Gene Finding

Algorithms in BioInformatics
Mandatory Project 2

Magnus Erik Hvass Pedersen (971055)
October 2004, Daimi, University of Aarhus

1 Introduction

The purpose of this report is to verify attendance of the author to the *Algorithms in BioInformatics* course at the department of computer science, University of Aarhus.

Following a description of the problem that is to be solved, a basic solution is given along with pseudo-code of the associated algorithms. The model is then extended to solve most of the additional problems posed in the project-description, after which the implementation of both basic and extended models is described. Finally, experimental results are presented.

The project includes an implementation of both the basic and extended alignment methods, and the files (including this report), can be found under *Algorithms in BioInformatics* at `http://www.daimi.au.dk/~u971055/`.

The reader is assumed to be familiar with the problem description as well as the course litterature, and [1] in particular. However, since [1] caters for more experienced bioinformatics researchers, its methods and their realizations are described in more detail here.

2 Gene Finding

Given two DNA sequences, we wish to determine if they contain similar genes, and if so, where they are located, and what changes are required for them to match.

We consider socalled *eukaryotic* genes which are sub-strings of a DNA sequence enclosed in start and stop *signals*, and consist of a number of *introns* and *exons* between these signals. As introns are non-coding, they are deemed irrelevant and we are therefore only interested in matching the signals as well as the exons, which code information that will eventually be *expressed* as physical features of the biological organism.

2.1 Signals

Since a DNA sequence is represented as a string of the four letters a, c, t, and g, the signals that encode the beginning and end of a gene, as well as the signals used in the transition from exon to intron and vice versa, must be provided for by combinations of these four characters. This implementation uses a simple model for identifying signals, in that sub-strings of the DNA sequence are simply compared to the following combinations of characters:

- $G^{start} = \{atg\}$ signals the beginning of a gene.
- $G^{stop} = \{taa, tag, tga\}$ signals the end of a gene.
- $D = \{gt\}$ signals the beginning of an intron.
- $A = \{ag\}$ signals the end of an intron.

And we shall use the following truth-values, to indicate where signals are present in the DNA sequences $S_a = a_1\, a_2\, \cdots\, a_N$, and $S_b = b_1\, b_2\, \cdots\, b_M$:

- $G^{start}_{i,j} \Leftrightarrow S_a[i-2\cdots i],\, S_b[j-2\cdots j] \in G^{start}$, that is, whether the sub-string $S_a[i-2\cdots i]$ equals atg, and similarly for the sub-string of S_b.
- $G^{stop}_{i,j} \Leftrightarrow S_a[i+1\cdots i+3],\, S_b[j+1\cdots j+3] \in G^{stop}$.
- $D^a_i \Leftrightarrow S_a[i\cdots i+1] \in D$, and similarly for D^b_j: $S_b[j\cdots j+1] \in D$.
- $A^a_i \Leftrightarrow S_a[i-1\cdots i] \in A$, and again A^b_j means that $S_b[j-1\cdots j] \in A$.

We shall say that queries in which the indices obviously cause the values to be false, are still permitted. So all of the above, including $G^{start}_{i,j}$ and $G^{stop}_{i,j}$ are allowed for $i \in \{1,\cdots,N\}$ and $j \in \{1,\cdots,M\}$.

To be in compliance with the referential data, these experiments consider the gene's start- and stop-signals to be (parts of) exons also, but this behaviour is easily switched on and off in the source-code.

2.2 Local Alignment

A method based on local alignment, is used in [1] to predict the above gene structure for homologous sequences.

In the case of gene finding, the basic method when using local alignment, is therefore to penalize and reward in accordance with the structure that we wish to uncover. More specifically, as introns are considered garbage, they are free of charge, with the exception that their beginning- and ending-signals are penalized by a factor of γ each, whereas a single gap in the alignment, is penalized by λ. Furthermore, when meeting a gene's start-signal (i.e. atg), we should start a new gene alignment, if the current possible alignments are too severely penalized (that is, there is no proper gene alignment so far).

To do this, first define the maximum similarity measure $S(i,j)$ for aligning sub-strings of S_a and S_b up to their i'th and j'th respective positions, as follows:

$$S(i,j) = \max_{h<i,\, k<j} Sim(a_h \cdots a_i, b_k \cdots b_j) \tag{1}$$

where $Sim(X,Y)$ is a measure on the similarity of the strings X and Y. To support introns, two additional measures are employed, namely: $I^a(i,j)$ and $I^b(i,j)$ for the strings S_a and S_b respectively. These are used to measure the score of starting or continuing an intron in position (i,j), whichever yields the highest score.

2.3 Recursive Equations

The three similarity measures described in the previous section, may be defined recursively as follows:

$$I^a(i,j) = \max \begin{cases} \iota: & I^a(i-1,j) \\ \iota: & S(i-1,j)+\gamma \quad \text{, if } D_i^a \end{cases} \tag{2}$$

$$I^b(i,j) = \max \begin{cases} \iota: & I^b(i,j-1) \\ \iota: & S(i,j-1)+\gamma \quad \text{, if } D_j^b \end{cases} \tag{3}$$

$$S(i,j) = \max \begin{cases} \varepsilon: & S(i-1,j-1)+\delta(a_i,b_j) & \\ \varepsilon: & S(i-1,j)+\lambda & \\ \varepsilon: & S(i,j-1)+\lambda & \\ \iota: & I^a(i-1,j)+\gamma & \text{, if } A_i^a \\ \iota: & I^b(i,j-1)+\gamma & \text{, if } A_j^b \\ & 0 & \text{, if } G_{i,j}^{start} \end{cases} \tag{4}$$

where $\delta(a_i, b_j)$ is the score of comparing the characters a_i and b_j, and the ι and ε annotations are detailed in section 2.7 below, and the base cases for the recursions are detailed in section 2.5.

Note that we do not perform any alignment of introns, even though it is possible to extend the recursive equations with this. The reason is that we are ultimately only interested in the exons, and will therefore discard the introns along with their associated gaps in the opponent DNA sequence, so it does not matter if each intron gives birth to (long) gaps there.

It is evident from the recursively defined equations, that they can be computed using *dynamic programming*, in which we compute a row at a time, beginning with row $i = 0$ up to $i = N$, and for each row i, compute from the leftmost elements $j = 0$ to the rightmost $j = M$. This of course yields quadratic time and space complexity.[1]

2.4 Score & Penalty

Since the intron-penalty γ is independent of the length of the intron, it must be rather large, so as to avoid that multiple consecutive gaps could pose as introns (of course, they would still need to be flanked by socalled *donor*- and *acceptor*-sites, i.e. introns must be of the form: $gt \cdots ag$).

For the comparison of characters, a socalled *Jukes-Cantor* score-matrix is used for δ, which means that a match yields $3k$ with $k > 0$, and a mismatch scores $-k$:

$$\delta(a,b) = \begin{cases} 3k & \text{, if } a = b \\ -k & \text{, if } a \neq b \end{cases} \tag{5}$$

If we then choose a gap-penalty such that $\lambda < -k/2$, then a mismatch scores higher than two gaps, so it rates better to have a substitution of one character in either one of the genes, than it does to have a gap in both.

[1] As noted in [1], Hirschberg's technique can be used to bring down the space complexity, from quadratic to linear space-usage.

2.5 Base-Case of Recursion

Because of their index-references, $I^a(0,j)$ is undefined for all $j \in \{1,\cdots,M\}$, $I^b(i,0)$ is undefined for all $i \in \{1,\cdots,N\}$, and $S(0,0)$ is also undefined.

As we are only interested in valid exons, it will become obvious in section 2.7 on back-tracking, that there are only certain paths in the above tables which will be traversed. In particular, the last line of Eq.(4) restarts the alignment of an eukaryotic gene, so upon back-tracking the tables, we should stop when encountering a cell (h,k) with $S(h,k)=0$ and where $G^{start}_{h,k}$ is true.

Assume that h' and k' is the lowest pair of indices such that $G^{start}_{h',k'}$ is true, and therefore is the first alignment of a potential gene's start-signals. Because everything before (h',k') can not be part of a legal gene, it is considered irrelevant as long as $S(h',k')$ is assigned a value of zero in Eq.(4). So the values for the previous positions in the I^a, I^b, and S tables (including their undefined values), must simply be chosen so that they will not interfere and cause $S(h',k') \neq 0$.

Even for the case of the two DNA sequences being identical: $S_a = S_b$, it will suffice to choose a very low number for the undefined values. For example, as done in this implementation which operates with 32-bit signed integers, we may set $S(0,0) = I^a(0,j) = I^b(i,0) = -2^{30}$, and with a matching-score of 9, the DNA sequences would require approximately 1.19e8 matches for the score to erroneously become positive without having met start-signals. A similar high count of penalties is needed for the integer to wrap around and become positive. So for all practical purposes, this choice of number for designating the undefined numbers is sufficient.

2.6 Back-Tracking

As is common with alignments obtained by dynamic programming, the actual alignment can be found by back-tracking the score-table from the maximum score and backwards through the table. With the above alignment method for gene finding however, we have three tables, which requires further explanation.

First off, we search $S(i,j)$ for a position (i',j') which maximizes $S(i',j')$ while also having stop signals after those positions in the DNA sequences:

$$(i',j') = \underset{(i,j):G^{stop}_{i,j}}{\operatorname{argmax}} S(i,j) \tag{6}$$

Once this position (i',j') has been found, we have to perform the actual back-tracking until we reach the first position[2] (h,k) for which $S(h,k)=0$, in which case we have obtained the gene with the highest similarity, starting in (h,k) and ending in (i',j'), excluding the enclosing start- and stop-signals. What remains is to find the intermediate steps that will bring us from (i',j') back to (h,k).

[2]By first position, is meant the highest h and k, and hence the first from the perspective of back-tracking.

2.7 Back-Tracking Algorithm

Having located (i', j') from Eq.(6), we need to back-track through our computations, to obtain the alterations that make one sequence match the other.

To begin with, we are situated at $S(i', j')$ as this was found to be the end of an optimal gene-alignment. From Eq.(4) we have that back-tracking may continue to a neighbouring position in the S-table, either $S(i'-1, j'-1)$, $S(i'-1, j')$ or $S(i', j'-1)$, but if the highest-valued sub-expression in Eq.(4) came from tracking an intron through tables I^a or I^b, then we should jump to the respective table and position, and continue back-tracking there. This way, when placed at cell $S(i, j)$ for some (i, j), we will continue aligning exons, or introduce introns when they improve the scoring.

When back-tracking the table I^a at position (i, j), we require:

$$D_i^a \wedge (S(i-1, j) + \gamma > I^a(i-1, j))$$

to be satisfied, before we have found the beginning of the intron, both in regards to a donor-site and optimal scoring – and can leave the I^a table and continue back-tracking the S-table again, this time from position $(i-1, j)$. Since the existence of all signals is ensured, and we always follow the path with the highest score, we obtain an optimal alignment with consistent signals. Back-tracking I^b works in the same way.

First recall that the I^a, I^b, and S tables are indexed with $i \in \{0, \cdots, N\}$ and $j \in \{0, \cdots, M\}$. The recursive back-tracking is then started by calling `BackTrack_S(i',j')` with (i', j') from Eq.(6), and the function being defined as:

- Function `BackTrack_S(i,j)`:
  - If $(i < 1 \wedge j < 1)$ terminate recursion (i.e. return from function).
  - If $(i > 0 \wedge j > 0)$ and $S(i, j) = S(i-1, j-1) + \delta(a_i, b_j)$ then flag a_i and b_j as being exons and call `BackTrack_S(i-1,j-1)`. Afterwards, terminate recursion.
  - If $(i > 0)$ and $S(i, j) = S(i-1, j) + \lambda$ then flag a_i as being an exon and call `BackTrack_S(i-1,j)`. Afterwards, terminate recursion.
  - If $(j > 0)$ and $S(i, j) = S(i, j-1) + \lambda$ then flag b_j as being an exon and call `BackTrack_S(i,j-1)`. Afterwards, terminate recursion.
  - If $(i > 0 \wedge A_i^a)$ and $S(i, j) = I^a(i-1, j) + \gamma$ then flag a_i as being an intron and call `BackTrack_Ia(i-1,j)`. Afterwards, terminate recursion.
  - If $(j > 0 \wedge A_j^b)$ and $S(i, j) = I^b(i, j-1) + \gamma$ then flag b_j as being an intron and call `BackTrack_Ib(i,j-1)`. Afterwards, terminate recursion.
  - Finally, if $(i > 0 \wedge j > 0 \wedge G_{i,j}^{start})$ and $S(i, j) = 0$, then set $(h, k) = (i, j)$ indicating the beginning of the gene, and terminate recursion afterwards.

The recursive function for back-tracking an intron in sequence S_a, is then:

- Function `BackTrack_Ia(i,j)`:
  - If $(i < 1)$ terminate recursion.
  - Flag a_i as being an intron.
  - If D_i^a and $(S(i-1,j)+\gamma > I^a(i-1,j))$ then call `BackTrack_S(i-1,j)` to end the intron, otherwise call `BackTrack_Ia(i-1,j)` to continue backtracking the intron.

And the recursive function for back-tracking an intron in sequence S_b is similar:

- Function `BackTrack_Ib(i,j)`:
  - If $(j < 1)$ terminate recursion.
  - Flag b_j as being an intron.
  - If D_j^b and $(S(i,j-1)+\gamma > I^b(i,j-1))$ then call `BackTrack_S(i,j-1)` to end the intron, otherwise call `BackTrack_Ib(i,j-1)` to continue backtracking the intron.

Depending on the desired behaviour, flag the gene's start-signals $S_a[h-2\cdots h]$ and $S_b[j-2\cdots j]$ as exons or introns, and equally so for the gene's stop-signals $S_a[i'+1\cdots i'+3]$ and $S_b[j'+1\cdots j'+3]$.

Deciding whether a legal gene was found, can be done by letting each of the functions return a boolean value indicating this. The function `BackTrack_S(i,j)` will then return a value of `true`, whenever the last of its cases was encountered, that is, when the start-signals of two aligned genes were found and $S(i,j) = 0$, and otherwise return false. This value will then be propagated back through the recursive calls, until it will finally be the value of the initial call to `BackTrack_S(i',j')`. Alternatively, we may initially set $(h,k) = (-1,-1)$, and if they are different after back-tracking has finished, then the beginning of a gene was clearly found. Note that we do not explicitly require $S(i',j') > 0$ for a legal gene to be found, all we need is for (i',j') to maximize Eq.(6), and for the last case of Eq.(4) to be met during back-tracking.

2.7.1 Directional Tables

The expressions in Eqs.(2,3,4) are calculated twice each with the above algorithms – once when computing the tables, and once during back-tracking. So to save computational time, we can store the results obtained when filling the tables I^a, I^b, and S, in some directional tables used during back-tracking.

That is, for each of the values $I^a(i,j)$, $I^b(I,j)$, and $S(i,j)$, store an identifier in tables B_{I^a}, B_{I^b}, and B_S, indicating which of the sub-expressions in Eqs.(2,3,4), respectively, that scored highest and thus identifies in which direction to back-track, and whether to switch table during back-tracking.

This implementation however, only uses such a directional back-tracking table for the S-table, because the sub-expressions for I^a and I^b are so easily computed, that it seems excessive to use storage to replace them.

It should be noted that since several sub-expressions may share the place as having the largest value, we may choose from any one of those viable paths during back-tracking. Storing this possibility in a table, can be done using a single bit for each possible back-tracking direction, and logically or'ing in new bits as needed. During back-tracking, a direction can then be chosen at random (or according to some preference), from the directions for which the designated bits have been set. This implementation uses such a bit-storage, although it selects the direction in which to continue the back-tracking, according to deterministic preference, thus following exons whenever possible.

2.7.2 Loops

The majority of recursive calls in the above functions, will simply decrement one or both of the indices i and j (while flagging the respective characters as introns or exons). The overhead involved in performing recursive function calls is therefore much higher than necessary, as the functions can be made into loops instead of being self-recursive, and thus only retain their mutual recursiveness, for changing which table is currently being back-tracked. The *loopified* version of `BackTrack_Ia(i,j)` becomes:

- Function `BackTrack_Ia(i,j)`:
  - Repeat the following while $(i > 0)$:
    - Flag a_i as being an intron.
    - If D_i^a and $(S(i-1,j)+\gamma > I^a(i-1,j))$ then call `BackTrack_S(i-1,j)` to end the intron, and afterwards return from `BackTrack_Ia()`.
    - Otherwise continue backtracking by decreasing the index:

$$i \leftarrow i - 1$$

And similarly for `BackTrack_Ib(i,j)` and `BackTrack_S(i,j)`.

2.8 Discarding Introns

Before back-tracking is performed and characters of the two DNA sequences are marked as being parts of either exons or introns, we initially mark them all as being parts of introns. After back-tracking has found valid genes, we then wish to output the exons for each of the sequences, thus discarding the introns. The algorithm that outputs the indicies for where exons begin and end, is as follows for S_a:

- First let a boolean $wasExon$ be false, and set the index i to 1.
- Repeat the following until $i = N$:

  - Let $isExon$ be a boolean signifying whether the character a_i is part of an exon or intron.
  - If $(isExon \wedge \neg wasExon)$ then output i as the beginning of an exon.
  - Otherwise if $(wasExon \wedge \neg isExon)$ then output $i-1$ as the end of an exon.
  - Assign $wasExon \leftarrow isExon$, and update the index $i \leftarrow i+1$.

- If $wasExon$ and $N > 0$, then the last exon ends at index N, so output N.

And similarly for S_b.

3 Extensions

The basic gene alignment method described above, can be extended in several ways, so as to uncover more likely structures for eukaryotic genes.

We are primarily interested in what *amino acids* are encoded by the exons. An amino acid is encoded by three consecutive *nucleotides* (each represented as one of the characters a, c, t or g) called a *codon*, and in [2, Table 1.2 p.10] it is illustrated how several different codons actually encode the same amino acids, for example ggg, gga, ggc, and ggt all encode *Gly* which is short for *Glycine* (see [2, p.3]). A better alignment method, should therefore take this into account when comparing the two DNA sequences.

Furthermore, gaps in exons are believed to mostly occur in multiples of three, as they would otherwise cause a socalled *frame-shift* of the following codons. Therefore, the combined length of the exons in a DNA sequence, should be a multiple of three.

3.1 Tracking Codon Position

To implement these requirements, we modify and extend Eqs.(2,3,4). First off, let us introduce a counter for which position in a codon we are considering, and denote it by $p \in \{1, 2, 3\}$. Since we assume no prior knowledge of where the signals of the DNA sequences are located, we can not assume that $S_a[h-2 \cdots h]$ and $S_b[k-2 \cdots k]$ are placed such that e.g. $h \mod 3 = 0$. The solution is therefore to compute the scores for each (i, j), assuming in turn that they are the 1st, 2nd, and 3rd position p in a codon. When all scores have been computed, we then back-track the tables so that the start and stop signals are aligned in regards to their codon-positions p.

3.2 Recursive Equations

We need three to each of the tables in Eqs.(2,3,4), one for each possible codon-position p. The following equations force gaps to occur in multiples of three,

although this is an artificial restriction, and should be replaced by a mere penalty on gaps of other lengths. The modified equations are:

$$I_p^a(i,j) = \max \left\{ \begin{array}{ll} \iota: & I_p^a(i-1,j) \\ \iota: & S_p(i-1,j)+\gamma \quad \text{, if } D_i^a \end{array} \right. \tag{7}$$

$$I_p^b(i,j) = \max \left\{ \begin{array}{ll} \iota: & I_p^b(i,j-1) \\ \iota: & S_p(i,j-1)+\gamma \quad \text{, if } D_j^b \end{array} \right. \tag{8}$$

$$S_p(i,j) = \max \left\{ \begin{array}{lll} \varepsilon: & S_{p\ominus 1}(i-1,j-1)+\delta(a_i,b_j) & \text{, if } p \neq 2 \\ \varepsilon: & S_{p\ominus 1}(i-1,j-1)+\delta(a_i,b_j) & \\ & +\Delta(i,j) & \text{, if } p = 2 \\ \varepsilon: & S_p(i-3,j)+3\cdot\lambda & \\ \varepsilon: & S_p(i,j-3)+3\cdot\lambda & \\ \iota: & I_p^a(i-1,j)+\gamma & \text{, if } A_i^a \\ \iota: & I_p^b(i,j-1)+\gamma & \text{, if } A_j^b \\ & 0 & \text{, if } G_{i,j}^{start} \wedge p = 3 \end{array} \right. \tag{9}$$

which are computed from row $i = 0$ up to N, and for each row we compute from column $j = 0$ up to M, and for each cell (i, j) we compute for $p = 1$ up to 3. The base case for the values that are undefined in the recursive equations, are again assigned as in section 2.5.

Notice that we define our alignment such that the start-signals occur when $p = 3$, which means that p counts the correct positions in codons, and we therefore have the middle of a codon whenever $p = 2$, so the comparison of amino acids can be done e.g. when $p = 2$ (also assuming of course, that $1 < i < N$ and $1 < j < M$), and is represented by the function $\Delta(i, j)$ (see sections 4.1 and 5 for details), thus comparing the amino-acids for sub-sequences $S_a[i-1 \cdots i+1]$ and $S_b[j-1 \cdots j+1]$. The decremental operator on p is cyclic:

$$p \ominus 1 = \left\{ \begin{array}{ll} 3 & \text{, if } p = 1 \\ p-1 & \text{, else} \end{array} \right.$$

3.3 Avoiding Stop-Codons

Stop signals are disallowed in the exons of a gene (and possibly in the introns as well), which can easily be ensured in the extended method, by severely penalizing its occurence. Since any codon must be aligned frame-wise with the start-codon, that is, their distance must a multiple of 3, then we will only penalize stop-codons for $p = 3$. It would perhaps seem that $\Delta(i, j)$ could be used for this, but as it is enclosed in a $\max(\cdot)$ expression, it is highly unlikely that the penalty will propagate to $S_p(i, j)$. Therefore, after $S_p(i, j)$ has been computed as in Eq.(9), we will assign it a penalty provided $p = 3$ and either $S_a[i-3 \cdots i]$ or $S_b[j-3 \cdots j]$ or both are in G^{stop}. The penalty is simply chosen to be the same as the *undefined* value in section 2.5, namely -2^{30}. If stop-signals should be disallowed from introns also, then I_p^a and I_p^b should be assigned penalties as well.

3.4 Back-Tracking

Since the start-signal is only allowed for $p = 3$, and we require the combined length of the exons to be a multiple of 3, we naturally wish to find the optimal alignment, by locating the stop signal for which $p = 3$. The best end-point of an optimally scoring gene, is therefore given by:

$$(i', j') = \underset{(i,j):G^{stop}_{i,j}}{\operatorname{argmax}} S_3(i, j) \tag{10}$$

Back-tracking is then performed in much the same way as in section 2.7, starting with a call to `BackTrack_S(i',j',3)`, which is given by:

- Function `BackTrack_S(i,j,p)`:
  - If $(i < 1 \wedge j < 1)$ terminate recursion (i.e. return from function).
  - If $(i > 0 \wedge j > 0 \wedge p \neq 2)$ and $S_p(i, j) = S_{p\ominus 1}(i-1, j-1)+\delta(a_i, b_j)$ then flag both a_i and b_j as being exons and call `BackTrack_S(i-1,j-1,p` $\ominus$ `1)`. Afterwards, terminate recursion.
  - If $(i > 0 \wedge j > 0 \wedge p = 2)$ and $S_p(i, j) = S_{p\ominus 1}(i - 1, j - 1) + \delta(a_i, b_j) + \Delta(i, j)$ then flag both a_i and b_j as being exons and call `BackTrack_S(i-1,j-1,p` $\ominus$ `1)`. Afterwards, terminate recursion.
  - If $(i > 2)$ and $S_p(i, j) = S_p(i - 3, j) + 3 \cdot \lambda$ then flag a_i, a_{i-1}, and a_{i-2} as being exons and call `BackTrack_S(i-3,j,p)`. Afterwards, terminate recursion.
  - If $(j > 2)$ and $S_p(i, j) = S_p(i, j - 3) + 3 \cdot \lambda$ then flag b_j, b_{j-1}, and b_{j-2} as being exons and call `BackTrack_S(i,j-3,p)`. Afterwards, terminate recursion.
  - If $(i > 0 \wedge A^a_i)$ and $S_p(i, j) = I^a_p(i - 1, j) + \gamma$ then flag a_i as being an intron and call `BackTrack_Ia(i-1,j)`. Afterwards, terminate recursion.
  - If $(j > 0 \wedge A^b_j)$ and $S_p(i, j) = I^b_p(i, j - 1) + \gamma$ then flag b_j as being an intron and call `BackTrack_Ib(i,j-1,p)`. Afterwards, terminate recursion.
  - Finally, if $(i > 0 \wedge j > 0 \wedge p = 3 \wedge G^{start}_{i,j})$ and $S_p(i, j) = 0$, then set $(h, k) = (i, j)$ indicating the beginning of the gene, and terminate recursion afterwards.

The recursive functions for back-tracking the introns, are very similar to those in section 2.7, with the exception being the inclusion of the codon-position p. Back-tracking an intron in sequence S_a, is therefore done as follows:

- Function `BackTrack_Ia(i,j,p)`:
  - If $(i < 1)$ terminate recursion.
  - Flag a_i as being an intron.

– If D_i^a and $(S_p(i-1,j)+\gamma > I_p^a(i-1,j))$ then call `BackTrack_S(i-1,j,p)` to end the intron, otherwise call `BackTrack_Ia(i-1,j,p)` to continue backtracking the intron.

And the recursive function for back-tracking an intron in sequence S_b is similar. Again, these recursive functions can be loopified as described in section 2.7.2, to improve performance.

4 Implementation

Implementation is done in *Microsoft Visual C++ .NET* and compiles to an *MS Windows* executable. There are two things to keep in mind during development: Performance and maintainability. Focus in this project has been on maintainability, because great increases in performance should first be obtained from abstract algorithmic improvements, before considering machine-dependent optimizations. However, section 4.5 briefly describes the importance of correct memory-usage, which is shown in section 5.3 to almost halve the execution time.

Maintainability is achieved through readable source-code which includes proper inline commenting. Furthermore, debugging is easened by adding assertions to most functions. These are automatically removed from the compiler-optimized release-build, and merely serve to trap erroneous situations, as well as indicating to the programmer, what parameters are valid for a given function.

Some code has been reused from the first mandatory project of this course, however modified and rearranged slightly, for example `LScore` and `LScoreMatrix`.

4.1 Classes

Starting with the simplest class, `LScore` is the abstract class for the δ-function, that matches individual characters, and is inherited in the `LScoreMatrix` subclass, which can read a score-matrix in socalled *Phylip*-format. The alphabet supported by the gene-finder, is automatically read from this matrix, and every unknown character encountered in the DNA sequences, is taken to be the first character defined in the score-matrix.[3] A score-matrix is also used for matching amino acids, with `X` indicating a stop-signal, and `*` indicating an unknown codon. Finding signals and codons in the DNA sequences, naturally assumes that their alphabet has the relevant characters.

The class `LSequenceDNA` is used to hold a DNA sequence, and reflects various aspects of such a sequence, including functionality for deciding where signals are present or absent, and whether the characters of the sequence are exons or introns, and what amino acids the codons potentially encode (this uses routines from `AminoAcid.h` to convert strings of characters to single letters designating amino acids). Note that a more efficient implementation would combine the 5 arrays of booleans used here, so as to save memory. The STL[4] has a template

[3] The function $\Delta(i,j)$ could take unknown characters into account when comparing entire codons, so that the most likely characters were chosen.

[4] C++'s Standard Template Library.

class named `bitstring<T>`, but the argument `T` is the size of the bitstring, so it is not suited as a member-field of a class due to `T` being required at compile-time, and a custommade but similar data-structure would therefore be required.

The class `LGeneFind` is the base-class for implementing the algorithms that compute the score tables and back-track them. The implementation follows the description closely, with the function `CalcAlignment()` containing the main-loop for computing the dynamic-programming tables, and using the functions `UpdateIa(i,j)` and `UpdateIb(i,j)` to compute the values for $I^a(i,j)$ and $I^b(i,j)$ respectively. Note that these functions are implemented (not specialized) in the sub-classes described below. The function `UpdateTrackAndMax()` simplifies the notation of the main-loop, by updating the temporary score-value, and resetting or or'ing in new directions for the back-tracking table. The function `BestGeneEnd()` locates the best ending position of a gene, and `CalcExons()` performs the back-tracking by calling the mutually recursive (and loopified) `BackTrackS`, `BackTrackIa`, and `BackTrackIb`, also implemented anew in each of the sub-classes.

4.2 Basic & Extended Algorithm

The two classes `LGeneFindBas` and `LGeneFindExt` derive from `LGeneFind`, and implement the basic and extended gene finding algorithms. Their base-class has a number of pure virtual functions, including `BackTrack(i,j)` which starts the back-tracking, `GetEndScore(i, j)` which returns the score $S(i,j)$ for the basic model, and $S_3(i,j)$ for the extended model, as used in locating the position (i',j') from Eqs.(6,10). The structure of both the dynamic programming algorithms but also the back-tracking algorithms lend themselves somewhat towards specialization, but implementing them wholly in each of the sub-classes, is considered to increase maintainability. An example of specializing functionality can however be found in the function `BestGeneEnd()`, which calls `GetEndScore()` mentioned above.

4.3 Entry-Point & Main Program

The entry-point is found in the file `bioinf_genefind.cpp`, which also holds the main program that prompts the user for input,[5] then opens the input sequence files and loops through them, allocating objects of `LSequenceDNA` and `LGeneFindBas/Ext` for each sequence pair to execute the appropriate functions, and then outputting the resulting exons in a GFF-formatted file.

4.4 Exceptions

To make the program more robust, exception handling was implemented at points where exceptions could be expected. In particular, a `try-catch` encloses the allocation of the gene-finder and sequence objects, as well as the alignment

[5] Unless a special testing flag has been set in the source-code, in which case default parameters are used, which is practical during development.

and back-tracking. This means that an exception arising in any of this, will cause that sequence to be ignored, and continue with the next without aborting the entire program – at least in theory.

To avoid memory-leaks, the classes are provided with simple mechanism (`DoAllocate()` and `DoDelete()` functions) for cleaning up after an allocation raised an exception due to the input sequences being too long. Unfortunately *MS Windows XP* appears to have problems handling this clean-up when the swap-disk has been exhausted, so even though the program continues, there is no memory available, and the rest of the input sequences will therefore be skipped, however gracefully.

4.5 Memory Usage

An important aspect with $O(N \cdot M)$ memory-usage, is of course to limit memory-usage by choosing appropriate (and small) data-types, but also to interleave memory correctly, to avoid *thrashing* the cache – this is especially important when working with large sequences, for which the necessary data will not fit in RAM, but must be swapped to disk. For example, the original implementation of `LGeneFindBas` had the following arrays for the tables S, I^a, I^b, and B_S:

```
int     **mS;
int     **mIa;
int     **mIb;
char    **mTrack;
```

For $N = M = 1e4$ and each `int` occupying 4 bytes, we therefore require $4e8$ bytes (approximately 381 MB) for each of the first three tables. So filling in these tables and first accessing single elements in `mS`, then in `mIa`, `mIb`, and finally in `mTrack`, we should expect very few cache hits. A better way of structuring this, is to have another class storing a cell of each table, for example:

```
class LCell
{
public:
    LCell   () {}

    int     mS;
    int     mIa;
    int     mIb;
};
```

and then have the following array-of-arrays in the `LGeneFindBas` class:

```
LCell   **mCells;
```

which is expected to exploit memory-caches far more, as the data for the different tables is now interleaved. `LCell` does not include `mTrack`, because it might cause misalignment[6] of the integers in `LCell`, which would require an additional 3

[6] For computational efficiency in retrieving and storing data on some machine architectures, data should be placed at an address which is modulo the size of the data-type.

bytes to fix.[7] However, the only way to find the implementation with the highest performance, is to actually benchmark them.

Watching the program execute, it is obvious that much time is spent allocating and de-allocating these large quantities of memory. Having the class `LCell` would also make resizing the tables easier upon reading longer DNA sequences, e.g. by using STL's `vector` class, either as:

```
std::vector<std::vector<LCell> > mCells;
```

where it should be noted that we must write a spacing between the last `> >` as it would otherwise be interpreted as the bit-wise operand `>>`. Or perhaps the following is more memory-friendly upon resizing in response to an increase in sequence lengths:

```
std::vector<LCell> mCells;
```

which would then be a single vector of size $N \cdot M$, thus requiring some simple arithmetic work to obtain an index k from (i, j) (namely $k = i \cdot M + j$).

4.6 Testing

First the implementation was tested for erroneous behaviour (i.e. *bugs*), which was aided by the extensive use of assertions, revealing errors in parameters e.g. as a result of incorrect loop-conditions. Various aspects of memory-usage were also tested, including the exhaustion of the swap-disk mentioned above, that revealed a weakness in the program where the exception-handling routine apparently does not get called, and therefore does not erase the allocated memory, thus creating a fatal memory-leak. Unfortunately, as the physical memory and swap disk are both exhausted, the debugger does not work in this special case, and it would be beyond the scope of this course to pursue a solution further.

Of equal importance is of course whether the algorithms compute what they are supposed to, in this case the alignment of eukaryotic genes in two DNA sequences. To this end, the data-files from section 5 were used and the validity of the output was checked by hand. That is, after running a few of the sequence pairs through the respective alignment algorithms, their outputs in form of exons, were checked to be valid in regards to signals in the sequences. That the extended algorithm proved much more accurate when compared to the referential data, increased the confidence in the correctness of the algorithms even further.

5 Experimental Results

To evaluate the quality of a gene finding method, we compare its results with referential data based on four measures:

[7] The implementation of the extended model in `LGeneFindExt` uses such an alignment fix, since it is just a matter of a single byte out of 40.

- TP (true positives) is the number of characters that were classified by the gene finding method as being parts of exons, and which are also classified as such in the referential data.
- FP (false positives) is the number of characters that were classified by the gene finding method as being parts of exons, but which are *not* classified as such in the referential data.
- TN (true negatives) is the number of characters that were classified by the gene finding method as not being parts of exons, and which are also classified as such in the referential data.
- FN (false negatives) is the number of characters that were classified by the gene finding method as not being parts of exons, but which *are* classified as being parts of exons in the referential data.

From this we may derive the socalled *correlation coefficient*:

$$CC = \frac{TP \cdot FN - FP \cdot FN}{\sqrt{(TP + FP) \cdot (TN + FN) \cdot (TP + FN) \cdot (TN + FP)}}$$

which equals -1 if all classifications of characters in the DNA sequences were incorrect, and $CC = 1$ if they were all correct.

In the following we shall use the provided data-set containing 87 homologous gene-pairs in the files `human_train.seq` and `mouse_train.seq`, for which the referential data is located in `train_all_true.gff`. We use the character scoring scheme in Eq.(5) with $k = 3$ for all experiments, and for the extended model, a *PAM 250* matrix from [3] is used. These score-matrices are located in `score01.txt` and `pam250.txt` respectively.

5.1 Displaying Results

A substantial amount of time was put into modifying the supplied *Python*-scripts to output these measures for each sequence pair, and then manipulating their output with various *Unix*-commands and so on, but the result of displaying the measures for each sequence pair was found to be of little value, as illustrated by the naturally erratic figure 1, which is difficult to read even though it only displays 31 out of 87 sequence pairs. So comparing such figures for even more sequence pairs became a moot point.

5.2 Various Penalties

A much more interesting result, comes from changing the penalties λ and γ, and see what effect this has on the qualitative measures of the method in question.[8] In the following, we only process sequence pairs for which $N \cdot M \leq 2.5e7$, so

[8] Note that all score/cost parameters (including the score-matrices) could be *heuristically* optimized, as is also indicated by the splitting of the data-set into a training- and test-set. This has not been attempted here however.

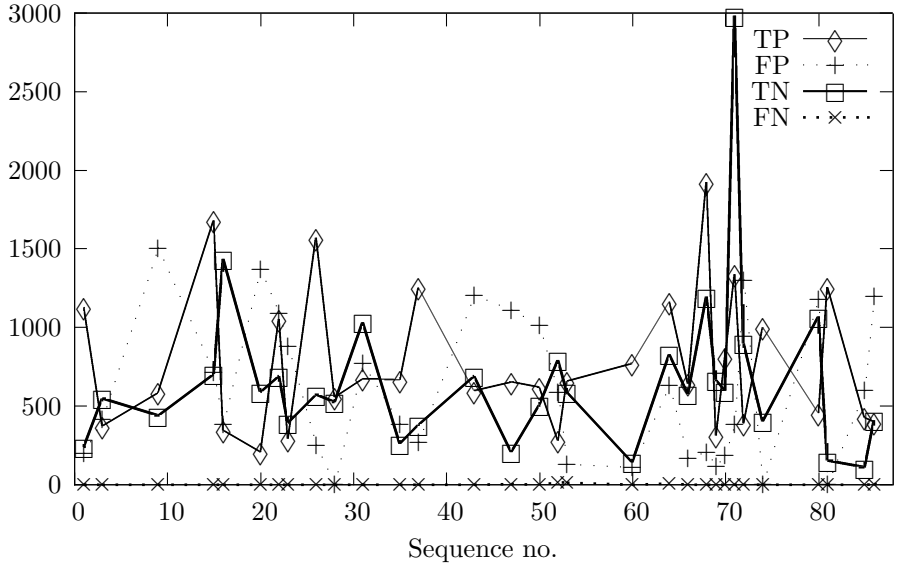

Figure 1: A run of the extended algorithm with $\lambda = -6$ and $\gamma = -30$, and only processing the 31 out of 87 sequences for which $N \cdot M \leq 3e9$. The presented measures are for the human genes only.

that both the basic and extended model process the same number of pairs, even though the basic algorithm is capable of working on much longer sequences with the same amount of computer memory.

The results in table 1 clearly show that the extended algorithm is superior over the basic algorithm, and moreover, that the extended algorithm is nearly consistent in its improvement of the number of false positives and true negatives, with increased penalties λ and γ.

5.3 Memory Usage

Two different ways of organizing the tables in memory were described in section 4.5. Experiments with the basic gene finding algorithm, where only sequence pairs for which $N \cdot M \leq 1e8$ (meaning that 83 out of 87 sequence pairs were processed) were considered, yields that the `LCell` arrangement uses roughly 1778 seconds to find the genes, while having multiple consecutive arrays uses 3497 seconds. So this fairly simple rearranging of memory, nearly cut the time-usage for the algorithm in half.

References

[1] Comparative Methods for gene structure prediction in homologous sequences, Christian N. S. Pedersen and Tejs Scharling, Technical Report, July 2 2002

[2] Introduction of Computational Molecular Biology, J. Setubal and J. Meidanis, Brooks/Cole Publishing Company, 1997

[3] Centre for Molecular and Biomolecular Informatics. `http://www.cmbi.kun.nl/bioinf/tools/pam.shtml`

	(λ, γ)	TP	FP	TN	FN	CC
Basic	(-6,-18)	49203/49355	87656/85838	40249/43897	27/436	0.3357/0.3422
	(-6,-30)	49205/49368	88287/86149	39618/43586	25/423	0.3324/0.3408
	(-6,-60)	49226/49208	89372/86854	38533/42881	4/583	0.3270/0.3332
	(-12,-36)	49206/49239	84821/83354	43084/46381	24/552	0.3511/0.3529
	(-12,-60)	49182/49256	85498/83776	42407/45959	48/535	0.3469/0.3510
	(-12,-120)	49122/49315	86779/84235	41126/45500	108/476	0.3385/0.3499
Extended	(-6,-18)	49109/49161	79525/78378	48380/51357	121/630	0.3775/0.3782
	(-6,-30)	49148/49268	80524/78877	47381/50858	82/523	0.3730/0.3779
	(-6,-60)	49158/49397	82494/79984	45411/49751	72/394	0.3625/0.3748
	(-12,-36)	49105/49124	68969/68491	58936/61244	125/667	0.4354/0.4320
	(-12,-60)	49085/49189	68923/68465	58982/61270	145/602	0.4353/0.4335
	(-12,-120)	49088/49191	64630/63612	63275/66123	142/600	0.4596/0.4610

Table 1: Accumulated quality measures for the basic and extended gene alignment methods with various gap- and intron-costs. Only DNA sequence pairs for which $N \cdot M \leq 2.5e7$ were considered, which means 60 out of 87 pairs. Measures are presented as human/mouse for both human- and mouse-genes respectively.