach

Before I started to look at any algorithm, I first had to get a notion of what the purpose of the algorithm is. For this, I needed a definition of an approximate tandem repeat. This is done in chapter 4. This chapter also contains the explanation of amino acid distance matrices.

The next step was to search for available existing algorithms. These algorithms were analyzed to judge their usefulness. The following aspects were being looked at.

- Method / approach: What method does the algorithm use? Is this method useful in specific cases only or can it be used for all different kinds of input?
- Is the algorithm mostly suited to find a specific kind of repeats, like shorter or longer repeats?
- Running time and algorithm complexity: what is the expected running time of the algorithm in terms of the input size?

An overview of the algorithms that looked rather promising can be found in chapter 5.

After this overview was made, we had to choose if we wanted to reuse any

of the available algorithms, combine multiple algorithms into a new one or make an entirely new algorithm from scratch. It turned out that there was one algorithm that was rather promising. With some modifications we could use it to find ATRs in big input files within a reasonable time. Sadly, the original developers of this algorithm did not want to give us the source code, because: “The algorithm is not complicated at all. It is easy to follow the algorithm to write the program. Thus, we cannot provide the code.” The implementation of the algorithm, with some changes, was indeed not very complicated, but the writers left out a lot of information, which left some gaps in the program that we had to fill in ourselves. Chapter 6 analyzes the performance and the problems of this algorithm into more detail.

Chapter 7 describes the chosen algorithm itself into detail. This algorithm was implemented at first, but I made several improvements that greatly improved the performance of the algorithm. These improvements are described in chapter 8.

Chapter 10 contains a performance analysis of our new algorithm.

Chapter 4

Definition of an approximate tandem repeat

4.1 Tandem repeat

Tandem repeats are repeating subsequences of a bigger sequence, which are immediately next to each other. If a certain repeating unit appears for example at the start of a sequence and at the end of a sequence, with different amino acids (non-repeating) in between, it would not be a tandem repeat.

The limitation to only consider tandem repeats is a rather important one. It narrows the possibilities a lot and enables the use of certain techniques or algorithms, while it makes other techniques or algorithms too complex.

4.2 Perfect tandem repeat

In the existing database some criteria have been set up already for PTRs. A sequence is only a PTR if it has a certain amount of repeats, based on the size of the repeating unit.

- If the unit size is 1, at least 5 repeating units are needed.

- If the unit size is 2, at least 4 (complete) repeating units are needed.
- If the unit size is 3, at least 3 (complete) repeating units are needed.
- If the unit size is 4 or more, at least 2 (complete) repeating units are needed.

For example, the following are PTRs:

- AAAAA (unit A, unit length 1, 5 complete units)
- ADADADAD (unit AD, unit length 2, 4 complete units)
- ADEADEADE (unit ADE, unit length 3, 3 complete units)
- ADEVADEVAD (unit ADEV, unit length 4, 2 complete repeating units and one incomplete unit)
- ADADAADADA (unit ADADA, unit length 5, 2 complete units)

But the following are not:

- AAAADAA (the “perfect” repeat is interrupted with a D)
- ADADADA (only 3.5 units, so the repeat is too short)

In a more formal way, we can define a PTR as the maximal substring $s_1s_2\dots s_n$ of an input string s , where $s_1 = s_2 = \dots = s_{n-1}$ and s_n is a proper prefix of s_1 .

Because we have different definitions for different values of the length of s_1 , which we will denote by $|s_1|$, there are additional requirements in order to call the subsequence a PTR:

- If $|s_1| = 1$ then $n > 5$
- If $|s_1| = 2$ then $n > 4$
- If $|s_1| = 3$ then $n > 3$
- If $|s_1| > 3$ then $n > 2$

We can put this in one formula as follows: $n > \max(6 - |s_1|, 2)$

4.3 Approximate tandem repeat

The definition of an ATR is not as trivial as the definition of a PTR. The sequence we saw before which was not a perfect repeat because of the interruption with a D might be an ATR. Even if we put another symbol in between, it might still be an ATR. But where do we draw the border? How many “wrong” symbols can a sequence contain before it is not considered an ATR anymore?

One possible solution is to use a distance function to measure the distance between two (consecutive) units. A rather well-known distance function is the Levenshtein distance, which is used in many fields of science. It counts the number of operations (insertions, deletions or substitutions) needed to go from a certain word to another word. For example, to go from “Levenshtein” to “Livingston” would take 6 operations, therefore the Levenshtein distance between these two words is 6.

- (0) Levenshtein
- (1) Livenshtein
- (2) Livinshtein
- (3) Livingshtein
- (4) Livingstein
- (5) Livingstoin
- (6) Livingston

So if in a sequence the subsequence “LEVENSHTEINLIVINGSTON” would appear, it could be seen as an ATR with a certain score of 6. But what does this 6 tell us? Take for example the following two words: “humbug” and “silver”. The distance between those two words is 6 as well, but the words have a lot less in common than the two words in the previous example. Actually, these two words do not have anything in common at all.

So another important factor is the length of the repeating unit. If the unit is longer, more errors should be allowed. This can be solved by not only

counting the mistakes and giving a certain (negative) score for them, but also counting the matches and giving them a certain (positive) score. If we give for example a score of -1 for a mismatch and a score of $+1$ for a match, then the Levenshtein / Livingston example would look as follows, with a $+$ marking a match and a $-$ marking a mismatch:

```
Levenshtein
+---+---+---+
Livingston
```

With 6 pluses and 6 minuses, the score would be $6 - 6 = 0$. The score for humbug / silver would be -6 though, as there are only 6 mismatches and no matches.

Another step that is often added, is having different penalties for deletions, insertions and substitutions. In the Levenshtein / Livingston example there are three substitutions, two deletions and one insertion. But we could also see it as follows:

```
Levenshtein
+---+---+---+
Livingston
```

In this case we have one match less, but one insertion and one deletion have been replaced by two substitutions. Depending on the different (positive and negative) scores, either could give a higher total score.

To be able to make a formal definition of an ATR, we introduce the distance function $D(s_i, s_j)$, which returns a numeric value for the distance between s_i and s_j . If $j = n$ it returns the smallest distance between s_n and any prefix of s_i . We can now define several possible definitions for an ATR.

- An ATR is the maximal substring $s_1s_2\dots s_n$ of an input string s , where $D(s_1, s_i) < C$, $1 < i \leq n$ and C is a certain constant value.
- An ATR is the maximal substring $s_1s_2\dots s_n$ of an input string s , where $D(s_i, s_{i+1}) < C$, $1 \leq i \leq n - 1$ and C is a certain constant value.
- An ATR is the maximal substring $s_1s_2\dots s_n$ of an input string s , where $D(s_i, s_j) < C$, $1 \leq i \leq n - 1$, $1 \leq j \leq n$ and C is a certain constant value.
- An ATR is the maximal substring $s_1s_2\dots s_n$ of an input string s , where $\sum_{i=2}^n D(s_1, s_i) < C$ and C is a certain constant value.
- An ATR is the maximal substring $s_1s_2\dots s_n$ of an input string s , where $\sum_{i=1}^{n-1} D(s_i, s_{i+1}) < C$ and C is a certain constant value.

All these definitions are slightly different from each other and it is not clear which one would be the most intuitive or the most useful. So we are forced to conclude that there is nothing such as *the* definition of an approximate tandem repeat, but that there might be different definitions, which can vary for different situations and different algorithms.

4.4 Amino acids

As we saw before, the input for our algorithm will be a set of protein sequences. These protein sequences consist of amino acids, which are denoted by letters of our normal alphabet. See Appendix II for the full list of standard abbreviations that are used. B, Z and J are ambiguous, as they are used to denote 2 different amino acids. Another special case is the character X, which means that the amino acid in the sequence could not be identified, or was perhaps unknown at the moment the sequence was entered into the database.

These 24 letters are the alphabet that forms the input strings for our algorithm. We ignore the ambiguity posed by B, Z and J, as we will regard them as any of the other letters of the alphabet. Only X will be treated differently. A sequence containing only X's should not be seen as a repeat. Therefore, any comparison with a X should be treated as a mismatch.

4.5 Amino acid distance matrices

Besides having different scores for substitutions, insertions and deletions, there is another possibility to get a more detailed score. Not every substitution is the same. A specific amino acid might have a higher chance to get substituted by a certain one than by another. For this purpose several different substitution matrices have been developed. The most common ones are PAM and BLOSUM.

An amino acid distance matrix is a 2-dimensional matrix which lists all possible amino acids in both dimensions. The cell (i, j) contains the probability that amino acid i will change into amino acid j (but sometimes the matrix is given the other way around, i.e. the probability that j changes into i).

4.5.1 PAM

PAM stands for Percent Accepted Mutations. The first model was made by Margaret Dayhoff in 1966. The matrices are based on known evolutionary changes of amino acids in proteins. A PAM matrix is always listed with a certain number, for example PAM 40 or PAM 120. This number is an indication of how many changes are accepted. The PAM 1 matrix is calculated by only comparing sequences with no more than 1% divergence. So a PAM 0 matrix would accept no changes at all and thus has only 1s on the diagonal and zeros on all other spots. A PAM n ($n > 0$) matrix is simply constructed by multiplying PAM 1 with itself $n - 1$ times. So the higher n , the more distant changes are permitted (with the same score). PAM 250 is generally

seen as a good measure for distant relationships.

4.5.2 BLOSUM

BLOSUM stands for BLOcks SUBstitution Matrix. BLOCKS is the name of the database on which the matrix is based. From the data in this database only very conserved regions are used, i.e. parts of proteins that did not change a lot. In these conserved regions the number of occurrences of each pair of proteins is counted. Based on this the matrix is created. BLOSUM is just like PAM accompanied by a number. This number indicates the percentage of similarity that is allowed between two different protein sequences in the database. If this number is lower, then only less similar proteins are used to calculate the matrix. Note that this is exactly the opposite of the PAM matrix numbers. The idea behind BLOSUM was to make a matrix that gives a better measure for distantly related proteins. BLOSUM matrices were first introduced in 1992. BLOSUM 62 is the default matrix used in BLAST 2.0. Although it was designed for detecting distant mutations, it still performs pretty well for closer relationships. BLOSUM 62 is comparable to PAM 160.

4.5.3 GONNET

Also in 1992, Gonnet, Cohen and Benner developed a new matrix, which is somewhat similar to PAM. A lot of new data was available since PAM was introduced. This data was used to estimate possible alignments of proteins. Based on these alignments a matrix was made. The matrix was used again to improve the alignments and a new matrix was made based on these new alignments. The values in a GONNET matrix are ten times the logarithm of the probability that the amino acids are aligned, divided by the probability that these amino acids would be aligned by chance.

Chapter 5

Existing algorithms

The first step was to make an overview of existing algorithms that could possibly be used to detect ATRs in protein sequences. For this overview, we also looked at algorithms that are mainly used to detect PTRs, but could possibly be adapted for detecting ATRs. Furthermore, we looked at algorithms for finding ATRs in other sequences (for example DNA sequences), which might be used for finding ATRs in protein sequences (after all, it is the same problem, only with a different alphabet).

One of the earliest methods employed is the use of Fourier transforms. In this method the periodicity of certain amino acids or groups of similar amino acids is analyzed. If a certain area contains a lot of amino acids that appear with the same interval, this points out a possible ATR. An example of this method is described by McLachlan [6].

Fourier transforms have some drawbacks though. To be able to detect a repeat, a Fourier method requires a relatively large number of repetitions. Furthermore, there are quite often different repeats, with different repeating unit sizes, that intervene with each other, which severely hampers the chances of these methods to detect the repeats.

Another approach is the use of alignments. For this a protein sequence

is compared to another protein sequence or to itself (self-alignment). These methods try to find local optima with the highest similarity, through the use of dynamic programming. For this a scoring algorithm is used with user-defined penalties for indels and gaps. This method often does not only detect tandem repeats but will also detect distant repeats.

There are also tools that combine both approaches. REPPER [7] is an example of this.

The algorithms in this chapter are sorted by year of publication.

5.1 Internal Repeats Finder

Year: 1999

Authors: Matteo Pellegrini, Edward M. Marcotte and Todd O. Yeates

Method: Self-alignment, modification of Smith-Waterman

Internal Repeats Finder [8,9] is a method for finding repeats in protein sequences. It is based on an extension of the Smith-Waterman algorithm. Therefore the basic technique is finding repeats through self-alignment.

After a repeat is found, a P-value is added, which indicates the significance of the alignment. Multiple results are shown, which are ranked by a certain score. This score is mostly based on the P-value (the smaller the P-value, the higher the score).

Internal Repeats Finder does not look for tandem repeats only, but also finds distant repeats. This probably takes extra running time, but the algorithm is still rather fast.

The tool has a web interface, but it is also possible to download the source code, which is written in C. This would probably ensure even faster running times than through the web interface.

One problem of the tool is that it is unable to handle sequences longer than 2000 proteins. It is not clear if this is a technical shortcoming of the algorithm or that it is just capped to avoid long running times on longer sequences.

Internal Repeats Finder can be found on the following URL:

<http://nihserver.mbi.ucla.edu/Repeats/>

5.2 TRUST

Year: 2004

Authors: Radek Szklarczyk and Jaap Heringa

Method: Statistical significance and transitivity

TRUST is an algorithm for finding repeats in protein sequences. TRUST is an acronym for Tracking Repeats Using Significance and Transitivity. TRUST is an ab initio method, which means that it does not rely on prior knowledge about the sequences that are being processed.

The TRUST algorithm tries to find repeats by making a self-alignment. Once a possible repeat is found, a P-value is calculated. This P-value is the chance that an alignment with the same score would have been obtained by a self-alignment in a random sequence of proteins. Alignments with a P-value lower than 1% are considered significant.

In the article about TRUST [10] it is stated that “It relies on a scheme to assess the statistical significance (P-value) of repeat alignment scores, as opposed to various parameters and arbitrary thresholds used by other methods.” However, the P-value of 1% seems quite arbitrary as well. Why is it 1% and not, for example, 0.5% or 2%?

After the different candidate repeats are found, transitivity is used to match

them to each other. This means that if a certain residue i is matched with a residue j , and j is aligned to k as well, then a correspondence between residues i and k is inferred.

One of the main goals of using transitivity is to be able to detect distant repeats. Because we are interested in tandem repeats only, the method is not really suited for us.

The method as it is right now does not seem very fast. The sample on the web site (TF3A_XENLA, length 366) takes several minutes to run. Part of this time might be caused by the web interface, but it is still rather slow.

TRUST can be found on the following URL:

<http://zeus.cs.vu.nl/programs/trustwww/>

5.3 REPPER

Year: 2005

Authors: Markus Gruber, Johannes Söding and Andrei N. Lupas

Method: Fourier transform and sequence self-comparison

REPPER (REPeats and their PERiodicities) is a combination of two tools. It tries to find repeats in protein sequences by using different approaches. FTwin tries to find periodicities by Fourier transform techniques. REPwin compares the sequence to itself (self-alignment) to find repeats.

FTwin assigns numerical values to amino acids that reflect certain properties, like hydrophobicity. Based on these values, it gives information on periodicities in the sequence.

In the article about REPPER the authors state once more that Fourier transforms can only find repeats if the number of repetitions is large enough. "FT is useful in detecting repeats of any size and nature, provided that the

analysis is made over a window sufficiently large to include many copies of the repeating unit and that the window contains only repeating units of one type.” [7]

REPwin on the other hand uses self-alignment which is focused on finding short tandem repeats. It uses the GONNET similarity matrix to give a score to a mutation of a certain amino acid into another one.

Both FTwin and REPwin use a sliding window to detect repeats of different sizes. The user needs to enter the window size before running the program. Larger windows will give more results, but this also leads to longer running times.

The REPPER tool is combined with two additional tools: PSIPRED, a secondary structure prediction tool, and COILS, a coiled coil prediction tool.

The running time is rather hard to judge. When running an example query in the web interface, it is impossible to see how much time goes into FTwin and how much into REPwin.

REPPER can be found on the following URL:

<http://toolkit.tuebingen.mpg.de/repper>

5.4 HHrep

Year: 2006

Authors: Johannes Söding, Michael Remmert and Andreas Biegert

Method: Transitivity and Hidden Markov Models

HHrep [11] is a method for finding repeats in protein sequences. Just like TRUST, HHrep uses transitivity to find repeats. As said before, transitivity is mainly used to find distant repeats, which are not the main focus for us. Therefore this method might also be too time-consuming.

HHrep is based on self-alignment, but there is a big difference with other methods that work like this. HHrep first constructs a profile Hidden Markov Model (HMM) from the sequence. The HMM is then compared to itself to find the alignments. To construct the HMM, several iterations of PSI-BLAST need to be run.

The tool is web-based. Through a web-interface you contact a server which performs the search. The tool does not seem very suited for automatic handling of large amounts of data though.

Another problem is that for a rather simple query (TF3A_XENLA, length 366) the program took more than half an hour to run. Running the same query again at another time did not give faster results.

HHrep can be found on the following URL:
<http://toolkit.tuebingen.mpg.de/hhrep>

5.5 Fire μ Sat

Year: 2006

Authors: Corné de Ridder, Derrick G. Kourie, and Bruce W. Watson

Method: Finite automata

Fire μ Sat [12,13] is an algorithm for finding micro-satellites in DNA sequences. A micro-satellite is a tandem repeat (either a perfect or an approximate one) with a short unit length. The unit (or *motif* as it is called in this article) can have a length of 2 to 5 nucleotides according to this article. There are other definitions that allow for a length of 1 to 6.

The algorithm constructs a state machine in which the transitions denote the handling of a character from the input sequence. Some states are marked as final states, these states indicate that a PTR or an ATR has been found.

The method has quite some limitations though. First of all, for every possible “motif” a new automaton has to be created. When creating this automaton, you need to know how many mutations are allowed. Allowing more mutations means a more complex automaton. Furthermore, the method is designed for short units only. Longer units would mean more complex and much bigger automata. The third problem is that the method is intended to be used for DNA sequences, which have an alphabet of only four characters. If we would try and use it for protein sequences, which have a basic alphabet of 20 characters, the number of transitions would be a lot higher as well.

The running time of the algorithm can at best be described as reasonable. If only one mutation is allowed, the running time is comparable to other algorithms. If two mutations are allowed for units of length 4 or 5, it is slightly slower already. If we would extend the algorithm for the use of protein sequences and longer repeats, it would end up extremely slow probably.

5.6 LocRepeat

Year: 2006

Authors: Xiaowen Liu and Lusheng Wang

Method: Pseudo-periodic partitioning

LocRepeat is a stand-alone program for finding “pseudo-periodic” repeats in DNA, RNA and protein sequences. A pseudo-periodic repeat is a region of the input sequence which shows strong signs of periodicity. In a pseudo-periodic repeat the different partitions are not compared to one unit, but instead they are compared to their direct neighbors. This approach allows for gradual change of the repeating pattern into a completely different substring.

The program consists of two parts. The first part uses a dynamic programming algorithm to find the subsequence of the input sequence with the

highest similarity score. This substring is used as input for the second step of the program. In this second step, a self-alignment is performed on the new input, which is obviously equal or smaller than the original input.

One of the drawbacks of the program is that it only reports one repeat per input sequence. If two repeats with different unit length are adjacent or close to each other, they might be reported as one big repeat. If two repeats are overlapping, one of them will probably not be reported at all.

The maximum input size for the program is 10000 characters (nucleotides or amino acids). The article suggests to split sequences longer than 10000 into shorter subsequences.

One of the big benefits of this method, from our perspective, is that it only looks for tandem repeats. This means that all results that are found are useful for us and that no computing time is wasted on finding distant repeats.

The article [14] states that the time complexity is $O(n^2)$. The authors did some speed tests which show that this time complexity is about right. It takes 26.9 seconds for a protein sequence of length 10000. The space complexity is $O(n^2)$ as well, but can be reduced to $O(n)$ at the cost of increasing the running time. The tests were done on a PC with a Pentium 4 3.4G CPU and 1GB memory.

The program can be downloaded freely from the following URL:
<http://www.cs.cityu.edu.hk/~lwang/software/LocRepeat/>

This implementation contains some bugs though. After clicking away some error messages, the program can still be used. After installing Visual C++ the errors were gone, so it is probably a library problem.

Chapter 6

Choosing an algorithm

None of the algorithms that we found matched our problem description exactly, but there was one that could possibly be adapted to fit our needs: LocRepeat.

The program has some severe drawbacks though, so we had to implement the algorithm ourselves so we could get the results that we wanted.

First of all, we tried to get the original source code from Dr. Lusheng Wang. He did not want to give us the source code and suggested to recreate it from the algorithm in the article [14].

6.1 Performance of LocRepeat

The article about LocRepeat contains running times for different lengths of input sequences. Furthermore, it is stated that the algorithm runs in quadratic time. The running times in the article (see table 1) confirm the quadratic running times. I used these running times and extrapolated them to get an estimate for the total running time for all sequences in the Swiss-Prot database.

For this, I first made an overview of the sequence lengths of all sequences in

the Swiss-Prot database. See Appendix I for this overview.

For each category, I extrapolated the average running time. Then I multiplied this average running time with the number of sequences with the appropriate length. Summing all categories, the total expected running time would be around 7 hours.

The LocRepeat program cannot handle sequences longer than 10000 proteins. If a sequence is longer, the article suggests to split it into smaller parts. If we split all sequences longer than 10000 proteins into subsequences, with an overlap of 5000 proteins each, the total expected running time will reduce, due to the quadratic running time.

The Swiss-Prot database contains currently only 5 sequences longer than 10000 proteins. If we split those, the total expected running time will be reduced by about 6 minutes.

Furthermore, I calculated if it would be worth splitting the input sequences into even smaller chunks. If we split up every sequence longer than 2500 proteins into chunks of 2000, we could gain another 26 minutes.

Even without the proposed splitting, the running time is 7 hours, which is acceptable.

Length in amino acids	2000	4000	6000	8000	10000
Running time in seconds	1.1	4.3	10.0	17.7	26.9

Table 1: Running times for protein sequences for LocRepeat. Table taken from [14]. Tests are performed on a 3.4GHz Pentium 4 with 1GB memory.

6.2 Problems of LocRepeat

The biggest problem that LocRepeat has, is that it can only report one repeat, the one with the highest score. At the end of the article, it is suggested already that the algorithm could be adapted to report more repeats. This is

one of the main things we had to change in our own implementation.

If our own implementation is able to report multiple repeats, this will also solve the problem that overlapping repeats are not reported in the original LocRepeat program.

Furthermore, we would like to have a program that is suitable for handling very large numbers of input sequences automatically. This is also a point on which our own implementation will be focused.

Chapter 7

LocRepeat algorithm

The LocRepeat program consists of two parts. The first part is the core algorithm of the program, which finds the “pseudo-periodic region” [15]. The second part performs a self-alignment on this region to split it into the different units.

In chapter 4.3 we saw that several different definitions for an ATR can be made. The definition that LocRepeat uses is similar to the last definition we made:

An ATR is the maximal substring $s_1s_2\dots s_n$ of an input string s , where $\sum_{i=1}^{n-1} D(s_i, s_{i+1}) < C$ and C is a certain constant value.

There are some differences though. The first one is only a small difference. Where we defined $D(s_i, s_j)$ as the distance function, with a higher score meaning a bigger biological distance between the two arguments s_i and s_j , is the function used in the article [14] a similarity measure, meaning a higher score for smaller distance between the arguments. This implies that the total sum shouldn't be smaller than a certain constant C , but bigger than a constant instead. LocRepeat narrows the solution set to just one subsequence of the input string: the one with the highest score.

The real difference is that the LocRepeat algorithm introduces another vari-

able, c , called the granularity factor. This granularity factor is multiplied by the size of the edges of the subsequence that are not matched to symbols in the sequence, but to spaces instead. $c/2$ needs to be bigger than the distance between any element of the alphabet and a space in the alignment. This condition ensures that the end of an alignment is not mapped towards spaces, which would create a self-alignment ending with one or multiple deletions. This leads to the following definition:

The output of LocRepeat is the substring $s_1s_2\dots s_n$ of input string s with the highest $\sum_{i=1}^{n-1} D(s_i, s_{i+1}) + \frac{c}{2} \cdot (|s_1| + |s_n|)$ for all possible substrings of s .

Now we define $w(i, j)$ as the maximum self-alignment value of a suffix t_j of $s[1, j]$, such that there are i letters at the right end of the self-alignment of t_j that are aligned with spaces.

$i = 0$ means that no single character is aligned with a space or in other words: the last character is matched with itself. This is a situation that we do not want to occur, therefore $w(0, j)$ is set to $-\infty$ for all j .

$i = j$ means that every single character is aligned with a space. As we saw before, $c/2$ by definition yields a higher score than aligning a character with a space. Because both ends of the sequence get the penalty $c/2$, the value of $w(j, j)$ is $c \cdot j$ for all j .

For $0 < i < j$ we compute $w(i, j)$ as follows. See [14] for a proof.

$$w(i, j)_{0 < i < j} = \max \begin{cases} c \cdot i \\ w(i, j-1) + \mu(s[j-i], s[j]) \\ w(i-1, j-1) + \mu(\Delta, s[j]) + c/2 \\ w(i+1, j) + \mu(s[j-i], \Delta) - c/2 \end{cases}$$

where:

- $s[k]$ denotes the k -th position in input sequence s (with the first symbol being $s[1]$)
- Δ denotes a space in an alignment
- $\mu(x, y)$ is the distance function for symbols x and y

The first step of the LocRepeat algorithm creates a 2-dimensional matrix with size n . Only one half of the matrix is used, as $i > j$ would make no sense. This would mean that there would be more characters matched to spaces than the length of the (sub)string.

The matrix is filled as follows:

```
for j = 1 to n do
  for i = j downto 1 do
    compute w(i, j)
```

Now we can also verify that $w(i, j)$ never uses values that are unknown at the point of calculation.

If $j = 1$, only $w(1, 1)$ is calculated. As it is of the form $w(j, j)$, its value will be $c \cdot j$ or simply c .

If $j > 1$ and $i = j$ then $w(i, j) = w(j, j) = c \cdot j$.

If $j > 1$ and $1 < i < j$ then $w(i, j)$ is calculated using the following three cells:

- $w(i, j - 1)$: because $j > 1$ we have $j - 1 > 0$ and because $i < j$ the value of $w(i, j - 1)$ has been calculated in the previous (outer) loop.
- $w(i - 1, j - 1)$: similar as the previous point and because $i > 1$ we have $i - 1 > 0$, so $w(i - 1, j - 1)$ has been calculated before as well.
- $w(i + 1, j)$: because $i < j$ we have $i + 1 \leq j$, so $w(i + 1, j)$ was the previous cell that was calculated.

If $j > 1$ and $i = 1$ then we get the special case $i - 1 = 0$ in the second item of the list above. As $w(i, j) = 0$ for $i = 0$, this value can also be calculated.

7.1 Example matrix

The article contains an example matrix for the sequence $s = \text{CAGAGT}$. The input variables are set as follows:

$j \backslash i$	6	5	4	3	2	1	0
C 1						C -2	$-\infty$
A 2					C -4	A -2	$-\infty$
G 3				C -6	A -4	G -2	$-\infty$
A 4			C -8	A -6	G 6	A -2	$-\infty$
G 5		C -10	A -8	G -6	A 16	G 7	$-\infty$
T 6	C -12	A -10	G -8	A 5	G 6	T -2	$-\infty$

Figure 1: Image taken from [4], showing the matrix created for sequence $s = \text{CAGAGT}$.

$$c = -2$$

$$\text{a match } (\mu(x, x)) = 10$$

$$\text{a mismatch } (\mu(x, y), x \neq y) = -10$$

$$\text{and a comparison with a space } (\mu(\Delta, x)) = (\mu(x, \Delta)) = -10$$

The matrix from the article is shown in figure 1. Several remarks can be made.

First of all, the rows are addressed as j and the columns as i . The values in the cells are addressed as $w(i, j)$ though. Usually, an element from a matrix is addressed as (row, column). In this example it is done the other way around, which is rather confusing.

Another problem with the matrix is that it actually contains an error. The value in cell $(i, j) = (3, 5)$ should be computed as follows:

$$w(i, j) = \max \begin{cases} c \cdot i \\ w(i, j - 1) + \mu(s[j - i], s[j]) \\ w(i - 1, j - 1) + \mu(\Delta, s[j]) + c/2 \\ w(i + 1, j) + \mu(s[j - i], \Delta) - c/2 \end{cases}$$

Filling in the values for $i, j, c, \mu(\Delta, x)$ and $\mu(x, \Delta)$, we get:

$$w(i, j) = \max \begin{cases} -2 \cdot 3 \\ w(3, 4) + \mu(s[2], s[5]) \\ w(2, 4) - 10 - 1 \\ w(4, 5) - 10 + 1 \end{cases}$$

Filling in the rest, we get:

$$w(i, j) = \max \begin{cases} -2 \cdot 3 \\ -6 + \mu(A, G) \\ 6 - 10 - 1 \\ -8 - 10 + 1 \end{cases}$$

$\mu(A, G)$ is a mismatch, so gets a score of -10 .

The maximum of these is the third one, with a total of -5 , but the matrix shows -6 .

7.2 Calculating the highest scoring region from the matrix

From the cell that contains the highest value in the matrix, the “pseudo-periodic region” can be calculated. For this, a backtracking mechanism is used. Through the matrix a path is constructed, starting from the cell with the highest value, ending at a cell with value $c \cdot i$. This path is based on the $w(i, j)$ function. The value of $w(i, j)$ is based on the maximum of the following four components:

$$w(i, j) = \max \begin{cases} c \cdot i \\ w(i, j-1) + \mu(s[j-i], s[j]) \\ w(i-1, j-1) + \mu(\Delta, s[j]) + c/2 \\ w(i+1, j) + \mu(s[j-i], \Delta) - c/2 \end{cases}$$

The component that actually makes the maximum contains the next step on the path.

If we take the example matrix from the article again, we can mark the path, starting at the highest scoring cell (2, 5). This is done in figure 2.

j \ i	6	5	4	3	2	1	0
C 1						C	
						-2	$-\infty$
A 2					C	A	
					-4	-2	$-\infty$
G 3				C	A	G	
				-6	-4	-2	$-\infty$
A 4			C	A	G	A	
			-8	-6	6	-2	$-\infty$
G 5		C	A	G	A	G	
		-10	-8	-6	16	7	$-\infty$
T 6	C	A	G	A	G	T	
	-12	-10	-8	5	6	-2	$-\infty$

Figure 2: The same matrix as in figure 1, but now showing the path from (2, 5) via (2, 4) to (2, 3).

The value of cell (2, 5) is 16, which is $w(i, j-1) + \mu(s[j-i], s[j])$. Therefore the path goes to cell $(i, j-1) = (2, 4)$. The value in (2, 4) is similarly based on cell (2, 3). The value in cell (2, 3) is equal to $c \cdot i$, so the path ends there.

The starting and the ending point of the path define the highest scoring subsequence as follows.

The end of the subsequence is the j -value of the starting point.

The start of the subsequence is based on two things. The j -value of the ending point is the endpoint of the first unit in the subsequence. The length of this first unit is the i -value of the ending point. So the start of the subsequence is $j - i + 1$, (with i and j the i and j -values of the ending point of the path).

In the example, the j -value of the starting point is 5.

The j -value of the ending point is 3, so the endpoint of the first unit is 3. The i -value of the ending point is 2, so the length of the first unit is 2. With unit length 2, the start of the subsequence is $3 - 2 + 1 = 2$.

With starting point 2 and ending point 5, the highest scoring subsequence in the example is AGAG. That subsequence will be used then as input for the second step, which performs a self-alignment on this subsequence.

7.3 Additional analysis of $w(i,j)$

The $w(i,j)$ function is based on four components. The maximum of these four components is chosen. The path through the matrix depends on which of these four components actually forms this maximum.

The four components of the $w(i,j)$ function are:

$$c \cdot i$$

$$w(i, j - 1) + \mu(s[j - i], s[j])$$

$$w(i - 1, j - 1) + \mu(\Delta, s[j]) + c/2$$

$$w(i + 1, j) + \mu(s[j - i], \Delta) - c/2$$

We will look into detail at all four components.

7.3.1 Component 1: $c \cdot i$

This is the value given to the cell if none of the other three options yield a higher result. c is a negative constant, called the granularity factor (actually, $c/2$ is the granularity factor). This constant (which can be chosen by the user) is used in two other components as well.

If $c \cdot i$ is the maximum of the four components, it will be the endpoint of a path through the matrix, if that path reaches this cell. Having $c \cdot i$ as maximum basically means the start of a new, empty alignment. We will see that the other three components are extensions of existing alignments.

The score for an empty alignment depends on constant c and on i . Because c is negative, this means that a higher i gives a lower (more negative) score. As we saw before, i marks the length of the first unit of an alignment and therefore also the number of characters over which the input sequence is shifted.

According to the article, the first and the last unit in a subsequence are not matched to other characters and therefore get a penalty score for aligning these characters to spaces. This penalty score is the granularity factor, $c/2$. Because the penalty score is added for both ends of the alignment, the starting score for an empty alignment is $c/2 \cdot i \cdot 2 = c \cdot i$.

One very big problem with the granularity factor was that the article about the algorithm [14] and the implementation of the program are not consistent with each other. According to the article about the algorithm, the granularity factor was trained using the Swiss-Prot database, resulting in a value of 0.52. When using the LocRepeat program though, the user can set the granularity factor, but this needs to be a negative integer value between -1 and -100 . 0.52 is neither negative, nor integer. The article about LocRepeat does not contain any information about conversion of the original granularity factor.

7.3.2 Component 2: $w(i, j - 1) + \mu(s[j - i], s[j])$

The second component takes the value from cell $(i, j - 1)$. The characters in the input sequence at position $j - i$ and position j are compared, using the distance function μ .

Cell $(i, j - 1)$ is the alignment with unit length i up to position $j - 1$ in the sequence. This component simply extends that alignment with one character, keeping the unit length the same.

Take for example cell $(i, j) = (2, 4)$ in the example matrix in figure 1. $w(i, j - 1)$ is the value in cell $(2, 3)$. This cell contains value -4 , which is $c \cdot i$, so it marks the start of an empty alignment.

Component $w(i, j - 1) + \mu(s[j - i], s[j])$ of cell $(2, 4)$ extends the (empty) alignment of cell $(2, 3)$ with one character. The A from $s[4]$ is matched to the A from $s[4 - 2]$, so we have an alignment like:

```
CAGAGT
 |
CAGAGT
```

When we arrive at $(2, 5)$, the current alignment can be extended even further. The G from $s[5]$ is matched to the G from $s[5 - 2]$ and thus giving alignment:

```
CAGAGT
 ||
CAGAGT
```

7.3.3 Component 3: $w(i - 1, j - 1) + \mu(\Delta, s[j]) + c/2$

This component gets its basic value from cell $(i - 1, j - 1)$. Just like the former component, this value is taken from “row” $j - 1$. The next character from the input sequence, $s[j]$, is not compared to the character that is shifted over i positions. Instead it is compared to a blank character. This means that this component marks an insertion in the alignment.

Another thing we see here is that the value is not taken from the same column i , but from column $i - 1$ instead. An insertion means that the length of the unit that we are trying to match needs to be extended as well. It is an important notion that column i does not only mark the length of the first unit at the end of a path through the matrix, but that it also marks the length of the “current” unit that we are trying to match. Later, when we try to simplify the algorithm, we will make very good use of this.

Besides adding the score of a (mis)match with a blank character, another factor is added: $c/2$. As we saw earlier, this is the granularity factor. Because the characters of the first and the last unit get a penalty, because they are not matched to other characters, we need to add the granularity factor another time if the current unit length is extended. Adding a gap extends the length of the last unit by one and thus the (negative) factor $c/2$ is added to the score.

In the example matrix we can take a look at cell $(3, 6)$ to see how an alignment is extended with a gap. Component $w(i - 1, j - 1) + \mu(\Delta, s[j]) + c/2$ gets its basic value from cell $(i - 1, j - 1)$, so cell $(2, 5)$. In the last paragraph we saw the alignment up to this cell:

```
CAGAGT
  ||
  CAGAGT
```

Now if we extend this alignment with a gap at the end, we get the following alignment:

```
CAGAGT
  ||:
  CAG AGT
```

7.3.4 Component 4: $w(i + 1, j) + \mu(s[j - i], \Delta) - c/2$

The fourth and last case marks a deletion. It takes the value from cell $(i + 1, j)$. This cell is on the same row j . Staying on the same row means

that there is no additional character added to the subsequence.

Cell $(i + 1, j)$ is in another column though. Where we saw in the last case that decreasing i by one marks an insertion, increasing i by one similarly marks a deletion. A deletion means that the next character from the shifted sequence will not be matched to a character from the input sequence, but to a blank character instead. Therefore the factor $\mu(s[j - i], \Delta)$, comparison with a space, is added.

Furthermore, because the length of the last unit is reduced by one, the granularity factor $c/2$ is subtracted from the result. This is also similar to an insertion, where the granularity factor was added instead.

For example, take a look at cell $(1, 5)$ in the matrix. It gets its basic value from cell $(2, 5)$, which is the alignment that we saw a few times already:

```
CAGAGT
  ||
  CAGAGT
```

Now if we add a deletion, we get the following alignment:

```
CAGAG T
  ||:
  CAGAGT
```

Chapter 8

Algorithm improvements

8.1 Improvement suggested by the author

The algorithm as shown so far can be improved in multiple ways. The original article suggests to store the start position of the subsequence in a separate array. As we saw before, the start position of the subsequence is based on both the i and the j -value of the end position of the path through the matrix. When doing this, only the current row and the previous row of the matrix need to be stored, therefore lowering the space complexity from $O(n^2)$ to $O(n)$. Adding this to the algorithm we get:

```
for j = 1 to n do
  for i = j downto 1 do
    compute w(i,j); compute start(i,j)
```

with $start(i, j)$ the start of the subsequence that ends at cell (i, j) .

The value of $start(i, j)$ depends on which component of $w(i, j)$ gives the actual maximum:

- If $c \cdot i$ is the maximum $\rightarrow start(i, j) = ji + 1$
- If $w(i, j-1) + \mu(s[j-i], s[j])$ is the maximum $\rightarrow start(i, j) = start(i, j-1)$

- If $w(i - 1, j - 1) + \mu(\Delta, s[j]) + c/2$ is the maximum $\rightarrow start(i, j) = start(i - 1, j - 1)$
- If $w(i + 1, j) + \mu(s[j - i], \Delta) - c/2$ is the maximum $\rightarrow start(i, j) = start(i + 1, j)$

8.2 Other improvements

The algorithm as we have it thus far only returns a subsequence of the original input sequence. The LocRepeat program performs a self alignment on this subsequence to split it into the different units. This is a process with running time $O(m^2)$, with m the size of the subsequence. Because m is smaller than or equal to n , the time complexity of the whole program is still $O(n^2)$.

One of the most important improvements for us was to report multiple alignments. Where LocRepeat only reports the highest scoring alignment, we want to list all alignments that score above a certain threshold score. If we would use the same approach as LocRepeat for this, we would end up with running time $O(n^2) + k \cdot O(m^2)$, where k is the number of subsequences found. As the number of possible subsequences is a quadratic function of the input size, the worst case running time would be close to $O(n^4)$.

One important thing which the creators of LocRepeat missed is that the algorithm already performs a self alignment on the original sequence. The clue is to store this alignment in a convenient way.

8.3 From a path in the matrix to a self alignment

We already saw that the core of the algorithm, the computation of $w(i, j)$, gives an indication about the alignment that we are trying to make. The four cases either mark a new alignment or extend the current alignment with a match, an insertion or a deletion. If we store this information into an array,

we can make the alignment easily.

The improvement suggested by the author marks a first step into the right direction already. Storing the start position of the subsequence greatly reduces the required storage space. If we store the full path through the matrix as well, we will require more storage space again, but it completely removes the need for the second step of the LocRepeat program: the self-alignment on the subsequence.

In the former chapter we briefly saw the creation of a self-alignment based on the component that forms the maximum in $w(i, j)$.

- If $c \cdot i$ is the maximum \rightarrow new alignment
- If $w(i, j - 1) + \mu(s[j - i], s[j])$ is the maximum \rightarrow extension with one character
- If $w(i - 1, j - 1) + \mu(\Delta, s[j]) + c/2$ is the maximum \rightarrow extension with an insertion
- If $w(i + 1, j) + \mu(s[j - i], \Delta) - c/2$ is the maximum \rightarrow extension with a deletion

Suppose we store N for a new alignment, E for an extension, I for an insertion and D for a deletion.

8.3.1 Example sequence

Take for example the following sequence with length 25:

CAGAGTAGTAGTAGAGAGTAGTAGT

Now suppose the algorithm would return the subsequence [2,25] as the highest scoring subsequence and the array with the path would be as follows:

NEEEEEEEEEEEEEEEEEEEEEEEEE

Note that the path through the matrix would actually be the reverse of this sequence.

This would lead to the following alignment:

```
CAGAGTAGTAGTAG AGAGTAGTAGT
  ||:|||||||:||||:|||||
NEEEEEEEEEEEEEEEEEEEEEEEEE
  ||:|||||||:||||:|||||
AG AGTAGTAGTAGAG AGTAGTAGT
```

The spaces in the first sequence mark a deletion (D) and the spaces in the second sequence mark an insertion (I). The vertical strokes on the second and fourth line mark a match, the semi-colons mark either an insertion or a deletion.

The improvement suggested by the author required us to store the start position of the subsequence leading to the current cell. Instead of storing this start position, we will store the two values that lead to this start position, namely i and j of the last cell of the path through the matrix.

As mentioned earlier, the i -value of any cell in the matrix is equal to the number of characters over which the sequence is shifted, and therefore equal to the size of the current repeating unit. An insertion in the algorithm means that i is increased by one, so the current unit size is also increased by one. A deletion works the opposite way: decreasing i by one means the current unit size is shortened by one. Now if we project this on our example, we get:

```

CAGAGTAGTAGTAG AGAGTAGTAGT
  |:|||||:||||:|||||
NEEEEEEEEEDEEEEEEEEEEE
  |:|||||:||||:|||||
AG AGTAGTAGTAGAG AGTAGTAGT

```

Unit size: 222333333333222223333333

```

      *      *      *
      +1     -1    +1

```

Using this unit size information, we can make an alignment, by counting how many characters we matched already (a substitution would count as a match for this matter). If that number is equal to the current unit size, a new unit is started and the number of matched characters is reset to zero. To keep track of the number of characters we matched already, we use `Current_Unit_Pos`. The algorithm looks as follows:

```

IF (N) -> Current_Unit_Size = i;
          Current_Unit_Pos = 0;

(E) -> Current_Unit_Pos ++;
      IF (Current_Unit_Pos == Current_Unit_Length) ->
          // The Current unit is "full"
          Current_Unit_Pos = 0;
      FI

(I) -> Current_Unit_Size ++;
      Current_Unit_Length ++;

(D) -> Current_Unit_Size --;
      IF (Current_Unit_Pos == Current_Unit_Length) ->
          // The Current unit is "full"
          Current_Unit_Pos = 0;
      FI
FI

```

Applying this to the example, we get:

```

CAGAGTAGTAGTAG AGAGTAGTAGT
  |:|||||||:||||:|||||
NEEEEEEEEEEEDEEEEIEEEEE
  |:|||||||:||||:|||||
AG AGTAGTAGTAGAG AGTAGTAGT
Unit size: 222333333333222223333333
Unit pos:  012123123123012121231231
          * * * * * * * *

```

The stars indicate the spots where the number of matched characters matches the current unit size and `Current_Unit_Pos` is reset to zero. In other words, the stars mark the end of a unit. We should add one more star to the start of the alignment, as we know already that the N marks the end of the first unit.

```

CAGAGTAGTAGTAG AGAGTAGTAGT
* * * * * * * *

```

From the combination of the original sequence and the stars, which mark the end of a unit, we can make the full self-alignment, including the partitioning into units. In the example, the last T is reported as the last (incomplete) unit.

```

CAGAGTAGTAGTAG AGAGTAGTAGT
* * * * * * * *
AG TAG AG TAG T
AG TAG TAG AG TAG

```

8.4 Storage space

The problem with the suggested change is that it would require quite a bit of storage space. For every cell a path is stored, which can have size n . We only need to store two rows of the matrix, but with n cells on the last row, this means that we are back to quadratic space complexity: $O(n^2)$.

Instead of storing the full path, we can choose to store the last position of every unit only (the positions of the arrows). In the example above this would mean we would store the following numbers:

```
CAGAGTAGTAGTAG AGAGTAGTAGT
* * * * * * * * *
3 5 8 11 14 16 18 21 24
```

This leads to another problem though. Because it is not known in advance how many units there are in the alignment, you would either have to use a dynamic array to store the positions or limit the maximum number of end positions you store for a repeat. A dynamic array leads to increased running time. Because there are only very few cases where the number of units is very big, we chose for the latter one and used a maximum of 100 units to be reported. The rest of the subsequence is reported as the last part, instead of the last (incomplete) unit. Example:

```
AAAAA... ..AAAAA   AAAAA
12345                99 100
```

The array of end positions would here be filled with the values 1...99. The first 99 units would be reported as “A”. The last unit would be reported as “AAAAA”, which is the remaining part of the subsequence.

Another reason why we store the end positions and not the full path is a technical one. To store the full path, we wanted to use 2 booleans. This seemed a logical choice, because there can be four different values (N, E, I and D). C++ uses 8 bits (1 byte) to store a boolean variable though. This means that the approach with 2 booleans used 2 bytes of storage space per position on the path.

An unsigned short integer uses the same 2 bytes of storage space. Therefore our solution that stores only the first 99 end positions is a lot more space efficient.

A small drawback is that an unsigned short can only store values up to 65535, but with the longest protein currently present in the Swiss-Prot database having a length of 35213 amino acids, we feel rather safe with this approach. Furthermore, the algorithm that is currently used for detecting perfect repeats limits the input size to 40000 characters. The LocRepeat program has a maximum input size of 10000 characters only.

It might have been possible to use a bit field to store the path. In that way, it would not take 2 bytes, but only 2 bits to store a value from the path. If we compare the total memory required for the longest protein, we can see that our current approach still outperforms this approach.

Bit field approach:

- 2 bits per path position
- Maximum path length: 35213
- $35213 \cdot 2 = 70426$ bits per cell in the matrix

Our approach:

- 2 bytes per end position
- Maximum number of end positions: 100

- $100 \cdot 2 \cdot 8 = 1600$ bits per cell in the matrix

The other thing we can see is that our approach takes constant memory space per cell in the matrix, so $O(n)$ in total. The bit field approach takes linear space per cell, so $O(n^2)$ in total.

8.5 Removing part of the matrix

Because of our definition of a tandem repeat, we do not need to calculate the whole matrix. A tandem repeat needs to have at least two complete units (or even more if the unit length is shorter than 4). In other words, a tandem repeat needs to have a length of at least twice the length of the first unit. Therefore, the unit length of a repeat can never be longer than half the size of the input sequence.

As we saw before, column i contains the scores of a self-alignment with unit length i . Because the maximum unit length is $n/2$, with n the size of the input sequence, we do not need to calculate the values of $i > n/2$.

Instead of the nested loop we had before, we now get:

```
for j = 1 to n do
  for i = min(j, round(n/2)) downto 1 do
    "algorithm"
```

$n/2$ needs to be rounded to an integer value, or the loop will never finish.

This improvement leads to a significant reduction of the number of values in the matrix that need to be calculated. If we would draw the matrix like in figure 1, the “left” half of the matrix can be left out. The top half of this left half was empty already and the bottom half was only half filled. This means that about 1/8 of the total matrix can be left out now. Because only half of the matrix was filled, this addition leads to a reduction of about one quarter of the number of values in the matrix.

8.6 Removing double sequences

Our algorithm might report some subsequences of the input sequence twice. Take for example the following subsequence:

```
AGTAGTAGTAGT
```

This sequence can be seen as a repeat with four units with unit length 3, but it could also be a repeat with two units with unit length 6.

To prevent reporting a subsequence twice, the scores of all subsequences with the same start and end position are compared and only the highest scoring one is reported.

8.7 Making distance matrices optional

Our first implementation of the algorithm used an amino acid distance matrix as an additional input file, similar to LocRepeat. This distance matrix would be used to calculate the distance between any two characters in the input sequence.

The use of a distance matrix is not always wanted though. In the first place, it is very difficult to determine which distance matrix to use. Different amino acids in a sequence might have gone through different levels of changes. Therefore you might want to use a PAM 40 matrix for example for a certain part of the input sequence and a PAM 120 matrix for another part.

We decided to make a second version of the program: one without matrices at all. In this version only three values are used: a score for a match, a score for a mismatch and a score for an insertion or a deletion.

8.8 Custom input settings

The program can be run with several arguments. One of those is a settings file, in which the following values can be set. If no settings file is specified

when calling the program, the standard values are used.

gapcost, standard = 8

This is the penalty for an indel (insertion or deletion).

matchscore, standard = 5

This is the score for a matching character.

mismatchscore, standard = -5

This is the score for a mismatch.

$c/2$, standard = -1

This is the granularity factor, which needs to be a negative integer.

THRESHOLDScore, standard = 0

This is the minimal score that a possible repeat needs to have to be reported.

MAXNUMBEROFRESULTS, standard = 1000

This is the maximum number of repeats that is reported per sequence.

8.8.1 Explanation of the standard values

The above listed standard values are not arbitrary numbers.

For most of the tests that I performed, I used the PAM 120 matrix. In this matrix, the gap score is defined as -8, so this leads to the standard gap penalty of 8.

To get the standard match score, I looked at the diagonal of the PAM 120 matrix, which are the scores for perfect matches. The value 5 was both the mode (the most occurring one) and the median (the middle one, if you put all numbers in ascending order). Therefore I took 5 as the standard match score. The standard mismatch score of -5 is simply the negative of the match score.

The standard value of the granularity factor is -1 , because this is the standard value in the LocRepeat program as well.

Finally, the threshold score and the maximum number of results are rather arbitrary. The threshold score of 0 makes sure that everything which does not have a negative score will be reported. The maximum number of 1000 results per sequence will probably never be reached for any known sequence. It is just present to be able to set the maximum if desired.

8.8.2 Changing the input settings

The number of results found for a set of input sequences will change when the input parameters are different. Therefore I performed some tests with different values for the custom input settings.

For this test I used a file which contains all protein sequences in Swiss Prot that have the organism type plants. The results were as follows.

First, I tested with the standard settings. Then I used the other version of the program, which allows the use of an amino acid distance matrix as an additional input parameter. The results show that using a matrix leads to a huge increase in the number of results. This is probably a lot of noise.

Standard settings: 9891 results

PAM 120 matrix: 245737 results

Next, I tried changing the granularity factor to see what the effect would be.

Standard settings, but $c/2 = -2$: 6930 results

Standard settings, $c/2 = -10$: 214 results

PAM 120 matrix, $c/2 = -10$: 43163 results

An interesting note is that changing $c/2$ to -10 resulted in mostly results with very short unit length, almost always the unit length was 1.

Another thing I tried was changing the scores for matches, mismatches and gaps.

Standard settings, match = 10: 93340 results

Standard settings, match = 2: 4898 results

Standard settings, mismatch = -10 : 7631 results

Standard settings, mismatch = -2 : 95224 results

Standard settings, gapcost = 2: 95684 results

Standard settings, gapcost = 16: 8342 results

PAM 120 matrix, gapcost = 16: 399401 results

Chapter 9

Performance

9.1 Expected running time

Before implementing the algorithm, we made an estimate of the running time required when using the whole Swiss-Prot database as input. These estimates were based on the typical running times given in the LocRepeat article and extrapolating them for the input sizes that were not listed. This led to an expected running time of about 7 hours if we would use the same hardware.

9.2 Observed running time

The first run on the entire Swiss-Prot database took about one and a half hours. This was done on a machine with similar — but slightly inferior — hardware as the machine mentioned in the article.

To be able to compare the running times even better, we performed some tests on a sequence with length 9159. In LocRepeat, the running times of both phases can be distinguished. Both phases took around 12 seconds on the machine I tested with, for a total of 24 seconds. The memory use of the first phase was negligible, but the second phase took over 1 GB of memory.

Running the same sequence in our algorithm, we got a running time of 7 seconds and a memory use of 10 MB only. In this time, we found multiple alignments, where LocRepeat reported only one.

Our algorithm still has a quadratic running time though. The target database, Swiss-Prot, contains a lot of short sequences, which are processed very fast. The longer sequences could have been split to speed up a run on the entire database, but we chose not to do this, to avoid loss of functionality. A (purely theoretical) repeat sequence of more than 10000 characters would have been impossible to detect if we would split the sequence into parts of 10000 amino acids.

Chapter 10

Conclusions

We developed a very fast algorithm for finding approximate tandem repeats in amino acid sequences. The algorithm is able to handle very large chunks of input data containing sequences up to 65535 characters.

The algorithm is based on another, existing algorithm. We significantly improved this algorithm, by removing redundant calculations and adding extra functionality. Removal of the redundant calculations led to a great reduction of the running time. The extra functionality that we added was the detection of multiple repeats in one input sequence, the ability to process large input files which contain multiple input sequences and the ability to handle input sequences of more than 10000 characters. These extra functions are performed in a running time which still greatly outperforms the original algorithm, both in running time and in memory requirements.

The algorithm is by no means limited to handling amino acid sequences, but can use any input string and search for repeating fragments in it. The algorithm can be run with a substitution matrix as additional input as well, allowing for different values for different substitutions.

References

- [1] Gary Benson; Tandem repeats finder: a program to analyze DNA sequences. *Nucleic Acids Research*, 1999, Vol. 27, No. 2, 573–580.
- [2] Philippe Le Flèche; A tandem repeats database for bacterial genomes: application to the genotyping of *Yersinia pestis* and *Bacillus anthracis*. *BMC Microbiology* 2001, 1:2 doi:10.1186/1471-2180-1-2.
- [3] Jaap Heringa; The evolution and recognition of protein sequence repeats. *Computers Chemistry*, 1994, Vol. IS, No. 3, 233–243.
- [4] MV Katti, R Sami-Subbu, PK Ranjekar and VS Gupta; Amino acid repeat patterns in protein sequences: their diversity and structural-functional implications. *Protein Science*, Vol 9, Issue 6 1203–1209.
- [5] Lugang Li, Renchao Jin, Poh-Lin Kok and Honghui Wan; Pseudo-periodic partitions of biological sequences. *Bioinformatics*, Vol 20, No. 3, 2004, 295–306.
- [6] McLachlan, A.D.; Analysis of periodic patterns in amino acid sequences: collagen. *Biopolymers* 16, 1271–1297, 1977.
- [7] Markus Gruber, Johannes Söding and Andrei N. Lupas; REPPER — repeats and their periodicities in fibrous proteins. *Nucleic Acids Research*, 2005, Vol. 33, 239–243.
- [8] Marcotte EM, Pellegrini M, Yeates TO, Eisenberg D.; A census of protein repeats. 1999, *J Mol Biol* 293, 151.

- [9] Pellegrini M, Marcotte EM, Yeates TO; A fast algorithm for genome-wide analysis of proteins with repeated sequences. *Proteins*. 1999 Jun 1;35(4):440–446.
- [10] Radek Szklarczyk and Jaap Heringa; Tracking repeats using significance and transitivity. *Bioinformatics* Vol. 20 Suppl. 1 2004, 311–317.
- [11] Söding J, Remmert M, and Biegert A; HHrep: de novo repeat detection and the origin of TIM barrels. *Nucleic Acids Res.* 34, W137–142.
- [12] Corné de Ridder, Derrick G. Kourie, and Bruce W. Watson; Fire μ Sat An Algorithm to Detect Microsatellites in DNA. *Stringology 2006*: 137–150.
- [13] Fire μ Sat: meeting the challenge of detecting microsatellites in DNA. *Proceedings of SAICSIT 2006*, Pages 247–256.
- [14] Xiaowen Liu and Lusheng Wang; Finding the region of pseudo-periodic tandem repeats in biological sequences. *Algorithms for Molecular Biology 2006*, 1:2 doi:10.1186/1748-7188-1-2.
- [15] Lugang Li, Renchao Jin, Poh-Lin Kok and Honghui Wan; Pseudo-periodic partitions of biological sequences. *Bioinformatics* Vol. 20 No. 3, 2004, 295–306.

Appendix I: Extrapolated running times for all sequences in the Swiss-Prot database

The data are from February 2008. Calculations are based on the running times in Table 1.

Length from	to	Number of sequences of this length	Extrapolated running time (in seconds)	Total running time for all sequences in this length category (in seconds)
1	50	6096	0.0006875	4.2
51	100	26411	0.002750	72.6
101	150	37499	0.006188	232.0
151	200	36275	0.01100	399.0
201	250	36388	0.01719	625.4
251	300	31932	0.02475	790.3
301	350	30992	0.03369	1044.0
351	400	27158	0.04400	1195.0
401	450	22497	0.05569	1252.8
451	500	18570	0.06875	1276.7
501	550	13412	0.08319	1115.7
551	600	9701	0.09900	960.4
601	650	8384	0.1162	974.1
651	700	5631	0.1348	758.8
701	750	4467	0.1547	691.0
751	800	3516	0.1760	618.8
801	850	2978	0.1987	591.7
851	900	3143	0.2228	700.1
901	950	2473	0.2482	613.8
951	1000	1879	0.2750	516.7
1001	1100	2623	0.3328	872.8
1101	1200	1779	0.3960	704.5
1201	1300	1433	0.4648	666.0
1301	1400	1310	0.5390	706.1
1401	1500	1032	0.6188	638.6
1501	1600	516	0.7040	363.3

Length from	to	Number of sequences of this length	Extrapolated running time (in seconds)	Total running time for all sequences in this length category (in seconds)
1601	1700	389	0.7948	309.2
1701	1800	333	0.8910	296.7
1801	1900	328	0.9928	325.6
1901	2000	256	1.100	281.6
2001	2100	158	1.213	191.6
2101	2200	229	1.331	304.8
2201	2300	208	1.455	302.6
2301	2400	142	1.584	224.9
2401	2500	102	1.719	175.3
2501	3000	261	2.475	646.0
3001	3500	235	3.369	791.7
3501	4000	88	4.400	387.2
4001	4500	42	5.569	233.9
4501	5000	49	6.875	336.9
5001	6000	47	9.9	465.3
6001	7000	20	13.5	269.5
7001	8000	22	17.6	387.2
8081	8081	1	18.0	18.0
8545	8545	1	20.1	20.1
8797	8797	1	21.3	21.3
8891	8891	1	21.7	21.7
9159	9159	1	23.1	23.1
13100	13100	1	47.2	47.2
18074	18074	1	89.8	89.8
22152	22152	1	134.9	134.9
34350	34350	1	324.5	324.5
35213	35213	1	341.0	341.0

Appendix II: Standard amino acid abbreviations

Amino acid	Standard abbreviation
Alanine	A
Arginine	R
Asparagine	N
Aspartic acid	D
Cysteine	C
Glutamic acid	E
Glutamine	Q
Glycine	G
Histidine	H
Isoleucine	I
Leucine	L
Lysine	K
Methionine	M
Phenylalanine	F
Proline	P
Serine	S
Threonine	T
Tryptophan	W
Tyrosine	Y
Valine	V
Asparagine or aspartic acid	B
Glutamine or glutamic acid	Z
Leucine or Isoleucine	J
Unspecified or unknown amino acid	X