Aligning a Splice Graph to a Genomic Sequence

Øyvind Ølberg

June 20, 2005

Abstract

This paper presents an algorithm for aligning a splice graph with a genomic sequence. The algorithm is an extension of the well studied bioinformatical problem of aligning two sequences. The algorithm presented will not exceed asymptotically running time for aligning a graph with a sequence as for aligning two sequences. Moreover, it will be shown that the algorithm is preset to only consume (at most) a user defined amount of memory, reducing the memory requirements to linear space, given reasonable assumptions.

Alternative splicing

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 continuous sequence, known as mature mRNA. The process of removing such is known as *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 create alternative isoforms. 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).

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[5][7]. It is common to fuse nodes with $indegree = outdegree = 1$ to make the graph more compact and readable. Definition 1 sums up the properties of a splice graph quite neatly.

Definition 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 splicing graph G and the whole graph G is the union of n such paths [5].*

Assembling a splice graph

The common way to reconstruct a mRNA sequence is by assemble EST sequences into a consensus sequences. This is usually done 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.

Assembling the sequences into a graph, rather than a consensus sequence, is usually done by constructing the graph as a $k - mer$ graph [5][7]. 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.

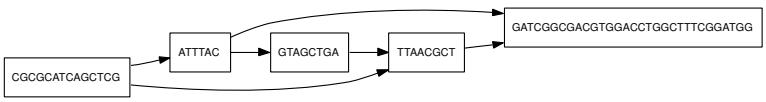

Figure 1: An example of a simple splice graph, visualized using graphviz[3].

Aligning a splice graph to a sequence

Aligning a graph to a sequence is a natural extension to aligning two sequences. Aligning two sequences, a genomic and an expressed sequence, can be done using a dynamic programming procedure to find an optimal alignment. This alignment can either be the best alignment involving the whole sequences (global) or two substrings of the sequences (local). Such an optimal alignment is an alignment with the highest possible score, given a scoring scheme. A scoring scheme describes the cost of aligning similiar/different nucleotides and inserting gaps into the sequences.

A brute force algorithm for finding the optimal alignment is to simply to create all possible alignments recursively and finding, in a case-by case fashion, the alignment with the highest score. Unfortunately there will be an exponential amount of possible alignments giving the algorithm a time complexity of $O(2^n)$. This can be avoided using dynamic programming. 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 the value in each cell in a dynamic programming table using recurrences, whose values are dependent on earlier calculations or explicitly formulated base cases.

Definition 2 *$V(i, j)$ is defined as the value of the optimal alignment of prefixes $S_1[1 \cdot i]$ and $S_2[1 \cdot j]$[4].*

Given a sequence $S_1[1 \cdot i]$ and a sequence $S_2[1 \cdot j]$, the algorithm defines an $(i + 1) \times (j + 1)$ matrix. Each cell (i, j) in the grid determines a position in the first and second sequence. As the algorithm progresses the score $V(i, j)$ will be stored in the appropriate cell. 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 $S_1[1..i - 1]$ and $S_2[1 \cdot j - 1]$, $S_1[1..i - 1]$ and $S_2[1 \cdot j]$ and $S_1[1..i]$ and $S_2[1 \cdot j - 1]$. Why does one only consider these three cases? Consider an optimal alignment of some prefix $S_1[1 \cdot i]$ and some prefix $S_2[1..j]$. 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 get the optimal alignment of prefixes $S_1[1 \cdot i]$ and $S_2[1 \cdot j]$ one now has three possibilities.

Definition 3 *The optimal alignment of $S_1[1 \cdot i]$ and $S_2[1 \cdot j]$ 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 $S_1[1 \cdot i]$ and $S_2[1 \cdot j]$. $V(i, j)$ being the score of the optimal alignment which simply equals the score of the best case. Definition 3 can then be formulated into a simple recurrence.

Recurrence 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].

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

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

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

An example of using recurrence 1 to fill a dynamic programming matrix is shown in figure 2.

		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

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

Figure 2: Aligning the sequence 'CACCCG' to the sequence 'CGTACT'. Calculating $V(i,j)$ for all cells (i,j). For instance is the value of $V(1,1)$ 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'. The optimal score for aligning the two sequences is in the bottom right corner. Using the traceback pointers (shown as arrows) the optimal alignment(s) can be reconstructed by tracing paths through the matrix from the bottom right to the upper left.

It is essential that the scoring scheme is defined in such a way that the optimal alignment produces is the most meaningful from a biological perspective. To achieve this, affine gaps are used as some common EST errors involve deletetions/insertions of multiple nucleotides. As the expressed sequences originate from a limited substring (gene) of the genome free end gaps are used to avoid spurious matches with other parts of the genome. To correctly align all the exons of the expressed sequence to the genome global alignment is used. Both affine gaps and free end gaps can be incorporated into recurrence 1 by adding/modifying the terms of the recurrence. The details are described in Gusfield [4].

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 introns and parts of the mRNA as several small gaps might be preferred instead of the large, true, one. There is no biological reason that an ordinary (not intron) gap should be scored in the exactly same way, another reason to treat introns differently. Secondly introns have certain biological signals, called splice sites, to mark their end and beginning. In the vast majority of cases an intron starts with a 'GT' and ends with a 'AG' (or 'CT' and 'AC' given reversed splice direction). These nucleotide pairs are often referred to as donor/acceptor pairs. This needs to be reflected in the recurrences. Thirdly since introns are spliced out of the mRNA sequence, one would expect introns to appear only in the sequences which has them, namely the DNA sequence, whereas ordinary gaps can appear in both sequences.

These concerns are taken into consideration in the algorithm designed by Mott[8] and used in his program EST_genome. The algorithm is a variant of the Smith-Waterman algorithm described earlier. The interesting addition is an explicit, separate cost for scoring of introns. Let $B(i)$ be the score of the best local alignment, found so far, ending in position i in the spliced sequence and $C(i)$ is 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.

Given a spliced sequence S, a genomic sequence G, the cost *splice* for inserting an intron with correct acceptor donor pairs and the cost *splice* for inserting an intron without correct acceptor donor pairs. the $score(i, j)$ function and the B term is defined as follows. Please note that in the algorithm described in this thesis the B/C terms are used in conjunction with global (rather than local) alignment.

Recurrence 2

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

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

The described extensions/modification to the global alignment problem is adequate to align a consensus sequence to a genomic sequence, but the aim is to extend the algorithm to include a graph along one axis of the dynamic programming table. A way of doing this is treating the splice graph as a partial order graph [6]. 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 trivial POA is fused to the graph according to each nodes position in the alignment. The fusing of nodes are done by following rules.

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.

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 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 *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[6] for computing an alignment. Note that recurrence 3 describes a global alignment, but it is equally feasible to define a recurrence for local alignment or other variants of dynamic programming.

Recurrence 3

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

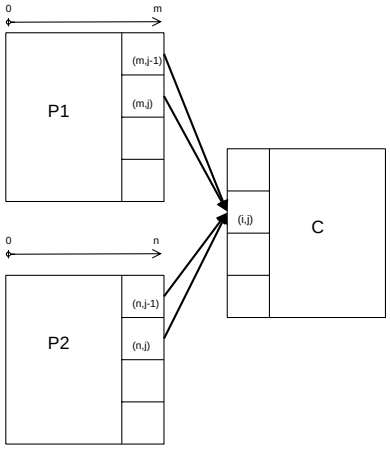

Figure 3: The figure shows the extended set of cells that must be examined for a cell in the first column of C in the case of multiple predecessors (P_1 and P_2)

A splice graph is equal to a partial order graph with respect to computing predecessor nodes, therefore recurrence 3 can be applied to splice graphs as well as partial order graphs. It should be noted that aligning a sequence and a graph in this way will align the highest scoring path trough the graph with the sequence rather than the graph as a whole. Firstly, this is much less computationally intensive (the alternative is a multiple alignment). Secondly, only aligning the best path makes the algorithm much more robust as any errors in the graph will have less of an impact on the final outcome.

An algorithm for aligning a splice graph to a sequence

Each node in the graph can be computed separately if the base cases are known. To facilitate this, each node is calculated in the order they appear in a topological sort. This ensures that every node having an inedge to a node V is computed before V. The first computation of every node is done using an alternate column strategy where only two columns are kept in memory each time. On each node the base cases are stored. The alternate column strategy does only allow us to compute the highest score, not find the optimal path. As such, we traverse the graph once more calculating the necessary nodes. This traversal starts at the node with the highest scoring cell and backtrack trough the nodes until the first node is reached. When backtracking through a single node that node is calculated using a full dynamic programming table (the base cases are stored on the node). Subsequently, an alignment is traced through the matrix and which cell in another node the alignment will need to be extended into is determined. The matrix can then be deleted before a new node is computed. In this way no more than one matrix (from a single node) needs to be in memory at any given time at the expense of having to compute the alignment of the graph and the sequence (at the most) two times.

The recurrences

The recurrences of the algorithm combine, in a natural way, the recurrence for basic global alignment extended with affine gaps, free end gaps, parts of the intron/splice terms (recurrence 2) by Mott[8] and POA (recurrence 3), 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 4 *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 3 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 2. Given a spliced sequence S and a genomic sequence G, the $score(i,j)$ function and the B term is defined as follows.

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

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

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

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

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

Reducing memory consumption

The size of a matrix on a given node is determined by the graph topology and the genomic sequence. This may result in very large matrices, possibly not able to fit in memory. The technique of only having a single matrix in memory can be naturally extended to only keeping a part of a matrix in memory. By using the FastLSA algorithm[1] a matrix can be converted into a grid of submatrices, where each submatrix can be computed as needed. However, this necessitates storing the base cases of the submatrices and recomputing parts of the main matrix to obtain the base case values. The number of submatrices the main matrix is to be divided into is a tunable parameter, allowing the user to determine a maximum size of a given submatrix. Both the horizontal and the vertical axis of the matrix may be split into several parts. Having filled in the values of the base cases, we can backtrack through the main matrix by recomputing the necessary submatrix. Starting in the bottom right corner, and ending in a submatrix bordering the left side of the main matrix, matrices are computed and a subalignment is traced through them. Which submatrices to recompute is determined based on where the subalignments end in the currently computed one, analogous to selecting which node to backtrack through next when tracing the path through the graph. When tracing the path through a sequence of nodes we are conceptually 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. Even if using FastLSA entails recomputing cells, only a constant amount more calculations are needed.

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 $S(m, n, k) =\leqq k \times (m+n) + BUF$. Let T(m,n,k) be the number of dynamic programming cells that need to be calculated. In the worst case $T(m, n, k) = m \times n \times \frac{k+1}{k-1}$[1].

Time and space complexity

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 $O(mn)$, equal to the standard dynamic programming procedure, since all cells of all matrices 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. 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 $O(qn) + S(m, n, k)$.

Implementation and Testing

The algorithm was implemented as a C++ program using the GNU GCC compiler[2]. The program can check both splice directions and/or both directions of graph to determine the optimal alignment if such information is not known before the program is run.

To test the accuracy of the algorithm a simple splice graph was constructed and subsequently aligned to a genomic sequence. The genomic sequence used was chromosome number 17 from the human genome. Two consensus sequences from SpliceNest[9] (Hs449264.1 and Hs449264.2) were extracted. As a basic test of accuracy each of the consensus sequences were inserted into a splice graph trivial graph. The resulting graph had three large nodes (corresponding to exons) but the end of the last exon were split into a few minor nodes. The few minor nodes were the result of slight dissimilarities between the overlapping exons in the two consensus sequences. 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 and find the optimal path containing the most/longest exons. According to SpliceNest, Hs449264.1 contained an alternative exon. Figure 4 shows the consensus sequences aligned with the genome. The tests were done using free end gaps, both graph directions as well as both forward and reversed splicing. The scoring scheme used was $matchcost = 1$, $mismatchcost = 1$, $opengap = 10$, $extendgap = 1$, $introncost = 40$ and $splicecost = 20$. The resulting alignment contained the correct exons aligned to the genome.

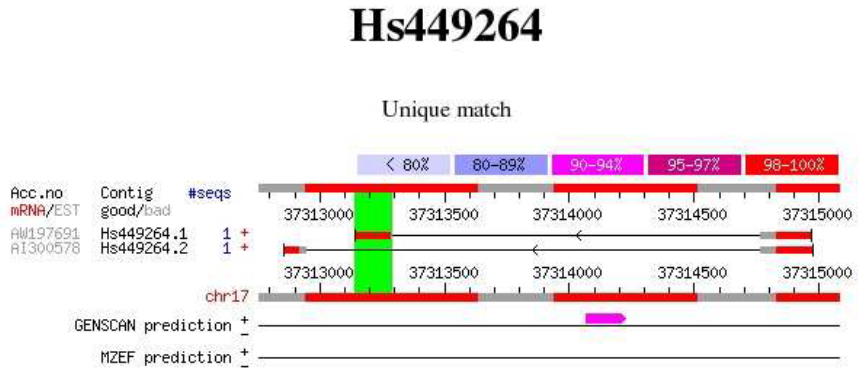

Figure 4: A screenshot showing Hs449264 from SpliceNest. The two consensus sequences aligned to the genome (chr17). Darker rectangles represents exons. The exon in the greyed out column is an alternative exon.

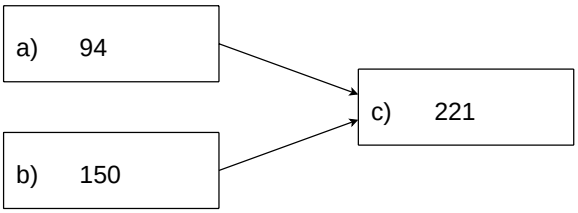

Figure 5: Overview of the splice graph constructed from the two sequences. The two first (a and b) exons are displayed flawlessly, while the exon contained in both consensus sequences (c) are not completely identical in both sequences. The small differences between the sequences are not shown on the figure and has no impact on the detected intron/exon boundaries. The number displayed on each node indicate the amount of characters stored on the node. Aligning this graph with the genomic sequences from chromosome 17 would result in an optimal path from the upper left corner in the matrix defined by exon b) to the bottom right of the matrix defined by exon c).

Throughput test

To test the speed of the algorithm random sequences were used. In this case 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.

The rand() function was used to generate semi-random numbers. The seed, srand(), was simply the current time (in milliseconds since 1970). 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 maximum amount of memory that could be spent, on a dynamic programming matrix, in the test was $\frac{1001\times15000\times4\times3}{1024\times1024} \approx 200$ MB. To include the subdivision of matrices in the test, the buffer was set to ≈ 35 MB. The results are displayed in table .

Concluding remarks

Based on the preliminary results the accuracy of the algorithm show that the correct intron exon boundaries are detected when aligning a splice graph to a genomic sequence. The algorithm only use $O(nm)$ time, the same as ordinary pairwise dynamic programming procedures. To further increase speed, it should be possible

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 1: A table showing the time (in seconds) used to compute problems of different size. The allocated buffer was set to ≈ 35 MB

to parallelize the algorithm along the lines described for FastLSA [1]. In addition the memory spent is a tunable parameter, allowing the user to select an amount that is available on the system, while only being a constant amount slower when using less memory than necessary to fit the largest matrix in memory.

Acknowledgments

Many thanks to Eivind Coward for helping me through the meticulous process of completing the master thesis this article is based on.

References

[1] 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.

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

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

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

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

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

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

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

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