Exact Multiple Sequence Alignment using Forward Dynamic Programming

- a thesis in Bioinformatics

Jesper Møjbæk, 20041074

Thesis supervisor: Christian N. S. Pedersen

Bioinformatics Research Center
Aarhus University
February 2010
Abstract

Multiple sequence alignment algorithms are widely used for comparing biological sequences. When an exact comparison between these sequences is needed, naive solutions are no longer sufficient, if the sequences grow in size and numbers due to time and space use. In this thesis I describe and implement a solution, based on forward dynamic programming which will return an optimal score, most often using less resources than the naive solutions. I show that the implementation outperforms a naive implementation and pushes the limits for exact multiple sequence alignment.

Resumé

Multiple sekvens alignment algoritmer bliver brugt i vid udstrækning til at sammenligne biologiske sekvenser. Når der er brug for en eksakt sammenligning af disse sekvenser, er naive løsninger ikke længere brugbare, da de kræver for meget plads og tid når sekvenserne vokser i længde og antal. I dette speciale beskriver og implementerer jeg en løsning baseret på forward dynamisk programmering der returnerer en optimal score, oftest ved et mindre resource forbrug end naive løsninger. Jeg viser at implementationen udkonkurerer en naiv løsning og skubber grænserne for multiple sekvens alignment
Preface

I began initial research in the spring of 2009, after having participated in several bioinformatics courses, it became evident for me that it was in the field of bioinformatics I would do my thesis. The interdisciplinary connection between biology and computer science is intriguing and I find it interesting when two different worlds can find common grounds where elements from both worlds are used.

A few people contributed to this thesis in some way or another. I would like to give special thanks to the following persons. My supervisor Christian N. S. Pedersen who has given me a lot of pointers and our discussions have helped me a lot during both the programming phase and the writing process. Jonathan Bech Bunde-Pedersen has given me much helpful input on my thesis and last Stefan Mckinnon Edwards for a thorough review.
Contents

Abstract ........................................... i
  Abstract ......................................... i
  Resumé .......................................... i

Preface .......................................... iii

1 Introduction .................................. 1

2 Biological Background ....................... 3
  2.1 Biological Prerequisites .................. 3
    2.1.1 DNA ................................... 3
    2.1.2 RNA ................................... 4
    2.1.3 Proteins ................................. 4
  2.2 Sequence Alignment and Biology ........... 5
    2.2.1 Determining Function From Structure .. 5
    2.2.2 Evolutionary Map ....................... 5
    2.2.3 Aligning RNA or Proteins? ............. 6

3 Sequence Alignment ........................... 9
  3.1 Pairwise Sequence Alignments .......... 9
    3.1.1 Scoring Sequence Alignments ........ 9
  3.2 Pairwise Sequence Alignment Methods ... 10
    3.2.1 Dot Matrix Analysis .................... 10
    3.2.2 Word Method ............................ 11
    3.2.3 Dynamic Programming .................. 11
    3.2.4 Comparison ............................. 12
  3.3 Multiple Sequence Alignment methods ... 13
    3.3.1 Dynamic Programming .................. 13
    3.3.2 Progressive Methods ................... 13
    3.3.3 Iterative Methods ...................... 14
    3.3.4 Probabilistic Methods .................. 14
    3.3.5 Comparison ............................. 14

4 Algorithms .................................. 17
  4.1 Dynamic programming algorithms .......... 17
    4.1.1 Time use ................................ 17
  4.2 The MSA Algorithm ......................... 17
Chapter 1

Introduction

Multiple sequence alignment plays a major role in different areas of bioinformatics. Areas such as function and structure prediction and the evolution of different genes make great use of alignment methods. The idea behind multiple sequence alignment is to compare three or more sequences and come up with some alignment which would get the highest score under the given conditions (i.e. the scoring scheme used). Whether they are alike, can be inferred from the scores and alignments of the sequences. This information can in turn be used to predict properties of some given biological sequence.

Numerous different methods for aligning multiple sequences have been devised, some of which are reviewed by Duret et al.[1] and Notredame[2]. All these different methods can be divided in two; approximate and exact methods. Multiple sequence alignment methods can be divided much more finely grained into various other groups. Whether they are exact or approximate merely states something about the result and not the method used. As the name implies, approximate methods can not guarantee an optimal solution to the problem at hand. Exact solutions will find an optimal solution, however, but often at the cost of computer power.

I want to examine how much the algorithm for aligning multiple sequences presented by Gupta et al.[3] can reduce the search space for a given problem. The algorithm tries to reduce the search space by using some predefined threshold value to cut off some of the entries that need to be visited in a dynamic programming table. I want to find out how big the reduction is and if it is at all a feasible approach for doing exact multiple sequence alignment in practice. In doing so, I will describe several approaches to multiple sequence alignment. I will also implement and examine the method devised by Gupta et al., in three different versions. Furthermore I want to look at different options for coming up with a good threshold value. My main focus will not be on the time and space use of the algorithm. However, this could be a limiting factor to my testing, so it will need to be given some attention.

In summary the questions I want to answer are the following

- How much is the search space reduced by using Gupta’s algorithm for aligning multiple sequences?
- Is the method feasible in practice?
- How much influence does the threshold value in Gupta’s algorithm have on the reduction of the search space
In this thesis I will show that the search space is reduced considerably, however only by a constant factor. I will show that the method indeed is very feasible in practice, and that this depends much on the threshold function.

I will start off by roughly explaining some biological background knowledge and explain why multiple sequence alignment is important in chapter 2. In chapter 3 different alignment methods and related material will be explained as will the pros and cons of the various algorithms. In Chapter 4 I will go into detail about the algorithms I implement and in chapter 5 the implementations of them. Chapter 6 will include experimental results and a discussion of them while I will conclude on my findings in Chapter 7.

The source code for my implementation can be found on http://www.daimi.au.dk/~jesperhm/. A JAR of the program can also be downloaded from this page. More information on running the program can be found in the appendix.
Chapter 2

Biological Background

There is a link between sequence alignment and biology, in that the sequences aligned in bioinformatics\(^1\) are either DNA (RNA in some cases) or protein sequences. This means the sequences often have some functionality in one or more aspects of life. A multiple sequence alignment can tell a great deal about the given sequences and their properties - and this is without actually doing any in vitro or in vivo tests on the actual biological sequences\(^2\).

In this chapter, I will start by going through some basic biology followed by explaining what the alignments can be used for. This should supply the necessary background knowledge to understand why we want to align sequences.

2.1 Biological Prerequisites

Biology is concerned with the somewhat broad area called life. All life is based on DNA (or RNA in some very basic cases). Proteins are fabricated on the basis of a DNA template and the proteins in turn are what carry out physiological processes within our bodies. Basically DNA is transcribed into RNA which is translated into protein. In this section I will describe this process and the different parts of it superficially. The information stated in this section is from Stryer\([4]\).

2.1.1 DNA

DNA is short for DeoxyriboNucleic Acid. All life on earth contains DNA and many believe that all life originate from the same DNA (or at least RNA, Walter Gilbert\([5]\)). That is, all DNA have a common ancestor, meaning that at some point back in time, it is believed that there was a single (very basic) life form, from which all life known today has evolved.

DNA is made of ribose molecules with one of the four nucleic acids; Guanine (G), Cytosine (C), Adenine (A) or Thymine (T) attached. These ribose molecules with an acid attached are linked together by phosphate groups. A sequence of several DNA molecules is called a strand. One strand is paired with another strand, based on their chemical properties, so that Guanine pairs with Cytosine and Adenine pairs with Thymine. A set of paired strands is called double stranded DNA and exists in a helical structure as shown in figure 2.1. DNA

---

\(^1\)Not only biological sequences can be aligned, even though this probably is the most common use of sequence alignment. One could imagine aligning several sequences of words, in order to compare them...

\(^2\)tests performed either inside the host or outside - i.e. in a petri dish
CHAPTER 2. BIOLOGICAL BACKGROUND

Figure 2.1: A DNA helix

contain the genes that determine our physical properties. These genes can be transcribed to RNA which in turn can be translated to protein (with exceptions - not all DNA codes for proteins, however in this thesis I will not dwell much on this).

2.1.2 RNA

Ribonucleic Acid (RNA) is very similar to DNA, except for the Thymine which is replaced by a Uracil (U). Also where the DNA often exists as double stranded DNA, RNA is usually single stranded. Furthermore the backbone ribose is a little different from that of the DNA hence the different name\(^3\). RNA is built from DNA in a process called transcription where a DNA strand is transcribed into an RNA sequence. Most RNA is translated into proteins which is described in the next section. RNA does have other uses and a specific structure, but I will not go further into this, as it is not relevant for this thesis.

2.1.3 Proteins

The Translation process translates RNA molecules into proteins. This is done by reading three RNA molecules at the time, called codons. Each codon codes for a amino acid. The observant reader will have realized that this would give a total of 64 amino acids \((4^3 = 64)\), but the codons are degenerative, meaning that some codons code for the same amino acids and some of them indicate when the ribosome\(^4\) should start and stop translating. Proteins consist of amino acids linked together by so called peptide bonds. There are 20 naturally occurring amino acids (a few more exist, but these are usually not included) which differ on properties such as size, acidity and hydrophobicity. Linked together these make up proteins which are fundamental building blocks of the body.

When working with proteins there are four different structures. The primary structure is the sequence of amino acids. The secondary structure is the internal structure of the amino acids. There are three different types of internal structures; beta sheets, alpha helices and loops. The beta sheet occurs when the amino acids hydrogen bond with other amino acids to develop a sheet like structure. Alpha helices also occur because of hydrogen bonds between the amino acids, but this pairing is slightly different and results in a helical structure. The last

\(^3\)deoxyribo- and ribo- which indicate that DNA has one less oxygen attached to the ribose unit than RNA.

\(^4\)This is a complex responsible for translating the RNA into protein.
internal structure, the loop, happens when unpaired amino acids not in beta sheet nor alpha helical structures make a chain forming a kind of loop. The tertiary structure of proteins is the spacial orientation of the different secondary structures. The tertiary structure is often related to as a subunit. Many proteins consist of two or more subunits and the spacial orientation of these is called the quartenary structure.

2.2 Sequence Alignment and Biology

Sometimes it might be difficult to see the point of sequence alignment. The goal might seem to be to just "align some sequences". But sequence alignment is an important part of biology for a just reason. In this section I will shed some light onto some of the reasons sequence alignment is of much use in biology. The various explanations are from Mount[9] and Nei and Kumar[10]. In recent years the sequencing methods have improved a lot and have become more economically feasible. Therefore the amount of data collected by molecular biologists outnumbers the resources available for analyzing the data. It is not feasible to analyze the data by looking at it, as it were when the first sequences were obtained (to some extent). For this reason computational alignment methods have been devised, so the task of aligning sequences would be less cumbersome. The results of a given alignment is used for determining how related the aligned sequences are. Usually some score (see chapter 3) is assigned to the alignment which is a measure of how similar the sequences are. Furthermore the given alignment can be used to find biological hot spots which are positions in the genes where mutations are frequent and/or might be significant with regard to different illnesses. This information can then be used to build phylogenies of the sequences which help biologists understand the evolution of the given sequences and ultimately in the bigger picture of life itself. Another aspect of sequence alignment, is that it may be possible to determine the function of a given sequence if it aligns well with another sequence with known function.

2.2.1 Determining Function From Structure

When information about the function of a given sequence is available, it is possible to determine the function of other sequences too. If other sequences align well with the sequence of known function, it is highly likely that it has the same function. In essence, this means that similar sequences usually have the same functions. That is they are most likely homologous and probably share a common ancestor. This is of great use when new sequences are discovered. Instead of running expensive in vitro tests to find the function of the given structure, it is much cheaper to compare them to other sequences with known properties. It will not eliminate the laboratory work of testing, but it will make it easier if some idea of what the properties of the sequences are exists.

2.2.2 Evolutionary Map

When the pairwise score between several sequences has been determined, it is possible to build a phylogeny which can tell us how species are related. This might seem like useless knowledge, but phylogenies actually help us fight diseases and illnesses. If we can determine

---

5A phylogeny is a depiction of the relatedness of a given set of sequences or species or some other set of entities which are related in some way. This is usually represented as some kind of tree.
that two sequences are related, and one of them is the cause of some disorder then it is very likely that the other sequence is also linked to the same disorder and hence needs the same treatment.

A phylogenetic tree is a tree indicating how sequences (or entities) are related. That is it shows their hereditary history. It shows when sequences diverge which is the same as when some mutation or change occurs that label them as two different species. A mutation is a single point in a sequence - be it DNA or protein - where the given nucleotide or amino acid has been changed into another. Typically the length of the branches between two sequences is proportional to the difference between them. An example of a phylogenetic tree is shown in Figure 2.2. It shows that B and C are equally related to A. Those three are closer to D than E. That is A, B and C has a more recent common ancestor than i.e. B and D.

An example of a phylogenetic tree of the life forms known today is shown in figure 2.3. Each of the kingdoms bacteria, archaea and eucaryotes are divided further into other subgroups and the subgroup called animals under eucaryotes is where humans and other animals belong. Animals are grouped close together with plants and fungi, meaning they are animals closest relatives with regard to the similarity of the DNA sequences. A phylogeny of all animals can also be made which would group i.e. humans closer to chimpanzees than to dogs. And zooming further in, a phylogeny over all humans would result in a map where different populations would be more related than others. i.e. People from China might be more related to people from Japan, than people from Denmark. Similar trees could be constructed for given viruses, to see how they have evolved over time.

### 2.2.3 Aligning RNA or Proteins?

Due to the degenerative property of codons, an alignment between two RNA sequences and the protein resulting from the translation might not give the same result. Because of the degenerative property of the codons, two different RNA sequences could actually code for the same amino acid sequence. Below, two not identical RNA sequences code for the same sequence of amino acids. Four positions differ in the RNA strands below.

![Figure 2.2: Example of phylogenetic tree made of the species A, B, C, D and E. The figure shows that B and C are closer related than A and B or A and C. A and B (and C) are in turn closer to D than to E.](image)
On the other hand, two RNA sequences, sharing the same number of similarities could also code for two different amino acid sequences as shown below. The DNA strands below also have four differences.

RNA:
AUGUUAGUUGCUCGA
AUGCUCGGGCAGAGA
Protein:
MLVAR
MLVAR

Such things need to be addressed before determining what to align. On a side note, it should be pointed out that whether DNA or RNA is aligned does not matter, since the only difference is that ‘T’ is swapped with ‘U’.

So the basis of all life dwells on these different types of sequences. It is therefore easy to see why one would want to compare and measure them. When comparing all these different sequences, different methods can be applied. In the next chapter, I will explain some different methods for comparing these sequences.
Chapter 3
Sequence Alignment

When aligning sequences they can be aligned locally or globally. The difference is that in a global alignment, entire sequences are aligned against each other. Local alignment only finds the best internal alignments which can be useful when finding similar active sites in two different protein sequences. The global alignments of the sequences may show that the sequences are not alike whereas the local alignment may indicate that the active sites\(^1\) are similar, meaning that they could have the same function. Implementing a global alignment method can often with few modifications be converted to a local alignment method. Therefore I will concentrate on discussing global alignment. In this chapter I will start off by explaining what pairwise sequence alignment is. Then I will describe some of the methods used when aligning sequences, both pairwise and multiple alignments. I will finish each of the two sections off by making a comparison of the methods stating the pros and cons of the different approaches.

3.1 Pairwise Sequence Alignments

Pairwise sequence alignment is, as the name implies, concerned with the alignment of two sequences. When an alignment is found, each index of the two sequences are scored against each other (see section 3.1.1). When scoring sequences in multiple sequence alignment, all the sequences in the alignment are scored pairwise with every other sequence. That is for three sequences there are three comparisons, for four sequences there are 6 and for five there are ten and so on. So basically the number of pairwise alignments is the number of possible ways one can pick two pairs from \(N\) \(\binom{N}{2}\) which is \(O(N^2)\), so the amount of pairwise alignments grows proportional to the square of the number of sequences. When giving the alignments a score some scoring function or distance matrix is used, as explained in the next section. In practice whole alignments are seldom created first and then scored, but instead they are scored during the alignment.

3.1.1 Scoring Sequence Alignments

When determining how alike two or more sequences are some kind of scoring model needs to be applied. This usually comes as some kind of distance matrix. There are very simple

\(^{1}\)An active site is some area in a protein where the function of the protein is utilized. It is usually the place of most interest in the protein.
distance matrices like the identity matrix (all zeros in the diagonal) shown in table 3.1(a) for
DNA where a match scores zero and everything else one (It could be one for a match and
zero for anything else, depending whether it is a maximizing or minimizing algorithm). This
distance matrix does not, however model the real world very well. For example, it assumes
that aligning an adenine with a thymine is just as good as aligning it with a guanine when
in fact it is much more likely that the adenine pairs with a guanine (that is it costs less
energy). A substitution matrix that takes such things into account could look like the one in
table 3.1(b).

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>C</th>
<th>G</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>G</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>T</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

There are other distance matrices and of course some for proteins too. I will not go further
into this area in this thesis though. In this thesis, I use the distance matrix shown in 3.1(b).

3.2 Pairwise Sequence Alignment Methods

Pairwise sequence alignment is, as explained in the previous section, concerned with the
alignment of two sequences. I will go through three different methods described by Mount[9].

3.2.1 Dot Matrix Analysis

The basics of the dot matrix method, is to fill out a two dimensional graph where each axis
is a sequence, starting from the first residue ending with the last. Then for each coordinate,
(x,y) where x and y are residues in each sequence, a dot is placed if they align. The graph
can be filled using dynamic programming (explained in section 3.2.3). When done the graph
will look something like the dot matrix in Figure 3.1 which is made using the two (more or
less random) sequences below using the Dotmatcher on the EMBOSS website[12].

AAAAATATATAGAGAACACCCCCAGAGAGCGGCAAAACGCGAGCGAGCTACTATCAGACCGACTCGACGCTCTG
AAGACAGATCTCGAGATCGGAGAATTGGTCACAAAAACGCGCTCTCTTATATAATATACGGCCGTTGTGAGAGAAAAATCTTCTC

The diagonal lines in the graph indicate subsequences that align with each other. The
longer the diagonal line, the longer the subsequences are. To avoid having single dots which
would be the case if all dots were shown, a sliding window is usually used. The number of
aligned residues in this window has to be above some threshold before a dot (or a line when
there are several aligned residues) is drawn.

The dot matrix method takes time $O(n \times m)$ where n and m are the lengths of the
sequences which seldom is a problem, unless the sequences are very long.
3.2. PAIRWISE SEQUENCE ALIGNMENT METHODS

3.2.2 Word Method

This method is a heuristic which takes tuples of length k where k is the number of residues to match. For each tuple it searches the other sequence for the occurrences of this tuple. When all occurrences of the same tuple in each sequence have been found, the residues in between each matching tuple are aligned, usually using dynamic programming. A modified version of this is used by BLAST[11], in order to quickly search through many sequences in a database. BLAST stands for Basic Local Alignment Search Tool, and is an algorithm for comparing sequences. The speed does come at a cost though. It is not as accurate as for example dynamic programming.

3.2.3 Dynamic Programming

I will describe dynamic programming a little more in depth than the other methods, due to the fact that the method by Gupta et al. is based on forward dynamic programming. Therefore it is very central to this thesis and deserves a little more time. Dynamic programming is originally a method used in the area of optimization in mathematics developed by Richard Bellman in the 1940’s[13]. It solved problems where several “good” decisions needed to be made. In computer science dynamic programming is a programming method based on dividing a problem into subproblems. Each of the subproblems are divided further into subproblems and so on until some base case is reached. Thus a given problem can be defined by some sub problems. An example of dynamic programming could be presented as a table of size \( n \times m \) (for two dimensions) where the entry in \([m],[n]\) is the solution to the problem. In order to get this solution the entries \([m-1],[n]\), \([m-1],[n-1]\) and \([m],[n-1]\) need to be calculated and so fourth, until the base case \([0,0]\) is reached. Instead of calculating this recursively the table can just
be filled from top to bottom which is called dynamic programming. Dynamic because each entry depends on the previous entries. A recursion for a dynamic programming table for two sequences, S1 and S2 could look like so

\[
D(i, j) = \begin{cases} 
0 & \text{if } i = 0 \text{ and } j = 0 \\
\min(D(i-1, j) + C, D(i-1, j-1) + S(S1_i, S2_j), D(i, j-1) + C) & \text{else}
\end{cases}
\]

In the above, D(x,y) is an entry in the dynamic programming table, S(S1_i, S2_j) is the score for aligning two bases (or proteins) and C is the cost for making a gap. A gap in sequence alignment is denoted with a '-' character. Gaps are used in alignments when the sequences do not align well with each other. Then one sequence can be pushed one index, by inserting a gap in it. This way the sequences might align better. The biological interpretation of a gap is that either a deletion or an insertion has happened during the evolution of one of the sequences. There are some boundary cases which have to be taken into account when programming the recursion. It is assumed that the best score is the minimum. The maximum could easily be implemented instead.

In Figure 3.2 the method of dynamic programming is visualized. In A the base case is processed. Then the top row is processed in B and C (the top row only needs the previous value to the left to be processed). In D, the next row can be filled. This is done until the whole table is processed, shown in E. The result can then be fetched from the entry in the bottom right hand corner. The dynamic programming method takes time \(O(n \times m)\), like the dot matrix method where \(n\) and \(m\) are the lengths of the sequences. In this thesis I use both forward and backward dynamic programming. Backward dynamic programming corresponds to the explanation of dynamic programming above. In forward dynamic programming a given entry in the table sends its value to all the subsequent entries. When an entry has received all values from previous entries, it can find its own value and propagate it to succeeding entries. So backward or forward merely states whether the entry values are passed or fetched.

3.2.4 Comparison

Each of the three methods perform well in different areas, none of them is perfect for every use. The dot matrix method will give the user a nice overview of the positions where the sequences align well. If there is some active site of 10-15 amino acids which both of the aligned sequences have in common for example, this will show up as a long series of dots in the plot. However when one does not want to go through the alignments manually this method is not the way to go. Furthermore, because the dot matrix method dwells on manual verification, it can be inaccurate due to fact that it can be difficult to see the differences (or decide which is the better) between two alignments by looking at the plots.

The word approach is very quick and effective when needing to align a sequence to many other sequences in a database for example. The result will not be 100% correct, but it will give a nice overview of the likeliness between the sequence and other sequences. Then some other exact alignment method can be used to find the exact scores if needed.

The last method, dynamic programming, is as time consuming as the dot matrix method (as expected as they are both two-dimensional tables that need to be filled), but gives a score, so that it is easier to compare several different alignments. Unless using local alignment\(^2\), finding good local alignments is not possible. This method can quite easily be extrapolated to \(k\) sequences which makes it feasible for multiple sequence alignment too (see section 3.3.1).\(^2\)

\(^2\)Can easily be implemented from a global alignment method.
3.3 Multiple Sequence Alignment methods

In this section, I will go through the multiple alignment methods explained by Mount[9] and at the end of the section try to elaborate on the pros and cons of the different methods. Because multiple sequence alignment, in some cases demands much computational power and time, there are a few more approximative methods than for the pairwise alignment methods.

3.3.1 Dynamic Programming

As mentioned earlier, the dynamic programming method can be used for an arbitrarily large number of dimensions. The computer power needed to calculate the dynamic programming table is the only thing setting the boundary. The number of indices that need to be computed grow exponentially proportional to $n^k$ where $n$ is the sequence length (assuming here that all the sequences have the same length) and $k$ the number of sequences. Thus both the time it takes to compute the number of entries in the table and the memory use becomes a problem quite quickly.

3.3.2 Progressive Methods

Progressive methods build an alignment by adding the sequences one at the time according to their pairwise scores, so usually the sequences that align best are added to the alignment first. The multiple star method (see a thorough explanation in chapter 4) is a good example
of this. Basically the multiple star algorithm starts with the sequence which aligns best with all the other sequences and then aligns the rest with that and insert gaps when necessary. Other methods exist as well, but common for them all, is that they start aligning the best first and end with the worst or the rest. These methods are usually very quick, so a lot of sequences can be processed. The scores however, are not always optimal.

3.3.3 Iterative Methods
Iterative methods often builds on top of a progressive method, so that for each sequence added to the alignment with a progressive method, the alignment is refined, by looking at subgroups of the different sequences. This way some potentially bad initial alignments can be countered. This is especially true for distant sequences where the progressive methods might halt a little. The cost of this comes of course in time use, as it takes longer.

3.3.4 Probabilistic Methods
Methods like expectation maximization (EM) or hidden Markov models (HMM) try to find the most likely alignment based on some assumptions. EMs are performed in two steps which are expectation and maximization. Before the first step some initial guess is made. In the first step, expectation, the likelihood of the alignment is calculated. Then in step two, the suggested alignment is adjusted maximizing the likelihood. The new adjusted alignment is then used in step one again. This process is continued until the difference between each alignment and the maximized version are below some predefined threshold. This will be the alignment returned.

HMMs are based on knowledge of other alignments. A model is trained on some known dataset\textsuperscript{3} before it is used on a set of sequences. The HMM model then finds the most likely alignment based on the coefficients.

3.3.5 Comparison
The methods presented each have their area of expertise.

The dynamic programming method gives an exact solution to a problem, but the time it takes grows exponentially with the number of sequences (time $O(n^k)$, for $k$ sequences of length $n$). For this reason dynamic programming is only possible for a very limited number of sequences. This method could be appropriate to use after a more coarsely grained method to filter out a few sequences, for which an exact alignment is needed.

Progressive methods thrive on the fact that they are very fast. An example is the multiple star alignment which in a matter of seconds can align a hundred sequences of length a hundred. An exact solution to this problem is not possible to calculate in anything near a reasonable amount of time\textsuperscript{4}. Therefore the approximate algorithms exist for a reason - it is not possible to align multiple sequences exact when the numbers get too big. The downside is of course obvious; they do not return an exact solution and less similar sequences will get even worse scores, due to the fact that initial alignments is based on the best scoring sequences.

\textsuperscript{3}The training is used to obtain coefficients for the probabilities of given alignments. These could be estimated without using a dataset to train on however.

\textsuperscript{4}A hundred sequences of length hundred will need $100^{100}$ table entries to be filled which is a great too many.
Iterative Solutions lie in between the afore mentioned. They do not have the speed of the progressive methods, nor do they return an exact solution.

Probabilistic methods like HMM are very useful when searching for known patterns or if the sequences being analyzed are much like the training set. So the more that is known about the sequences the better these methods get. HMM are quite time consuming, due to the fact that they go through all alignment possibilities, but the result could be more realistic than using some scoring matrix (see section 3.1.1). EM on the other hand does not need any information about the sequences beforehand, but if they have a common motif, it is likely that this could be found using this method.

Table 3.2 summarizes the mentioned methods. The precision of probabilistic methods can not be measured the same way the other methods can, due to the fact that the other methods often use a scoring function and the probabilistic ones use a training set or some threshold in case of the EM.

<table>
<thead>
<tr>
<th>Method</th>
<th>Precision</th>
<th>Speed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dynamic Programming</td>
<td>Exact</td>
<td>Slow</td>
</tr>
<tr>
<td>Progressive</td>
<td>Approximative</td>
<td>Very fast</td>
</tr>
<tr>
<td>Iterative</td>
<td>Approximative</td>
<td>Fast</td>
</tr>
<tr>
<td>Probabilistic</td>
<td>*</td>
<td>Slow</td>
</tr>
</tbody>
</table>

At the end of the day, it is impossible to name a given method as the best because they all have different areas where they are useful.

I have now described different sequence alignment algorithms which has shown that they each have their forces and caveats. In the next chapter, I will go into detail about the algorithms I implement.
Chapter 4

Algorithms

In this chapter I will explain the outline of the algorithms I implement. I will provide some pseudo code if needed. Furthermore I will make some comments on the time use of the algorithms. Even though it is not a major concern of mine in this thesis, it is still quite important that it is actually possible to run the algorithms in a humane time frame.

4.1 Dynamic programming algorithms

To compare my algorithms (The MSA and Multiple Star algorithms which are described in this chapter) with an exact solution, I have implemented naive solutions for aligning two, three and four sequences. All of them have been implemented using backward dynamic programming, starting at the end corner. The reason I did not implement a version for k sequences, is that I did not want to complicate the code which would increase the risk of introducing errors. The reason for filling the tables starting at the end corner is that the MSA algorithms needs the property from the two sequence algorithm that in any given entry of the table, it is possible to read the score of the alignment from that point to the end corner of the table. The algorithms for three and four sequences are built on top of the two sequence version, why they are also started at the end corner.

4.1.1 Time use

As mentioned in chapter 3, filling out a dynamic programming table takes time $O(n^k)$, k being the number of sequences and n the sequence lengths assuming they all have the same length. For the algorithms I implement that means $O(n^2)$, $O(n^3)$ and $O(n^4)$.

4.2 The MSA Algorithm

The MSA algorithm is presented in the article by Gupta et al.[3]. The cited paper presents an optimized version of a the multiple sequence alignment algorithm by Carrillo and Lipman[7]. The algorithm is similar to Dijkstra’s shortest path algorithm from 1959[8]. In the algorithm they try to to cut of some of the search space in the dynamic programming table away by means of a threshold value, thereby reducing the number of entries that need computing.
4.2.1 Steps of the Algorithm

This section summarizes the steps of the algorithm.

1. **Make threshold value**
   A threshold value needs to be defined. This is a number which is definitely larger than or equal to the optimal score of the alignment.

2. **Make \(O(K^2)\) alignments (assuming \(K\) sequences)**
   For each pair of sequences \((i,j)\) I make an alignment using dynamic programming, filling the dynamic programming tables starting at the end corner. In each entry, I note the score from there to the end corner. These tables are saved as \(F_{ij}\).

3. **Initialize Priority queue**
   A priority queue of vertices where the front of the queue is the vertex with the smallest weight is initialized. A priority queue is a queue data structure where elements are stored according to some value. The lowest ranked value item is always at the head of the queue. Here the value determining the position in the queue is the distance from the start corner which is the score from the start corner to the position of the given vertex. This is also called the weight of the given vertex.

4. **Initialize Trie**
   A trie is initialized. A trie is a prefix tree where each node contains a single coordinate and a leaf symbolizes a vertex where the path to that leaf goes through internal nodes corresponding to the coordinate of the vertex.

   Each internal node of the trie is a coordinate for a vertex (or more, since several vertices can share some coordinates) and each leaf corresponds to a vertex and contains the weight of the given vertex. Figure 4.1 shows an example of a trie where each internal node corresponds to an entry of a vertex coordinate and the leaf holds the weight (here \(W\)) of a given vertex. As can be seen some of the leaves share internal nodes on their path to their leaf with other leaves. It is apparent that any given internal node, can at most have \(n\) child nodes, \(n\) being the length of the sequences. The depth of the trie is the number of sequences. Therefore the upper bound for the number of internal nodes is in the order of \(O(n^k)\) which also is the upper bound on the number of leaves. So hopefully the trie will not be filled which would make the data structure and algorithm redundant in regard to a naive solution.

   The trie is needed so that the weight of a given vertex can be found, in case it is already in the queue. Every time a vertex is added to the queue, it is added to the trie too.

5. **The iterative process**
   Starting with the vertex in the start corner (corresponding to \((0,0,0)\) for three dimensions), with distance 0. This vertex is inserted into the priorityqueue, with key 0 (which is its distance) and also into the trie. In each iteration the vertex \(v\) in the queue with the lowest key is extracted. If the weight of the vertex plus the sum of all the values of \(F_{ij}\), for all pairs \((i,j)\), at the given point is less than the defined threshold value, the algorithm proceeds to the next step with the extracted vertex. The equation below
4.2. THE MSA ALGORITHM

explains this too where the best possible score needs to be below the threshold value. \( v \) is the extracted vertex

\[
\text{best possible score} = v.\text{Distance}() + \sum_{i<j} F_{ij}(v.\text{Point}(i,j)) \quad (4.2.1)
\]

If the vertex is suitable to proceed to the next step, all the neighbors of \( v \) are added to the trie and the queue, if they are not already there. Then their distances are updated if necessary which it is if they have been encountered before. That is if the cost of the edge between \( v \) and the given neighbor plus the distance of \( v \) is less than the distance already noted in the neighbor (on initialization of a given neighbor - a vertex - the distance is set to \( \infty \)). When done with this step the iteration proceeds to extract the vertex with the lowest distance from \( Q \) and do the same for all its neighbors. Termination is achieved either when the extracted vertex \( v \) is the same as the vertex corresponding to the end corner or when \( Q \) is empty. In the latter case no alignment with score less than the threshold value has been found.

The pseudo code below describes what happens in each iteration of the algorithm.

```plaintext
Init queue \( Q \)
Init trie \( T \)
Vertex \( s = \text{Vertex}((0,\ldots,0), 0) \)
Vertex \( t = \text{Vertex}((N_1, N_2, \ldots, N_K), \text{"infinity"}) \)
Insert \( s \) in \( Q \)
insert \( s \) in \( T \)
while \( Q \) is not empty) {
    \( v = Q.\text{extract}() \)
    if (\( v == t \)) {
        "Done", return
    }
    if (\( v.\text{Distance} + \sum_{i<j} \text{F}_{ij}(\text{point}) \leq U \)) {
        for ("All neighbours, \( n \), of \( v \)) {
            \( w = T.\text{find}(n) \)
            if (\( w == \text{null} \)) {
                \( w = \text{new Vertex}(n, \text{"infty"}) \)
                \( Q.\text{insert}(w) \)
                \( T.\text{insert}(w) \)
            }
            if (\( v.\text{Distance} + \text{Cost}(\text{Edge}(v,n)) < w.\text{Distance} \)) {
                \( w.\text{Distance} = v.\text{Distance} + \text{Cost}(\text{Edge}(v,n)) \)
                \( Q.\text{decrease}(w, w.\text{Distance}) \)
            }
        }
    }
    "No alignment found", return
```

4.2.2 The MSA Algorithm Visualized

In order to better explain the MSA algorithm, the basics of the algorithm are visualized in figure 4.2 for two sequences. The coloring scheme is as follows: Red is a vertex which has been polled from the priorityqueue and processed. Orange means the vertex is in the priorityqueue. Grey is a vertex which is unreachable because it can never be part of an optimal alignment.
20

CHAPTER 4. ALGORITHMS

Figure 4.1: Each node corresponds to a single entry of the coordinate for a given vertex. A leaf holds the weight of a given vertex and corresponds to a vertex where the path to that leaf from the root is its coordinate.

In A the base case is processed and in B its neighbors are added to the priorityqueue. In C the next vertex is polled from the queue. The same happens in D and E. In F a whole row is colored gray, because it cannot be reached. The weight found in the first gray vertex from the left is above the threshold value, so that the rest cannot be reached. In G a few steps have been taken and more vertices have been added to the priorityqueue or processed. The alignment is done when the vertex in the bottom right hand corner has been processed. As can be seen in H a lot of the table has been colored gray meaning it is unreachable and does not need to be processed. This is exactly where the MSA algorithm becomes powerful. At this point, the priorityqueue is not (necessarily) empty. Because the priorityqueue always returns the vertex with the lowest weight, it is guaranteed that the score of that vertex cannot be any lower (if so, some other vertex must be before that one in the queue). So when the polled vertex is the bottom right hand corner vertex we are done, since it does not have any neighbors.

4.2.3 Implemented Algorithms

I implemented three different algorithms, all based on Gupta’s MSA algorithm. The actual implementations are described in chapter 5. This section describes the different approaches. The differences are the data structures used to keep track of the vertices and the ones used for the queue.

MSA

The MSA implementation uses the Java PriorityQueue for the queue data structure. The queue polls the vertices according to their weight with the lowest weight at the head of the queue. Instead of the trie the Java HashMap data structure is used.

MSACoor

MSACoor is short for MSA Coordinate which indicates that the algorithm does not use a queue, but uses a topological ordering of the coordinates, so that when a given vertex is
4.2. THE MSA ALGORITHM

The MSA algorithm steps visualized. Red is a vertex that has been polled from the priority queue and is finished being processed. Orange is a vertex which is in the queue. Grey is an unreachable vertex. The algorithm progresses as shown in figures A to H. As some vertices are colored gray many more gets colored gray quickly.

Figure 4.2: The MSA algorithm steps visualized. Red is a vertex that has been polled from the priority queue and is finished being processed. Orange is a vertex which is in the queue. Grey is an unreachable vertex. The algorithm progresses as shown in figures A to H. As some vertices are colored gray many more gets colored gray quickly.

The MSA algorithm is visualized in figure 4.3. If the figure is compared to the visualization of the MSA algorithm in figure 4.2, the difference is a bit easier to see. MSACoor processes each layer of the table until some vertices have been discarded. For two dimensions, the layer is the half square starting at the top most row and ending in the left most column. Each entry in this layer which is one entry wide, is processed before another layer is started, unless any entries have been discarded. The MSA algorithm on the other hand, processes the ones with the lowest weight which gives the different paths shown in the figures. I use a TreeMap to keep track of the vertices and a HashMap instead of the trie specified in the MSA paper, to keep track of all vertices. Due to the way the algorithm is implemented, I check if the vertices could be part of an optimal solution when they are polled and not when they are inserted into the TreeMap. This check probably should have been done before they are added to the TreeMap. Furthermore, this implementation does not need a HashMap in order to keep track of the vertices, since the vertices are fetched from the TreeMap by their coordinate. The reason for these implementation subtleties, are that the MSA implementation was the foundation for the MSACoor implementation.

**MSATrie**

This implementation is an implementation of the exact data structure specified in the paper by Gupta et al.. I use a trie to keep track of visited vertices and the queue is a bucket queue with a number of buckets corresponding to the threshold value. The front of the queue is the first non-empty bucket.
Differences between MSACoor and MSATrie

Since these are the main algorithms (and because MSA and MSATrie are essentially based on the same principles), I will not include MSA in the comparison. MSACoor is based on the topological sorting, as mentioned in previous section. This means that the ordering in which the vertices are polled from the queue is not determined by the weight of the given vertex at all. In MSATrie on the other hand, the vertices are polled on the basis of their weight. So the actual order in which the vertices are processed by the two algorithms, could possibly vary very much. In theory the number of processed vertices should not vary considerably, since MSACoor will eventually discard routes below the threshold, albeit possibly later than MSATrie.

One could imagine some set of (unequal) sequences in which MSATrie would process only the diagonal (aside from a few vertices at the beginning). This set of sequences will also be processed quickly by MSACoor, but the point at which the diagonal does not create any new neighbors will start a little later. This is due to the layer processing property explained earlier.

4.2.4 Threshold Value Functions

I have implemented several different threshold functions. Basically I have used three functions, constant value (based on an approximative score), polynomial function and an exponential function. The latter two do not depend on the input, so how close their function value can get
4.2. **THE MSA ALGORITHM**

to the optimal score will determine how good they are. This is of course bound to have some random aspect which I cannot control. Therefore I have performed some tests, described in chapter 6, measuring the threshold values. I will describe each of the functions in the following.

### 4.2.5 Threshold Functions in the Algorithms

When using the functions which are not a constant value, I increment the functions until it is above the score summed over all the pairwise scores from the start corner, taken from $F$ (which is all the pairwise alignments with a value for the distance to the end corner in each entry). If the function value is below this value, it will also be lower than the optimal score and hence will be incremented. When this is done the algorithm is run with the given value. If no alignment is found, the algorithm will be incremented and the algorithm run with the new value. This is done until a solution is found.

Sometimes pure brute force is better than the clever way which is the reason why I decided to try some incremental threshold functions. I could easily come up with many other functions than the two I have chosen, but the time frame of this thesis does not permit a test of all possible functions therefore I have chosen polynomial and exponential functions. These two using different coefficients will supply a variety of functions that grow at very different rates.

#### Constant Function

This function merely returns a constant which is defined upon initialization. When using an approximative alignment method to determine the threshold function, the value is plugged into the constant threshold function. This method only starts aligning the sequences once, so vertices are not visited multiple times in different alignment sessions. However the value supplied could be so much larger than the actual optimal score that it has to investigate a huge number of vertices.

#### Polynomial Function

A polynomial function which iterates through $a \times x^y$ where $a$ and $y$ are constants supplied when initiating the function and $x$ is incremented until an alignment is found. The advantages of using a function for determining the threshold value, is that it will never be much larger than the optimal score (unless some obscene input values are used). The disadvantage is of course that it might need a few runs of the algorithm before a suitable value is found.

#### Exponential Function

Basically the exponential function is a growing threshold function, much like the polynomial function, except that it grows much faster. It iterates through $a \times y^x$ where $a$ and $y$ are constants supplied and $x$ is incremented until an alignment is found. The function has the same advantages and disadvantages as the polynomial function, it is just a different function that could find good uses where the polynomial function might halt a bit.
4.2.6 Constructing Vertex Neighbors

Each entry in the graph has $2^k - 1$ neighbors where $k$ is the number of sequences. The coordinates of these neighbors can be found by taking the current point and adding either 0 or 1 to each entry. Not all zeros need to be added though, since it would result in the same coordinate of the current point. To create all combinations, all the binary numbers from 1 to $2^k - 1$ can be created. The $i$'th digit in each binary number corresponds to the digit to add to the $i$'th coordinate of the current point.

4.2.7 Time use

In the following I will use the original algorithm data structure for analysing time use. This means I expect `Put` and `Get` operations of vertices to take time $O(\log n)$, which they will using trie structures. The found time boundaries in this section will still hold for the algorithms I implement. The only problem is that they use hashing. This means, that the performance is only expected and not guaranteed.

Worst case

In a worst case scenario all vertices of the graph are visited. This implies that each vertex is added to the priority queue which is $O(\log n^k)$ for each vertex, because the trie holds at most $n^k$ vertices. Furthermore, for each vertex, all its neighbors are generated which corresponds to $O(2^k)$ in each step. For each of the generated neighbors a lookup in the trie is needed which takes time $O(\log n^k)$. There are $O(n^k)$ steps (one for each vertex) so we get the following

$$\text{Worst case time} = O(O(n^k) \times (O(\log n^k) + (O(2^k) \times O(\log n^k)))) = O(2^k(\log n^k)n^k) \quad (4.2.2)$$

So the worst case time is a factor $2^k \log n$ worse than the naive method. However, the overhead is much greater though, so in practice, in worst case, it is far slower than a naive implementation.

Best case scenario

In a best case scenario only vertices on the path will be visited which is $O(n)$ steps. The time spent in each step is $O(2^k \log n)$, and $O(\log n^k)$ for adding the vertex. This gives us the following

$$\text{Best case time} = O(n) \times (O(\log n^k) + (O(2^k) \times O(\log n^k))) = O(n(\log n^k)2^k) \quad (4.2.3)$$

This is still an exponential time use, but since $k$ never is a very large number this can be considered as $O(n \log n)$. It should be noted that the estimation of a threshold value may be the limiting factor in a best case. i.e. the multiple star method runs in $O(k^2 \times n^2)$ time. In practice this is never an issue though, since the time it would take in the best case is very little.
Average runtime

It is very difficult to give an average runtime for the algorithm as it depends highly on the threshold value and the input sequences.

4.3 Multiple Star algorithm

The Multiple Star algorithm presented by Gusfield[14], approximates a multiple alignment of more than two strings. The algorithm works as follows. First all sequences are aligned pairwise with all other sequences. The sequence totaling the lowest score is chosen to be the center sequence. Now all sequences are aligned against this center sequence. The alignments are saved in an array of alignments. After the first alignment is done, whenever a gap is inserted into the center sequence, a gap is also inserted into the other sequences that have already been aligned with the center sequence. The center sequence used for the alignment is always the one in the saved array with (possibly) inserted gaps. Meaning, the sequences will be aligned to different versions of the center sequence as the algorithm proceeds. When done the whole array of sequences with inserted gaps is the alignment. To find the score all positions are scored pairwise with all other sequences.

In his paper, Gusfield argues that the approximated score can never exceed $2\left(\frac{k-1}{k}\right) \times S_O$, $k$ being the number of sequences and $S_O$ being the optimal score. This means the score of this method will never be more than $2 \times S_O$. The time use of this algorithm is very minimal compared to various exact solutions to the multiple string alignment problem (see chapter 3). There are $O(k^2)$ alignments which each take $O(m^2)$ time, given there are $k$ sequences of length $m$. This gives us the time $O(m^2 \times k^2)$. In practice $k$ is never that big so the algorithm finishes in a few seconds unless given really many (several hundred) and long sequences.

In the next chapter I will go through my actual implementations of the algorithms presented in this chapter.
Chapter 5

Implementation

In this chapter I will start by discussing my choice of programming language. Then I will explain the actual implementations. In doing so, I will include some of the code used to implement the algorithms. The chapter will finish with a section on validating the correctness of my implementations followed by how I determined the optimal threshold function coefficients.

5.1 Choice of Programming Language

I chose to implement the algorithms using Java[15]. The reasons for this are several. First of all Java is very accessible. The API is well documented and it is widely used so there is a massive Java community which means that any problems I might run into others probably encountered too. Second, I have been using Java for a few years and know it quite well, so it would not be necessary to learn a new language. Third, it is easy and fairly quick to implement, memory need not be allocated and garbage collection is automatic. This means I could spend my time actually implementing the algorithms instead of writing prerequisite code. There are of course downsides to Java as well. The memory usage can be quite steep and the speed could be found better in i.e. C or C++. In this thesis however the main focus is not on speed nor memory usage, so the downsides mentioned can be ignored. The time it takes to write the code could probably be better with an interpreted language such as python or ruby, but they are not as widespread and even though speed is not a priority, they are really a lot slower than Java. The speed gained in the production using such languages is probably not that big in my case anyway, since the code is quite compact.

5.2 Implementations

The algorithms to implement are well described in the literature, but from pseudo-code to actual code can be quite a jump. A single line of code in the pseudo-code can potentially be a lengthy process. An example of this is the following line from chapter 4:

\begin{verbatim}
for ("All neighbors, n, of v") {
    ...
}
\end{verbatim}
All these "neighbors" have to be created first. Another problem is that the whole data structure has to be implemented. For the MSA algorithm all the pairwise alignments (called F) need to be calculated and stored in some manner. The vertices need to be defined in some sane manner too. In this section I will describe how I implemented the algorithms.

5.2.1 Naive implementations

Alignment

The main method generating the alignment for the three sequence alignment method can be seen in listing 5.1. In the code I have three nested for loops which correspond to the backward dynamic programming method of filling out a table from the end corner to the start corner. It is very straightforward and simple which should make the risk of introducing errors smaller. I simply iterate over the three dimensions and in each iteration all options are put into an array, from which the minimum is selected at the end as the integer to put into the dynamic programming table. s1, s2 and s3 are the three sequences being aligned. The implementation for four strings is basically one more loop iterating over the extra dimension, so I will not go further into detail with it.

```
for (int i = dynTable[0][0].length - 1; i >= 0; i--) {
    for (int j = dynTable[0][0].length - 1; j >= 0; j--) {
        for (int k = dynTable[0][0].length - 1; k >= 0; k--) {
            int[] values = new int[7];
            for (int m = 0; m < values.length; m++) {
                values[m] = Integer.MAX_VALUE;
            }
            if (i != dynTable.length - 1 && j != dynTable[0].length - 1 &&
                k != dynTable[0][0].length - 1) {
                values[0] = dynTable[i + 1][j + 1][k + 1] +
                scoringFunction.score(s1.charAt(i), s2.charAt(j)) +
                scoringFunction.score(s1.charAt(i), s3.charAt(k)) +
                scoringFunction.score(s2.charAt(j), s3.charAt(k));
            } else {
                values[0] = Integer.MAX_VALUE;
            }
            if (i != dynTable.length - 1 && j != dynTable[0].length - 1 &&
                k == dynTable[0][0].length - 1) {
                values[1] = dynTable[i + 1][j + 1][k] +
                scoringFunction.score(s1.charAt(i), s2.charAt(j)) + 2*gapCost;
            } else {
                values[1] = Integer.MAX_VALUE;
            }
            if (i == dynTable.length - 1 && j == dynTable[0].length - 1 &&
                k == dynTable[0][0].length - 1) {
                dynTable[i][j][k] = 0;
            } else {
                dynTable[i][j][k] = Utilities.min(values);
            }
        }
    }
}
```

Listing 5.1: Naive implementation for three sequences. Backward dynamic programming is used in the naive implementations. Three nested for-loops iterate through all vertices until the end vertex is reached.
5.2. IMPLEMENTATIONS

Traceback

For the traceback functionality of the naively implemented methods I simply start at the start corner (The optimal score is in this corner because they are filled starting at the end corner) and go through the preceding entries looking at which one could be the one that resulted in the score in the end corner. The first one that could result in this is added to the alignment. Then the same is done for that entry. And so on until the top left hand corner is reached.

5.2.2 Data Structure for the MSA Algorithms

The three MSA algorithms all make use of the same data structure (more or less). In this section I will try to elaborate the structure.

In all the MSA implementations, the `align()` method takes a boolean argument which indicates whether an alignment needs to be made, using traceback. The vertex is implemented as a class, with fields for the coordinate, the weight and the sum of the coordinates (used to compare the vertices).

5.2.3 MSA Alignment

The first implementation of the MSA algorithm (which I ingeniously called MSA...) is shown in listing 5.2. It is based on the PriorityQueue from Java 1.5. A major problem in their implementation is that in order to update an item in the queue, it must first be removed and then inserted again (there is no update method for the Java PriorityQueue). It takes time linear to the number of elements in the queue to remove a specific element. Therefore the algorithm is very slow if an item is removed and then inserted again. Another option is to leave the item and add a new one, with the correct weight. This, however, increases the memory use. Another problem, is that it is not possible to fetch a specific element from the queue. It is only possible to remove a specific element. Therefore a second form of lookup table was needed for the case where a vertex in the neighbor list has already been encountered and is already in the queue\(^1\). In this case it must be fetched, so that the weight can be set accordingly. Here I have used a HashMap implementation by java. This performs gets and puts in constant time (expected), there is however some memory use. The code corresponds much to the pseudo code presented in chapter 4.

Traceback

If the traceback boolean is true, all the touched vertices are kept in the map. If done so it is easy to reverse the alignment process, by going through the "negated" neighbors of the end corner, and looking which one can contribute to the score in the end corner. First one found is appended, and then the negated neighbors of that vertex is iterated through etc. Finally the start corner is reached and we are done. The code in listing 5.3 does exactly that.

\(^1\)This structure is actually also used in the original algorithm by Gupta where the trie is used in the same manner as I here use the HashMap.
private boolean run(boolean internalAlignmentMode) {
    while (Q.size() != 0) {
        v = Q.poll();
        verticesAdded++;
        if (v.equals(t)) {
            score = v.getWeight();
            return true;
        }
        if (!internalAlignmentMode) {
            map.remove(v.getKey());
        }
        int F_sum = Utilities.sumF(v.getPoint(), F);
        if (v.getWeight() + F_sum <= thresholdScore) {
            neighbours = Utilities.newPoints(v.getPoint(), binaryArray, endCorner);
            for (int i = 0; i < neighbours.size(); i++) {
                n = new Vertex(Integer.MAX_VALUE, neighbours.get(i));
                if (!v.equals(n)) {
                    temp = map.get(n.getKey());
                    if (temp != null) {
                        n = temp;
                    }
                    int cost = v.getWeight() + Utilities.cost(v, n);
                    if (cost < n.getWeight()) {
                        n.setWeight(cost);
                        Q.remove(n);
                        Q.add(n);
                        map.put(n.getKey(), n);
                    }
                }
            }
        } else {
            map.remove(v.getKey());
        }
        score = -1;
        return false;
    }
}

Listing 5.2: MSA implementation of the Gupta algorithm which uses Java's PriorityQueue and a HashMap for the data structure.
public String[] getAlignment() {
    for (short[] biar : binaryArray) {
        tempArray = new short[binaryArray.get(i).length];
        int j = 0;
        for (int bi : biar) {
            if (bi == 1) {
                tempArray[j] = -1;
            } else {
                tempArray[j] = -2;
            }
            j++;
        }
        negatedBinaryArray.add(tempArray);
        i++;
    }
    current = t;
    t.setWeight(score);
    sbArray = new StringBuilder[sequences.length];
    for (int j = 0; j < sequences.length; j++) {
        sbArray[j] = new StringBuilder();
    }
    while (!current.equals(s)) {
        ArrayList<short[]> neighbours =
            Utilities.newPointsTraceBack(current.getStartPoint(), negatedBinaryArray, startCorner);
        for (short[] ne : neighbours) {
            temp = new Vertex(Integer.MAX_VALUE, ne);
            if (map.containsKey(temp.getKey())) {
                n = map.get(temp.getKey());
                if (current.getWeight() == n.getWeight() + Utilities.cost(n, current)) {
                    for (int k = 0; k < sequences.length; k++) {
                        if (n.getCoordinate(k) + 1 == current.getCoordinate(k)) {
                            sbArray[k].append(sequences[k].charAt(n.getCoordinate(k)));
                        } else {
                            sbArray[k].append('-');
                        }
                    }
                    current = n;
                    break;
                }
            }
        }
    }
    String[] alignments = new String[sbArray.length];
    for (int j = 0; j < alignments.length; j++) {
        alignments[j] = sbArray[j].reverse().toString();
    }
    return alignments;
}

Listing 5.3: MSA implementation of traceback

5.2.4 MSACoor

The MSACoor implementation uses a different approach than the other two implementations (see chapter 4). The difference between the implementation of the MSACoor algorithm and the MSA algorithm, is the way I define the comparator in Vertex (which is called Vertex2 in MSACoor - yes the name fulfills all programming conventions). In listing 5.4, the code for the comparison is seen. The first thing I do is check the sum of the coordinates of the vertices being compared, and if they are different, I return the difference. If not I iterate through the coordinates and return the difference when and if one is encountered. If not 0 is returned. This should ensure that the vertices are fetched from the TreeSet in the correct order. I used the TreeSet in this implementation, because it has a remove() method that takes constant time (expected). This comes at a price of memory, because it needs a lookup key which in this case is the string version of the coordinate for a given vertex.
public int compareTo(Object o) {
    if (((Vertex2) o).getSum() != this.sum) {
        return this.sum - ((Vertex2) o).getSum();
    }
    for (int k = 0; k < point.length; k++) {
        if (point[k] != ((Vertex2) o).getPoint()[k]) {
            return point[k] - ((Vertex2) o).getPoint()[k];
        }
    }
    return 0;
}

Listing 5.4: Vertex comparison for MSACoor. First the sum is compared and the difference returned if they are different. If not each point is compared until two non-equal points are found.

The alignment iterative process and the traceback functionality is the same as for the MSA implementation, with the exceptions stated above, so I will not go further into this.

5.2.5 MSATrie

For the MSATrie implementation I did an implementation which follows Guptas paper[3] closely. The iterative alignment process is the same as for the MSA algorithm. The difference is that instead of a TreeMap, the bucket queue implementation is used and instead of the hashmap, the trie is used.

The Trie

In order to keep track of the visited vertices (the job of the HashMap in MSA and MSACoor) I need a Trie (see description of a trie in chapter 4). The trie is defined in a recursive like data structure. Meaning that the actual trie only has an explicit reference to one single node, the root. The root then has references to other nodes, as its children. A node is defined by the following:

- The depth in the trie where the node is located
- The coordinate (a single integer which corresponds to a single entry in the vertex coordinate. This is -1 if it is the root)
- The weight which is -1 if it is internal. Else it is a leaf, corresponding to a vertex
- A hashmap of its children, with the child nodes coordinate entry as the key
- Reference to its parent node

The trie makes use of the following methods (the class includes other methods, but they make use of these methods)

traverseInsert(short[] key, int weight, Node current)
traverseToNode(short[] key, Node current)
traverseToRootAndDelete(Node current)

The method traverseInsert(...) traverses the trie from the root to the leaf corresponding to the vertex I want to insert. On the way it inserts any nodes not already present in the
5.2. IMPLEMENTATIONS

trie. \texttt{traverseToNode(...)} simply traverses the trie from the root to a given leaf. Either to fetch the weight or to check if the trie contains the vertex. \texttt{traverseToRootAndDelete(...)} traverses the trie from a given node (often a leaf node) and removes any inner nodes with no children on the way to the root. The use of the trie is to fetch the weight of a given vertex which cannot be done by the queue data structure which is described next.

The Queue

The queue class implementation is quite simple. It consists of a number of buckets corresponding to the threshold value. There is no reason to have any more than that, since, they will never be used. Each bucket consists of a linked list. This list keeps vertices with the weight corresponding to the bucket number. When polling the element with the lowest weight, the first bucket with any vertices in it is simply fetched and the first vertex in the linked list in that bucket is removed and returned. To change the weight of a vertex the vertex can be fetched by its old weight and moved to a new bucket. The old weight needs to be fetched from the trie, hence why we need it.

5.2.6 Multiple Star

Opposite the other implementations, this is not an exact solution. The specifics of the algorithm are explained in chapter 4. In this section I will go through the implementation of it.

Alignment

I start off by finding the center sequence, by aligning each sequence to every other sequence, summing up the pairwise scores for each sequence. The sequence with the lowest total score after aligning it with all other sequences is chosen as the center sequence. Then all the sequences need to be aligned to the center sequence again and added to a column of sequences. The code implementing this is shown in listing 5.5. I keep all the sequences in two lists, \texttt{alignedWithCenterList} and \texttt{notAlignedWithCenterList}. The names make a nice hint at what the two lists contain. I poll sequences from \texttt{notAlignedWithCenterList} until it is empty and align each of them with the center sequence. After I have aligned the two, using dynamic programming I insert any newly inserted gaps in the center sequence into the sequences already present in \texttt{alignedWithCenterList} at the same index as in the center sequence. When done I add the center sequence to \texttt{alignedWithCenterList}. 
private String[] MakeColumns(int centerSequence, 
    LinkedList<String> notAlignedWithCenterList) {
    String center = sequences[centerSequence];
    ArrayList<String> alignedWithCenterList = new ArrayList<String>();
    while (!notAlignedWithCenterList.isEmpty()) {
        String s = notAlignedWithCenterList.poll();
        String[] seqs = new String[2];
        seqs[0] = center;
        seqs[1] = s;
        TwoStringAlign aligner = new TwoStringAlign(seqs, scoringFunction, gapCost);
        center = aligner.align(true);
        alignedWithCenterList.add(aligner.getAlignment()[1]);
    }
    return alignedWithCenterList.toArray(new String[0]);
}

Listing 5.5: Multiple Star Implementation

Traceback

Traceback for the multiple star method is very trivial. alignedWithCenterList contains the alignments at the end. The score can be calculated by summing all pairwise scores in each column.

5.3 Test of correctness

To test the correctness of my implementations, I have implemented naive dynamic programming solutions to three and four string alignments (see section 5.2.1). These naive implementations are not tested against any other algorithms (then those would need to be tested too etc.). I have tested them by feeding them some simple, short strings which could be aligned by hand. Furthermore, the implementation of these algorithms is much less complex than the MSA algorithms (and the multiple star algorithm for that matter), so the risk of making errors is quite a bit less.
5.3. TEST OF CORRECTNESS

5.3.1 MSA Algorithms

Since all the MSA implementations all should return an optimal alignment score, I have tested them together. I have tested them on three and four sequences against the naive solutions. The test setups were 1000 runs of each three and four sequences of length 30 and with a mutation rate of 0.5. None of the MSA results differed from the results from the naive implementations.

5.3.2 Multiple star algorithm

According to Gusfield[14] the Multiple star algorithm has an upper bound ratio between the Multiple Star score and the exact score of $\frac{S(\text{MultipleStar})}{S(\text{Exact})} = \frac{2(k-1)}{k}$ where k is the number of sequences. I made various tests to ensure my implementation was kept below this threshold. I did more than ten thousand alignments with 5 sequences of length 25 and a mutation rate of 0.4. None of these scored above the upper bound score which I used the MSACoor implementation together with Upper bound defined by Gusfield to find. Figure 5.1 shows a graph of 100 alignments of three, four, five and six sequences of length 30 and with a mutation rate of 0.4. The “Max” line (at y=1) indicates the maximum value according to the maximum defined score, argued by Gusfield. The reason it is 1 and not some other number, is that the numbers are normalized around this value. It is quite clear that the higher the number of sequences the further away from the upper bound the score is. It indicates that the upperbound specified by Gusfield seldom is reached and is very generous the more sequences are used. Though it is never possible to show that my implementation is always below the upper bound score, I am fairly confident that it is.

![Figure 5.1: Correctness test of the Multiple star implementation with 3, 4, 5 and 6 sequences of length 30 and a mutation rate of 0.4. The line plot called Max is the upper limit for the multiple star method result, according to Gusfield. It is clear that the more sequences are tested, the further from the maximum value the multiple star result is. Meaning for many sequences the upper bound is very generous.](image-url)
5.4 Determining Threshold Function Coefficients

The two threshold functions described in chapter 4, polynomial and exponential, can take different coefficients. In this section I will show how these coefficients were chosen. Since the time available to me in this thesis was not unlimited, I have not done a thorough test of all possible coefficients. The values tested are shown in the following. The data I use for finding the optimal coefficients are not chosen to be respective for any specific data set. Therefore my choice of coefficients will be biased towards datasets that are similar to the ones in the following experiments. However the experiments will probably give some better coefficients than just guessing.

5.4.1 Polynomial

Since the polynomial function does not grow quite as fast as the exponential function, I decided to include a constant factor too, so that the tested values would be \( a \) and \( y \) in \( a \times x^y \). I begin finding the optimal value for \( y \) by letting \( a \) be 1. Figure 5.2 shows a setup with exponent values of 1.5, 2, 2.5, 3, 3.5 and 4. The setup measures percentage of touched vertices as a function of sequence length using five sequences with a mutation rate of 0.3. It shows that with longer sequences, the different exponent values do not make much difference, but for shorter sequences the lower exponents are better. I have chosen 2 as the exponent to use in my experiments. It is the highest value which is consistently low for all sequences. I defer that the faster the function is growing, without the number of touched vertices being too great, the better. One might wonder why the exponent value 4 fluctuates as it does. This is because some threshold value is used from sequence length 25 to sequence length 35. At length 35 it is incremented into some value which lies much above the true score why more vertices need to be touched.

Next I find a good constant which is depicted in figure 5.3. The setup is the same as the results in figure 5.2. The exponent used is 2. It is clear, that the longer the sequences are, the less the constant matters. I therefore simply choose 1.

5.4.2 Exponential

For the exponential function I only tested different base values, \( y \) in \( y^x \). The setup also measures vertices touched as a function of sequence length using five sequences and a mutation rate of 0.3. The results can be seen in figure 5.4. The values tested are 1.1, 1.3, 1.5, 1.7, 2, 2.2. The reason for the chosen values are that I learned that higher values are of no use since they grow too quickly. The 0.2 difference between the values chosen is arbitrary. It shows that 1.1 is the best base value to use, as it is a little better than 1.3. The rest simply grow too quickly, and will therefore have way too high values in many cases. 1.1 is the same as a 10% increase in the threshold value for each iteration.

In summary the polynomial threshold function is \( x^2 \) and the exponential function is \( 1.1^x \).

In the next chapter I will go through my experiments.
5.4. DETERMINING THRESHOLD FUNCTION COEFFICIENTS

Figure 5.2: The exponent of the polynomial threshold function is varied from 1.5 to 4. The setup is on five sequences with a mutation rate of 0.3, with the sequence length from 10 to 75. The figure shows the percentage of touched vertices as a function of the sequence length. The low values for the exponent give similar results. I chose to use the value 2, because it was the highest score that was consistently low with regard to touched vertices.

Figure 5.3: The constant of the polynomial function is here being evaluated, starting at 1 and ending at 10. The setup measures the percentage of touched vertices as a function of the sequence length using five sequences of varying length and a mutation rate of 0.3. The figure shows that the choice of constant does not make much difference, hence why I simply chose 1.
Figure 5.4: The base value of the exponential function is determined in this test by measuring the percentage of touched vertices using five sequences, a mutation rate of 0.3 and an increasing sequence length. The values tested start at 1.1 and end at 2.2. It is quite clear that the higher the value, the worse. Therefore I have chosen 1.1 as the base of the exponential function which is the same as a 10% increase in each iteration.
Chapter 6
Experimental Results and Discussion

In this chapter I will go through all my experiments. The experiments I have chosen to do should give me some idea about the questions I ask in the introduction. I start off by explaining how I generate my test sequences. Then I quickly summarize over the experiments. In the following sections I describe each of my experiments and their results. At the end of each experimental section I will conclude on that given experiment and try to elaborate on my findings. At the end of the chapter, I will put my results together.

6.1 Test Sequences

6.1.1 Sequence Generation

In order to test a given setup I create some random DNA sequences. My input parameters to the sequence generator are the number of sequences, the mutation rate and the length of the sequences. I start by generating one sequence by randomly picking a base for each position of the sequence. The other sequences (however many is determined by the number of sequences defined before) are then generated by making mutations to a copy of the first sequence. The mutation rate determines the number of mutations and even though it is random, I make sure that there are never less than the given percent of mutations. Also the mutation is not allowed to mutate into the same base as it was before. This generates sequences that I am fairly sure have the mutations I specify.

6.1.2 Choice of Sequences

All the experiments are run on DNA sequences. The main reason for not using protein sequences is that it requires a bit more time due to the more complex scoring function. Furthermore sequences consisting of 20 different different amino acids are more complex than DNA, with only four bases. There is no reason to complicate the experiments with complexity of the sequences when the goal is not to test the biological correctness of the algorithms or scoring functions.
6.2 Experiments

To test the MSA algorithms, I made a few different tests. Table 6.1 shows what tests have been run. The distance matrix used in all alignments is the extended distance matrix shown in table 3.1(b) in chapter 3. All the tests have been run on a 2.50 GHz Intel Core 2 Duo Laptop with 4GB RAM running Windows Vista (64bit). Unless specified the tests are all run using the -Xmx2064m option, specifying that the heap size would not exceed 2GB RAM. Some of the experiments will overlap each other and, in part, show the same results. Some of the experiments also use quite short sequences which is due to the high memory use when the number of sequences is increased.

Table 6.1: Tests performed on the MSA algorithms

<table>
<thead>
<tr>
<th></th>
<th>Sequence length</th>
<th>Number of sequences</th>
<th>Mutation rate</th>
<th>Time</th>
<th>Vertices Touched</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optimal threshold value</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>MSACoor VS. MSATrie</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Touched Vertices</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Cloud Size</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Threshold Functions</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MSACoor VS Naive</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

6.2.1 Optimal Threshold Value

With regard to traceback, I want to find out if it is possible to run the algorithm twice where the second run will use the score from the first run (which will be the optimal score) as threshold value. If the number of touched vertices is much less in the second run, it could be possible to remember the vertices in the second run and do a traceback from that, without the cost being too great. This also tells me something about the importance of the threshold value. I have run several tests where I vary the sequence lengths, number of sequences and the mutation rate. A section for each is included in this section.

Sequence Length

This test is run with five sequences, a mutation rate of 0.2 and a varied sequence length. The results are shown in figure 6.1. The graph shows the ratio between the number of added vertices using both multiple star result and optimal value as threshold function (one alignment for the same sequences for each). I have added a mean line to the graph which is between 0.1 and 0.2. Meaning on average, with the given setup, the second run processes less than a factor 0.2 of the number of vertices that the first run does. More important for a practical use is the extra time it takes which is much less, as figure 6.2 shows (same run as figure 6.1). The figure shows the total time to run both algorithms and the time to run the MSACoor with the multiple star value as threshold value. The extra time it takes to run the algorithm once more, with the optimal score as a threshold function is negligible. In fact it is actually very difficult to even see the plots for the total time, given that the plots for the total time is only a little higher than the time the first run took.

Number of Sequences

The previous experiment is only run on five sequences, so to see what difference it makes, I also did an experiment where I vary the number of sequences. The results are shown in
The setup is run ten times using a sequence length of five, a mutation rate of 0.7 and a varying number of sequences. As with the previous experiment, I here show the ratio between the exact score and the multiple star result as threshold values. The figure shows the average of the ten runs. It shows that for five sequences the time is almost the same as for the run with a Multiple Star result. The reason for this is due to the randomness of the experiments. In the rest of the cases the ratio is less than 0.5, which still makes it feasible to use this method for a traceback functionality for several sequences.

**Mutation rate**

Figure 6.4 shows that when the mutation rate is increased (above 0.2 at least), the time/vertices touched ratios do not differ much. The setup is five sequences of length 40 and an increasing mutation rate. When the mutation rate is very low the ratio is higher which is due to the difference of the multiple star threshold and optimum score is very small. This is also the reason why the trend lines are above most of the measured points. It would be more fair to make the trend lines according to measurements with a mutation rate above 0.1. In that case it would be close to 0.1 for both the time and vertices touched ratios.

**Conclusion of Optimal threshold value**

In every experiment the second run with the optimal score as threshold value, touches far less vertices and takes less time. It is apparent, that the closer the threshold function gets to the optimal score, the quicker and more efficient the algorithm is. It is also proven that this method may be used to do a traceback without it being too time consuming, because the extra time it takes to run the algorithm once more is little, one could remember all the vertices and do a traceback.
CHAPTER 6. EXPERIMENTAL RESULTS AND DISCUSSION

Figure 6.1: MSACoor run with Optimum score as threshold value. The graph shows the ratio of the number of added vertices between the first MSACoor run (using Multiple star method for threshold value) and the second MSACoor run (using the optimal score as threshold value). There are five sequences and the mutation rate is 0.2. The figure shows that the second run processes less than a factor 0.2 of the number of vertices that the first run does.

Figure 6.2: MSACoor run with Optimum score as threshold value. The graph shows the total time for both the MSACoor runs, one using the Multiple star method for threshold value and the other using the optimal score (from the MSACoor) and the time used by the MSACoor with Multiple star method threshold value alone. There are five sequences and the mutation rate is 0.2. It is clear that the extra time it takes to run the algorithm once more with the optimal score as a threshold value is negligible.
6.2. EXPERIMENTS

Figure 6.3: MSACoor run with Optimum score as threshold value. The graph shows the time ratio between the MSACoor run with Multiple Star result as threshold value and the MSACoor with the optimum as threshold value. I did 10 runs with the number of sequences varying from three to ten. The sequence lengths were set to five in order to make it possible, in a reasonable amount of time. The mutation rate is set to 0.7. The figure shows the average from the ten runs where most of them is around 0.3-0.5.

Figure 6.4: MSACoor run with Optimum score as threshold value. The graph shows the time/vertices touched ratio between the MSACoor run with Multiple Star result as threshold value and the MSACoor with the optimum as threshold value. The sequence length is 50 and there are five sequences. The mutation rate is varied in each run. The red and blue trend lines are the averages for the time and vertices touched ratios respectively. Except for the ratios at low mutation rates they all lie around 0.1.
6.2.2 MSACoor VS MSATrie

Other than the different data structures used to make these two implementations, they also differ in the way they work. As explained in chapter 4, MSACoor uses a topological ordering of the vertices and MSATrie uses the weight to determine the ordering of the queue. The respective implementations are explained in chapter 5. In this section I look at the number of touched vertices, the time use and the memory use of the two implementations.

Vertices Touched

To compare the MSATrie and the MSACoor algorithms I ran both of them and noted the number of touched vertices. The results can be seen in figure 6.5. The setup is run with four sequences, a mutation rate of 0.3 and a varying sequence length. The difference is minimal, however MSATrie does add a few vertices less than MSACoor. This could indicate that MSATrie uses a bit less memory than MSACoor or that it is a bit faster, since it does not need to compute as many vertices.

Time Use

In figure 6.6 the same setup as the previous experiment can be seen except here I have measured the time. It shows that MSATrie is a bit slower than MSACoor for shorter sequences, but for longer sequences the time difference becomes greater.

Memory Use

I wanted to test the memory use. This is a bit more tricky since, it is very difficult to measure how much memory the application uses. Instead I use the -Xmx128m flag for the JAVA VM which specifies that the Java virtual machine may use no more than 128 MB of memory. Then I see which method can get through the longest sequences, using five sequences and a mutation rate of 0.3 before getting a memory overflow exception. I ran this setup ten times for both MSACoor and MSATrie and on average the sequence lengths of the MSACoor runs were 40 longer than those of MSATrie. MSATrie is apparently horribly inefficient memory wise, compared to MSACoor.

Conclusion on MSACoor VS MSATrie

My implementation of the MSATrie algorithm is shown to be inferior to the MSACoor implementation in every aspect (except vertices touched where the difference is minimal). I find no reason whatsoever to use the method.
6.2. EXPERIMENTS

Figure 6.5: MSACoor VS MSATrie with number of touched vertices. Using four sequences, a mutation rate of 0.3 and varying sequence lengths. The difference between MSACoor and MSATrie is minimal, but MSATrie adds a little less vertices.

Figure 6.6: MSACoor VS MSATrie with time. Using four sequences, a mutation rate of 0.3 and a varying sequence length. The time use of MSATrie grows significantly compared to MSACoor when the number of sequences are increased.
CHAPTER 6. EXPERIMENTAL RESULTS AND DISCUSSION

6.2.3 Touched Vertices

One of the main points of interest in this thesis is the number of vertices touched using these algorithms and how much this number could be reduced compared to a normal dynamic programming implementation where all vertices need to be processed. I vary the sequence lengths, number of sequences and the mutation rate to see how these influence the number of vertices touched. I only use the MSACoor implementation, since it has proven to be the fastest, uses less memory and the number of touched vertices does not vary significantly from MSATrie.

Function of Number of Sequences

Figure 6.7 shows that the number of touched vertices is on average a factor 0.15 less than the total number of vertices that would have been processed by a naive implementation. I ran the algorithm five times with the same setup which was a sequence length of six and a mutation rate of 0.7. The number of sequences were varied from three to nine. Figure 6.8 shows a run using the same setup as the experiment shown in figure 6.7, except that I only ran it for nine sequences, but 40 times, to see if I could get a better picture of the fraction of added vertices for many sequences. It shows that the percentage of touched vertices fluctuates between zero and 0.4. The mean is just above 0.10, so most of the values are much lower than 0.4.

Function of Sequence Length

This test is run with five sequences, a mutation rate of 0.3 and a varied sequence length. The results are shown in figure 6.9 and it shows that the longer the sequences, the smaller the percentage of touched vertices is. Above sequence length of 50 almost all points are below the average of 0.005. So the longer the sequences the more efficient the algorithm is.

Function of Mutation Rate

Not surprising, as the mutation rate is increased, the number of added vertices increases too. This is also exactly what figure 6.10 shows. The setup includes five sequences of length 40 and a mutation rate starting at 0.0 and incremented by 0.02. When the mutation rate hits 0.4 the number of touched vertices start to increase drastically. The fraction of touched vertices seems to decrease at the end which might be because of how the generation of the test sequences work. Because they are all made from the same sequence, a mutation rate above some specific value, might not result in sequences that are more diverse. When the mutation rate is 1.0 for example, there is only three options at each index in the new sequences, due to the fact that I restricted the mutation to a base that is not already in that position in the original sequence.

Conclusion on Vertices Touched

One of the major aspects of this thesis is the investigation of the number of vertices touched. The experiments in this section show that there is a major reduction in the number of vertices that need to be touched. The number declines as the sequences grow in length and grows as the mutation rate is increased. Even though it is far lower than the total number of vertices, it is a constant factor factor lower. This is best seen in figure 6.9 where it is fairly constant.
when the sequence length exceeds 40. Because it is a constant factor less, it will still scale quite badly.

![Figure 6.7: MSACoor with fraction of touched vertices with regard to the total number of vertices as a function of the number of sequences. Sequences of length 6 with mutation rate of 0.7. 5 different runs. Each number in the legend corresponds to a run. The turquoise colored line is the average from all the runs. The number of vertices touched is on average a factor 0.15 less than the total number of vertices.](image1)

![Figure 6.8: MSACoor with percentage of touched vertices with regard to the total number of vertices. Nine sequences of length six and a mutation rate of 0.7 was used. 40 runs in total with the same setup. The mean value is shown with the blue line. The percentage of added vertices fluctuates between zero and 0.4 with an average around 0.10.](image2)
CHAPTER 6. EXPERIMENTAL RESULTS AND DISCUSSION

Figure 6.9: MSACoor with fraction of touched vertices with regard to the total number of vertices as a function of the sequence length. Five sequences and a mutation rate of 0.3 was used. It is clear that the longer the sequences, the less the number of touched vertices is. It flattens out after the length of around 40. After this it is fairly constant.

Figure 6.10: MSACoor with fraction of touched vertices with regard to the total number of vertices as a function of the mutation rate. Five sequences of length 40 was used. When the mutation rate is increased, the number of touched vertices increase too. The reason the fraction decreases art higher mutation rates could have something to do with the way my test sequences are generated.
6.2. EXPERIMENTS

6.2.4 "Cloud" size

The cloud is what I call all the vertices in the queue at any given time. This is what actually sets the limits on the possibilities of the algorithm with regard to memory use. In this section I try to see how this cloud behaves when changing the different parameters. Figure 6.11 shows that the form of the cloud size over time is the same for different runs, but of course the number of vertices in it vary according to the sequences being aligned.

In the following experiments I measure the size of the cloud as the algorithm progresses, as a function of the number of sequences, sequence length and the mutation rate. Given the fact that each run for a set of sequences is never the same, the data on the x-axis is normalized, so that it lies between zero and one. I call it algorithm progress where zero is not begun and one is done.

Function of Number of Sequences

Figure 6.12 shows, as a general rule that the more sequences there is, the bigger the cloud gets, relative to the total number of vertices in the search space. The setup is using a sequence length of six, a mutation rate of 0.7 and a varying number of sequences.

Function of Sequence Length

Generally, the longer the sequences, the smaller the size of the cloud is as a percentage of the total number of vertices. Of course there is some irregularities which can be expected with random data. This can be seen in figure 6.13. The setup is using five sequences, a mutation rate of 0.3 and a varying sequence length.

Function of Mutation Rate

As expected, the more diverse, the sequences are, the larger the percentage of touched vertices in the cloud gets. Figure 6.14 shows this, albeit with some deviations. The setup is using five sequences, a sequence length of 40 and an increasing mutation rate. Though a mutation rate of 1.0 falls below three of the other runs, the tendency is that the larger the mutation rate, the bigger the cloud is.

Conclusion of Cloud size

The cloud size as a percentage of the total number of vertices grows as the sequences get more numerous, as the mutation rate grows and as the sequence length grows. This is not surprising as it is expected, that more vertices need to be touched for more diverse, longer and numerous sequences.
CHAPTER 6. EXPERIMENTAL RESULTS AND DISCUSSION

Figure 6.11: MSACoor with touched vertices. 5 sequences of length 50 with mutation rate of 0.5. 10 different runs. The general shape of the cloud size as a function of the number of iterations done.

Figure 6.12: MSACoor with percentage of total vertices in the cloud. 5 sequences of length 50 with mutation rate of 0.5. 10 different runs. It is apparent that the more sequences, the bigger the cloud gets.
6.2. EXPERIMENTS

Figure 6.13: MSACoor with percentage of total vertices in the cloud. 5 sequences of length 50 with mutation rate of 0.5. 10 different runs with varying sequence lengths. It shows that the longer the sequences, the smaller the size of the cloud gets, relative to the total number of vertices gets.

Figure 6.14: MSACoor with percentage of total vertices in the cloud. 5 sequences of length 40 and a varying mutation rate. 10 different runs. The figure shows that the tendency is that the higher the mutation rate, the larger the cloud size is relative to the total number of vertices.
6.2.5 Varying Threshold Functions

The threshold value has shown to be quite important, therefore I will test three different functions. Constant value (being both multiple star alignment result and the exact score), an exponential function and a polynomial function. As described in chapter 4 the major difference between the functions, is that the exponential and polynomial functions do not depend on the input whereas the constant threshold function (when using multiple star alignment score or similar at least) is based on the input sequences. This should be visible in some of the tests, as the polynomial and exponential functions should have sweet spots where the function value is little above the optimal score for each set of sequences. The reason I have tested the algorithm with the exact score as threshold value too, is that it will show how much more can be gained by getting a lower threshold value, as the exact score would be the optimal threshold value. It should be noted that the number of touched vertices by the non constant functions is the total number of vertices touched in all iterations.

Function of Number of Sequences

Figure 6.15 shows a setup where I have increased the number of sequences of length six with a mutation rate of 0.7 and measured the percentage of touched vertices. It shows that the polynomial function is the best all round and the constant (multiple star method score) is the worst. As the sequences get more numerous the percentage of touched vertices get lower and the differences between the different threshold functions becomes smaller. Compared to the exact value, it is not much below the other values at a higher number of sequences. This hints that at some point a good threshold value does not make much difference anymore.

Function of Sequence Length

In figure 6.16 I have tested the threshold functions percentage of touched vertices as a function of the sequence lengths. The setup uses five sequences, a mutation rate of 0.3 and a varying sequence length. The constant function (multiple star) touches a little more vertices than the others which are fairly equal.

Function of Mutation Rate

Figure 6.17 shows that the more diversity in the sequences the better the polynomial and exponential threshold functions seem to be. The setup is using five sequences of length 40 and a varying mutation rate. As the mutation rate gets above 0.5 the number of touched vertices using a constant threshold function (multiple star result) grows quicker than that of a polynomial and exponential function. The latter two are comparable to the exact value.

Conclusion on Threshold Functions

The polynomial and exponential functions outperform the constant function (Multiple star result), on every experiment. In figure 6.16 the difference between the three functions is less as the sequence length gets above 20, even though the multiple star threshold function is a little higher. I find it quite surprising though that a function that does not use the sequences being aligned can be the source of a much better threshold value, than an algorithm as the multiple star algorithm. The experiments also show that not much more can be gained by finding improved threshold functions, compared to the polynomial and exponential functions.
In the experiments, they do not touch many more vertices than the exact value. Looking at the number of sequences, it suggests that there might be a limit as to when a better threshold value will improve the algorithm drastically.

Figure 6.15: MSACoor with percentage of touched vertices as a function of the number of sequences for different threshold functions using sequences of length six with mutation rate of 0.7. The polynomial and exponential functions touch less vertices than the constant function (multiple star value) for most of the runs.

Figure 6.16: MSACoor with percentage of touched vertices as a function of sequence length for different threshold functions using five sequences with a mutation rate of 0.5. As the sequence length gets above 20, the differences become smaller, however the constant function using the multiple star result is a bit behind.
CHAPTER 6. EXPERIMENTAL RESULTS AND DISCUSSION

Figure 6.17: MSACoor with touched vertices as a function of mutation rate for different threshold functions using five sequences of length 40. When the mutation rate is increased, the constant threshold function (multiple star result) touches far more vertices than the exponential and polynomial functions. They grow too, but much more modest. They are not much above the exact threshold value.

6.2.6 MSACoor VS Naive

To find out if the implementation is actually quicker than a naive solution, I have run a setup with MSACoor (both constant threshold and polynomial function) against my naive implementations for three and four sequences. Both with a mutation rate of 0.5. I start with sequences of length 50 and 90 respectively. Figure 6.18 shows that MSACoor with the multiple star result as a threshold value is slower than the naive solution for three sequences, but that the polynomial function is a little quicker. The trend lines added are there to make it easier to see the differences between the implementations. Figure 6.19 is run on four sequences, and here both the multiple star threshold value (on average, following the trend line) and the polynomial function threshold value are quicker than the naive solution. The polynomial function is considerably faster.

Conclusion on MSACoor VS Naive

This experiment has shown that with the correct threshold function, the MSACoor algorithm is much quicker than the naive solution and could be used as an alternative. The polynomial threshold function is again here proven to be much quicker than the constant function using multiple star result. It does not show how the method scales with more sequences though.
6.2. EXPERIMENTS

Figure 6.18: MSACoor versus naive implementation for three sequences of varying length, with a mutation rate of 0.5. This test shows that for three sequences the naive implementation is still quicker than MSACoor using the multiple star result as a threshold value, but using the polynomial threshold function, is a little quicker.

Figure 6.19: MSACoor versus naive implementation for four sequences of varying length, with a mutation rate of 0.5. It shows that the polynomial threshold function is considerably faster than the naive solution and the constant threshold value function (multiple star result) is marginally quicker.
6.3 Discussion

I will use this discussion section to recap on the experiments. It should lead the way for my conclusion in next chapter.

It is clear that the MSACoor algorithm outperforms the original implementation by Gupta et al.
[3] (the implementation I call MSATrie). I suspect this is due to the extra amount of bookkeeping needed for maintaining the queue. If the threshold value used is the optimal score, a lot of time and effort can be spared which can be useful for traceback functionality. The threshold functions (polynomial and exponential) generally outperform the multiple star result as a threshold value. The larger the parameters, sequence length, mutation rate and number of sequences, get, the better they get, compared to the multiple star result. For many sequences the upper bound for the multiple star result gets close to $2S$ where S is the optimal score. This could explain why they get better and better compared to the multiple star value.

The threshold function experiments also hint that even with a perfect threshold value, the time and space use will still be substantial for larger data sets.

The number of touched vertices is a constant factor less than the total number of vertices that would need to be processed with dynamic programming (more or less). This is due to the fact that the number of vertices in the cloud is based on a cut surface through the search space which grows as $O(nk^{-1})$ (this is straight forward as the cut surface is as big as the whole search space apart from one dimension). This is the maximum size for the cloud, so this size is probably seldom reached.

Looking at the time use which in the end is what will determine if it will be usable, it is highly competitive at four sequences. Since I have no naive implementation for more than four sequences, I can unfortunately not verify whether this speedup scales with number of sequences, but looking at the difference between three and four sequences it would seem so.

In the last chapter I will conclude on my results with respect to my questions in the introduction.
Chapter 7

Conclusion and Future Work

The results I base my conclusions on are obtained using test data. Using true DNA sequences might have given some different results, but it would have been very difficult to do measurements as functions of mutation rate and sequence lengths, if I did not have proper control over these factors. Therefore I am aware that the results might not show the true properties of the algorithms, however I believe they give a good indication of them.

7.1 Conclusion

The experiments in chapter 6 show that the MSA algorithm by Gupta et al.[3] lessens the number of vertices needed to be processed significantly. It has not been possible (and probably is not) to find the exact number, but it lies around 0.1, according to my results. Furthermore the choice of threshold function is shown to make a big difference on the percentage of touched vertices and the amount of time needed when the number of sequences is not too big.

Ultimately the algorithm only pushes the limit of how long and how many sequences can be aligned. It still grows exponentially and is not feasible for very long or especially very many sequences.

To summarize over the points of interest I stated in the introduction I will go through each of them:

• How much is the search space reduced by using Gupta’s algorithm for aligning multiple sequences?
  When the sequence lengths are long enough (more than 40) and the mutation rate is above 0.3, the number of vertices touched is around a factor 0.1 less than the total number of vertices. Even though this is quite a reduction, it is only a constant factor which means that it will still scale exponentially with the number of sequences.

• Is the method feasible in practice?
  The method will find an optimal solution to a given problem, and in many cases it will do it quicker than the naive solution. Therefore I think it is very feasible. The only problem is that the worst case running time, is worse than the naive solution. I am, however unsure as to how likely it is to get a set of sequences resulting in a worst case running time.
• How much influence does the threshold value in Gupta’s algorithm have on the reduction of the search space?

The threshold value has a major influence on the running of the algorithm. Looking at the difference of my threshold functions versus the optimal values used as threshold functions, it is still possible to gain something, if one could find better threshold functions. The scaling problem will still be an issue, even with a perfect threshold function (the exact score), but it could be possible to push the limits even further. When many sequences need to be aligned, the threshold values do not seem to have the same impact as with less sequences though.

Looking at exact multiple sequence alignment algorithms in general, This method can be very useful, since finding an exact alignment is very time consuming and if the limit of how many sequences can be processed can be pushed a little further it will give users of multiple alignment programs some more options.

I implemented two different versions, MSACoor and MSATrie. The result showed that MSACoor was superior to the MSATrie implementation. If the MSACoor was implemented optimally, it would not need two hash tables, as with my implementation. Therefore it could have been even better than I have shown it to be compared to MSATrie. The overhead in maintaining the datastructure of the MSATrie algorithm just seems to be too steep.

7.2 Future Work

I would like to implement a leaner version of the MSACoor algorithm, perhaps in C++. It could be interesting to see how efficient and quick the algorithm could be implemented. It will of course have the same scaling, but maybe the limit will be pushed even further than it already is. If this was to be done, it would be really exciting to test the algorithms on some proper dataset, i.e. BAliBASE.[17]

It could also be interesting to do some more thorough testing on threshold functions, since it has been proven that these have quite an impact on the performance of the algorithm.

If many more experiments were conducted, it could be possible to find some values for the parameters (mutation rate, sequence length and number of sequences) where the algorithm performs better than for other values. In this way, the algorithm could be used when it would be most optimal or different versions/threshold functions could be applied in different situations. These parameters can easily be found by analyzing the sequences.

Also the traceback method implemented might be implemented more efficiently, so that all vertices need not to be saved. In this thesis backtracking was mainly implemented for correctness testing. Maybe Hirschbergs’[16] idea could be used in some modified way.

One last thing that would be nice, is a GUI version of the implementation or at least some more options in the command line version. The functionality to align proteins is in the source code, but since it was not the main focus of the thesis to apply the algorithms on various sequences, this part has been left out of the final version.

---

1BAliBASE is a database of manually made sequence alignments designed for evaluating multiple alignment programs.

2Hirschberg devised a method for doing tracebacks in linear time. The specifics will not be explained in this thesis.
Bibliography


Appendix

The Java Application

The source code and .jar file can be downloaded from http://www.daimi.au.dk/~jesperhm/. The source files include an ANT build file which can be used to build the JAR. The project is also created as an eclipse project, why it might be useful to open it in eclipse. The application needs to be run with Java 1.6 or newer and will only align DNA sequences.

Using the Java Application

The JAR can be run as follows:

```
java -jar MSA.jar [sequences] -alignment
```

[sequences] is all the DNA sequences seperated by space. -alignment is optional if the alignments need to be returned.