Aligning a Splice Graph to a Genomic Sequence

by

Øyvind Ølberg

Thesis submitted in partial fulfillment of the requirements
for the degree of *Master of Science*.

University of Bergen
Department of Informatics
June 1, 2005

This thesis is the product of one year of designing and implementing an algorithm to solve the assigned problem. It would not have been possible without the invaluable assistance of Eivind Coward. I would also like to extend my thanks to my fellow master students, if not for whipping me on then for making the frequent pauses from work more enjoyable (table tennis anyone?).

The source code cannot be found in this thesis, it is only available on a separate CD.

Øyvind Ølberg
June 1, 2005, Bergen

Contents

List of Tables

List of Figures

Chapter 1

Brief introduction and problem definition

DNA molecules can be subdivided into genes, a subunit containing all the information necessary to produce a protein. The complete collection of genes and intergenic DNA (DNA between the genes) in a cell is known as the cellular genome. In eukaryotic organism coding sequences (exons) in a gene is often interrupted by non-coding sequences (introns). After the transcription of a gene introns are removed from the primary transcript and exons are assembled into a sequence, known as mature mRNA. The process of removing such introns is called splicing. However, the splicing process can be performed in several ways, concatenating/removing different substrings, resulting in several similar, but different, mature mRNA sequences. As these sequences are similar, integrating them into a graph (a splice graph) provides a compact way of viewing them while simultaneously highlighting the differences as bifurcations in the graph.

Given such a splice graph we want to determine the gene from which the mRNA sequences making up the graph comes from. This can be done by aligning the sequences in the graph with a genomic sequence. However, we do not expect a perfect match. Firstly, the spliced sequences do not contain introns and will only be similar to substrings of the genomic string. Secondly, the process of extracting the mature mRNA from a cell is an error prone and cumbersome process. One approach is to identify small parts of a sequence, known as EST (Expressed Sequence Tag) sequences, one at a time before assembling to reconstruct a full sequence. The reconstructed mRNA sequence may then also contain errors, which are preserved when the sequence is inserted into the splice graph.

The problem is then to correctly align a splice graph onto a genomic sequence while at the same time keeping both time and space complexity at a minimum.

The problem to be solved is briefly described in the first chapter. The

second chapter focuses on the biological fundamentals, with particular emphasis on EST sequences. The following two chapters describes existing tools, algorithms and data structure which can be used to align sequences and graphs. In the next chapter the new algorithm is presented, with implementation details described in the succeeding chapter. The algorithm is then tested both for accuracy and speed in the following chapter before possible extensions are discussed. Finally, the last chapter sums up the test results along with some concluding remarks.

Chapter 2

Biological background

The following introduction barely scratches at the surface of the rich field of microbiology, and is only intended as background information for presenting the problem to be solved in this thesis. For a more comprehensive introduction, read an introductional textbook to the field[19].

2.1 DNA

DNA molecules carry the genetic information necessary for the organization and functioning of all living cells and control the inheritance of characteristics. At the molecular level DNA consists of a sequence of repeating substructures (a polymer) called nucleosides. Nucleotides are phosphate esters of nucleosides, a purine or a pyrimidine base linked glucosidically to ribose or deoxyribose. In DNA four different bases are used: adenine (A), cytosine (C), guanine (G) or thymine (T). These are linked together by 3',5'-phosphodiester bridges. The four bases form an alphabet of four letters from which information can be coded using a sequence of nucleotides. In the Watson-Crick double-helix model two complementary strands are wound together in a right-handed helix and held together by hydrogen bonds between complementary base pairs. The complementary base pairs are A bonded with T and C with G. An effect of this selective pairing is that one strand of DNA can be reconstructed using the other, helping to preserve the correct information. This is also exactly what is done in the replication process. The two strands part and from each strand a complimentary strand can be synthesized resulting in two identical DNA molecules.

2.2 The central dogma of molecular biology

Figure 2.1 outlines the general information flow within the cell, where reading a DNA strand as a recipe ultimately leads to the assembly of proteins.

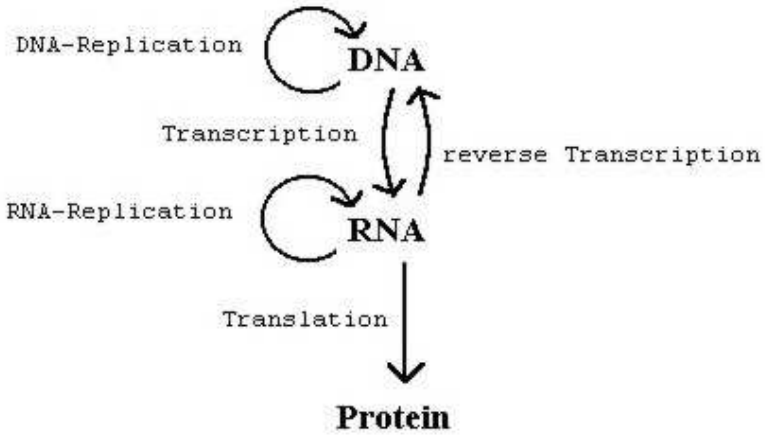

Figure 2.1: The central dogma of molecular biology, extended to include RNA replication and reverse transcription

There are three main processes in the central dogma. The first is *replication*, the copying of a parental DNA molecule to form an identical daughter DNA molecule. The second is *transcription*, where parts of the information of a DNA molecule is copied onto a RNA molecule. The third is *translation* where the information encoded in a RNA molecule is translated into a particular sequence of amino acids, a protein. The central dogma has later been extended to include RNA replication and reverse transcription.

The fundamentals of information flow within a cell is suggested by the central dogma. Nonetheless there are some additional details, relevant to this thesis, that warrant a closer inspection.

2.3 Gene expression

DNA molecules can be subdivided into genes, a subunit containing all the information necessary to produce a protein. The complete collection of genes and intergenic DNA (DNA between the genes) in a cell is known as the cellular genome. Most of the genome contains stretches of DNA that do not code for genes, so-called junk DNA which may have regulatory or other functions. Many genes can be further subdivided into coding sequences of base pairs (exons) and non-coding sequences of base pairs (introns). Genes usually have a number of relatively long introns interspersed with a limited number of shorter exons. It is important to note that even if all cells (with a

few exceptions) contain the whole genom of the organism only a small part is expressed(transcribed to mRNA) at any given time.

In eukaryotic organisms, such as humans, the DNA molecule is located within a nuclear envelope at the center of the cell. To use the information encoded in the double helix it is necessary to transport it out of the envelope to the parts of the cell able to produce proteins. This is done by creating a RNA copy of the DNA in a process called transcription. The strands intertwined in the double helix temporarily disconnects at the required area and the nucleotides of one strand are copied to synthesize a RNA molecule. This RNA molecule will leave the nuclear envelope and proceed through the cytoplasm to a ribosome. At the ribosome the mRNA molecule will be translated into a sequence of amino acids in a process appropriately known as translation. Such a sequence is known as a protein (polypeptide).

The process of transcription will create a complementary RNA copy, known as a *primary transcript*. In a post processing step introns are removed from the primary transcript in a process called *splicing*. Splicing involves cutting the RNA at certain places (splice sites), removing the introns, then reassembling the exons to a continuous sequence. The edited molecule is referred to as a *mature mRNA* molecule. The amount of DNA devoted to introns varies greatly from species to species and from gene to gene. The trend is that higher organisms, such as humans, have more, and longer, introns than other organisms. As an example, in most mammalian cells, only about 1% of the DNA sequence is copied into a functional RNA (mRNA). The remaining 99% includes both introns and junk DNA.

2.4 Alternative splicing

Although some eukaryotic mRNA transcripts produce a single mature mRNA, which in turn leads to the production of a single protein (or more precise, a single polypeptide), some produce more using *alternative splicing*. As the name implies, alternative splicing performs the post processing step differently to produce alternative forms of mature mRNA. Different mature mRNA can be produced by selecting novel acceptor/donor sites, resulting in alternative ends for introns and implicitly different exons. This includes taking the acceptor site of one intron and attach it to the donor site of another, thereby possibly removing several introns and exons. Other features resulting in alternative splice variants are when an intron is retained in the mature mRNA or if the transcription process itself has alternative start and stop positions (producing alternative 3' and/or 5' ends). Finally, primary transcripts from different genes can be spliced to each other in a form of splicing known as trans-splicing (the regular form of splicing is known as cis-splicing). The RNA may also be changed after splicing in a process called RNA editing. RNA editing modifies single nucleotides or inserts short

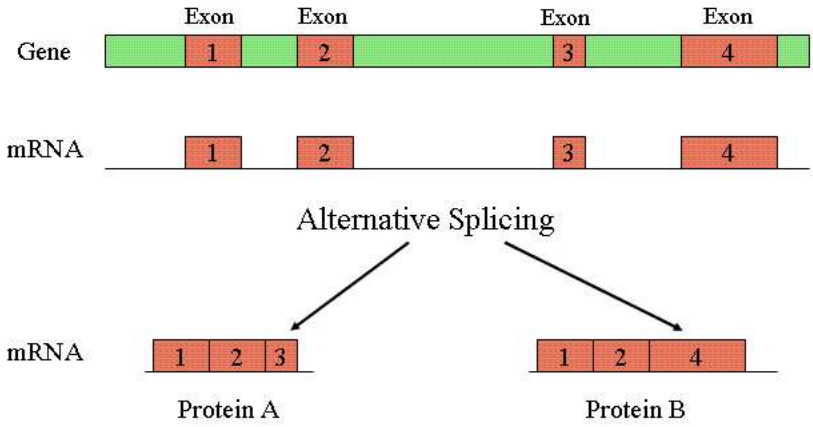

Figure 2.2: The figure demonstrates how one DNA molecule produce two different mRNA molecules due to alternative splicing.

sequences and is though to be highly regulated.

Alternative splicing is an important mechanism for modulating gene function. It effectively expands the 'vocabulary' of the gene allowing it to make various kinds of mRNA depending on the current needs. Alternative splicing has been implicated in many processes, including sex determination, apoptosis and acoustic tuning of the ear[15]. The number of human genes estimated to be alternatively spliced is a rapidly increasing number as scientist discover new variants. Estimates have been made claiming that up to 60% [15] of human genes are alternatively spliced. After the 'discovery' that the human genome contains only $\approx$ 32000 genes instead of the $\approx$ 100000 estimated earlier, much attention has been paid to alternative splicing to explain the discrepancies of the estimates.

2.5 Detecting alternative splicing

To differentiate between different splice variants the expressed sequences must be analyzed. The process of recovering the expressed sequences is not a trivial problem in itself. In practice a whole expressed sequence is not available as a single entity, rather it is reconstructed from small parts, known as EST (Expressed Sequence Tags) sequences, where each EST ideally represents a small part of the sequence. Sequencing EST sequences is relatively cheap, libraries of them are growing rapidly. EST sequences are clustered into sets corresponding to a single gene.

EST sequences are error prone. Many types of errors are introduced in

the sequencing process, a point that will be illustrated in the description of the EST sequencing pipeline [12]. Errors can occur in the laboratory while sequencing or can be natural occurring. Some features are not errors per se, but makes it harder to classify and cluster the EST sequences. One such feature is the presence of pseudogenes. Pseudogenes are genes that have been copied, and conserved as separate gene, during the evolutionary history of the genome. Most pseudogenes never get transcribed, but a minority are. Separating expressed sequences from pseudogenes from expressed sequences from the original gene is difficult.

2.5.1 The EST sequencing process

1. Reverse transcriptase reads RNA and, using it as a template, constructs a complementary DNA molecule.

2. Polymerase chain reaction (PCR) is used to amplify the cDNAs. The enzyme catalyzing the amplification is DNA polymerase. It needs a short starting sequence, known as a primer, to start the amplification process. Sequences that resemble each others reverse complements can anneal and function as primers for the polymerase. The resulting clone contains parts of both sequences and is known as a *chimera*.

3. The cDNA is inserted into a small circular DNA molecule known as a *plasmid*.

4. The plasmid is inserted into vector organisms, usually a bacterium, When the bacterium replicates, the plasmid is also replicated. Descendants of a single bacterium can be isolated and thus provide multiple copies of a single plasmid.

5. A primer is selected, usually from the plasmid sequence in the vicinity of the inserted DNA, as a starting point for DNA polymerase to generate nucleotide sequences of random lengths. Random termination is achieved by inserting a small amount of dideoxyNTP. When the polymerase uses dideoxyNTP instead of an ordinary base (A,C,G or T) the transcription terminates.

6. To separate the molecules they are diffused through a Polyacrylamide gel. The speed of diffusion varies with sequence length, and this allows the molecules to be separated.

7. The gel is scanned and the labeled molecules identified. The terminating ddNTP (from step 5) sequences are labeled differently. The output is a chromatogram containing curves representing the intensities of the labels. Each curve represents a single nucleotide.

To produce the full sequence of a clone the primers have to be designed in such a way that a new primer starts immediately after the last sequenced region. This process is more cumbersome than regular EST sequencing, but has been used to good effect[12].

2.5.2 Base calling

After the EST sequences are read they have to be analyzed. The first step is to interpret the chromatogram produced, a process known as base calling. The quality of the chromatogram is low both at the start and end ($500-800$ bases) which poses problems. Discerning the nucleotides from background noise can be a problem, and single nucleotide errors occur. Another source of error is known as 'stuttering', stretches of repeated nucleotides may cause the polymerase to slip back and reprocess a small part of the sequence. The result, if it happens in step 5, is small insertions and/or quality degradation. If the cDNA forms a secondary structure it will affect the speed through the Polyacrylamide gel and the resulting sequence may be distorted. After the sequences are generated, they are usually annotated with information such as clone-ID, read end (5' or 3') or sequence type (full length or EST).

To minimize errors, several quality control processes are done. The most unreliable parts, such as the start of the sequence, can be cut or masked from the sequence. Contamination from chimeric or vector sequences are also masked out by comparing directly with the vector sequence or to genomes of common vector organisms. In addition to errors introduced in the sequencing process, natural occurring features are also a source of problems. These often produce short repeated sequences that should be masked out. Also among these features is alternative splicing with features such as alternative exon composition and retained introns.

2.5.3 Clustering

Having cleaned up the sequences the next step is clustering them, ideally sequences from the same gene should end up in the same cluster. To decide which sequences are to be clustered, one can compare the sequences to the genomic sequences and cluster them based on the genomic position they map to [1]. This is computational expensive, since genomic sequences are very long, and the genomic sequence might not be available, instead sequences are often compared against each other. A common problem in clustering EST sequences is that genes from the same families, and homologous genes, often have similar sequences. Trying to separate them might erroneously put EST sequences from the same gene into different clusters. The number of sequences in each cluster will vary, and so will the gene positions they are

[1] Another approach is to cluster EST sequences according to originating primary transcript

mapped to. Parts of the gene with few or no sequences are said to be low coverage regions. In these cases the clustering process might not be able to merge clusters representing sections of a gene. Parts of the genome may belong to more than one gene (known as sense-antisense transcription), typically on opposite strands, and might make it difficult to separate sequences from such parts into the right clusters.

The final step is using the sequences of a cluster in an attempt to reconstruct the original mRNA sequence, or at least as close to the original sequence as possible. An alternative approach is to construct a graph from the sequences to elucidate different splice variants. The common way to assemble the sequences is by finding a Hamiltonian path through an overlap graph, a graph where each sequence is a node and there exists a directed edge between two nodes if the sequences overlap. That is, if a prefix of some sequence S_2 matches some suffix of a sequence S_1 there will be an edge from S_1 to S_2. Errors in the EST sequences will affect the overlap graph, pruning it will usually produce better results. Algorithms for constructing a splice graph will be described in a later chapter.

2.5.4 Finding alternative splice variants

Finding alternative splice variants seems deceptively easy in theory; by comparing expressed sequences (mRNA or cDNA) from a given gene insertions/deletions that indicate alternative exon usage can be identified. In practice the issue is more complicated. There are two basic approaches with respect to which sequences to compare. One can compare the transcripts with each other, identifying divergent exon patterns. A second approach is to try to map transcripts directly onto a genomic sequence, where insertions/deletions are indicative of alternative splicing. Neither is without flaws. Comparing EST sequences with each other suffer from the errors associated with them, for instance might EST sequences from different genes be compared because they have erroneously been put in the same cluster due to, for instance, the sequences being derived from paralogous genes. Comparing directly to the genome will remove many such problems but also introduce new ones. Firstly, comparing a sequence against the whole genome is computationally expensive. Secondly, by mapping a short (possible erroneous) sequence to a larger sequence will result in many false negatives indicating alternative splicing[15][1]. A third option is to combine both methods in a hybrid approach[1].

Reconstructed mRNA sequences, also known as EST contigs, are not only useful in comparing alternative splice variants. For organisms where the gene structure is not yet determined expressed sequences can be mapped onto the genome to locate genes. The drawback with this approach is that genes tend to be very unevenly expressed to the extent that some are only expressed in some tissue or states. Even if the expressed sequences are put

in normalized libraries, determining genes solely based on EST sequences is hard[12].

Chapter 3

String alignments

When trying to determine the relation between two strings, a natural comparison criterium is how similar the strings are. We may not expect them to be equal, so methods for exact string matching are not useful. This chapter describes how to do inexact matching, where some measure of similarity/dissimilarity is computed to describe the relation of the strings. However, if one is to discuss strings it is necessary to first define exactly what a string is in this context, using the following definitions from Gusfield[6].

3.1 Definitions

Definition 3.1 *A string S is an ordered list of characters written contiguously from left to right. For any string S,$S[i \ldots j]$ is the contiguous substring of S that starts at position i and ends at position j of S. In particular $S[1 \ldots i]$ is the prefix of string S that ends at position i, and $S[i \ldots |S|]$ is the suffix of string S that begins at position i, where $|S|$ denotes the number of characters in string S.*

Definition 3.2 *$S[i \ldots j]$ is the empty string if $i > j$.*

Definition 3.3 *A proper prefix, suffix, or substring of S is, respectively, a prefix, suffix or substring that is not the entire string S, nor the empty string.*

Definition 3.4 *For any string S, $S[i]$ denotes the i'th character of S.*

3.2 Alignments and edit distance

Frequently one wishes to measure the difference between two strings, for example in comparing biological sequences. There are several ways to formalize the notion of string difference. One such way is known as edit distance. Given two strings S_1 and S_2, the edit distance is the minimum of

edit operations necessary to transform one string to the other. The allowed operations are (1) replace one symbol with another, (2) delete k consecutive symbols and (3) insert k consecutive symbols. A sequence of such operations transforming S_1 to S_2 (or vice versa) is known as an edit transcript. The edit transcripts with the fewest operations are known as optimal transcripts. Note that if symbol S_1[i] is equal to symbol S_2[i] no operation is necessary, hence it does not influence the edit transcript in any way.

Definition 3.5 *The edit distance between two strings is defined as the minimum number of edit operations, insertions, deletions and substitutions, needed to transform the first string into the second. For emphasis note that matches are not counted[6].*

An alternate way of representing this problem is by maximizing the score of the alignment of the two strings rather than minimizing the edit operations required. An alignment of two strings is a scheme of writing one string on top of another to illustrate the relationship between strings, or parts of them. Given two strings S_1 and S_2, an alignment is obtained by inserting dashes into S_1 and S_2 so that the characters of the resulting strings can be put in one-to-one correspondence to each other. As long as the order of symbols within the strings are preserved any symbol in S_1 can be aligned to any symbol in S_2. Though in general only the alignment(s) achieving the highest score are interesting. Finding the optimal alignment where each string in its entirety is involved is known as a global alignment.

Definition 3.6 *A (global) alignment of two strings S_1 and S_2 is obtained by first inserting chosen spaces (or dashes), either into or at the ends of S_1 and S_2 and the placing the two resulting strings one above the other so that every character in either string is opposite a unique character or unique space in the other string [6].*

Aligning two equal symbols will typically yield a positive score, while aligning unequal symbols or inserting dashes yield a negative score. By summing these values along an alignment a total score is obtained. There can be, and usually are, more than one alignment having the maximal score. Which, if any, are the most useful is dependent upon the specific problem for which the alignment representation is used. Tuning the values of the scores will produce different optimal alignments.

Definition 3.7 *Let Σ be the alphabet used for strings S_1 and S_2, and let Σ' be Σ with added character '_' denoting a space. Then for any two characters x, y in Σ', score(x,y) denotes the value obtained by aligning character x against character y [6].*

Definition 3.8 *For a given alignment A of S_1 and S_2, let S_1' and S_2' denote the string after the chosen insertion of spaces, and let L denote the (equal) length of the two strings S_1' and S_2' in A. The value of the alignment A is defined as $\sum_{i=1}^{l} score(S_1'[i]S_2'[j])$ [6].*

An alignment usually uses a similarity measure, a higher score indicates a higher degree of similarity between the sequences, whereas edit distance uses a distance measure. The more edit operations required to transform one sequence to the other, the higher the dissimilarity. Given a function to compute the score; a distance function will define negative values for mismatches or spaces and then aim at minimizing this distance. A similarity function will give high values to matches and low values to spaces and then maximize the resulting score. It is, however, perfectly possible to formulate an alignment representation using a distance measure. In the similarity framework one can easily distinguish among the different possible mismatches and also among different kinds of matches. For this purpose a scoring matrix can be defined where different symbols aligned yield different scores depending on the value of the symbols in question. For instance, when comparing strings of amino acids aligning two Tryphtophanes may be more important than aligning two Alanines. Assigning the aligning of two Thryphtophanes a higher score makes it more likely to be included in the optimal alignment.

Definition 3.9 *Given a pairwise scoring matrix over the alphabet Σ, the similarity of two strings S_1 and S_2 is defined as the value of the alignment A of S_1 and S_2 that maximizes total alignment value. This is also known as the optimal alignment of S_1 and S_2 [6].*

```
EX__ON
I NTRON
```

Figure 3.1: Using a score of 1 for a match, -1 for a mismatch, and -2 as a penalty for each space in the alignment of 'intron' and 'exon' will (among others) produce the alignment displayed. The score for this alignment is: $-1-1-2-2+1+1=-4$

From a modeling viewpoint alignment and edit transcript differ. Edit transcripts models specific events, such as mutations, while an alignment only model the relationship between strings. Edit transcripts specify the process of transforming one string to another whereas an alignment only describe the end product. As such the alignment description is more neutral and it is what will be used throughout the remaining part of this text.

3.3 Sequence alignment

As mentioned in the previous chapter biological sequences, such as DNA, RNA or proteins, can be looked upon as a string of characters from a limited alphabet. Each character is an abstraction from a biological term, nucleotides or amino acids, that are arranged linearly in fixed position in larger molecules. Using this symbolic notation to represent a sequence it is clearly evident that a (biological) sequence equals a (as defined previously) string. Thus DNA sequences can be considered strings of characters from the alphabet T,G,C and A. Such strings can be aligned to determine the similarity of the DNA sequences using a similarity score.

Having determined that two sequences can be aligned, the question is now why an alignment would be useful and what information an alignment, in particular spaces, provides. The obvious reason for aligning biological sequences is to see how similar/different they are from each other, how these similarities are explained varies depending on what type of sequences that is analyzed.

When aligning two protein sequences the operations of aligning characters, inserting characters or deleting characters can be viewed as 'evolutionary' operators. As such, two different characters aligned (a mismatch) can be viewed as a mutation in one of the sequences changing one character to another. Similarly, insertion and deletions can be treated as mutational events. Assumed that the mutation rate is a constant, the total similarity of the protein sequences is a measurement of the time since the two sequences diverged from a common ancestor.

When comparing expressed sequences to DNA the differences between the strings are explained in another way. Apart from sequencing errors, the spaces in the alignment can be explained from the fact that introns are spliced out from the expressed sequences. This will result in large regions of spaces broken off by regions of characters (exons). When aligning an expressed sequence to the DNA sequence it is derived from, the expressed sequence should match regions(exons) in the DNA very well.

An alignment will be sensitive to the scoring scheme used, particulary in how continuous regions of spaces, known as gaps, are treated. Gaps will be discussed more thoroughly later.

Various approaches, besides alignments, have been applied to problems such as gene recognition. However, methods such as using statistical codon usage[21] have only had limited success[4]. For determining exon structure, algorithms combining alignments with combinatorics[4][14] or heuristic tools[11], such as BLAST, do exist. They may save computational time while still doing correct predictions.

As biological sequences are treated as strings, the terms string and sequence will be used interchangeably thorough the remainder of the text.

3.4 Algorithm for finding optimal alignment(s)

A brute force algorithm for finding the optimal alignment is simply to create all possible alignments and determine, in a case-by case fashion, which alignment has the highest score. Unfortunately, there will be an exponential amount of possible alignments giving the algorithm a time complexity of $O(2^n)$.

Luckily, an optimal alignment of two sequences can be found without explicitly enumerating all possible alignments. The approach builds on the classic algorithm by Needleman & Wunsch. It uses dynamic programming to find optimal alignments for a given scoring scheme. The main idea is that results found early in the computation can be reused in later calculations, thereby reducing the time complexity to $O(nm)$. The algorithm computes and stores the value of each cell in a dynamic programming table using recurrences, whose values are dependent on earlier calculations or explicitly formulated base cases. The original algorithm did not impose any restrictions on the penalty assigned to a gap of a certain length. A gap is, informally, one or more consecutive spaces. For reasons of computational speed this was later rectified to assigning a cost function linear in the number of deleted (inserted) residues.

3.4.1 The recurrence

Definition 3.10 *$V(i,j)$ is defined as the value of the optimal alignment of prefixes $S_1[1\ldots i]$ and $S_2[1\ldots j]$[6].*

Given a sequence $S_1[1\ldots i]$ and a sequence $S_2[1\ldots j]$, the algorithm defines an $(i+1)\cdot(j+1)$ matrix. Each cell (i,j) in the grid determines a position both in the first and second sequence. As the algorithm progresses the score $V(i,j)$ will be stored in the corresponding cell (i,j). The speedup over the brute force method is achieved by not completely recalculating $V(i,j)$ for each cell but rather let the value of $V(i,j)$ depend on previously calculated entries stored in the matrix. The key is to formulate a set of recurrences computing $V(i,j)$ using the recursive relation $V(i,j)$ has with $V(i-1,j-1)$, $V(i,j-1)$ and $V(i,j-1)$. These entries correspond to having calculated the optimal alignment of the prefixes $S_1[1\ldots i-1]$ and $S_2[1\ldots j-1]$, $S_1[1\ldots i-1]$ and $S_2[1\ldots j]$ and finally $S_1[1\ldots i]$ and $S_2[1\ldots j-1]$. Why is only these cases considered? Consider an optimal alignment of some prefix $S_1[1\ldots i]$ and some prefix $S_2[1\ldots j]$. The last character in the alignment must either be a character of $S_1[i]$ aligned with $S_2[j]$, a character $S_1[i]$ aligned with a gap in S_2 or a gap in S_1 aligned with $S_2[j]$. It follows that the alignment, before adding the last character, must be the optimal alignment of that prefix: $V(i-1,j-1)$, $V(i-1,j)$ and $V(j,i-1)$ respectively. To compute the optimal alignment of prefixes $S_1[1\ldots i]$ and $S_2[1\ldots j]$ there now exist three possibilities.

Definition 3.11 *The optimal alignment of $S_1[1 \ldots i]$ and $S_2[1 \ldots j]$ is produced from one of the following cases.*
V(i-1,j-1) + $S_1[i]$ aligned with $S_2[i]$
V(i,j-1) + inserting a space after $S_1[i]$
V(i-1,j) + inserting a space after $S_2[j]$

All which will produce an alignment of the prefix $S_1[1 \ldots i]$ and $S_2[1 \ldots j]$. Being the score of the optimal alignment, $V(i, j)$ simply equals the score of the best case. Definition 3.11 can then be reformulated into a simple recurrence.

Recurrence 3.1

$$V(i,j) \leftarrow \max \begin{cases} V(i-1,j) & +gapcost \\ V(i,j-1) & +gapcost \\ V(i-1,j-1) & +score(i,j) \end{cases}$$

where gapcost is the penalty of inserting a gap, and $score(i, j)$ is a function calculating the score of aligning $S_1[i]$ and $S_2[j]$.

$$score(i,j) \leftarrow \max \begin{cases} match & if S_1[i] = S_2[j] \\ mismatch & otherwise \end{cases}$$

As in any other recursion, base cases must be defined.

$V(0,i) = i \cdot gap$
$V(0,j) = j \cdot gap$

This is clearly correct as the score of aligning the prefix $S_1[1 \ldots i]$ with a gap of length i starting before S_2 must be proportional with the length of the gap (given linear gap penalty). The same principle applies when aligning $S_2[1 \ldots j]$ with a gap of length j.

Note that the base cases are calculated and inserted into the first row and the first column of the matrix. This extra column and row is why the matrix has a size of $(i + 1) \cdot (j + 1)$ rather than just the expected $i \cdot j$, this allows for inserting spaces before the start of the sequences.

3.4.2 The steps of the algorithm

Having established the recurrence, the different steps of the base algorithm can be explained.

1. Initialization.
2. Matrix fill (scoring).
3. Traceback (alignment).

The two first steps have already been discussed. Initialization is simply declaring the matrix and filling in the values of the base cases. Matrix fill is simply applying recurrence 3.1 to the remaining cells in the matrix. Having filled the matrix, the score of the optimal (global) alignment of $S_1[1 \dots i]$ and $S_2[1 \dots j]$ is $V((i+1),(j+1))$.

Finding the optimal score is sufficient for some application but usually it is the optimal alignment itself that is interesting. The last step locates the alignment(s) corresponding to the optimal score. By also storing pointers for each cell(i,j) as the table is computed one can easily follow a path through the matrix corresponding to an optimal alignment. Traceback takes the current cell and looks to the neighboring cells that could be direct predecessors. If the value $V(i,j)$ equals $V(i-1,i-1) + score(i,j)$ a pointer to cell $(i-1,j-1)$ is stored, similarly pointers to cell $(i-1,j)$ and cell $(i,j-1)$ can be stored. It is possible for a given cell, excluding the base cases, to have all three pointers. During the initialization pointers are set for the base cases, for $V(i,0)$ a pointer is set to $V(i-1,0)$ for all $i > 0$, for $V(0,j)$ a pointer is set to $V(0,j-1)$ for all $j > 0$.

To find an optimal (global) alignment one follows the traceback pointers from the bottom right corner to the upper left corner. As one starts tracing the path at the end of the alignment one builds up the alignment backwards starting with the last character in the alignment and ending with the first one.

An also perfectly valid solution is to compute the traceback in a separate step after the matrix fill step. Given a scoring scheme and the value of the optimal alignment $V(i,j)$, the cell which precedes (i,j) in the alignment can be calculated using the same approach as for setting pointers. The complete alignment can then be determined by applying the same principle recursively until the upper left corner is reached. Calculating in this manner has the advantage of not using additional memory to store the pointers. It is also fully compatible with more advanced variants of dynamic programming procedures to compute optimal alignments.

3.4.3 The complete algorithm step by step

Finding the optimal alignment of $S_1[1 \dots i]$ and $S_2[1 \dots j]$ where S_1 equals 'CGTACT' and S_2 equals 'CACCCG'. The scoring scheme is 1 for a match, -1 for a mismatch and $((-1) \cdot n)$ for a gap of length n. The three steps are illustrated in the following figures.

1. Initialization Fig. 3.2.

2. Matrix fill (scoring) Fig. 3.3 and Fig. 3.4.

3. Traceback (alignment) Fig. 3.5.

		C	G	T	A	C	T
	0	-1	-2	-3	-4	-5	-6
C	-1						
A	-2						
C	-3						
C	-4						
C	-5						
G	-6						

Figure 3.2: The first step, initializing the base cases of the matrix

		C	G	T	A	C	T
	0	-1	-2	-3	-4	-5	-6
C	-1	1					
A	-2						
C	-3						
C	-4						
C	-5						
G	-6						

Figure 3.3: The second step, Calculating V(i,j) for all cells i,j. V(1,1) is shown in red. The value of V(1,1) is the maximum of : V(0,1) +gap penalty, V(1,0) + gap penalty and V(0,0) + the score of aligning character 'C' with character 'C'. (shown in red,yellow, blue and green (respectively))

The time complexity of the algorithm is proportional with the product of the length of the strings.

Theorem 3.1 *Given string S_1 of length m and S_2 of length n, the optimal alignment of S_1 and S_2 can be found using $O(nm)$ time.*

		C	G	T	A	C	T
	0	-1	-2	-3	-4	-5	-6
C	-1	1	0	-1	-2	-3	-4
A	-2	0	0	-1	0	-1	-2
C	-3	-1	-1	-1	-1	-1	0
C	-4	-2	-2	-2	-2	0	0
C	-5	-3	-3	-3	-3	-1	-1
G	-6	-4	-2	-3	-4	-2	-2

Figure 3.4: The second step (continued), filling the matrix. The completely filled matrix. The optimal (global) alignment is in the bottom right corner (shown in red)

		C	G	T	A	C	T
		←	←	←	←	←	←
C	↑	↖	↖←	←	←	↖←	←
A	↑	↖↑	↖	↖←	↖	←	↖←
C	↑	↖↑	↖↑	↖	↑	↖	↖←
C	↑	↖↑	↖↑	↖↑	↖↑	↖↑	↖
C	↑	↖↑	↖↑	↖↑	↖↑	↖↑	↖↑
G	↑	↑	↖	←	↖↑	↑	↖↑

Figure 3.5: The last step, finding the alignment(s) corresponding to the optimal score. A complete table of pointers represented as arrows pointing to another cell. A cell on the optimal path having more than one pointer indicates that there are more than one solution

Proof: Consider the three steps of the algorithm. When initializing the table $n + m$ cells are filled corresponding to inserting gaps of length $1 \ldots n$ and $1 \ldots m$. Given a linear gap penalty one does a constant amount of work in each cell, which equals a total time of $O(n + m)$ for the first

step. When filling the $n \cdot m$ matrix one also does a constant amount of work per cell. In any given cell only the value of 3 other cells are considered, a constant number. Filling the matrix thus only takes $3 \cdot (nm)$ time, keeping the asymptotical running time $O(nm)$. Finally, the worst case running time of the traceback step is tracing the longest alignment possible, as this will necessitate visiting the most cells. The longest alignment possible, given that it is not possible to align gaps to gaps, is an alignment where the aligned sequences do not overlap at all, see figure 3.6. In such an alignment every character in S_1 and S_2 is aligned with a gap. The total number of cells visited are then $m+n$, a running time of $O(n+m)$ since one still only perform a constant number of operations in each cell.

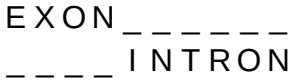

Figure 3.6: The longest alignment possible when aligning INTRON and EXON

The $O(mn)$ of the fill matrix step dominates the $O(n+m)$ running time of the two other steps giving the whole algorithm a total running time of $O(mn)$.

Any algorithm that uses dynamic programming to calculate sequence alignment will always have at least $O(nm)$ worst case running time, as all cells must be calculated. Removing a cell from the calculation will result that any alignment 'passing through' this cell will not be considered. Even if some cells seems hardly ever useful (such as the bottom left one) there is no way to beforehand exclude it, with absolute surety, from the calculation without possibly excluding the best alignment from being computed in the algorithm.

Theorem 3.2 *Given string S_1 of length m and S_2 of length n, the optimal alignment of S_1 and S_2 can be found using $O(nm)$ space.*

Proof: The relative high memory requirements of the algorithm is due to having to store the complete matrix with the $V(i,j)$ values for all i,j. Since the amount of space stored in each cell is a constant, a number and possibly up to three pointers, the total amount of space used is $O(nm)$.

Note

This algorithm computes a global alignment of two strings. The variants of this algorithm being presented in this chapter will use the same recurrence, base cases, and way of penalizing spaces as this algorithm unless otherwise

specified. The algorithm presented in this section will be referred to as the 'base algorithm'.

3.5 Gap penalties

Having briefly explained linear gap penalty in the previous section more advanced, possibly more biologically correct, gap penalties can be examined. To do so one has to have a more precise definition of a gap.

Definition 3.12 *A gap is any maximal, consecutive run of spaces in a single string S of a given alignment. A gap may begin before the start of S, in which case it is bordered on the right by the first character of S, or it may begin after the end of S, in which case it is bordered on the left by the last character of S.*

A gap penalty is simply the cost of inserting a single such gap. Usually the penalty of a gap is independent from the cost of any other gaps in the alignment, referred to as local gaps. By including a term that reflects the gaps in the alignment the overall distribution of spaces can be influenced, implicitly influencing the overall shape of the alignment.

The most general gap penalty is scoring a gap with an arbitrary gap cost. The main drawback of using an arbitrary gap cost is the added time complexity. This makes other schemes, where the gap cost is dependant on the length of the gap, a more viable option.

3.5.1 Affine gap penalties

In the examples so far a linear gap penalty has been used; if the penalty for inserting a single space into a string is x and the length of the gap is n, the gap penalty is the linear function $x \cdot n$. However, other functions for gap cost is possible. How a gap penalty is chosen depends on how the differences between the aligned string are explained. From a biological standpoint deletions/insertions can in some instances be modeled as mutations. A single mutation might give rise to multiple deletions/insertions as opposed to just one. A mutational event might even give rise to a variable size gap, where gap length can be relatively random within a set of constraints. With a linear gap penalty a mutation resulting in a gap of length two would be penalized the same amount as two gaps of length one from two separate mutations. Such a gap penalty would have to comply with the following condition for gaps of length r and s.

$$gapcost(r + s) \leq gapcost(r) + gapcost(s)$$

A gap penalty fulfilling this condition is known as a concave gap penalty. Note that linear gap penalty is a concave gap penalty. To get lower cost for

longer gaps one can use an affine gap penalty. The simplest affine model is to have, in addition to a cost for each space in the gap, a separate cost for starting the gap.

$$gapcost = gapopen + (gapextend \cdot gaplength)$$

Alignments using an affine gap penalty tend to have fewer, longer gaps than those who use linear gap penalty. More advanced affine gap function can include terms where the cost of extending a gap explicitly decreases as the length increases, for instance by having the penalty for inserting space number n equal to $(gapextend \cdot \log n)$[17].

3.5.2 Free end gaps

Another variant is to have free end gaps in a global alignment. Given an alignment of string S_1 and string S_2. An end gap is a gap inserted before the first character of S_1 or S_2, or a gap inserted after the last character of S_1 or S_2. The reason for having no cost for these gaps are apparent when aligning a short sequence, such as a mature mRNA sequence, with a longer one, such as a long DNA sequence. The reason for aligning the two sequences is that they are expected to be similar in some way, the smaller sequence matches some region of the larger. Having no cost for end gaps makes it more likely that this is achieved instead of having the smaller string broken up into pieces matching smaller regions along the larger string. Having no cost for end gaps does not affect the general gap cost in any way, it can be linear, affine or whatever seems appropriate for the application. Since this variant is somewhat in-between global - and local alignment it is often referred to as semi-global alignment.

Having described different gap schemes, we now turn to how the recurrence of the base algorithm must be changed, in each case, to incorporate these schemes.

3.6 Variants of the base algorithm

Several modifications to the base algorithm can be made, and are done in practice, to 'tune' the alignment to suite our current needs.

3.6.1 Local Alignment

In many cases sequences might not be very similar in their entirety but have regions which are highly similar. This problem might surface when comparing DNA strings to each other or comparing different proteins. The problem of finding such regions are formally defined in definition 3.13.

Definition 3.13 (Local alignment problem) *Given two strings S_1 and S_2, find substrings A and B of S_1 and S_2, respectively, whose similarity (optimal global alignment value) is maximum over all pairs of substrings from S_1 and S_2. A and B may be empty strings[6].*

These regions might be correctly aligned to each other using the base algorithm but, as the goal of that algorithm is to maximize the *global* alignment score, it is no certainty. The positive score of the similar substrings might drown in the penalty of the dissimilar substrings.

Given the strings S_1 and S_2 of length n and m, respectively, there are $O(n^2m^2)$ possible pairs of substrings. Putting all pairs into the base algorithm and finding the highest scoring alignment would then consume $(n^2m^2) \cdot (nm)$ time $= O(n^3m^3)$. However, the problem can be solved much faster, in $O(mn)$, time by inserting another term in the recurrence of the base algorithm. This time bound was obtained by using an algorithm by Smith & Waterman. The new algorithm is almost identical to the base algorithm except that a new term, zero, is inserted into the recurrence. Since the worst score possible, if the substrings are allowed to have length zero, is zero, any contribution from negative prefixes can be removed by not allowing negative cell values in the matrix. Negative values are replaced with zero, hence the new zero term in recurrence 3.2. Note that this is the recurrence when the gap penalty is linear.

Recurrence 3.2

$$V(i,j) \leftarrow \max \begin{cases} V(i-1,j) & +gapcost \\ V(i,j-1) & +gapcost \\ V(i-1,j-1) & +score(i,j) \\ 0 & \end{cases}$$

The base cases are also changed, before they would be negative to represent insertion of gaps, now they will be zero. This is due to the fact that any gap inserted into the beginning of either sequence is a potential prefix to a substring, one with a negative value. Viewing free end gaps in similar manner makes it clear that a prefix starting before the start of S_1 or S_2 is a negative number, the values in the corresponding cells are therefore set to zero.

As for the base algorithm, the scoring scheme will influence which local alignment is optimal. Scoring a match as one, and mismatches/gaps as zero, the optimal local alignment is the longest common subsequence. If mismatches/gaps are given a large negative score, and each match still a score of one, the optimal local alignment will equal the longest substring. Usually the interesting cases are somewhat in between these and an appropriate scoring scheme must be defined.

3.6.2 Free end gaps

Modifying any scoring scheme to accommodate free end gaps is a relatively simple procedure. Firstly, as no gaps preceding the first character in either sequence should be penalized, the base cases in the recurrence should all be initialized to zero.

$V(i,0) = 0$
$V(0,j) = 0$

Secondly, as no gap succeeding the last character of either sequences should be penalized we cannot content ourselves to look for the optimal alignment score in cell (n,m). The cell (x,y) containing the optimal alignment score will be the highest scoring cell in the last column/row, since any gap from cell (x,y) to cell (m,n) would not be penalized. These modifications will work with the other gap schemes mentioned here.

3.6.3 Arbitrary gap penalties

We continue with the gap scheme that seems most demanding, implementing arbitrary gap weights. As per the base algorithm we compare prefixes $S_1[1 \ldots i]$ and $S_2[1 \ldots j]$. Any alignment of such prefixes falls into one of the three following categories.

1. Character $S_1[i]$ is aligned to a character strictly to the left of $S_2[j]$, the alignment ends with a gap in S_1.
2. Character $S_1[i]$ is aligned to a character strictly to the right of $S_2[j]$, the alignment ends with a gap in S_2.
3. Character $S_1[i]$ is aligned opposite to $S_2[j]$, this include both the case that $S_1[i]$ equals $S_2[j]$ and the case where they are different.

Definition 3.14 *Define $E(i,j)$ as the maximum value of any alignment of type 1; define $F(i,j)$ as the maximum value of any alignment of type 2; define $G(i,j)$ as the maximum value of any type of alignment of type 3; and finally define $V(i,j)$ as the maximum value of the three terms $E(i,j)$, $F(i,j)$ and $G(i,j)$.*

A set of recurrences establishing $V(i,j)$, and which takes into account the three cases above, can now be formulated.

Recurrence 3.3

$$V(i,j) \leftarrow \max \begin{cases} E(i,j) \\ F(i,j) \\ G(i,j) \end{cases}$$

$G(i,j) = V(i-1,j-1) + score(S_1[i], S_2[j])$
$E(i,j) = \max_{0 \le k \le j-1}(V(i,k) - gapcost(j-k))$
$F(i,j) = \max_{0 \le l \le i-1}(V(l,j) - gapcost(i-l))$
The base cases are, assuming that end gaps are not treated differently. $V(i,0) = -gapcost(i)$
$V(0,j) = -gapcost(j)$
$E(i,0) = -gapcost(i)$
$F(0,j) = -gapcost(j)$

If end gaps are free the base cases are.

$V(i,0) = 0$
$V(0,j) = 0$

Theorem 3.3 *Assuming that* $|S_1| = n$ *and* $|S_2| = m$*, the recurrences can be evaluated in* $O(nm^2 + mn^2)$*[6].*

Proof: As usual we have a $(n+1) \cdot (m+1)$ matrix where we fill in one $V(i,j)$ value at the time. We need to examine one cell to evaluate $G(i,j)$, j cells of row i to evaluate $E(i,j)$, and i cells of column j to evaluate $F(i,j)$. Therefore, for any fixed row, $\frac{m \cdot (m+1)}{2} = \Theta(m^2)$ cells are examined to evaluate all E values in that row, and for any fixed column, $\frac{n \cdot (n+1)}{2} = \Theta(n^2)$ cells are examined to to evaluate all the F values of that column, Since we have n rows and m columns the total running time is $O(nm^2 + mn^2)$[1][6].

3.6.4 Affine gap penalties

When using affine gap penalties instead of arbitrary gap weights we keep the terms E, F and G but modify the recurrences slightly.

Recurrence 3.4

$$V(i,j) \leftarrow \max \begin{cases} E(i,j) \\ F(i,j) \\ G(i,j) \end{cases}$$

$G(i,j) = V(i-1,j-1) + score(S_1[i], S_2[j])$
$E(i,j) = \max(E(i,j-1), V(i,j-1) - gapopen) - gapextend)$
$F(i,j) = \max(F(i-1,j), V(i-1,j) - gapopen) - gapextend$

The base cases are, assuming that end gaps are not treated differently.

$V(i,0) = -gapopen - (gapextend \cdot i)$
$V(0,j) = -gapopen - (gapextend \cdot j)$
$E(i,0) = -gapopen - (gapextend \cdot i)$
$F(0,j) = -gapopen - (gapextend \cdot j)$

[1]Note that this proof from Gusfield[6] uses columns instead of rows, and vice versa, compared to the rest of the algorithms in this thesis

If end gaps are free the base cases are.

$V(i,0) = 0$
$V(0,j) = 0$

The changes from arbitrary gap weights are in how we treat the $E(i,j)$ and $F(i,j)$ terms. Except for the opening cost, the cost of a gap of length $n+1$ equals the one for a gap of length $n+gapextend$. This is in contrast in arbitrary gap penalties were there is no predictable relationship between a cost of a gap of length n and a gap of length $n+1$. The implications of this is that, for each cell, no more than **one** previous value of E an F and the cost of opening a gap to compute $E(i,j)$ and $F(i,j)$ have to be examined. This is a constant number of comparisons unlike the number of comparisons for an arbitrary gap which were proportional to column/row length.

Theorem 3.4 *The optimal alignment with affine gap weights can be computed in $O(nm)$ time, the same as for optimal alignment without a gap term (base algorithm).*

Proof: We can compute $V(i,j)$, $G(i,j)$, $E(i,j)$ and $F(i,j)$ in constant time. As we have an $(n+1)\cdot(m+1)$ matrix this translates into a $O(nm)$ running time.

Refining the recurrence

The G term in the recurrences, both for arbitrary and affine gap penalties, is unnecessary. The G term only depends on previously calculated values of V, not G. The V term can then be computed on the basis of the E,F and a previously computed V term. Removing G does not effect execution speed, but avoids storing the values of G in a table saving space. Recurrence 3.5 is equivalent to recurrence 3.4.

Recurrence 3.5

$$V(i,j) \leftarrow \max \begin{cases} E(i,j) \\ F(i,j) \\ V(i-1,j-1) + score(S_1[i], S_2[j])) \end{cases}$$

$E(i,j) = \max(E(i,j-1), V(i,j-1) - gapopen) - gapextend)$
$F(i,j) = \max(F(i-1,j), V(i-1,j) - gapopen) - gapextend$

3.7 Further extension to incorporate introns

As explained earlier, introns are parts of the sequence removed from the pre mRNA in a post transcriptional process. When aligning a DNA sequence to

a mature mRNA, derived from the DNA sequence, one would expect that long substrings in the DNA would be aligned with gaps. However, there are compelling reasons to treat introns differently from ordinary gaps.

Introns can be very long, particulary in higher eukaryotes, so an affine gap penalty with a very low cost, even nothing, to extend a gap should be used. Otherwise we risk spurious alignment between parts of an intron and parts of the mRNA as several small gaps might be preferred instead of a large, true, one. There is no biological reason that an ordinary (non intron) gap should be scored in the exactly same way, another reason to treat introns differently. Secondly introns have certain biological signals, known as splice sites, to mark their end and beginning. In the vast majority (99%) of cases an intron starts with a 'GT' and ends with a 'AG'. This needs to be reflected in the recurrences. Thirdly since introns are spliced out of the pre mRNA to form a mature mRNA sequence, introns would be expected to appear only in the sequences which has them, the DNA sequences, whereas ordinary gaps can appear in both sequences.

3.7.1 Incorporating an intron term into the recurrence

These concerns are taken into consideration in the algorithm designed by Mott and used in his program EST_genome[16]. The algorithm is a variant of the Smith-Waterman (local alignment) algorithm described earlier. The interesting addition is an explicit, separate cost for scoring introns. Let $B(i)$ be the score of the best local alignment, found so far, ending in position i in the spliced sequence and $C(i)$ be the genome coordinate to which $B(i)$ refers. If an intron starts with 'GT' and ends with 'AG' this is referred to as a acceptor-donor pair. Recurrence 3.1 is then replaced with the following recursions.

Recurrence 3.6

$$V(i,j) \leftarrow \max \begin{cases} V(i-1,j) & +gapcost \\ V(i,j-1) & +gapcost \\ V(i-1,j-1) & +score(i,j) \\ B & \\ 0 & \end{cases}$$

Given a spliced sequence S and a genomic sequence G the $score(i,j)$ function and the B term is defined as follows.

$$B \leftarrow \max \begin{cases} B(i) - splicecost & if C(i), j \text{ are a donor-acceptor pair} \\ B(i) - introncost & otherwise \end{cases}$$

$$(B(i), C(i)) \leftarrow \max \begin{cases} (V(i,j), j) & if V(i,j) > B(i) \\ (B(i), C(i)) & otherwise \end{cases}$$

$$score(i,j) \leftarrow \max \begin{cases} match & if S[i] = G[j] \\ mismatch & otherwise \end{cases}$$

The first equation is identical to the one for local alignment except for the new term B used to identify introns. The second equation differentiates the cost of inserting an intron depending on whether it has a proper acceptor/donor pair. Even if the best local alignment ending in i does not match the splice consensus perfectly, if the difference $intron cost - splice cost$ exceeds the extra cost incurred to make it so, the alignment will respect the correct boundaries. The third equation simply updates $B(i)$ and $C(i)$ if a higher scoring local alignment ending in i in the spliced sequence is found.

The alignment can still be computed in $O(nm)$ time as only a constant more operations are done per cell in the dynamic programming matrix. As default parameters Mott[16] suggest setting $splicecost = 20$ and $intron cost = 40$. The value can of course be changed but one problem remains, exons shorter then *splicecost* may be skipped. Introns on both sides of the exon will then be combined into one large intron. On the other hand, intron penalties should always be higher than the longest expected random match, typically $10 - 15$ base pairs.

3.8 Memory saving techniques

When computing large alignments limiting memory use is often a more pressing issue than limiting the spent computing the alignment. Any algorithm storing a $n \cdot m$ matrix will use $n \cdot m$ space. For instance a $5000 \cdot 5000$ matrix will use $5000 \cdot 5000 \cdot 4$ bytes space, assuming that each cell entry is a 4 byte integer, which equals roughly 100 MB. There are ways to reduce space consumption, though it will involve recomputing values and thus be more time consuming.

3.8.1 Computing the optimal alignment score in linear space

Looking at recurrence 3.1, a given value $V(i,j)$ is only dependent on $V(i-1, j-1)$, $V(i, j-1)$ and $V(i-1, j)$. For the purpose of finding $V(i,j)$ the rest of the matrix need not be known. By only storing columns j and $j-1$, we have access to all the values needed to compute $V(i,j)$ for any i in column j. After computing the whole column, the entries of column j is simply written over the values in column $j-1$ and column $j+1$ is calculated based on these values, see fig.3.7. This alternating scheme is repeated until the last column is reached and the optimal score is found. By not storing more than two columns at once the space complexity is kept linear $O(m)$. This approach is equally well suited to using two rows instead of columns, adjusting the space complexity to $O(\min(m,n))$ by choosing the one using

the least space, columns or rows. Using even less space can be achieved by using just one column/row and a single value. However, when making another column/row superfluous the space consumption still remains linear, there is no asymptotic improvement in computational complexity.

a)

	C	G
	-1	-2
C	1	-0
A	0	
C	-1	
C	-2	
C	-3	
G	-4	

b)

	G	T
	-2	-3
C	0	-1
A	-1	-1
C	-2	-1
C	-3	-2
C	-2	-3
G	-3	-3

Figure 3.7: Figure a) shows the calculation of a value $V(i, j)$ keeping only the current and previous column in memory. Figure b) shows that the previous 'current' column is now the 'previous' column (note the characters horizontally along the matrix) and a new column has been calculated based on these values. The numbers in the example is the same as in Fig. 3.4.

Having computed the score of the best alignment there remains the problem of finding an actual alignment achieving this score. No pointers have been stored, nor is it possible to reconstruct the path through the matrix solely based on the last two columns/rows. Therefore the algorithm described above is only useful if just the score of the best alignment is needed, not the alignment itself.

3.8.2 Finding the alignment in linear space using Hirschberg's algorithm

To obtain the alignment, in addition to the score, an algorithm designed by Hirschberg can be used. It was originally designed to cope with a different problem, the longest common subsequence, but has been adapted to solving the problem of aligning two sequences[18]. The main idea is to use a divide & conquer approach, recursively dividing the matrix in two, and calculate small parts of the alignment one at the time. This process results in recomputation of cells, equaling more time spent calculating the alignment, however it can

be shown that in the worst case the time spent is only $2 \cdot c \cdot mn$[6], for some constant c, and the asymptotic time complexity remains at $O(mn)$.

Definition 3.15 *For any string S let S_r denote the reversed string[6].*

Definition 3.16 *Given strings S_1 and S_2, define $V_r(i,j)$ as the similarity of the strings consisting of the first i characters of S_r1, and the string consisting of the first j characters of S_r2 [6].*

Calculating the score $V_r(n,m)$ can be done with the same space- and time complexity as calculating $V(n,m)$.

Given sequences S_1 and S_2 with length m and n, respectively. The $m \cdot n$ dynamic programming matrix is divided in two, also dividing S_2 in two, creating two $m \cdot \frac{n}{2}$ submatrices, $(\frac{n}{2}) \cdot m$ and $(n - \frac{n}{2}) \cdot m$. The alignment of the upper half of the original matrix is computed using the 'ordinary' strings while the alignment of the lower half is computed using the reversed strings. Note that the two submatrices overlap in row $\frac{n}{2}$. The values in row $\frac{n}{2}$ are stored for alignments of both submatrices as well as traceback pointers. Row $\frac{n}{2}$ will then contain the values of an optimal alignment from $(0,0)$ to $(\frac{n}{2},k)$, for $0 \leq k \leq m$. Correspondingly, the optimal alignment, computed using the reversed strings, from (n,m) to $(\frac{n}{2},k)$, for $0 \leq k \leq m$, is also stored. Since only two rows are stored the space complexity is kept linear. To find the optimal alignment $V(n,m)$ we now need to find the two alignments $V(\frac{n}{2},k)$ and $V_r(\frac{n}{2},m-k)$ for some position k on row $\frac{n}{2}$, that maximizes the optimal alignment score for the whole matrix.

Definition 3.17 *Let k^* be the position k that maximizes $[V(\frac{n}{2},k)+V_r(\frac{n}{2},m-k)]$[6].*

From definition 3.17 it follows that $V(n,m)$ equals;

Recurrence 3.7 $V(n,m) = \max[V(\frac{n}{2},k^*) + V_r(\frac{n}{2},m-k^*)]$

Since row $\frac{n}{2}$ is stored for both the lower and upper submatrix finding k^* is simply a matter of iterating over the row to find the cell $(\frac{n}{2},k)$ which maximizes $[V(\frac{n}{2},k) + V_r(\frac{n}{2},m-k)]$. Having found k^* we now compute a small part of the alignment. Traceback pointers stored for cell $(\frac{n}{2},k)$, one for each matrix, allows us to store a subpath $L_{\frac{n}{2}}$ of the optimal path L.

Definition 3.18 *Let $L_{\frac{n}{2}}$ be the subpath of L that starts with the last node of L in row $\frac{n}{2}-1$ and ends with the first node of L in row $\frac{n}{2}+1$ [6].*

Locating position k^* takes $O(nm)$ time since computing the submatrices takes, at the most, $O(nm)$ time and finding k^* in row $\frac{n}{2}$ takes time proportional to the length of the row, $O(m)$, a total time consumption of

$O(nm)$. Having done this we now have two new submatrices A and B where the dimensions of A are $(\frac{n}{2}-1)\cdot k_1$ where k_1 is the position in S_2 corresponding to the last node of L in row $(\frac{n}{2}-1)$. The dimensions of B are $(n-(\frac{n}{2}+1)\cdot(m-k_2)$ where k_2 is the position in S_2 corresponding to the first node of L in row $(\frac{n}{2}+1)$.

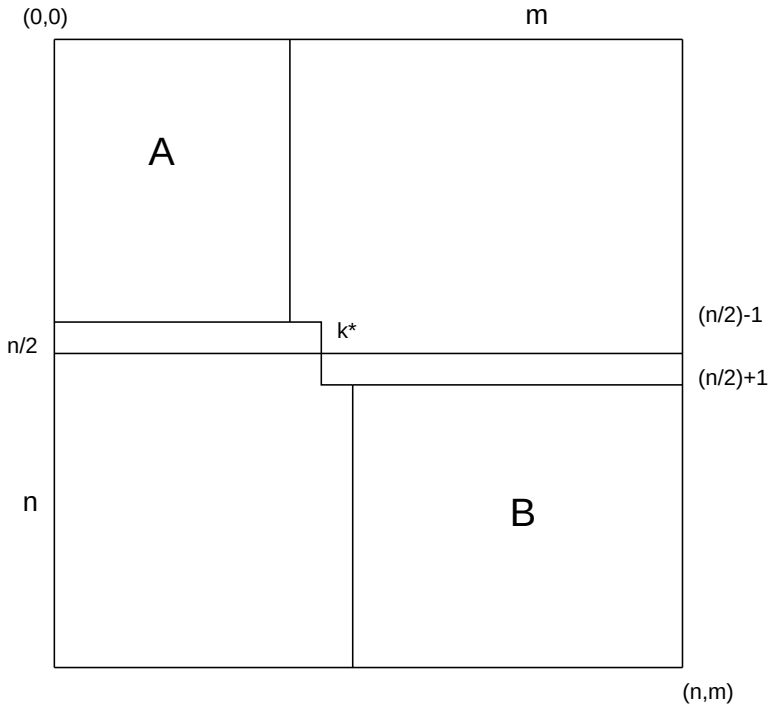

Figure 3.8: The figure shows how a matrix is divided at row $\frac{n}{2}$ and the value k^* is calculated. The subpath from row $(\frac{n}{2}-1)$ to $(\frac{n}{2}+1)$ can then be found. The matrix is now divided into two submatrices, A and B, to which the Hirscberg algorithm is recursively applied.

Large pieces of the original matrix, everything but A and B, have been eliminated from the consideration of which cells lie on the optimal path. The algorithm recursively subdivides the A and B matrices into smaller matrices, using the same technique, until a continuous optimal path L has been found. It is immediately obvious that further subdividing matrices A and B leads to values in both A and B being recomputed.

The alignment computed this way is a global alignment, a local a alignment can be found if we can find the maximal scoring substring *Alpha* and *Beta* in linear space. The endpoint of the substring can easily be found, as it corresponds to the highest scoring cell in the matrix. To find the start point, either one must use reversed dynamic programming or implement a

special pointer scheme. Once found the two substrings define a submatrix the Hirschberg algorithm can be applied to.

For a more thorough examination of the algorithm, including proofs, see [18] or [6].

3.8.3 Finding the alignment in linear space using FastLSA

A second algorithm that reduces the memory requirements for computing the dynamic programming matrix is FastLSA[2].

In the same spirit as the Hirschberg algorithm, FastLSA divides the dynamic programming matrix into parts, where each part is calculated separately as needed. The FastLSA algorithm does not deal with reversed strings, each submatrix is calculated in the usual way. If a single submatrix is to be calculated autonomously the base cases of the recurrences, in addition to the substrings defining the matrix, needs to be known. To facilitate this the algorithm calculates and stores the needed rows/colums of the base cases in grid cache lines. To fill these cache lines with values the algorithm first have to compute most of the cells in the matrix, everything but the bottom right submatrix, using the linear space method using two columns/rows.

The number of submatrices the main matrix is subdivided into is a tunable parameter. The matrix can be divided both horizontally and vertically, as opposed to the Hirschberg algorithm which only divide vertically *or* horizontally. The matrix may also be divided into more than two parts. The number of submatrices the main matrix is subdivided into might affect performance both time wise, the number of recomputations needed, and space wise, the number of rows/columns that needs to be cached. Dividing one sequence, along the row of the matrix, into k parts and the other, along the column of the matrix, into t parts one gets $k \cdot t$ submatrices. This necessitates storing $k - 1$ columns and $t - 1$ rows of length m and n, respectively, in addition to the base cases for the main matrix.

We examine the recursive division of a matrix in the algorithm step by step.

1. *Allocate a base case buffer.* Initially a maximum amount of memory to be spent on a dynamic programming matrix is set. This amount might reflect the total memory available or some other constraint.

2. *A dynamic programming matrix, or submatrix, is given and base cases filled in*

   - If the whole matrix is able to fit in the base case buffer the matrix is calculated as per the base algorithm and the alignment produced. Either the entire optimal alignment is found or a new submatrix is selected for computation.

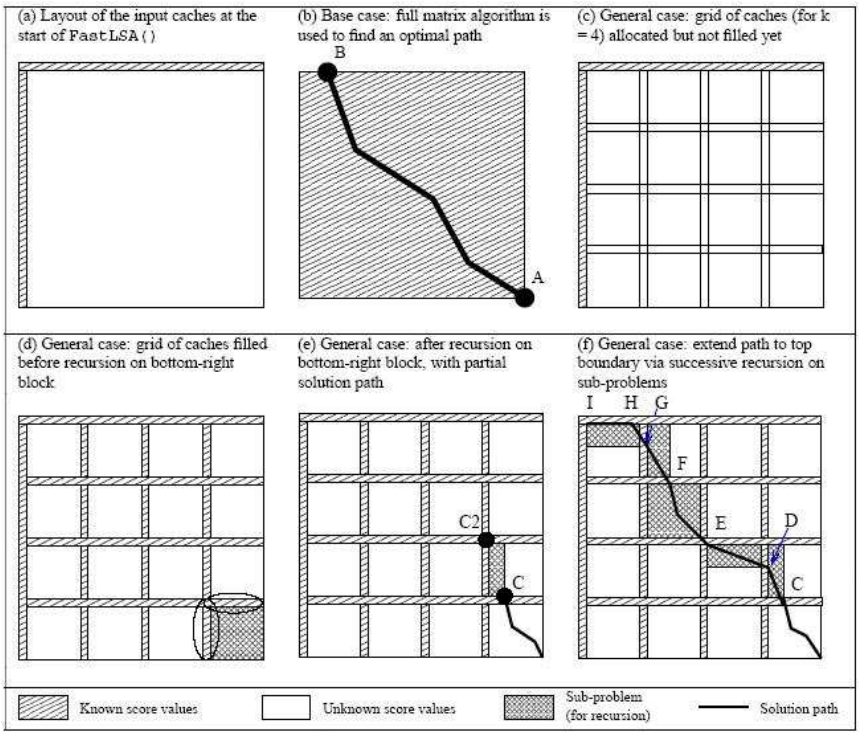

Figure 3.9: This excellent figure[2] demonstrates different cases of the fastLSA algorithm.

- If the matrix does not fit in the base case buffer it is subdivided into a (tunable) number of smaller submatrices and step 3 executed.

3. *Allocating the cache.* Grid caches are allocated corresponding to the number of submatrices.

4. *Filling the cache.* The matrix is calculated to fill the row/column cache lines for the submatrices. Starting with the bottom right matrix step 2 is executed once again.

As step 2 & 4 indicates the algorithm is a recursive, divide & conquer algorithm and as such analogous to the Hirschberg algorithm. If the buffer is small enough or the matrix very large, multiple levels of recursion might be necessary.

However the steps described above does not describe how to find the optimal alignment once we have divided the matrix into multiple submatrices. Having recomputed a given submatrix we can calculate a subpath of the optimal alignment. If the alignment produced does not extend all the way

to the upper left corner of the original matrix (assuming global alignment) we have to compute a subalignment of at least one more submatrix. To extend the alignment further we have to recompute the necessary parts of a submatrix adjacent to the endpoint of the subalignment. This translates to that if the subalignment ends at cell $(0, X)$, $X > 0$, in the submatrix we recompute the submatrix to the left of the current one down to, and inclusive, row X in the case of a horizontal gap or down to row $X - 1$ given a diagonal move. Similarly an alignment ending in $(X, 0)$, $X > 0$, will necessitate a recomputation of the matrix above. If the alignment ends in $(0, 0)$ either the one to the left, the one above or the diagonal one is chosen as the next submatrix. Which of those matrices to recompute can be determined if we, in addition to values, store traceback pointers for the cached cells. Recalculating matrices in this way can conceptually be viewed as being analogous to backtracking from a single cell in the matrix. Once a correct move, say a diagonal move, has been found other moves (horizontal or vertical) will never be considered later in the backtracking. Correspondingly, some submatrices will never have to be recomputed in the FastLSA algorithm. Given that we wish to compute a global alignment, the first submatrix to recompute during the traceback step is the one that contains the cell with the optimal alignment score, the last is the submatrix containing cell$(0, 0)$.

Lemma 3.1 *Given a dynamic programming matrix defined by two sequences S_1 and S_2 of length m and n, respectively. Let $S(m, n, k)$ be the maximum number of dynamic programming cells that need to be stored in order to align the sequences using $k - 1$ cached columns and $k - 1$ cached rows of length m and n, respectively. Let the size of the reserved buffer be BUF. Then $S(m, n, k) \leqq k \cdot (m + n) + BUF$[2].*

As can be seen from lemma 3.1 the space required increases only linearly with increasing matrix size. Still, the space requirement is higher than for the Hirschberg algorithm but the time required for recomputation is lower. As the lemma 3.2 shows the upper time bound decreases as k increases.

Lemma 3.2 *Given a dynamic programming matrix defined by two sequences S_1 and S_2 of length m and n, respectively. Let $T(m, n, k)$ be the number of dynamic programming cells that need to be calculated in order to align the sequences using $k - 1$ cached columns and $k - 1$ cached rows of length m and n, respectively. The total execution time of FastLSA is proportional to $T(m, n, k)$. In the worst case $T(m, n, k) = m \cdot n \cdot \frac{k+1}{k-1}$[2].*

Example 3.1 *For $k = 4$, $T(m, n, k) = m \cdot n \cdot \frac{4+1}{4-1} = m \cdot n \cdot \frac{5}{3}$.*

In practice, FastLSA has the advantage that it can be tailored to use a given amount of memory allowing small matrices to stay in the processor cache and larger matrices to stay within main memory. This translates into

faster execution as retrieving information from the cache is faster than from main memory which in turn is faster than retrieving from a harddrive (virtual memory). For instance, it is about one million times slower retrieving a value from a harddrive than from main memory.

Chapter 4

Graphs

4.1 Splice graphs

The traditional approach to analyze different splice variants is to compare them directly in a case by case fashion. However, with the amount of genes found being alternatively spliced this approach becomes cumbersome and inflexible. Some genes produce as many as thousands of different transcripts, making a list of all transcripts difficult to build and analyze. Moreover, such a list does not show the relationship between different transcripts and does not show the overall structure of all transcripts. A better way to represent this information conveniently, is collecting all splice variants and inserting them into a single data structure, a splice graph[7][13]. Definition 4.1 sums up the properties of a splice graph quite neatly.

Definition 4.1 *Let $S_1, \ldots, S_n$ be the set of all RNA transcripts for a given gene of interest. Each transcript S_i corresponds to a set of genomic positions V_i with $V_i \neq V_j$ for $i \neq j$. Define the set of all transcribed positions $V = \bigcup_{i=1}^{n} V_i$ as the union of all sets V_i. The splicing graph G is the directed graph on the set of transcribed positions V that contains an edge (v, w) if and only if v and w are consecutive positions in one of the transcripts S_i. Every transcript S_i can be viewed as a path in the splice graph G and the whole graph G is the union of n such paths [7].*

It is common to fuse nodes with $indegree = outdegree = 1$ to make the graph more compact and readable. Each node will then, ideally, represent an exon. The splice graph can be constructed based solely on expressed sequences, genomic sequences are not necessary, which can be considered an advantage. This advantage stems from the fact that even as genomic sequencing advances rapidly a large number of EST data can still not be mapped onto genomic sequences[7]. In addition there are several organisms which have available EST data but no genome sequencing project. The disadvantage of a the splice graph is that, currently, most applications/tools

work on consensus sequences rather than graphs. However, as a transcript can be viewed as a path in the graph, it should be possible to extract the necessary information.

4.1.1 Assembling a splice graph

Assembling consensus sequences from EST clusters is commonly done using an overlap graph (as described in the EST clustering process). Assembling the sequences into a graph, rather than a consensus sequence, is usually done by constructing the graph as a $k-mer$ graph [7][13]. The key is to break the data down into parts of a fixed length (length k). Each node is represented by a $(k-1)-tuple$ and each edge by a $k-tuple$. Sequence variation and alternative splicing is represented as bifurcations in the graph. Any errors in the EST sequences used to construct the graph will cause a serious 'blurring' effect, adding erroneous edges to the graph hiding the real exon structure. To build a directed acyclic graph (DAG), we must assume that every $k-tuple$ is unambiguously defined in the consensus sequences, if the $k-tuple$ is repeated the graph will contain a cycle. In order to weed out erroneous edges, good error correction algorithms are vital. The specifics may vary, but in general information such as splice sites, the weight of the edges (the set of sequences supporting it) or alignment information for overlapping sequences is used to remove the errors that might occur.

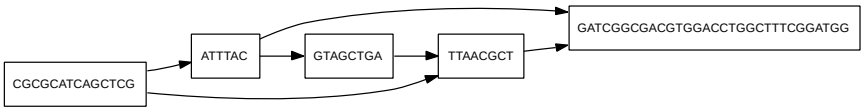

Figure 4.1: An example of a simple splice graph, visualized using graphviz[5].

4.2 Partial Order Graphs

4.2.1 Motivation

A partial order graph is a way of representing a multiple sequence alignment. A multiple sequence alignment is, as the name implies, aligning several sequences to each other. As each sequence must be aligned to all other sequences, the time complexity quickly rises to levels where even supercomputers struggle. Aligning two sequences S_1 and S_2 of length m and n, respectively, gives a running time of $O(mn)$ as proved in the previous chapter. Aligning three sequences S_1, S_2 and S_3 of length m, n and l has a time complexity of $O(mnl)$, cubic complexity. Instead of a (2D)matrix we

then have a cube with $(m \cdot n \cdot l)$ entries to fill inn. To generalize, the time complexity of aligning m sequences of length n is $O(n^m)$.

Given the high complexity of aligning many sequences one usually attempts a less computationally intensive, but perhaps less accurate, approach. These heuristic approaches are usually based on some form of progressive alignment. The essence of progressive alignment is to construct the final alignment based on a series of pairwise alignments. Each pairwise alignment is combined with another pairwise alignment until the last two are combined into the final alignment. Having N sequences result in $\frac{N}{2}$ alignments, these alignments are combined into $\frac{N}{4}$ alignments and so on. A crucial decision is which alignments to combine in each step, usually the most similar sequences are aligned first. When comparing proteins this translates into comparing the sequences that diverged last, a phylogenetic tree is therefore often used to guide the alignment. However, progressive alignment is a greedy method and might end up in a local minimum, either because the sequences are aligned out of order or because alignment errors happening early in the process get locked in. Another problem is choosing the alignment parameters, such as gap weights, for the alignments. In practice, pairwise dynamic programming is not applied to align alignments directly. The alignments are reduces to a single sequence each, a one dimensional profile. This inevitably leads to loss of information[10] as the sequences participating in the alignment are 'averaged' out. However, as we, in all the mentioned dynamic programming variants, align sequences to each other not alignments this has been a necessary procedure. By only doing multiple pairwise alignments the asymptotic time complexity is kept quadratic.

The partial order graph[10] is designed to prevent information loss, or degeneracy, and be alignable using pairwise dynamic programming. A partial order graph is also designed to preserve two pieces of information crucial to a (multiple) alignment: what sequence positions are aligned to each other, and the ordering of these positions within the sequences themselves.

4.2.2 Building the partial order alignment(POA)

A partial order graph is, in addition to being a graph, effectively a multiple alignment representation, and is therefore also referred to as a partial order alignment (POA). Starting with a single sequence S_1, it is converted to a linear graph with each character in S_1 stored on a separate node. A node storing a character $S_1[i]$ has an edges from the node storing $S_1[i-1]$ and an edge to the node storing $S_1[i+1]$ as long as $0 < i < n$, where n is the length of S_1. The node storing $S_1[0]$ has only an edge to $S_1[1]$ and similarly the node storing $S_1[n]$ has only an edge from $S_1[n-1]$, given that $S_1[n] \neq S_1[0]$. A POA consisting of just one sequence is referred to as a trivial POA. From this scheme it is easily inferred that a partial order graph is a direct acyclic graph.

Building a multiple alignment entails adding sequences to the graph, one by one. The process of adding a new sequence S has several steps, first the new sequence is converted to a trivial POA format. In the next step the trivial POA is aligned with a graph G, which may also just be a single sequence in trivial POA format. After the alignment is done, nodes in the new trivial POA is fused to the graph according to node position. Given two nodes, V and W, that were aligned to each other in the alignment step and hence are to be placed in the same position in the graph. If information about which sequences each node originated from is interesting (say, if one wants to reconstruct a particular sequence from the graph at some later point), the information from both V and W are stored in a list on the fused node. The fusing of nodes is done by the following rules, redundant edges are removed afterwards.

1. If V and W store the same character they are fused into a single node.

2. If V and W are aligned, but store different characters, and node V is aligned to a node X in G whose character is equal to V then G and V are fused.

3. If V and W are aligned to, but store different characters, and V is not as yet aligned to any node in Gwhose character equals W, then V and W are recorded as being aligned(mismatching) to each other.

4. Unaligned letters are not altered.

A trivial POA obeys a true, one dimensional, ordering of the characters. That is, for two nodes V and W, $i < j$ XOR $j < i$. The $i < j$ relation translates into 'there is a directed path from i to j'. A general POA only obeys such a linear ordering within (linear) regions, where each node has at the most one inedge and outedge. A general POA may contain nodes i and j such that NOT $i < j$ AND NOT $j < i$, there exist no path between i and j. This is illustrated in figure 4.2 displaying both a trivial and a non-trivial partial order graph. A non-trivial POA may contain bifurcations to indicate gaps or mismatched characters. The graph is not guaranteed to obey a linear ordering, rather it is said to obey a *partial* ordering, hence the name partial order graph.

4.2.3 Aligning a POA to a sequence

Adding a new sequence to the graph, building the POA, involves aligning the graph to a sequence. Building a graph from N sequences, involves making $N - 1$ alignments. The alignments we have examined sofar have been alignments of two sequences, rather than a graph and a sequence, and

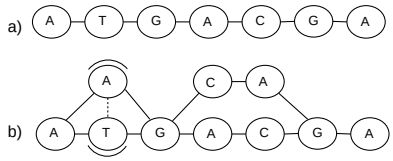

Figure 4.2: Figure a) shows a trivial POA where $i < j$ XOR $j < i$. Figure b) shows a general (non-trivial) POA where $i < j$ AND NOT $j < i$.

a new algorithm have to be designed to accommodate the change from a sequence to a graph.

Rather than creating a completely new algorithm, ordinary pairwise sequence alignment can be extended in a natural way to incorporate a graph along the horizontal axis of the dynamic programming matrix. Given a graph G and a sequence S. Aligning some position in G containing only a single node V to some character c in S can be done in the usual way. In fact, any linear region in the graph can be computed in the same way. A linear region is just a sequence of nodes corresponding to one substring, S'. A submatrix can be constructed aligning S' to S. As such the whole graph can be subdivided into connected linear regions. The partial order structure is thus transferred to the (2D) matrix by forming additional dynamic programming matrices corresponding to bifurcations in the graph. These matrices are connected to each other exactly as the branches in the POA are connected. Now we must extend the base algorithm at the bifurcations, as the cells corresponding to these nodes will have an (analogous to a multiple alignment at that node) extended set of possible moves. In the base algorithm ,a value for a given cell (i, j) could be derived from the values of (maximum) three other cells, the three moves possible was a diagonal$(i-1, j-1)$,a horizontal$(i-1, j)$ and a vertical$(i, j-1)$. These moves are valid for a matrix stored at a node, except for the junctions were multiple matrices 'fuse'. Here we must extend the moves to include moving to each matrix in the junction. In the simplest case, where two matrices meet at a junction (figure 4.3, the moves are extended to include two horizontal moves (one from each matrix), two diagonal moves(one from each matrix) and a single vertical move (on the current matrix).

In a linear region of a POA a node W has (at most) one predecessor node, a node with a directed edge to W. If a node is in a bifurcation of the graph it may have more, one from each matrix fusing at the junction.

Definition 4.2 *Given a node V, let ρ be the set of predecessor nodes of V. Let w be a node in ρ. Finally, let P denote the number of predecessor nodes in ρ.*

From this we can establish a recurrence[10] for computing an alignment.

Note that recurrence 4.1 describes a global alignment, but it is equally feasible to define a recurrence for local alignment or other variants of dynamic programming.

Recurrence 4.1

$$V(i,j) \leftarrow \max \begin{cases} V(w,j) & +gapcost \ \forall \ w \ \epsilon \ \rho \\ V(i,j-1) & +gapcost \\ V(w,j-1) & +score(i,j) \ \forall \ w \ \varepsilon \ \rho \end{cases}$$

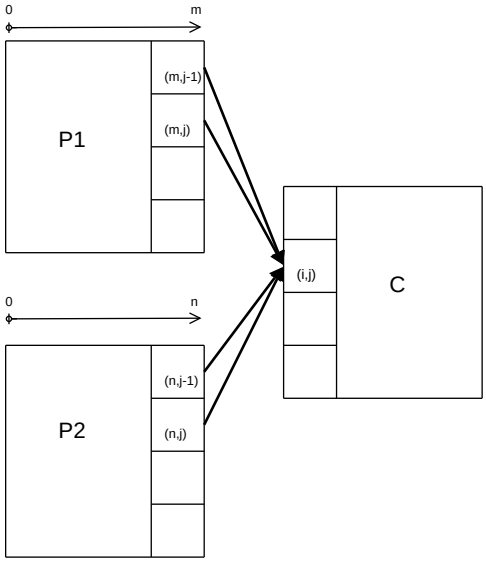

Figure 4.3: Computing a cell stored in a matrix on a node (C) (in a splice graph) with multiple predecessors (P_1 and P_2)

4.2.4 Computational complexity

As the algorithm only differs from the base algorithm in the case of multiple predecessor nodes, the source of any increase in time complexity must be the cases where $P > 1$. Each predecessor node adds two more possible moves to the total number of moves, one horizontal and one diagonal move. The total number of moves that have to be calculated in recurrence 4.1, given P predecessor nodes, is then $2 \cdot P + 1$. Aligning a sequence S, of length m, to a partial order graph G, with n nodes, where the average number of predecessor nodes per node in G is $\bar{P}$, the total time complexity is $O(mn \cdot (\bar{P}+1))$. The time complexity increases only linearly with $\bar{P}$. The space complexity of the algorithm is identical to the base algorithm, $O(mn)$, using the same reasoning.

The $O(mn)$ time complexity is not asymptotically higher than 'regular' dynamic programming, tests have also shown it to perform well in practice[8]. However, partial order alignment suffer from one weakness also associated with progressive alignment, the order of the sequences added to the graph influences the final outcome. As with progressive alignment this can be corrected by adding the sequences in the 'right' order, provided there is one. For instance, when aligning proteins one can use a guide tree[9].

4.3 Treating a splice graph as a POA

Both splice graphs and partial order alignments are graph representations. However, a splice graph is not completely identical to a POA. A splice graph represents a set of sequences whereas a POA represent an alignment of sequences. In a POA, if two aligned characters mismatch, they are recorded as being aligned but unequal. A splice graph can, in principle, also be build from a multiple alignment, if two characters mismatch they are simply assumed to be in different exons.

Even if a POA and a splice graph does not represent the exact same thing, the procedure for aligning a POA to sequence can be applied to splice graphs. Analogous to a POA, a splice graph contains linear stretches of nodes (often fused to a single supernode) interrupted by bifurcations. Thus recurrence 4.1 can be applied to a splice graph equally well, using the same reasoning as for a POA.

Chapter 5

Aligning a splice graph to a genomic sequence

5.1 The goal of the algorithm

Given a splice graph, which incorporates information about different splice variants, we wish to map the graph to a genomic sequence. Mapping a graph as opposed to a single sequence takes into account a much richer set of overlapping sequences and increases the accuracy of the mapping (needs testing). Mapping the graph rather than one consensus sequence at the time will save significant computation time as each consensus sequence usually have regions (exons) in common with other consensus sequences.

Definition 5.1 *Given a splice graph G and a genomic sequence S. Let S' be a path from some vertex V, with no inedges, traversing the graph to some node V',with no outedges, in G. The optimal alignment of S and G is the alignment of S and some path S', which achieves the highest score.*

The alignment produced is the alignment of the optimal path through the graph and the genomic sequences. An alignment of the whole graph and a sequence would be a multiple alignment, which is time consuming to construct. Another reason, besides computational speed, to not make an alignment of the whole graph, is that the splice graph might not be free of errors. Consensus sequences from paralogous genes or pseudo genes might be assembled into the same splice graph. The result is that some sequences will match regions in one gene while other sequences will match regions in a different gene. In this case, trying to map a whole splice graph to a single gene makes no sense.

5.1.1 Decomposing the graph

A basic approach to aligning a graph to a sequence is to enumerate all paths through the graph, with each path corresponding to a sequence, and

then pairwise align each such sequence to the genomic sequence to find the highest scoring alignment. However, this makes the graph structure itself redundant. It would be easier to use at set of consensus sequences rather than go to the trouble of building the graph before extracting the sequences again. It also means that, since the consensus sequences in all likelihood share several exons, the alignment between these exons and the genomic sequences will be computed several times. A significant speedup can be achieved by aligning the graph structure directly to the genome, making it a much better idea.

5.1.2 Basic approach to the algorithm

The basic components of the algorithm is relatively easily explained but have some interesting implications. Given that only parts of the genome are expected to align well with the nodes of the graph local alignment is an option. However, we want to match a whole graph, not find the highest scoring stretch of nodes with $indedges = outedges = 1$. The highest scoring of such a stretch of nodes would in most cases just be the largest exon, the stretch with the longest substring stored on it. Rather than trying to combine several (non-overlapping) local alignments to produce a full path through the graph, I have chosen to use global alignment. The obvious problem is then to have the alignment match the exon/intron patterns assumed to exist in the genome. To resolve this problem the alignment is strongly encouraged to match long gaps in one sequence (expressed sequence), having correct donor/acceptor pairs, with introns in the other (genomic) sequence. In order to make this happen the scoring scheme needs to make such alignments cost effective.

Scoring scheme

The scoring scheme uses a single value both for matching nucleotides and mismatching nucleotides. This is common when comparing genomic sequences as opposed to comparisons of proteins where scoring matrices are the norm. EST sequences are susceptible to insertions/deletetions as well as single nucleotides being misinterpreted. The logical choice then seem to be to opt for an affine gap penalty. The downside to this is using a constant amount more time and space, space usually being the scarcest commodity of the two. As we shall later see this disadvantage can be offset by the need to only keep the matrix of single node in memory at any time. An alignment of a graph of spliced sequences, containing no introns in the ideal case, to a genomic sequence containing introns would be expected to contain long gaps in the graph sequence opposite the introns in the genomic sequence. Making the gap penalty low enough to map the exons correctly, with valid acceptor/donor pairs, while at simultaneously having gaps that

are not presumed to be introns getting too cheap, are strong arguments for treating introns as a separate case from 'ordinary' gaps. The intron/splice terms used by Mott[16] adequately satisfies these constraints. The last point taken into consideration is that when aligning a long genomic sequence to the graph we expect the graph to map only to a subsequence of the genome (a gene), making end gaps free will hinder sporadic matches to the genomic string outside the expected range. As such, free end gaps should only be enabled for the splice graph, not the genomic sequence. Note that the different terms may effect each other, a high penalty for opening a gap may result in introns without valid acceptor/donor sites, if $intron cost - splice cost$ is too low. Setting the right values in a scoring scheme may be as important as the algorithm itself.

5.1.3 Combining existing algorithms

Having outlined the basic requirements of a new algorithm, the next step is to use the existing algorithms at our disposal to create an algorithm capable of aligning a splice graph to a genomic sequence. Comparing a node to a sequence equals comparing two strings, the one stored at the node and the genomic sequence. At node level we can then apply the following algorithms.

- Pairwise sequence alignment using dynamic programming
  - Global alignment
  - Affine gap penalties
  - Free end gaps
- Special type of gap representing introns (Mott[16])
- Memory reduction techniques
  - Calculating optimal score using linear space
  - Finding the optimal alignment using either FastLSA or the algorithm by Hirschberg

However, we have more than simple collection of nodes, the nodes are locked within a larger structure, a graph. Within the graph the nodes are connected through edges, and the matrices stored on each node are similarly connected to each other.

- Generic graph algorithms such as depth first search and topological sort
- Partial order alignment, to calculate the base cases of a given node dependant on predecessor nodes

- Calculating optimal score using linear space and store the last column of each node.

Finally, some of these must be slightly adapted to be suitable. For instance, one must be able to trace the alignment over several nodes, including affine gaps, and be able identify introns correctly. The trick is to combine the algorithms in a good manner and provide the little extra it takes to make everything work. For instance, at the graph level we can first sort the graph using topological sort. When we subsequently compute the nodes according to the order specified by the topological sort it is guaranteed that no node V is processed before all predecessor nodes of V have been computed. Then we can compute each node using linear space and store the last column. When using POA the base cases of any given node V can then be computed based on the last column stored on the predecessor nodes of V. Subsequently, each node in the graph can be calculated autonomously.

5.2 The Algorithm

Given a splice graph G and a genomic sequence S as input. Every node in G with indegree=0 gets an inedge from an artificially created startnode. Likewise every node in G with outdegree=0 gets an outedge to an artificially created endnode. Having these additional nodes makes it easy to both start and end the algorithm at an appropriate time and place. The startnode contains the first column of the base cases, making it unnecessary to store them on every first node in a potential path through the graph. The endnode contains no information but, as will be demonstrated, is simply a termination criteria. A more precise description, given a splice graph G. An artificial node V is inserted at the start of the graph having outedges to all nodes in G which previously had no inedges. Similarly an artificial node W is inserted at the end of the graph having inedges from all nodes in G which previously had no outedges. V is known as a *startnode* whereas W is known as a *endnode*.

5.2.1 Aligning a splice graph with a sequence

As part of the problem definition we want to align a sequence with a graph. The contents of an alignment have been described earlier in some detail, both for sequences and POA. When aligning a graph with a sequence we use ideas from Partial Order Graphs extending the alignment problem to accommodate splice graphs as previously described. Conceptually the splice graph can be viewed as graph with each node storing a dynamic programming matrix.

Fusing nodes into supernodes

For simplicity, continuous stretches of nodes with indegree=outdegree=1 are fused into a single supernode, as in figure 5.1. A supernode will ideally correspond to an exon barring sequencing errors.

Definition 5.2 *A stretch of continuous nodes with indegree = outdegree = 1 can be fused into one node storing all the necessary information.*

Fusing nodes are done by the following rules, given a current node V and a current supernode S.

- if V has indegree=outdegree=1 it is fused with the current supernode S. Move to next node W following outedge from V.
- if V has outdegree>1 it is fused with the current supernode S. S is then closed. Following each edge from V new supernodes are started.
- if V indegree>1 S is closed. V is added to a new supernode T.
- if V has indegree>1 and outdegree>1 S is closed. V is added to a new supernode T which closes afterwards. Following each edge from V new supernodes are started.

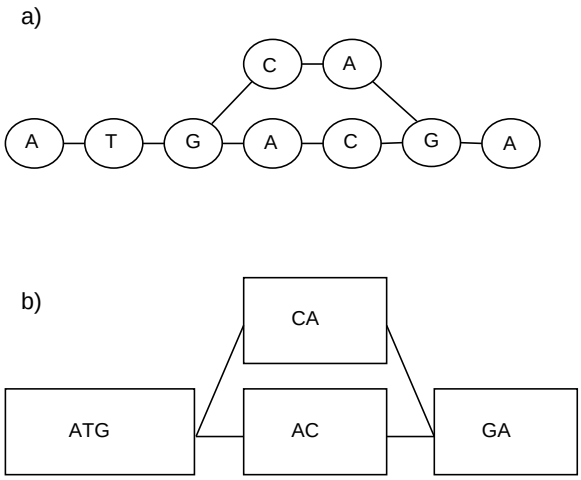

Figure 5.1: Figure a) shows a simple splice graph. Figure b) shows an equivalent representation of the splice graph where the nodes with indegree=outdegree=1 have been fused, where possible, into supernodes (shown as rectangles) without losing information.

Computing a node

Having fused all nodes according to definition 5.2, we will from now on use the term node to denote such fused nodes. The two dimensions of a matrix is determined by the substring (exon) stored at the node (1^{st} dimension) and the length of the genomic sequence along the other axis (2^{nd} dimension). Using the intron term from Mott means that introns will span a number of cells within a given column, as intron only occur in the genomic sequence. Therefore introns will never span different nodes, and can be calculated independently of the rest of the graph. Furthermore, each node can be computed individually and, except for the base cases on each node, completely autonomously. The base cases are determined from predecessor nodes or, if there is no predecessor, calculated as per standard for aligning two sequences. The only node in the splice graph without predecessors is the startnode. However, the nodes must be calculated in a certain order to get the base cases right. For a given node V with a set of predecessor nodes Q, all members of Q must be calculated before V. The cells in the first column of the matrix stored on V is calculated as in recurrence 4.1 where the values of the predecessor nodes equal the values stored in the last column in the matrix on that node. The values of the last column of a predecessor node and the first column of the current node is the interface through which the nodes interact when doing the alignment. An obvious way to guarantee that each node is calculated in the right order is by doing a topological sort of the graph. The startnode, having no inedges, will be the first node in the sorted sequence of nodes while the endnode, having no outedges, will be the last node. Starting with the startnode, and following the topological ordering of the graph, the dynamic programming matrices on the nodes are calculated one by one until the endnode is reached. Assuming a global alignment, the cell containing the score of the optimal alignment will be the bottom right cell on one of the nodes connected to the endnode. If there is more than one node connected to the end node the optimal value is found and the traceback start at the cell on that node. When having end gaps free for the splice graph we must find the optimal score as the highest value in the last column of a node. Note that if end gaps also were free in the genomic sequence, we would have to search the last row of *all* nodes as well.

The recurrences

The recurrences of the algorithm combine, in a natural way, the recurrence for the base algorithm (recurrence 3.1) extended with affine gaps(recurrence 3.4), free end gaps, parts of the intron/splice terms(recurrence 3.6)by Mott[16] and POA(recurrence 4.1), the last when calculating the first column of a new node. Since introns are only found in the genomic string the effect of the intron/splice terms will be confined to a one node and as such will never

add to the set of predecessor nodes for cells in the first column.

Recurrence 5.1 *The general case.*

$$V(i,j) \leftarrow \max \begin{cases} E(i,j) \\ F(i,j) \\ G(i,j) \\ B \end{cases}$$

$G(i,j) = V(i-1,j-1) + score(S_1[i],S_2[j])$
$E(i,j) = \max(E(i,j-1), V(i,j-1) - gapopen) - gapextend)$
$F(i,j) = \max(F(i-1,j), V(i-1,j) - gapopen) - gapextend)$

When computing the first column of a node, fusing recurrence 4.1 with the general case, the G and F terms must be computet for all predecessor nodes.

$G(i,j) = V(p,j-1) + score(S_1[p],S_2[j]) \forall p$
$E(i,j) = \max(E(i,j-1), V(i,j-1) - gapopen) - gapextend)$
$F(i,j) = \max(F(p,j), V(p,j) - gapopen) - gapextend) \forall p$

The B,C and $score(i,j)$ terms are unchanged from recurrence 3.6. Given a spliced sequence S and a genomic sequence G the $score(i,j)$ function and the B term is defined as follows.

$$B \leftarrow \max \begin{cases} B(i) - splicecost & if C(i), j \text{ are a donor-acceptor pair} \\ B(i) - introncost & otherwise \end{cases}$$

$$(B(i), C(i)) \leftarrow \max \begin{cases} (V(i,j), j) & if V(i,j) > B(i) \\ (B(i), C(i)) & otherwise \end{cases}$$

$$score(i,j) \leftarrow \max \begin{cases} match & if S[i] = G[j] \\ mismatch & otherwise \end{cases}$$

Having end gaps free in the splice graph will make some of the base cases zero.

$V(i,0) = -gapopen - (gapextend \cdot i)$
$V(0,j) = 0$
$E(i,0) = -gapopen - (gapextend \cdot i)$
$F(0,j) = 0$

5.2.2 The traceback

Having filled all the matrices on all nodes and found the highest scoring cell we now traverse, as per the base algorithm, the traceback pointers in

the usual way. Note that the traceback will only trace the path through supernodes which lie on the optimal path. Given a sequence S of length n and a graph G of length m. By the length of the graph we mean the sum of the length of substrings stored on the nodes of G. The time needed to find the path is $O(m+n)$, theorem 3.1 (and corresponding proof) can be applied in this case also.

5.2.3 Space complexity

At the start of the algorithm space is reserved for all matrices on all nodes. The consequence of using affine gap penalties is that we have to use three matrices (V,E and F) for each node to calculate the alignment score. Yet another matrix is necessary to store the traceback pointers. Total space consumption is thus $4(m \cdot (n+1))$ for each node, where m is the length of the string stored at the node and n is the length of the genomic string. Since the space requirement for the whole splice graph equals the sum of the nodes including the startnode, this can be reformulated as $4((m+1)\cdot(n+1))$, where m equals the length of the string mapped over all nodes and m equals the length of the genomic string. As mentioned earlier the space consumption of the base algorithm is the product of the length of the two strings in addition to an extra/row column for the base cases, a total of $(n+1)\cdot(m+1)$ cells,where n is the length of one string and m is the length of the other. Adding a table for traceback pointers and affine gap penalties to the base algorithm brings the total up to $4(m \cdot (n+1))$. The amount of space used is ostensible the same, but note that length parameter n describes graph in one case and a sequence int the other case. As a graph usually will contain multiple sequences n will, in practice, be large for the graph.

5.2.4 Time complexity

Even if there are more terms to be calculated then in the base algorithm, it is still a constant number. Given a sequence S of length n and a graph G of length m. As aligning a splice graph with a sequence entail aligning the *whole* graph to a sequence, the complexity remains at $O(mn)$ (theorem 3.1), albeit with a (constant)higher number of operations per cell compared to the base algorithm. How much higher depends on the average number of predecessor nodes in the graph $\bar{P}$. Note that $\bar{P}$ denotes the number of predecessors in a graph before the nodes are fused into supernodes.

5.3 Reducing memory consumption

The naive algorithm calculates the alignment properly but uses a prohibitive amount of resources doing so. The time complexity cannot be reduced, except by a constant amount, based on the need to examine all cell in the

dynamic programming matrices. By not examining all cells, even the most unlikely to be part of the optimal alignment, we might not discover the optimal alignment. If only an approximate optimal alignment is needed a heuristic algorithm is preferable(faster) but it would be hard to develop an accurate one, in terms of finding correct exon/intron boundaries. Even if the time complexity cannot be reduced we can reduce the memory consumption.

Storing Traceback pointers use memory, there can be up to four (including intron) pointers for a given cell. It is also unnecessary to store pointer for nodes which the optimal alignment does not pass through, as these will never be needed. However, there is no way to exclude such nodes before backtracking is commenced. A way of reducing the memory requirements is then to not store traceback pointers, instead the traceback is calculated from the values in the matrix when needed. However, we do store special intron pointers. We store an intron pointer each time the B term is the largest term in recurrence 5.1, that is when an intron is (ideally)found in the genomic sequence. As each intron is large we can possibly avoid examining an excessive number of cells during the traceback computation and since the number of introns in practice are low, compared to sequence length, not much space is used.

5.3.1 Optimal alignment score using linear space

As shown earlier, finding the score of the optimal alignment can be done using linear space. However, we want the alignment as well as the score. By combining this approach with a traceback scheme that can solve smaller bits independently, one by one. First we traverse the topologically sorted graph using linear space dynamic programming storing the last column on each row. By storing the last column base cases for the dynamic programming on a given node can be inferred using recurrence 5.1. A more intuitive approach would have to store the base cases, first column/row, directly, but that would necessitate having to store traceback pointers. This is because we would not know which nodes the values for the base case came from without pointers. We do not need to store the first row as the values there can be calculated from the value in the cell in which the first column/row overlap. Note that, in addition to being more aesthetically pleasing, using supernodes is a condition for reducing the memory requirements, otherwise the matrices on each node would contain only one column(stored) and the memory requirements would be unchanged. Having found the optimal alignment score in cell(i, j) on a node V we recalculate the necessary part of the matrix. Since the optimal score was in cell (i, j) the dimensions of that dynamic programming matrix is $i \cdot j$. If the alignment exits node V in cell (x, y) then is we must consider all predecessor nodes and find which which move(s) lead to the value in cell(x, y). Having found the correct node we recompute parts of the matrix on that node, then add another piece to the global alignment. We do this

until the startnode is reached. For a graph G with i (super)nodes aligned with a sequence S of length n, where the dimensions of the matrix on the largest node are $m \cdot n$ containing k introns, the memory usage is $3 \cdot ((i \cdot n) + (m \cdot n)) + k$. The memory requirement are highly dependant on the size and shape of the graph. For instance, if the EST used to build the graph have sequencing errors that have not been removed we get many nodes storing only one character. Each such node will store the last(only) column and memory usage will increase rapidly. Another point is that if we have a very large node we must somehow manage fit the whole matrix of this node in memory. Recalculating parts of matrices consumes more time than the base algorithm, in the worst case all nodes lie on the optimal path and all cells of every matrix must be recomputed. This equals having to calculate the matrix twice, doubling the worst case time, $2 \cdot c \cdot mn$ for some constant c moves per cell, while the *asymptotic* worst time remains at $O(mn)$.

5.3.2 Using Hirschberg's algorithm

Applying Hirschberg's algorithm to the problem of aligning a splice graph to a sequence could further reduce memory consumption. There are, conceptually, two ways we could apply the recursive divide&conquer scheme to the graph. We could view the graph as a single entity to be calculated, dividing the graph into two sets of nodes, by recursively removing a single edge from the graph. Alternatively we could apply Hirschberg's algorithm to a single node, having calculated base cases using linear space as described. However, either of these cases are difficult to implement in practice. Dividing the graph into two (equal size) sets of nodes is impossible for certain graph topologies. It is always possible to divide the graph in two by having a single node in one part and the rest of the graph in the other. This is reminiscent of applying Hirschberg's algorithm to a single node, since each recursive call would single out a specific node. In this way we do not have to store the last column of each node instead we recompute all the values in each iteration, given n nodes the first node is computed only once while the last i computed n times. As this is too time consuming, applying Hirschberg's algorithm on a node level basis seems a more feasible approach. A single node contains one dynamic programming matrix and in principle should be analogous to the single matrix we get when aligning two sequences. Hirscberg's algorithm computes the path from the bottom right of the matrix to the upper left of the matrix, or some defined submatrix (when computing local alignment for instance). The problem with computing a single node in a graph is that we do not know in which cells the subalignment for the matrix stored at the node starts/ends. If we first compute the optimal alignment score using linear space we have one coordinate, in which cell the alignment ends, but not the other. Without knowing both these coordinates, which implicitly defines the dimensions of the submatrix to be computed, we cannot apply

Hirschberg's algorithm. Since Hirschberg's algorithm is unusable under our circumstances, we turn to fastLSA to reduce the memory requirements.

5.3.3 Using the fastLSA algorithm

When using FastLSA to compute a the matrix on a node we do not have to know in which cell the alignment ends beforehand when backtracking. The alignment on a given node might end in any submatrix bordering the (left) edge of the matrix. As such the algorithm can be applied to the problem of aligning a splice graph to a genomic sequence.

By calculating the alignment score using linear space, and storing the last column on each node, we are already doing something like the FastLSA subdivision of the matrix. Conceptually we are only dividing one axis (with the genomic string along the other axis) and caching values at each bifurcation in the graph. However, we have no control over the size of the submatrices we divide into, it is determined by the graph structure. As FastLSA recursively subdivides the matrix, we can divide each node into submatrices if necessary. This second level of division can divide one axis, or both, and can be controlled by parameters, such as available memory. Since we are using affine gap penalties as well as the B and C terms we need to cache these values in addition to the score $V(i, j)$.

Intron pointers make even more sense when we use FastLSA, if we have to check each cell to find the start of the intron we might have to recompute subtmatrices along the way that do not yield any result. By using intron pointers we can possible skip whole submatrices that otherwise would have to be calculated and eliminated in the traceback calculation.

5.4 Summing up

The designed algorithm integrates ideas from a variety of algorithm and fuses them into an algorithm capable of globally align a splice graph to a sequence using limited memory. Given a sequence S and a splice graph G with lengths m and n, respectively. The length of a graph is here the combined length of the substrings on every node in the graph. The time complexity of the algorithm is $O(mn)$ since all cells must be examined at least once and only a constant number of operations are done on each cell.

The worst case memory bound is a combination of two things; (1) storing a column for each node and (2) storing a matrix/submatrix during the traceback. Let q be the number of fused nodes in the splice graph. If every (fused) node store only a single character $q = n$ and the space complexity of (1) is $O(qn) = O(mn)$, equalling the base algorithm. However, having such a graph in practice is highly unlikely. The sequences making up the graph can not have a any identical character at any given position. Since the alphabet only contains four letters (A,C,G & T) any position covered by at least five

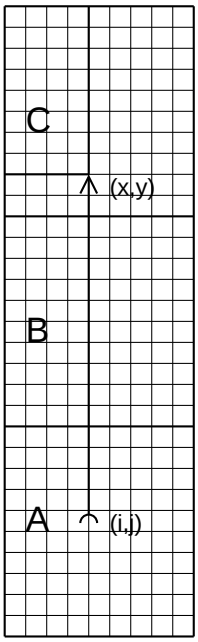

Figure 5.2: Given that there is an intron between the cells (x, y) and (i, j). If we do not have an intron pointer we would have had to calculate submatrix B but using intron pointers mean that we only have to recompute (parts of) matrix C where the bottom right corner is cell (x, y).

sequences would guarantee that at lest two letters would be fused. As such, even completely random sequences should still cause a fair number of node fuses. As for (2) the space complexity is identical to the FastLSA algorithm. Given a node in G storing the largest sequence S_1 and the genomic sequence S_2, of length m and n, respectively. Let $S(m, n, k)$ be the maximum number of dynamic programming cells that need to be stored in order to align the sequences S_1 and S_2 using $k - 1$ cached columns and $k - 1$ cached rows of length m and n, respectively. Then $S(m, n, k) \leqq k \cdot (m + n) + BUF$. The total space complexity is then $O(qn) + S(m, n, k)$.

Lee et.al[10] have also made a variant of POA adapted to aligning expressed sequences. This SPLICE POA program analyzes a multiple sequence alignment consisting of expressed sequences aligned to genomic to identify splices and exons. However, I have found no article describing how the algorithm used, nor how exons/introns are identified.

Pseudo Code

This section contain fragments of a program implementing the algorithm. The different methods show how the computation of a single node is done,

and how a graph level procedure decides the order in which the nodes are computed. The first method calculates the dynamic programming matrix for a given node and stores the last column. The second is a method checking if an intron (may) exist at given indexes. The third piece of pseudo code shows a function calculating nodes (using the first method) in a certain order. This is simple example of how to do dynamic programming for a graph. The following code only outlines the calculations (no optimizations) and use a few auxiliary functions not given source code for.

1: void compute_Node(int *segnr*){
Calculate alignment of a given node(*segnr*) to a sequence. Note that the base cases of the start node is filled in a separate step(in the align() method). functions.}
2:
Require: $segnr > 0$
3: node *current* $\Leftarrow$ graph.getnodefromlist(*segnr*)
4: list *pred_list* {set of predecessor nodes of the current node}
5: predecessor node *curr_pred*
6: string *spliceg* {string stored at node}
7: string *genome* {genomic string}
8: **for** $i = 0$ to *splice_length* **do**
9: **for** $j = 0$ to *genome_length* **do**
10: **if** $i = 0$ AND $j = 0$ **then**
11: **if** *curr_pred* = *start_node* **then**
12: $V(0,0) \Leftarrow -startgap - extendgap$
13: $E(0,0) \Leftarrow -startgap - extendgap$
14: **else**
15: $V(0,0) \Leftarrow pred\_list.get\_Max\_V(i,j)\_Last\_Column(j) - extendgap$
16: $E(0,0) \Leftarrow pred\_list.get\_Max\_V(i,j)\_Last\_Column(j) - extendgap$
17: **end if**
18: **else if** $j = 0$ **then**
19: $V(i,0) \Leftarrow V(i-1,0) - extendgap$
20: $E(i,0) \Leftarrow E(i-1,0) - extendgap$
21: **else if** $i = 0$ **then**
22: $E(0,j) \Leftarrow \max(E(0,j-1), V(0,j-1) - startgap) - extendgap$
23: $F(0,j) \Leftarrow \max(pred\_list.get\_Max\_F(i,j)\_Last\_Column(j),$
$pred\_list.get\_Max\_V(i,j)\_Last\_Column(j-1) - startgap) - extendgap$
24: $b \Leftarrow bcost(C[i], j, 0, y, B)$
25: $V(0,j) \Leftarrow \max(\max_{0 \leq curr\_pred \leq pred\_list.size}(curr\_pred.get\_V(i,j)\_Last\_Column(j-$
$1) + score(splice[0], genome[j-1])), E, F, b)$
26: **if** $V(0,j) > B[0]$ **then**
27: $C[0] \Leftarrow j$
28: $B[0] \Leftarrow V(i,j)$
29: **end if**

```
30:       else
31:         E(i,j) ⇐ max(E(i,j−1), V(i,j−1) − startgap) − extendgap
32:         F(i,j) ⇐ max(F(i−1,j), V(i−1,j) − startgap) − extendgap;
33:         b ⇐ bcost(C[i], j, i, y, B)
34:         V(i,j) ⇐ max(V(i−1,j−1)+score(splice[i], genome[j−1]), E, F, b)
35:         if V(0,j) > B[0] then
36:           C[0] ⇐ j
37:           B[0] ⇐ V(i,j)
38:         end if
39:       end if
40:     end for
41:     if j = genome_length then
42:       for q = 0 TO spliced_length do
43:         current.Store_value(q, j)
44:       end for
45:     end if
46: end for
47:
```

```
1: int bcost(int i, int j, int x, bool Splice_direction, int *B) {
   Checking for introns.}
2:
3: string genomic {genomic string}
4: if j > 1 AND i > 1 AND i < genomic_length − 1 then
5:   if Splice_direction = Forward_Splicing then
6:     if genomic[i] =' G' AND genomic[i + 1] =' T' AND genomic[j −
       2] =' A' AND genomic[j − 1] =' G' then
7:       return(B[x] − splicecost)
8:     else
9:       return(B[x] − intron cost)
10:    end if
11:  end if
12: else if Splice_direction = Reverse_Splicing then
13:  if genomic[i] =' C' AND genomic[i+1] =' T' AND genomic[j−2] ='
     A' AND genomic[j − 1] =' C' then
14:    return(B[x] − splicecost)
15:  else
16:    return(B[x] − intron cost)
17:  end if
18: end if
19: returnMIN_INTEGER
20:
```

```
1: void align() {
```

```
   Doing a topological sort then calculating all nodes in order, code actually
   doing the top oplogical sort not shown.}
2:
3: graph theGraph
4: list topsort {list containing indexes to nodes in the graph}
5: if theGraph.size > 0 then
6:    topsort ⇐ theGraph.do_Topological_Sort
7:    setup(topsort[0],freeendgaps) {initializing base cases for the start node,
      possibly using free end gaps}
8:    for i = 0 TO list.size do
9:       compute_Node(int segnr) {compute a node}
10:   end for
11: end if
12:
```

Chapter 6

Implementing the algorithm

6.1 Programming language

The programming language chosen for this project is C++. The C programming language, originally developed for unix, is a procedural all purpose language and is compiled rather than interpreted. Being compiled it is fast and confers the advantage that an executable can distributed to people without access to a compiler. Except for a few details, C++ is a superset of C with additional features including support for object oriented programming. Object oriented programming provides features such as encapsulation, polymerization, data hiding and inheritance allowing for easy reusage of code and more readable and secure programs. C++ provides this while being only negligible slower, overall, than C. As opposed to languages such as Java, C++ allows the user to control how memory is allocated/deallocated. This have been very useful when allocating/deallocating space for dynamic programming matrices to minimize the memory spent, and to avoid spending time running a garbage collector in the background. There are several commercially available compilers for C++, as well as free compilers such as the GNU GCC[3] compiler, available for a multitude of platforms. Combining speed, user level memory management, a free compiler and object oriented design, C++ had all the necessary features for creating this program.

6.2 Class overview

Figure 6.1 shows the different classes in the program. A brief description of the function of each class follows.

- **runMe** contains the main() method which controls the flow of the program.
- **parser** parses input; config file, splice graph and sequence.

- **EST_genomePOALinSpace** contains all the necessary algorithms for computing alignment score and traceback.
- **segmentlistLinSpace** represents the internal graph structure used in the program, where all nodes are fused into supernodes where possible.
- **segmentLinSpace** represents a single node in the internal graph.
- **splicegraph** represent a splicegraph where each node stores a single character
- **vertex** represents a node in the splicegraph.
- **gridnet** represents a dynamic programming matrix divided into pieces(submatrices).
- **grid** represents a submatrix in the gridnet.
- **synthgen** generates semi-random sequences for testing.

In addition to the aforementioned classes, there are three 'structs' (C-style), used as placeholders for data.

- **introntracker** represents the start/stop indexes of an intron in a matrix.
- **path** contains postions and subalignment during backtracking.
- **config** contains the config data read from file.

6.3 Program flow

6.3.1 Parameters

To impose constraints on the alignment, or on how the program is to compute them, there is a number of user defined parameters that can be set. An important feature is to be able to tune the scoring scheme to suite the current needs. Note that entries marked 'int' are all positive integers while 'bool' denotes a boolean variable. All these variables are read from a config file.

- int **match**, the cost of aligning two identical letters.
- int **mismatch**, the cost of aligning two different letters.
- int **openGap**, the cost to open a gap, independent of gap length
- int **extendGap**, the cost to extend a gap ,a penalty proportional to the number of spaces in the gap.

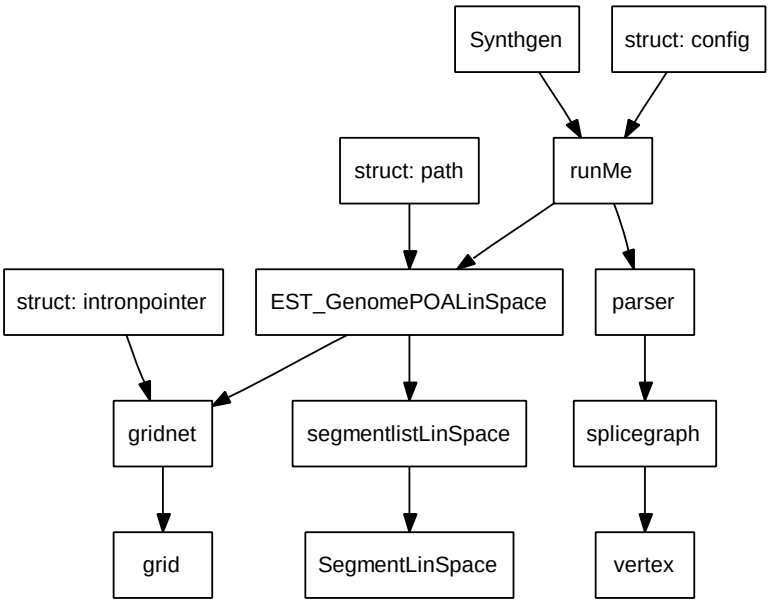

Figure 6.1: Figure showing the different classes in the program

- int **splice**, the cost for inserting an intron, with correct splice sites GT/AG (or CT/AC if the splice direction is reversed
- int **intron**, the cost for inserting an intron, without proper splice sites.
- bool **useFreeEndGaps**, gaps before/after the start/end of the splice graph is not penalized.

There are also a number of parameters related to memory management and input.

- int **bufferSize**, the maximum size (in bytes) of memory the programm can allocate for dynamic programming tables.
- int **divideCol**, the number of pieces the sequence along the vertical axis is divided into (creates submatrices).
- int **divideRow**, the number of pieces the sequence along the horizontal axis is divided into (creates submatrices).
- bool **useAutoSplit**, the algorithm will divide a gridnet (automatically) in such a way that all grids are guaranteed to fit into the allocated memory buffer
- bool **onlyForwardSplice**, forces the program to just check the forward splice direction instead of both (saves time).

- bool **onlyReversedSplice**, forces the program to just check the reverse splice direction instead of both (saves time).
- bool **onlyForwardGraph**, forces the program to just check the forward graph direction instead of both (saves time)
- bool **onlyReversedGraph**, force the program to just check the reversed graph direction instead of both (saves time).

Reversing the graph translates into reversing the edges of the DAG and traversing it from opposite direction. The default behavior of the program is to check both directions of the graph using forward splice direction. The highest scoring graph direction is the realigned using reversed splice direction. By setting a parameter such as 'onlyForwardSplice=true, consideration of reversed splice sites are not made. If both onlyForwardSplice=true as well as onlyReversedSplice=true the program will print an error message as these options are mutually exclusive.

Finally it is possible to have the program output (to screen) information about what is currently being computed, indirectly displaying the progress of the alignment procedure as it traverses the graph.

- bool **writeProgress**

6.3.2 Input/output

The program makes no attempt to build a splice graph from a set of EST sequences, this is a outside the scope of this thesis. Instead the graph is built from the consensus sequences, the input is a (multiple) alignment of the sequences to be used. If no alignment is available it can be built iteratively, reminiscent to building a POA, by aligning one sequence at the time to the graph.

The program offers numerous outputs both to file and screen. The three files the program outputs each have slightly different purposes. The .aln file outputs the alignment in a human readable way, displaying the pairwise aligned characters (60 characters per line) and also the total number of characters displayed up to, and included, that line at the end of each line. The .nat file also contains alignment information, but one that is used as input to the program. The .dot file describes the splice graph in the DOT language. The DOT language is a way to describe directed graphs and is compatible with the Graphviz[5] tool. See figure 4.1 for an example on how Graphviz displays graphs.

6.3.3 Typical program flow

This section briefly describes how a typical run of the program might look like from the perspective of the classes involved.

Input and setup

The main() method in class runMe reads the genomic sequence and the set of sequences corresponding to a splice graph from a file. The data is sent to an instance of the parser class. The parser object parses the data and creates data structures (splice graph and genomic sequence), if necessary. The splice graph is then passed as an argument to an instance of segmentListLinspace which converts the splice graph to the internal representation used by the program. The most notable features of this representation is that all nodes that can be, are fused into larger nodes (segments) and that artificial startnodes/stopnodes are inserted. Each segment also stores a substring and the dimension of the dynamic programming matrix that will be made when aligning this node to a sequence. Memory is also allocated to store the last column of the would be matrix. Subsequently the main() method creates an instance of a EST_genomePOALinSpace object with parameters read from the config file and the data structures to be aligned, notably the genomic sequences and the segment graph.

Execution

The process of aligning a sequence to a splice graph has several distinct steps. To calculate the alignment score the segments are first sorted, topologically, in a sequence. Traversing the sequence of segments, each segment is computed using linear space and the last column on each segment is stored. For a given segment, the information needed to construct the matrix is transferred to the the EST_genomePOALinSpace object before the computation is done. Having calculated all segments, the segments connected to the endnode are examined to locate the cell with the optimal score. Once found, a special struct, known as *Path*, is created to represent the alignment at the current stage in the traceback. In addition to alignment this struct stores an index to a segment in the graph being/about to be processed, as well as the bottom right coordinate of the matrix on that segment. This is not all, information about which submatrix(grid) one is currently computing and whether the last part of the optimal alignment computed ended in the middle of a horizontal/vertical gap is also stored in the Path struct.

To compute the alignment, parts of the matrix must be recomputed, from the upper left corner to the coordinate supplied by Path. If this matrix is small enough to fit in the allocated buffer it can be done in the ordinary way using a full two dimensional dynamic programming matrix. Note that the program still creates an instance of a gridnet to compute the matrix, though no values are cached or recalculated. Having calculated the matrix we can trace the subalignment from the bottom right corner to some cell(i, j) in the first column. From which segment, and the coordinate in that segment, the cell preceding cell(i, j) in the global alignment is stored in Path and

the subalignment computed is concatenated to the total so far. Then, using information from the Path struct, the current matrix is deleted and a new segment is computed in the same fashion. If an aligment ends in mid-gap the gap must be completed when starting the backtracking on the new segment before all backtracking moves are viable again.

If the part of a dynamic programming matrix, which are necessary to recompute, is too large to fit in the allocated buffer, we need to subdivide the matrix into smaller submatrices. In the implementation, the set of all sumatrices from a given matrix form a gridnet where each submatrix is known as a grid. The gridnet is recomputed to fill in rows/columns that are stored as base cases for each grid. Note that having k grids, only $k-1$ have to be recomputed, as recomputing the bottom right one would not lead to any base cases being computed. Each submatrix is then, if it can be fitted in the buffer, recomputed as necessary using the same approach as for a whole matrix. Starting at the grid in the bottom right corner of the gridnet we trace the alignment through a number of grids before reaching a grid in the leftmost column of the gridnet. Moving between gridnets can be viewed as moving between different segments, and correspondingly the current position is duly recorded by the Path struct. When we can trace the alignment to another segment we are done computing the current one. This translates to having reached a grid in the leftmost column of the gridnet in which the traceback traverses beyond the first column.

When we reach the segment corresponding to the startnode and reach cell$(0,0)$ the alignment have been reconstructed. Since we have free end gaps we must extend the alignment with gaps at both ends so that the final alignments spans the entire length between cell $(0,0)$ in the start node and the bottom right corner of the node containing the cell with the optimal alignment score.

Output

After the alignment is computed, control is again passed back to the main() method. It decides what to output, in terms of file formats, and then passes the request back to the instance of EST_genomePOALinSpace. The instance of EST_genomePOALinSpace then proceeds to write output to file.

6.3.4 Special considerations for the implementation of the algorithm

This section covers some implementation specific issues and how they have been handled.

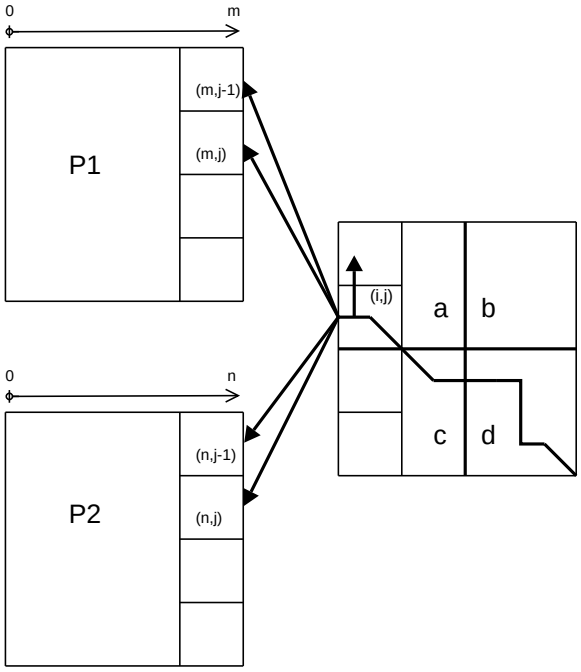

Figure 6.2: Backtracking through a node subdivided into submatrices a, b, c and d using the FastLSA algorithm, with predecessor nodes P1 and P2. Cell(i, j) must consider five possible moves to determine which cell is next on the optimal path. The possible moves are two diagonal moves, two horizontal moves and a vertical move.

Data structures

Some of the classes describing the main data structures used, segmentLinSpace or gridnet for instance, do not contain all the actual data (in these cases matrices) only the information necessary to reconstruct the data. If the data was stored on objects of these classes it would have to be accessed through function, at least if we want data encapsulation. In the process of calling such a function there is some overhead in the actual function call as opposed to accessing the data as a local/class variable. Usually this is of little consequence, but if such functions are called for each cell in the matrix, often millions of times in practice, the overhead is significant and therefore should be avoided if possible. For instance if we are to compute the dynamic programming matrix of a node the dimensions and the string stored at the node would be extracted from a node object and the actual matrix would be constructed in the class (EST_genomePOALinSpace) which does the calculating. This is feasible because we do not need the matrix after we have computed a subalignment through it. Persistent data, such as the last

column of the matrix on each node, is stored on a node object as the base values are needed in constructing a matrix at some later point.

Intron pointers

When storing intron pointers we would like to not store more than necessary while still maintain fast lookup. Declaring a whole matrix using a two dimensional array would provide fast lookup, but would contain mostly blank cells and thus be a waste of memory (we might as well store all traceback pointers in the array). The solution chosen for this program is using a two dimensional vector, a vector is a dynamic array that can grow/shrink as needed. The first dimension is the length of the string stored at the node, and never changes. Each cell in the array can be viewed as a 'bucket' where intron pointers are put as they are discovered. The second dimension is increased as the number of introns increases. As the number of introns rarely gets very high, we expect many of the buckets to be empty. When we need to find a value in the vector during backtrace, we supply the coordinates (i, j) to the cell and find the right bucket using the first coordinate. However, all values in the bucket have to be examined in the worst case. This happens if we do not find a pointer corresponding to the second coordinate. Even if this rarely causes the execution speed to noticeably decline, an additional refinement can be made. Since the values in the vector are inserted in the order they are found when filling the table they will be automatically sorted on the second coordinate, and thus we can use a binary search to lower the worst time of a search from $O(n)$ to $O(\log n)$ where n is the number of intron pointers stored in a bucket.

Dividing a matrix into submatrices using FastLSA

FastLSA will recursively subdivide a dynamic programming matrix into parts that are small enough to fit in the buffer. In the program a matrix stored on a node will not be subdivided more than once, conceptually halting the division after the second level. Each submatrix created must be small enough to fit into the allocated buffer. Since no traceback pointers are stored, which submatrix to backtrack through next must be calculated rather then checked via a pointer. To facilitate this calculation the grid caches are logically not the base cases of a given submatrix, but rather the last column/row of the bordering matrices. This is analogous to the storing of the last column of a node as opposed to storing the first, in the last case we would have to store pointers in addition to the values to be able to do traceback. Another problem surfacing when applying FastLSA to a graph, a sequence of nodes, is that the division parameters will either have to be set separately for each node or a single set of parameters will have to be applied to all nodes. Setting parameters, manually, for each node for a large graph

is infeasible, and using the same parameters for every node does not take into account the varying sizes of nodes. The solution is letting the program decide in how many pieces a matrix should be divided into. The decision is based on the size of the allocated buffer, since each node is only subdivided once it is vital that all submatrices are small enough to fit into memory. In an effort to keep the shape of the submatrices as quadratic as possible, the largest sequence is divided first if the matrix need to be split up. This principle is applied recursively until all submatrices fit into the buffer. Note that the program also makes it possible for a user to specify the division parameters his/herself overriding the automatic division described here.

Number of optimal alignments found

There can be, and usually is, more than one alignment having the optimal score. These alignments can be very similar, the only difference might be, for instance, the position of a specific gap. The number of such alignments grows exponentially with the amount of mismatches/gaps that can be 'moved around' in the alignment. Computing all of these alignments that are not significantly different wastes a lot of time. The only interesting alignments would be those that are significantly different from each other. However, defining 'significally different' is hard and incorporating it into the traceback procedure even harder. When aligning a splice graph with a genomic sequence we try to force the alignment to comply with the intron/splice boundaries. Since introns play such a major part in the alignment, if we have multiple alignments sharing the same exon/intron assembly we can assume that any differences between them are not significant. Therefore, only a single optimal alignment needs to be calculated.

Computing the recurrences

Since the vast majority of total time spent running the program is used in computing the recurrences, these should be as efficient as possible. Firstly, the amount of function calls made inside the recurrence(s) should me minimized. Functions, such as the score(i,j) and finding the maximum from a set of values, should be reduce to their components and inserted directly into the recursion. More code will be generated but, since the functions replaced are small, not excessively so. Another way to limit the number of comparisons in the recurrences, implemented in this program, is to compute the special cases in separate loops. Instead of having a single loop iterating over all cells in a matrix, we have separate loops for the different base cases as these are computed somewhat differently (multiple predecessor nodes etc.). Had we only one loop, each cell would have to be checked to determine if it were a base case of some type resulting in many unnecessary comparisons.

The traceback

Using an affine gap penalty combined with the graph structure makes the traceback step a bit more cumbersome. To avoid spending memory resources on storing pointers I have instead chosen to calculate the traceback. Given the scoring scheme and the value optimal alignment $V(i,j)$ one can calculate, using the same approach as for setting pointers, the cell which precedes (i,j) in the alignment. However, some additional cells may be checked. Firstly, one must check for an intron pointer indicating an intron ending in the current cell. Also, since an affine gap penalty is used, more cells must be checked in addition to $V(i,j-1)$, $V(i-1,j)$ and $V(i-1,j-1)$ to find the predecessor cell of (i,j). Cells $E(i,j-1)$ and $F(i-1,j)$ must be checked to see whether a gap spanning more than one cell ended in (i,j). If this is the case we cannot directly move to $E(i,j-1)$ or $F(i-1,j)$ and repeat the general checking procedure. Having moved to $E(i,j-1)$ or $F(i-1,j)$ the traceback is in the 'middle' of a gap. This results in, if we had for instance moved to $E(i,j-1)$, that any diagonal or horizontal moves cannot be considered before the cell where the gap was opened is reached. In this case successive vertical moves will be made as long as $E(i,x) = E(i,x-1) - extendgap$, for $0 < x < j$, until $E(i,x) = V(i,x-1) - startgap - extendgap$. Then the traceback can be continued as normal from cell (i,x). Long gaps can span different nodes (horizontally) or different gridnets (horizontally and vertically). When the traceback of a node/grid ends with an unclosed gap this is noted in the Path struct. When the traceback of the next node/grid is started the start of the gap is located before ordinary traceback commences.

Chapter 7

Testing

To test the performance, both memory usage and time spent, of the program a number of synthetic sequence was generated and aligned. For checking the accuracy of the algorithm a test using real data was made.

7.1 Test using synthetic data

For the purpose of testing speed and memory usage of the program, synthetic data was generated. By doing multiple test with such data covering all features of the program, reliable average time will be obtained. The test sets generated is not intended to simulate real data, but merely to cover all the features (introns, graph structure etc.) that the program supports, in such a way that all cases are examined.

7.1.1 Generating the data set

Synthetic data was generated by using the following fairly rudimentary model. An genomic string of length n was constructed by adding n nucleotides (A,T,C or G) to the string, where each nucleotide was randomly drawn from an uniform distribution. In order to generate a splice graph, a spliced string was generated first. This string, representing the spliced mRNA, is simply the first $\frac{1}{5}$ of the genomic sequence concatenated with the last $\frac{1}{5}$ of the genomic sequence. Aligning the spliced string with the genomic should result in an intron, splitting the spliced sequence into two parts. To further promote this, acceptor/donor pairs were inserted into the genomic sequence at the appropriate positions. Subsequently the splice string was modified slightly by taking into account the following parameters applied to every position X in the sequence.

- MutateX, a % probability of mutating the nucleotide at this position to a different (random) nucleotide.

- InsertX, a % probability of inserting a (random) nucleotide at this position.
- DeleteX, a % probability of deleting the nucleotide at this position.

This will generate a string similar to the genomic string, but with variability controlled by the previous parameters. The spliced sequence can now be divided an mapped onto the nodes of a splice graph. Determining the graph structure

7.1.2 The test

In this case a 1000 character string was generated as the genome string. The spliced string was made by making a copy of the genome, with a 1% probability of a character mutating, a 1% probability of deleting between $1-25$ characters (randomly generated number), and 1% probability of inserting between $1-25$ characters (randomly generated number). The spliced string was parsed into a splice graph. The splice graph started out with a single node containing the whole string before being divided. At every 1000 character a new edge was inserted to (current position + 250) making the graph (mainly) consisting of a node of 750 characters followed by an (optional) path through a node of 250 characters before a new node of 750 characters. Subsequent tests with a 2500, 5000, 7500, 10000, 12500, 15000 and 17500 character long genome string were also done. Each test was performed 50 times and the average time spent was recorded. All test were made with forward splicing, forward graph direction and without free end gaps enabled. Table 7.1.2 shows the time used in calculating the alignment, the average length of the spliced string (as well as the minimum/maximum length of the spliced string in the 50 iterations) and the length of the genome string.

Test Notes

The rand() function was used to generate semi-random numbers. The seed, srand(), was simply the current time (in milliseconds since 01.01.70). All test were done on an AMD Athlon XP 2000 with 512 MB of RAM running Windows XP. The clock() function from the standard C++ library was used to measure the time spent. All times given in the tests exclude the time being used for string generation.

The largest node (in any test run) contained 1001 characters, the largest genome size was 17500 characters, each entry in the matrix used 4 bytes (integer) and 3 tables (E,F,V) were needed for each matrix. The maximum amount of memory spent, on a dynamic programming matrix, in the test was then $\frac{1001\times15000\times4\times3}{1024\times1024} \approx 200$ MB. As the buffer size was ≈ 350 MB (350 000 000 bytes) no matrix were ever subdivided. To see how subdividing matrices would affect performance the buffer was reduced to ≈ 35 MB, one tenth of

Test using synthetic data				
Time(sec)	Spliced(avr)	Spliced(min)	Spliced(max)	Genome
0.148	379	329	454	1000
1.05532	1001	868	1137	2500
3.19138	2009	1855	2218	5000
6.1817	2989	2757	3217	7500
10.9303	4030	3698	4118	10000
16.1953	4999	4774	5274	12500
23.3844	5988	5519	6297	15000
32.2715	6990	6311	7380	17500

Table 7.1: A table showing the time (in seconds) used to compute problems of different size. The allocated buffer was set to $\approx$ 350 MB

the previous buffer size, and the same test setup was used to generate the graph/sequences. The results are displayed in table 7.1.2.

Test using synthetic data				
Time(sec)	Spliced(avr)	Spliced(min)	Spliced(max)	Genome
0.15962	400	338	453	1000
1.01888	1012	875	1139	2500
3.13748	1998	1759	2127	5000
6.12864	2983	2712	3252	7500
10.8959	3994	3753	4354	10000
16.2832	5006	4721	5263	12500
23.3511	6002	5698	6295	15000
31.9303	6995	6603	7592	17500

Table 7.2: A table showing the time (in seconds) used to compute problems of different size. The allocated buffer was set to $\approx$ 35 MB

Memory versus Speed

When comparing table 7.1.2 and table 7.1.2 they seem remarkably similar. There are some minor differences, probably due to the randomly generated sequences. The subdivision of matrices do not seem to make an impact on the running time, if anything the algorithm uses less time. This the opposite of the expected result, when we subdivide matrices more cells have to be computed resulting in more time spent. However, how the program satisfies the memory requirement set by the user must be taken into account. When the program requests a chunk of memory, the operating system (OS) acts as a memory broker allocating memory to all programs running. When allocating the large chunk of memory ($\approx$ 350 MB) the OS decided that, even

if enough main memory was available, that 350 MB would be requisitioned from pool of slower, virtual memory. This seems to be enough to cancel out the advantage, in terms of the number of values having to be computed, of not having to subdivide any matrices.

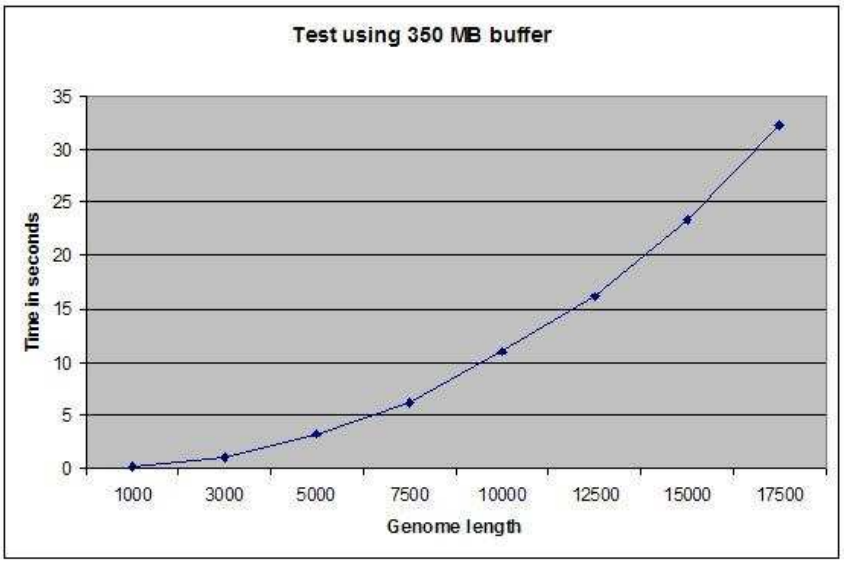

Figure 7.1: Comparing the time used (y-axis) versus the length of the genome (x-axis) In all cases the problem was small enough to be solved in allocated buffer (main memory) without subdividing the dynamic programming matrixes.

A very rudimentary approximation of the computational complexity, one which ignores backtracking, is the product of the strings involves as they determine the size of the dynamic programming matrix. Result are given in figure 7.3 for each of the six tests, with a genome size of 1000,3000,5000,7500, 10000, 12500, 15000 and 17500 respectively. Note that the value used for spliced length is the average length in each test run, from the test using $\approx$ 350 MB buffer. The shape of the two graphs (fig. 7.3 and fig. 7.2) are similar, indicating that the time spent computing the matrix is by far the most time consuming part of the algorithm.

7.2 Test using real data

In contrast to the synthetic benchmark, this test concentrates on accuracy, not speed. Using real sequences the goal is to check whether a splice graph can be mapped correctly to a genomic sequence, including valid acceptor/donor pairs.

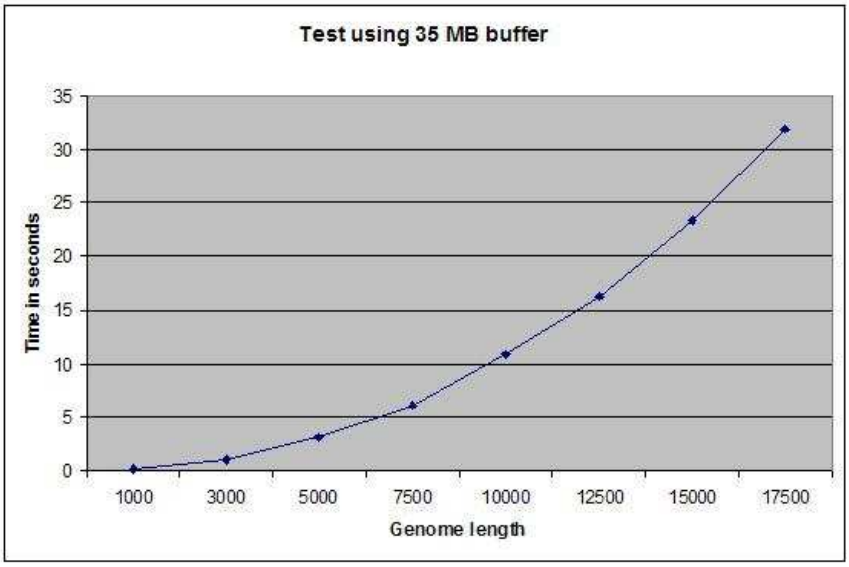

Figure 7.2: Comparing the time used (y-axis) versus the length of the genome (x-axis) the allocated buffer was small enough so that some matrices had to be subdivided.

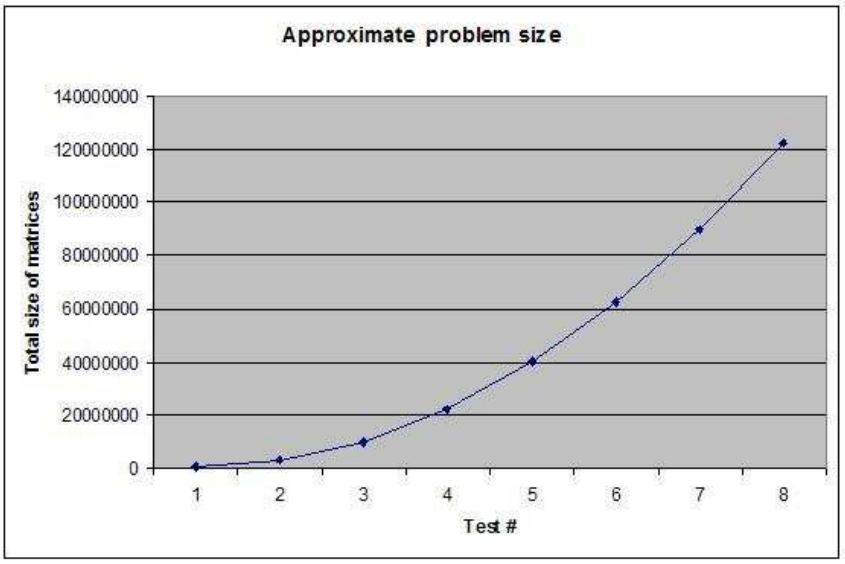

Figure 7.3: The time spent computing a matrix of size (genome length)$\times$ (spliced length)

7.2.1 The test

The genomic sequence used is chromosome number 17 from the human genome. Two consensus sequences from SpliceNest[20] (Hs449264.1 and Hs449264.2) were extracted. As a basic test of accuracy each of the consensus sequences (as trivial graphs) were aligned to the genomic sequence. Both directions of the graph and both forward/reversed splicing were used. The goal was to find the correct intron/exon boundaries compared to SpliceNest. The sequences extracted were relatively small, permitting repeated testing to be done quickly. According to SpliceNest, Hs449264.1 contained an alternative exon. Figure 7.4 shows the consensus sequences aligned with the genome.

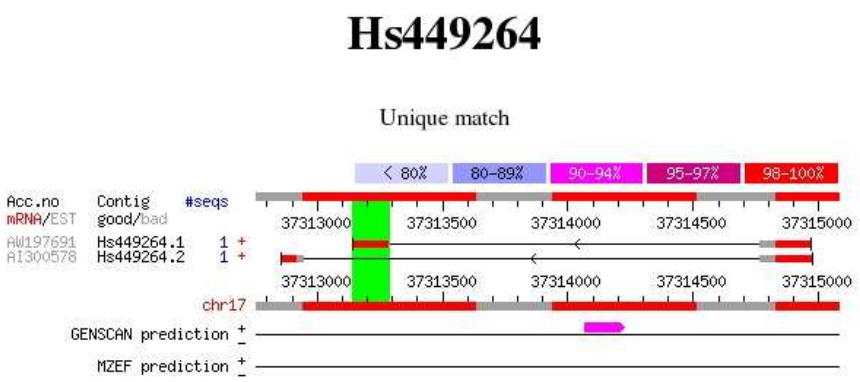

Figure 7.4: A screenshot showing Hs449264 from SpliceNest. The two consensus sequences aligned to the genome (chr17). Darker rectangles represents exons. The exon in the greyed out column is an alternative exon.

The tests were done using free end gaps, both graph directions as well as both forward and reversed splicing. The scoring scheme used was $matchcost = 1$, $mismatchcost = 1$, $opengap = 2$, $extendgap = 1$, $introncost = 40$ and $splicecost = 20$. Both sequences matched the intron/exon boundaries in SpliceNest. Two exons were found in each case, with gaps at both ends (corresponds with figure 7.4). Introns are indicated with '>' (forward splicing) and '<' (reverse splicing). Note that introns are truncated in the alignments below, the (total) length of the intron is displayed as a number instead.

```
Hs449264.1

Alignment score:319
Alignment length:2331
```

```
____________________________________________________________ 60
CAAAGTGCTGGGATTACAGGAGTGAGCCACTGCACCCGGCTCATACAATGTTTAAGCCAG 60

____________________________________________________________ 120
AAGTCACCTTAAAGACCACTTAGTCCCACTCCCTCTTTTTACAGAAGAAACTCAGGCCCA 120

____________________________________________________________ 180
AGAAGGACAGTGACCAGTCTGAGGTCACAACCCAAAGCTGGCAGGGCTGGGGTGAAAAGC 180

____________________________________________________________ 240
TAGGCTCCTGTGACCTCCTTCTGTGGCCATCCCACCTGCCTCATTCCATTTCTGGGGATA 240

____________________________________________________________ 300
GAGATGTCTGTGGGGTTTTTTCCTCCCCATTTTTTCTCCCCCTAGTGTCCAGCATTCGTT 300

____________________________________________________________ 360
AAATACCTAAGAACTACTTAGTGACCTCCCTCCCTCGAGACCTGGGAGGAGGCTGGGATC 360

_______________________________TGTTAGTTCAGGTATAAGACTGATACATAG 420
TCTCCTAGGATGGGGGCTAGAAGTGGGAGCTGTTAGTTCAGGTATAAGACTGATACATAG 420

ACTGTTTTCCTGGCCCAGGCCCATCTGTCACCAACTGACTTTGTATGTGGCTCAGGAGAC 480
ACTGTTTTCCTGGCCCAGGCCCATCTGTCACCAACTGACTTTGTATGTGGCTCAGGAGAC 480

CTCTAGGCAAGGGAACGAGGTTCCTAAGAACCCTGACTCAGGTGAGGTTCTGGCCTCTCT 540
CTCTAGGCAAGGGAACGAGGTTCCTAAGAACCCTGACTCAGGTGAGGTTCTGGCCTCTCT 540

<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< 600
CTCCAGAACATAGGAAATTCAGGTGTAGCTGCATGAAACCTCTCCCTAGAAGAGGAATGG 600

.....1470........<<<<<<<<<<<<CTCCAGCCTGGCTGACAGAGCGAGTCTGTC 2040
.................TGCCCCACTGCAC_TCCAGCCTGGGTGACAGAGCGAGTCTGTC 2040

TCAAACGTCTCAAAAACAAACAAACAAACAAAAAACCTCTTTGAAGGGGATTAAGGGGAT 2100
TCAAACGTCTCAAAAACAAACAAACAAACAAAAAACCTCTTTGAAGGGGATTAAGGGGAT 2100

GTGTCCCAATTAACTAGCAAACACCTGTGAATGCCAGCTTTGTGTCAAGGGGAGAGGGAT 2160
GTGTCCCAATTAACTAGCAAACACCTGTGAATGCCAGCTTTGTGTCAAGGGGAGAGGGAT 2160

TTGTATAGCTAAAAGATATTTATTGCTTAGGAAAAACATGACTGACCTCAATGCATCCTC 2220
TTGTATAGCTAAAAGATATTTATTGCTTAGGAAAAACATGACTGA_ATAAATGCATCCTC 2220

ATT_________________________________________________________ 2280
ATTAAATAAAAAATGCTCCAGTTTCTGTTGTCAGGGAGTTTATTAGAGGGAAAGGTAAGG 2280
```

```
_____________________________________________________ 2332
CCCCCACTTTTCTCCACCTGCCCCTGGCATGATACAAGGCAGAAGGGCAGGT 2332

Hs449264.2

Alignment score:287
Alignment length:2331

____________________________________________________________ 60
CAAAGTGCTGGGATTACAGGAGTGAGCCACTGCACCCGGCTCATACAATGTTTAAGCCAG 60

_____________________________________ACAGAAGAAACTCAGGCCCA 120
AAGTCACCTTAAAGACCACTTAGTCCCACTCCCTCTTTTTACAGAAGAAACTCAGGCCCA 120

AGAAGGACAGTGACCAGTCTGAGGTCACAACCCAAAGCTGGCAGGGCTGGGGTGAAAAGC 180
AGAAGGACAGTGACCAGTCTGAGGTCACAACCCAAAGCTGGCAGGGCTGGGGTGAAAAGC 180

TAGGCTCCTGTGAC<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< 240
TAGGCTCCTGTGACCTCCTTCTGTGGCCATCCCACCTGCCTCATTCCATTTCTGGGGATA 240

......1816.....<<<<<<<<<<<<<<<CTCCAGCCTGGCTGACGGAGCGAGTCTGTC 2040
...............ATTGCCCCACTGCAC_TCCAGCCTGGGTGACAGAGCGAGTCTGTC 2040

TCAAACGTCTCAAAAACAAACAAACAAACAAAAAACCTCTTTGAAGGGGATTAAGGGGAT 2100
TCAAACGTCTCAAAAACAAACAAACAAACAAAAAACCTCTTTGAAGGGGATTAAGGGGAT 2100

GTGTCCCAATTAACTAGCAAACACCTGTGAATGCCAGCTTTGTGTCAAGGGGAGAGGGAT 2160
GTGTCCCAATTAACTAGCAAACACCTGTGAATGCCAGCTTTGTGTCAAGGGGAGAGGGAT 2160

TTGTATAGCTAAAAGATATTTATTGCTTAGGAAAAACATGACTGAATAAATGCATCCTCA 2220
TTGTATAGCTAAAAGATATTTATTGCTTAGGAAAAACATGACTGAATAAATGCATCCTCA 2220

TTAAATAAAAA_____________________________________________ 2280
TTAAATAAAAAATGCTCCAGTTTCTGTTGTCAGGGAGTTTATTAGAGGGAAAGGTAAGGC 2280

_____________________________________________________ 2331
CCCCACTTTTCTCCACCTGCCCCTGGCATGATACAAGGCAGAAGGGCAGGT 2331
```

Even if the sequences aligned are relatively small and uncomplicated (no tiny exons for instance), the accuracy of the alignment process appears to be high. As such, the algorithm satisfies the constraints put upon it; to find the optimal alignment of a graph (trivial graphs in these cases) and a genomic sequence.

Chapter 8

Future Extensions

8.1 Parallelism to speed up the aligning process

Having more than one processor performing the alignment calculation could result in significant time speed up, if done right. By 'right', we mean firstly that the problem should be of such a nature that several parts of the problem can be calculated independently and simultaneously. Secondly, that the speed up factor is approximately linear with respect to the number of processors used.

Speedup of a single matrix, derived from a single node, can be done by the means described by the authors of the FastLSA algorithm. Parts of the matrix can be computed in parallel. The trick is making the problems submitted to each processor of similar size (granularity), such that no processor is idle for long periods of time.

The graph structure itself can promote another level of parallelism. The first pass of the algorithm through the graph will calculate all nodes, storing the last column, to find the best alignment score and the node containing the cell which has this score. Before calculating a certain node we must compute all nodes having an outedge to this node, this is done by calculating each node in the order of which they appear in a topological sort of the graph. If we calculate a certain node(A), and suppose that this node has two outedges to two other nodes (B and C), we can then calculate the two other nodes (B and C) simultaneously provided they have no other indexes from nodes not calculated yet.

Heuristic approaches are also a possibility. During backtracking, while calculating the path through a certain node we can 'guess' which node will be the next to be visited. Rudimentary heuristics could choose for instance; the node with the best(highest) value in the last column, the node with the best average score in the last column, the node which has the most entries which the first column of the currently computed note gets its score from or the node which lies on the longest/shortest path (depends on how similar we

expect the strings to be) from the start node depending on the number of nodes or length of string. However, using heuristics might lead to computing nodes that will not be used, and the depth of the matrix (length of genome string) computed for the chosen node might be larger than required.

An obvious side effect of having several nodes computed at the same time will be added space consumption as several nodes will be in memory simultaneously.

8.2 Aligning two splice graphs

A natural extension to aligning a graph with a sequence is aligning two graphs. Two similar splice graphs could be an indication of paralogous genes, given that the graphs are derived from different genes. Comparing two splice graphs allows us to compare sets of expressed sequences from different genes without knowing the genomic sequence.

For instance, we can compare a splice graph derived a gene in the human genome with a splice graph derived from a gene in the mice genome If the alignment score is high, the genes may be ortologous. The algorithm may (as the current one do) only align the best paths through the graphs or the whole graph structures may be involved. A combination is also possible, the alignment score obtained from aligning the best paths through the graphs can be used to indicate possibly high similarity. If the similarity is high enough then the (whole) graphs may be aligned.

Aligning two graphs is no trivial matter. In the current algorithm one dimension (genomic sequence) of the dynamic programming matrix is always of constant size, whereas a new algorithm would have graphs defining both dimensions. Extending the dynamic programming procedure to incorporate two graphs would be the main challenge of a new algorithm. As the graphs both represent expressed sequences the intron terms should be removed from the recurrence

Chapter 9

Concluding Remarks

An algorithm has been designed to solve the problem of aligning a splice graph to a genomic sequence. The algorithm incorporates many features found in other algorithms to solve the problems posed by the variability of the biological sequences and the graph structure. An alignment of two sequences, or a graph and a sequence, is a way to do inexact matching using a similarity score. Finding an optimal similarity score is only possible within the context of a scoring scheme. The scoring scheme shapes the optimal alignment by influencing the optimal score, having a good scoring scheme is equally important to having a good algorithm.

As the throughput tests show, the time spent computing the alignment is approximately $c \cdot n \cdot m$ for some constant c, where n is the sum of the lengths of the strings on each node in the graph and m is the length of the genome. In fact, it has been shown the asymptotical time complexity is no higher than for standard pairwise alignment, $O(nm)$. Memory is a scarce commodity when using dynamic programming to align sequences. Therefore the algorithm applies memory conserving techniques to limit memory use, allowing large alignments to be done using the program. The user can tune the memory requirements of the algorithm to suit the resources at his/her disposal while experiencing only a negligible loss in performance.

The test using real sequences show that the algorithm can detect intron/exon boundaries successfully. However, more testing needs to be done to test how accurate the algorithm is. An possible error source is erroneous graphs, build from EST sequences containing errors, which may result in the optimal score being found. The parameters of the algorithm may also be a source of errors. Depending on how the parameters are set, the scoring scheme may produce unpredictable results. In particular, exons shorter than the splice term (default value 20) may be skipped entirely.

Finally, there are many possible extensions of the algorithm. Reduced computational time (parallelism) and more advanced forms of alignment (aligning two graphs) are possibilities.

Bibliography

[1] C.Grasso, B.Modrek, Y.Xing, and C.Lee. Genome-wide detection of alternative splicing in expressed sequences using partial order multiple sequence alignment graphs. In *Proceedings of the Pacific Symposium on Biocomputing*, pages 29–41, 2004.

[2] Adrian Driga, Paul Lu, Jonathan Schaeffer, Duane Szafron, Kevin Charter, and Ian Parsons. FastLSA: A fast, Linear-Space, Parallel and Sequential Algorithm for Sequence Alignment. In *International Conference on Parallel Processing (ICPP), Kaohsiung, Taiwan*, 2003.

[3] GNU GCC website, http://gcc.gnu.org.

[4] Mikhail S. Gelfand, Andrey A. Mironov, and Pavel A. Pevzner. Gene recognition via spliced sequence alignment. *Proc. Natl. Acad. Sci. USA*, 93:9061–9066, 1996.

[5] The Graphviz website, http://www.graphviz.org.

[6] Dan Gusfield. *Algorithms on strings trees, and sequences.* The Press Syndicate of the University of Cambridge, 1999.

[7] Steffen Heber, Max Alekseyev, Sing-Hoi sze, Haixu Tang, and Pavel A. Pevzner. Splicing graph and est assembly problem. *Bioinformatics*, 1:1–8, 2002.

[8] Timo Lassmann and Erik L.L. Sonnhammer. Quality assessment of multiple alignment programs. *FEBS Lett.*, 2002.

[9] Christopher Lee and Catherine Grasso. Combining partial order alignment and progressive multiple sequence alignment increases alignment speed and scalability to very large alignment problems. *Bioinformatics*, 2004.

[10] Christopher Lee, Catherine Grasso, and Mark F. Sharlow. Multiple sequence alignment using partial order graphs. *Bioinformatics*, 18:452–464, 2002.

[11] L.Florea, G.Harzell, and Z.Zhang. A computer program for aligning a cdna sequence with a genomic dna sequence. *Genome Res*, 8:967–974, 1998.

[12] Ketil Malde. *Algorithms for the analysis of expressed sequence tags.* PhD thesis, Department of informatics, University of Bergen, 2005.

[13] Ketil Malde, Eivind Coward, and Inge Jonassen. A graph based algorithm for generating est consensus sequences. *Bioinformatics*, 21(8):1371–1375, 2005.

[14] Andrey A. Mironov, James Wildon Ficket, and Mikhail S. Gelfand. Frequent alternative splicing of human genes. *Genome Res*, 9:1288–1293, 1999.

[15] Barmak Modrek, Alissa Resch, Catherine Grasso, and Christopher Lee. Genome-wide detection of alternative splicing in expressed sequences of human genes. *Nucleic Acids Res.*, 29(13):2850–2859, 2001.

[16] Richard Mott. EST_GENOME: a program to align spliced DNA sequences to unspliced genomic DNA. *Comput. Appl. Biosci.*, 13:477–478, 1997.

[17] Richard Mott. Local sequence alignments with monotonic gap penalties. *Bioinformatics*, 15(6):455–462, 1999.

[18] Eugene W. Myers and Webb Miller. Optimal alignments in linear space. *Comput. Appl. Biosci.*, 4:11–17, 1988.

[19] David L. Nelson and Michael M. Cox. *Lehninger Principles of Biochemistry.* Worth Publishers, third edition, 2000.

[20] SpliceNest website, http://splicenest.molgen.mpg.de/.

[21] Edward C. Uberbacher and Richard J. Mural. Locating protein-coding regions in human dna sequences by a multiple sensor-neural network approach. *Proc. Natl. Acad. Sci. USA*, 88:11261–11265, 1991.