Available online at www.sciencedirect.com
SCIENCE DIRECT
Information and Software Technology 47 (2005) 965–978
INFORMATION AND SOFTWARE TECHNOLOGY
www.elsevier.com/locate/infsof

Engineering a software tool for gene structure prediction in higher organisms

Gordon Gremme [a], Volker Brendel [b,c], Michael E. Sparks [c], Stefan Kurtz [a,*]

[a] Zentrum für Bioinformatik, Universität Hamburg, Bundesstrasse 43, 20146 Hamburg, Germany
[b] Department of Statistics, Iowa State University, Ames, IA 50011-3260, USA
[c] Department of Genetics, Development and Cell Biology, Iowa State University, Ames, IA 50011-3260, USA

Available online 8 November 2005

Abstract

The research area now commonly called 'bioinformatics' has brought together biologists, computer scientists, statisticians, and scientists of many other fields of expertise to work on computational solutions to biological problems. A large number of algorithms and software packages are freely available for many specific tasks, such as sequence alignment, molecular phylogeny reconstruction, or protein structure determination. Rapidly changing needs and demands on data handling capacity challenge the application providers to consistently keep pace. In practice, this has led to many incremental advances and re-writing of code that present the user community with confusing options and a large overhead from non-standardized implementations that need to be integrated into existing work flows. This situation gives much scope for contributions by software engineers. In this article, we describe an example of engineering a software tool for a specific bioinformatics task known as spliced alignment. The problem was motivated by disabling limitations in an original, ad hoc, and yet widely popular implementation by one of the authors. The present collaboration has led to a robust, highly versatile, and extensible tool (named GenomeThreader) that not only overcomes the limitations of the earlier implementation but greatly improves space and time requirements.

Keywords: Computational biology; Genome annotation; Similarity-based gene structure prediction; Intron cutout technique; Incremental updates

1. Introduction

Modern biology research is characterized by the ability to study questions from a genome-wide perspective. Whereas only a decade ago a research project would typically focus on a single gene or pathway, it is now possible to view and evaluate the same genes and pathways in the context of all the genes of an organism, mapped onto the chromosomes that constitute the species' entire genetic blueprint. Of course, these possibilities require prior correct identification and annotation of all the genes, a challenging problem that has not been entirely solved [7,8]. Whereas obtaining the genetic blueprint, or, more technically, genomic DNA sequencing and assembly, is a mostly hands on, experimental process, gene annotation is largely computational, involving both statistically based prediction methods and integration of various sources of experimental and knowledge-based evidence.

This paper illustrates the development of a versatile tool for gene structure prediction, named *GenomeThreader*. We describe the algorithms utilized by *GenomeThreader*. The main algorithmic contribution of this paper is the *intron cutout technique*, which allows prediction of gene structures stretching over large regions of a genome or chromosome. Such gene structures are often present in vertebrate genomes. The intron cutout technique consists of an efficient filtering step and a dynamic programming step, and we describe how to combine them.

Unlike most papers on similar topics written for the bioinformatics community, we do not stop with the algorithms, but continue with the description of implementation aspects. We consider these aspects very important, because only well engineered software tools can cope with the ever-changing requirements and fast growing data sizes in molecular biology.

We tried to keep the description of these implementation aspects generic to allow applications to problems other than gene structure prediction. Some details and ideas presented in the implementation sections may be straightforward or even be folk knowledge for an experienced computer scientist with focus on efficient implementation of algorithms. Nevertheless, we think that it is worthwhile to describe them here for the following reasons: First, it is interesting for a general computer scientist to see how the application of software engineering principles leads to robust and versatile software, solving an important problem in bioinformatics. Second, in the fast growing and interdisciplinary field of bioinformatics, software

* Corresponding author.
E-mail address: kurtz@zbh.uni-hamburg.de (S. Kurtz).

0950-5849/$ - see front matter © 2005 Elsevier B.V. All rights reserved.
doi:10.1016/j.infsof.2005.09.005

is often developed by researchers without formal education in computer science. These researchers are mostly not aware of certain implementation techniques and software engineering principles. This paper gives a source for otherwise undocumented techniques and software engineering principles applied to a particular problem in molecular biology.

The paper is organized as follows: Section 2 gives a brief introduction to the basic biological concepts needed to understand the paper. Section 3 introduces the computational problem addressed by the *GenomeThreader* software, namely the spliced alignment problem. Section 4 describes how to compute optimal spliced alignments. Section 5 introduces the intron cutout technique, which allows prediction of gene structures stretching over large regions of a genome of a higher organism. Section 6 explains how to compute a consensus spliced alignment from a set of spliced alignments. Section 7 is devoted to implementation and software engineering aspects. We describe the data structures implemented in *GenomeThreader*, sketch interfaces and test strategies and shortly describe the software development tools we employed. Some evaluation and performance results are given in Section 8. Section 9 closes with a discussion and an outline of future work.

2. Biological background

It suffices to review a few basic concepts of molecular biology for the reader not familiar with the subject. For a more thorough introduction, the reader is referred to textbooks of molecular biology [2,15].

Chemically, DNA is a polymer composed of four different types of nucleotides, denoted by A, C, G, and T. In the computational context of this work, we treat each DNA molecule as a string over the alphabet {A, C, G, T}. These strings can be as short as 100 symbols and as long as several million. The long strings represent the chromosomes of a species, and the entire set of all strings (chromosomes) comprise the genome of that species. Of note is that most DNA exists as an antiparallel helix of two complementary DNA molecules. Here, complementarity is defined by the consistent pairing of A's with T's and C's with G's on the opposing strands, and antiparallel refers to chemical directionality of the molecule. Thus, for example, ACCGTT pairs with AACGGT.

Genes are certain substrings of the chromosome strings. Here, we only consider protein-coding genes—parts of the genome that encode information for proteins, which are another type of polymer consisting of 20 different amino acids. The familiar genetic code describes the translation from the nucleotide alphabet into the amino acid alphabet. The underlying cellular processes are quite complicated, involving first a process of transcription, which generates a copy of a genic portion of genomic DNA as an RNA molecule (pre-mRNA, yet another polymer, but for our purposes we may consider it an exact copy of specified parts of the genomic DNA string). See Fig. 1 for a schematic explanation of the process. A curious feature of most genes in animals and plants is that the RNA molecule undergoes a process called splicing by which certain stretches (called introns) are cut out of the original molecule and only the remaining parts (exons), in their original linear order, provide the basis for translation into protein, the mRNA. The processed RNA can be sampled experimentally, either as full-length molecules (termed cDNA; the term results from the fact that, for experimental reasons, the RNA is reverse transcribed back into the complementary DNA string) or as fragments (termed ESTs—Expressed Sequence Tags).

The computational approach to gene finding discussed in this paper consists of aligning cDNAs and ESTs to genomic DNA (gDNA, for short) and thereby identifying the exons and

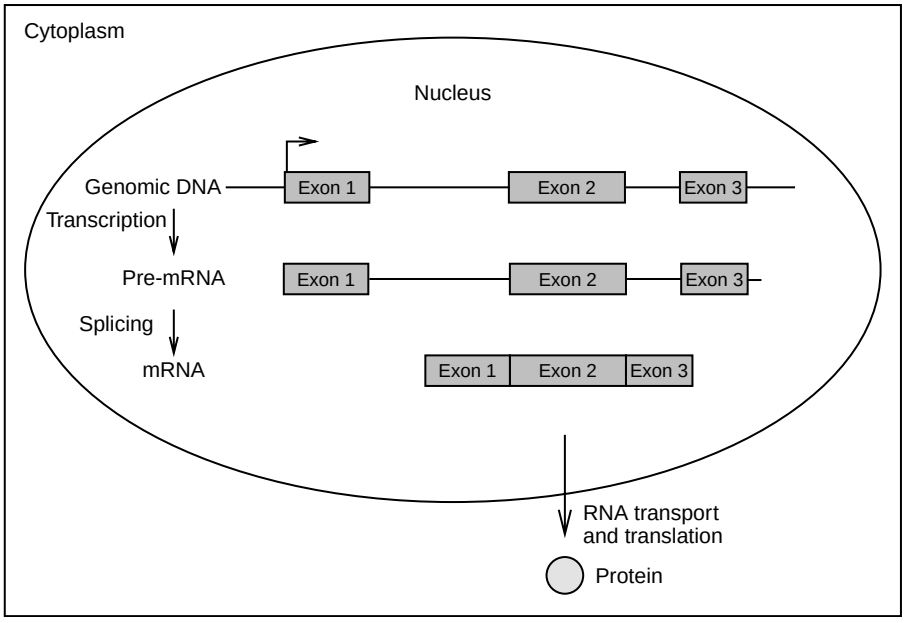

Fig. 1. Gene expression (simplified). More details are given in Refs. [2,15].

introns of genes. The problem is non-trivial because in practice the alignments sought are not necessarily of exactly matching strings. Because of natural variations (termed polymorphisms) and sequencing errors, matching sequences should tolerate several percent of single symbol mismatches as well as differences arising from insertions/deletions (indels). Solutions to such alignment tasks are of great practical importance, both for genome projects in the public domain and for projects within the pharmaceutical and biotechnology industries. The data provided for input to *GenomeThreader* software come from public domain projects. Internationally maintained public databases such as the database resources of the National Center for Biotechnology Information (NCBI) [22] or the EMBL Nucleotide Sequence Database [12] provide access to very large numbers of DNA, RNA, and protein sequences. For example, at the time of writing this article, the entire genomes of human, chimpanzee, mouse, rat, and dog are available for download, as well as several million EST sequences for human and mouse, respectively. The genomes of two plants (*Arabidopsis thaliana*, a laboratory model organism, and rice) are also available, and sequencing of the worldwide most important crop, maize, has begun. The plant EST and cDNA collections are not as extensive as for human and mouse for any given species, however, the cumulative numbers for related species are also in the millions. These sequences can be used for gene structure annotation provided the alignment algorithms are robust with respect to sequence divergence between related genes in the different species.

3. The computational problem

We now formulate the computational problem solved by the *GenomeThreader* software. In the simplest case, the input to *GenomeThreader* consists of one (typically long) gDNA sequence (supplied in any one of the most commonly used sequence file formats) and a set of cDNA/EST sequences (depending on the application, this could, for example, be a single, newly derived sequence or a large set consisting of thousands or even millions of individual sequences (each uniquely identified in the public databases)). The gDNA sequence could be several million symbols, whereas each cDNA/EST sequence would typically be about 500 symbols long and maximally about 20,000 symbols. It is unknown how many of the cDNA/EST sequences will match the gDNA sequence in some location. The alignment problem thus can be divided into two subproblems: first, identification of the cDNA/EST sequences and corresponding gDNA locations that may constitute high-quality matching pairs, and second, derivation of the optimal alignment (delineating the exons and introns in the gDNA). In *GenomeThreader*, the first task is solved by fast string matching algorithms based on enhanced suffix arrays [1], with a subsequent chaining phase combining several consistent matches. The second task involves application of classical dynamic programming [3]. The idea is to take an expressed gene product (a cDNA/EST or a protein) and perform a 'backward calculation' of the biological process shown in Fig. 1. The goal is to reveal the (previously unknown) gene structure from which the (known) product was derived. That is, one aligns the product against the gDNA allowing for introns. Therefore, this kind of alignment is called *spliced alignment*. This problem has been extensively considered over the last 10 years. Our algorithm is closely related to the *GeneSeqer* algorithm [6,20]. Other recent programs with similar capabilities are *GMAP* [23], *Genomewise* [5] *BLAT* [13], *Spidey* [21], and *sim*4 [10].

Different spliced alignments in the same region of the gDNA may not be mutually consistent. Inconsistencies of particular biological interest are different assignments of exons and introns, which may indicate physiologically significant alternative splicing. Therefore, a third task solved by *GenomeThreader* is the derivation of all possible alternative transcripts covering a particular gDNA region that are consistent with some, but not necessarily all cDNA/EST alignments in that region. This is done by a method described in Ref. [11].

3.1. Basic notions

We consider sequences over an alphabet Σ. The *length* of a sequence s, denoted by $|s|$, is the number of symbols in s. $s[a]$ is the ath symbol of s. If $a \leq b$, then $s[a\ldots b]$ is the substring of s beginning with the ath symbol and ending with the bth symbol. If $a > b$, then $s[a\ldots b]$ is the empty sequence. The *edit distance* of two sequences s and s' is the minimum number of insertions, deletions, and replacements of single symbols required to transform s into s'.

3.2. The spliced alignment problem

We consider the problem of computing an optimal spliced alignment of a gDNA $g = g[1\ldots n]$ and a cDNA/EST sequence $c = c[1\ldots m]$, both over the alphabet $\Sigma = \{A, C, G, T, N\}$, where N is the undetermined symbol.

A spliced alignment is characterized by a subset of n exon states ex_t, $t \in [1,n]$ and n intron states in_t, $t \in [1, n]$. Each of the states ex_t and in_t describes the status of position t in g. In each exon state, an output column $\begin{bmatrix}\alpha\\ \beta\end{bmatrix}$ for α, $\beta \in \Sigma \cup \{-\}$ is generated. In each intron state, an output column $\begin{bmatrix}\alpha\\ \cdot\end{bmatrix}$ for $\alpha \in \Sigma$ is generated. We use the symbol '–' for denoting deletions. That is, $\begin{bmatrix}-\\ \beta\end{bmatrix}$ denotes the deletion of symbol β from sequence g, while $\begin{bmatrix}\alpha\\ -\end{bmatrix}$ denotes the deletion of symbol α from c. The symbol '.' stands for a symbol spliced out of the gDNA.

Consider a sequence $Q = q_1, q_2, \ldots, q_k$ of intron and exon states, and let $A = \begin{bmatrix}\alpha_1 & \alpha_2 & \ldots & \alpha_k\\ \beta_1 & \beta_2 & \ldots & \beta_k\end{bmatrix}$ be the corresponding sequence of column outputs in these states, i.e. $\begin{bmatrix}\alpha_i\\ \beta_i\end{bmatrix}$ is the output in state q_i. (Q, A) is a spliced alignment of g and c if we obtain g from $\alpha_1\alpha_2\ldots\alpha_k$ and c from $\beta_1\beta_2\ldots\beta_k$ after deleting all

```
ACCGTCAAGTT–CG
|  |        ||  ||
AGC  . . . . . TTACG
```

Fig. 2. A spliced alignment where the sequence of intron and exon states is left implicit. The gDNA sequence is shown in the upper lines. Each column of the form $\begin{bmatrix}\alpha\\ \cdot\end{bmatrix}$ corresponds to an intron state. All other columns correspond to exon states. Matching symbols are denoted in the second row with the symbol '|'. Insertions and deletions are shown using the '–' symbol.

occurrences of the symbols '–' and '.'. Fig. 2 shows an artificial spliced alignment with two exons enclosing one intron.

Because we consider an optimization problem, we assign weights to each state transition in the state sequence Q and to each output column in the alignment A. The state transitions are weighted by a function w as follows:

$$
\begin{aligned}
w(ex_t, ex_{t+1}) &= \log((1-P_{\Delta_g})(1-P_{D(t+1)}))\\
w(in_t, ex_{t+1}) &= \log(P_{A(t)}(1-P_{\Delta g}))\\
w(ex_t, in_{t+1}) &= \log((1-P_{\Delta g})P_{D(t+1)})\\
w(in_t, in_{t+1}) &= \log(1-P_{A(t)})\\
w(ex_t, ex_t) &= \log(P_{\Delta g})\\
w(in_t, ex_t) &= \log(P_{A(t)}P_{\Delta g})
\end{aligned}
$$

for $t \in [1, n-1]$ (first four lines) and $t \in [1, n]$ (last two lines), respectively. All other transition weights are set to $-\infty$. $P_{\Delta g}$ denotes the probability of deleting a single symbol in g. See Fig. 3 for a graphical overview of the weight assignments. $P_{D(t)}$ reflects the probability that t is the first position of a *donor site* in g and $P_{A(t)}$ reflects the probability that t is the last position of an *acceptor site* in g. The general term for a donor or acceptor site is a *splice site*. The terms donor and acceptor site are biologically motivated. For the discussion here, it suffices to know that a donor site indicates the start of an intron, and an acceptor site indicates the end of an intron in the genomic sequence. An intron is completely specified by giving the corresponding pair of donor and acceptor sites. The calculation of the parameters $P_{D(t)}$ and $P_{A(t)}$ follows Bayesian splice site models (BSSM) described in Refs. [6,18]. Therefore, $P_{D(t)}$ and $P_{A(t)}$ are called *BSSM parameters*.

While the weight of a state transition depends on the position t, the weight of an output column is independent of t: An output column $\begin{bmatrix}\alpha\\ \beta\end{bmatrix}$ generated in an exon state is assigned a weight $w\left(\begin{bmatrix}\alpha\\ \beta\end{bmatrix}\right)$ as follows:

$$
w\left(\begin{bmatrix}\alpha\\ \beta\end{bmatrix}\right) = \begin{cases} \sigma & \text{if } \alpha, \beta \in \Sigma\backslash\{\mathrm{N}\}, \alpha = \beta \\ \mu & \text{if } \alpha, \beta \in \Sigma\backslash\{\mathrm{N}\}, \alpha \neq \beta \\ \nu & \text{if } \alpha, \beta \in \Sigma, \alpha = \mathrm{N} \text{ or } \beta = \mathrm{N} \\ \delta & \text{otherwise} \end{cases}
$$

σ denotes the identity weight, μ the mismatch weight, ν the weight for alignment columns involving undetermined symbols, and δ the deletion weight. In an intron state in_t the column $\begin{bmatrix}g[t]\\ \cdot\end{bmatrix}$ with weight 0 is generated.

The sum of the weights of all state transitions and all output columns of a spliced alignment (Q, A) is its weight, denoted by $w(Q, A)$. The spliced alignment problem is to find a spliced alignment of g and c with maximum weight, denoted by $w(g, c)$. A spliced alignment (Q, A) of g and c satisfying $w(Q, A) = w(g, c)$ is called an *optimal spliced alignment*.

Table 1 gives an overview of the parameters required to determine the weight of a spliced alignment.

4. Computing optimal spliced alignments

As with many problems in biological sequence comparison, the spliced alignment problem can be solved by a dynamic programming (DP) algorithm. This computes two $(m+1)\times$

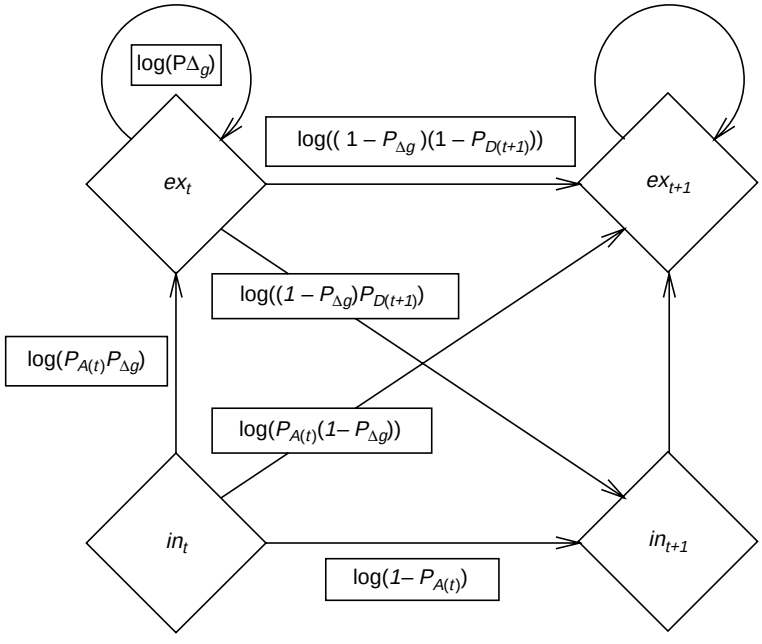

Fig. 3. States and state transitions of a spliced alignment. Adapted from Ref. [20].

Table 1
Parameters determining the weight of a spliced alignment

Parameter	Notation	Default
Initial exon state probability	$w(ex_1)$	0.5
Probability of inserting a gap in gDNA	$P_{\Delta g}$	0.03
Identity weight	σ	2.0
Mismatch weight	μ	−2.0
Weight for alignment positions involving undetermined symbol N	ν	0.0
Weight for deletions	δ	−4.0
Splice site parameter	$P_{D(t)}$, $P_{A(t)}$	From BSSM

$(n+1)$-matrices E and I such that following holds for all $i \in [0, m]$ and $j \in [0, n]$:

- E_i^j is the maximum weight of any spliced alignment of $g[1\ldots j]$ and $c[1\ldots i]$ such that the state sequence ends with an exon state.
- I_i^j is the maximum weight of any spliced alignment of $g[1\ldots j]$ and $c[1\ldots i]$ such that the state sequence ends with an intron state.

Obviously, $w(g,c) = \max(E_m^n, I_m^n)$. To simplify the computation, we introduce an additional exon state ex_0 and intron state in_0, and define $w(ex_0, ex_1) = w(in_0, ex_1) = \log(w(ex_1))$ and $w(ex_0, in_0) = w(in_0, in_1) = \log(w(in_1)) = \log(1 - w(ex_1))$. Here, $w(ex_1)$ denotes the *initial exon state probability*. Now each matrix entry can be computed by the following recurrence:

$$E_i^j = \max \left\{ \begin{array}{l} \max\{E_i^{j-1} + w(ex_{j-1}, ex_j), \\ \quad I_i^{j-1} + w(in_{j-1}, ex_j)\} + w\left(\begin{bmatrix} g[j] \\ - \end{bmatrix}\right) \\ \max\{E_{i-1}^{j-1} + w(ex_{j-1}, ex_j), \\ \quad I_{i-1}^{j-1} + w(in_{j-1}, ex_j)\} + w\left(\begin{bmatrix} g[j] \\ c[i] \end{bmatrix}\right) \\ \max\{E_{i-1}^{j} + w(ex_j, ex_j), \\ \quad I_{i-1}^{j} + w(in_j, ex_j)\} + w\left(\begin{bmatrix} - \\ c[i] \end{bmatrix}\right) \end{array} \right\}$$

$$I_i^j = \begin{array}{l} \max\{E_i^{j-1} + w(ex_{j-1}, in_j), \\ \quad I_i^{j-1} + w(in_{j-1}, in_j)\} \end{array}$$

for $i > 0$ and $j > 0$. Additionally, $E_i^j = 0$, for $j = 0$ or $i = 0$, $I_i^j = 0$ for $j = 0$, and $I_i^j = -\infty$ for $i = 0$. The first row of the I matrix (case $i = 0$) is set to $-\infty$, because cDNAs/ESTs, theoretically speaking, do not contain introns. This is also the reason why a value in matrix I only depends on two other values. Inspection of the data dependencies in the recurrence shows that each matrix entry only depends on a constant number of entries in the previous row or column. Hence, the matrices can be computed column by column or row by row. Each entry can be computed in constant time. Hence, both matrices can be computed in $O(mn)$ time, which also gives the time bound for determining $w(g, c)$. An optimal spliced alignment is recovered by tracing back from the entry $\max(E_m^n, I_m^n)$ to an entry in its multi-way maximum that yielded it, determining which entry gave rise to that entry, and so on back to the entry E_0^0. This requires saving backtrace information for each matrix entry, and leads to an algorithm that takes $O(mn)$ space. The backtracing procedure can be organized in such a way that a spliced alignment of g and c is computed in time proportional to its length.

index j	1	2	3	4	5	6	7	8	9	10	11		12	13
gDNA g	A	C	C	G	T	C	A	A	G	T	T	-	C	G
EST c	A	G	C	.	.	.	.	.	.	T	T	A	C	G
index i	1	2	3							4	5	6	7	8
states Q	ex_1	ex_2	ex_3	in_4	in_5	in_6	in_7	in_8	in_9	ex_{10}	ex_{11}	ex_{11}	ex_{12}	ex_{13}
out. weights	σ	μ	σ	0	0	0	0	0	0	σ	σ	δ	σ	σ

Fig. 4. Adapted from Ref. [20]. Hypothetical alignment of a gDNA g = ACCGTCAAGTTCG with an EST sequence c = AGCTTACG. The gDNA position j is in the range [1,13] and the EST sequence position i is in the range [1,8]. As one can see from the optimal state sequence Q, positions 4–9 of the gDNA have been assigned intron status.

Fig. 4 shows a (hypothetical) optimal spliced alignment including output column weights. The same alignment is shown in Fig. 5 as a path in the superimposed matrices E and I.

5. The intron cutout technique

When predicting the gene structure for genomic sequences of vertebrates (e.g. human or mouse) or plants one is often faced with the problem of long introns. Some known introns consist of several 10,000 or even 100,000 bases (e.g. [2,15]), and thus the dynamic programming algorithm described in Section 4 is too slow and requires too much space. On the other hand, an intron does not contribute to the overall weight of a spliced alignment. Therefore, we could skip most of the internal parts of introns in the dynamic programming algorithm, if we knew the intron locations. While the exons of a potential gene structure should be highly similar to the EST sequences derived from this genomic locus, the introns should be devoid of any but chance matches to the EST sequences. Thus, the idea is to apply a *similarity filter*: this first finds approximate matches between the gDNAs and the ESTs. Several of these matches are combined into a chain if they are compatible with each other, i.e. if they could serve as parts of a spliced alignment. On the gDNA these chains provide candidates for exons. All stretches of the gDNA not covered by a chain are considered as potential introns. They are cut out before applying dynamic programming. See Fig. 6 for a graphical explanation of this idea. In the backtracing phase of the dynamic programming algorithm, the previously cut out parts of the introns are inserted back. This produces a complete spliced alignment and thus retains the properties of the DP algorithm, allowing recognition of the exact exon/intron boundaries. Most important, the cutout technique considerably

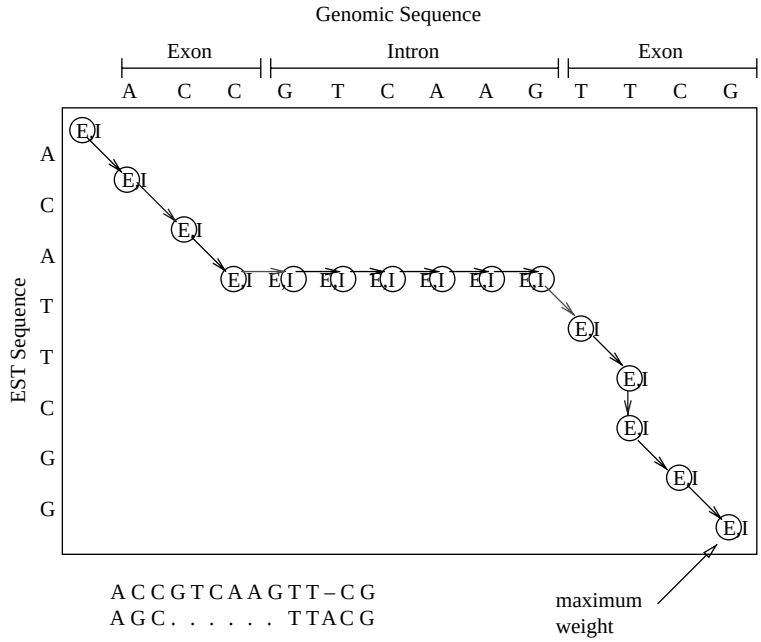

Fig. 5. An optimal spliced alignment of the sequences g = ACCGTCAAGTTCG and c = AGCTTACG represented by a path in the superimposed matrices E and I. Each node is only represented by showing the symbol E and I at the corresponding coordinate of the DP-matrices. The states of the optimal state sequence are circled. The optimal path through the matrix starts at E_0^0 and ends at E_m^n. The computed gene structure of the artificial gDNA is shown on top of the matrix.

reduces the effort for the dynamic programming algorithm. Note however, that the technique is heuristic: if an exon does not contain a sufficiently long and well conserved match, it is cut out, which leads to an incorrect gene structure prediction.

Although the cutout technique is conceptually simple, we are not aware of any software tools fully employing it for predicting gene structures. In the following, we first describe how to identify parts of the gDNA where to possibly apply the intron cutout technique. The idea is to first efficiently compute matches between the gDNA and the ESTs, and then to chain these. The chain suggests regions of the gDNA to cut out.

5.1. Computing matches

We consider *maximal approximate matches* between the gDNA g and the EST sequence c. Formally, a *maximal approximate match* is a pair of substrings $g[j\ldots r]$ and $c[i\ldots h]$ which is *left maximal* and *right maximal*. Left maximality means that $j=1$ or $i=1$ or $g[j-1]\neq c[i-1]$. Right maximality means that $r=n$ or $h=m$ or $g[r+1]\neq c[h+1]$. We are only interested in maximal approximate matches of some minimum length $\ell_{\min}$ with some maximum number of differences $d_{\max}$. That is, we require that $\min(r-j+1, h-i+1)\geq \ell_{\min}$ and $d\leq d_{\max}$, where d is the edit distance of $g[j\ldots r]$ and $c[i\ldots h]$. A standard approach to compute these approximate maximal matches is the *seed-and-extend* approach. This relies on the fact that a maximal approximate match contains at least one maximal exact match of length $\lfloor \ell_{\min}/(d_{\max}+1)\rfloor$ or longer. This is called an *exact seed*. Each maximal approximate match can be derived from an exact seed by extending this to both sides in sequence g and c. The extension is performed by a dynamic programming algorithm that allows up to $d_{\max}$ errors. See Ref. [14] for a description of the technical details. This seed-and-extend approach is implemented in the program *Vmatch* (http://www.vmatch.de/), and we utilize *Vmatch* for computing maximal approximate matches.

The basic concept of *Vmatch* is to preprocess a set of database sequences (in our case the gDNA) into an enhanced suffix array, which provides a very powerful index structure for string matching [1]. This index structure is stored on file and computed only once. Unlike traditional hashing methods (which first generate exact matches of some fixed length k and then extend these to maximal matches), *Vmatch* directly computes maximal exact matches. As a consequence, it is considerably faster than tools utilizing hashing methods.

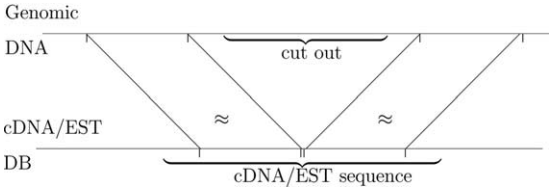

Fig. 6. A graphical explanation of the intron cutout idea.

5.2. Chaining the matches

To derive a potential exon in the gDNA, usually several approximate matches have to appear in collinear order. Therefore, the next step of our similarity filter is to chain the approximate matches. To clarify this step, we introduce some new notions. Because a match always refers to the sequences g and c, we denote it by the left and right boundaries. That is, the matching substrings $g[j\ldots r]$ and $c[i\ldots h]$ are denoted by

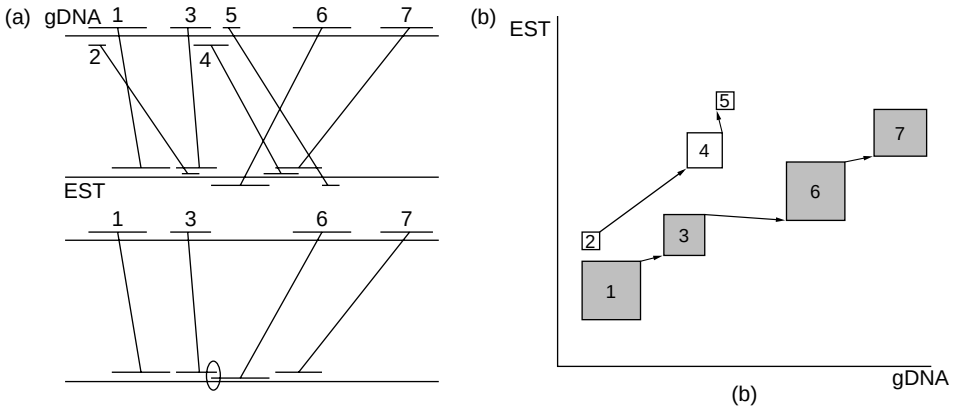

Fig. 7. Given a set of matches (upper left figure), an optimal global chain of collinear (possibly) overlapping matches (lower left figure) can be computed, e.g. by computing an optimal path in the graph in (b) (in which not all edges are shown). The overlapping part of match 3 and match 6 is circled.

$([j\ldots r], [i\ldots h])$. For any pair of matches $f=([j\ldots r], [i\ldots h])$ and $f'=([j'\ldots r'], [i'\ldots h'])$ we define a function $gap(f,f')=\max\{0, j'-r-1, i'-h-1\}$ and a binary relation $\ll$ as follows: $f\ll f'$ if and only if $j<j'$, $i<i'$, $r<r'$, $h<h'$, and $gap(f,f')\leq m$. m is the (user defined) *maximum gap width*. If $f\ll f'$, then we say that f *precedes* f'. Note that the definition of $\ll$ allows for overlaps between matches both in g and c. We want to account for these and define the *overlap* of f and f' by

$$ovl(f,f') := 2(\max(0, r-j'+1) + \max(0, h-i'+1))$$

For a given set M of matches, a *chain* is a sequence $f_1, f_2, \ldots, f_q$ such that $f_a\in M$ for $a\in[1,q]$ and $f_a\ll f_{a+1}$ for $a\in[1,q-1]$. f_1 is called *start fragment* and f_q is called *end fragment*. To obtain the score for a chain, we score matches: Each maximal approximate match $f=([j\ldots r], [i\ldots h])$ is assigned a positive score. This is defined on the basis of an optimal alignment of $c[i\ldots h]$ and $g[j\ldots r]$ without any spliced out symbols and exon/intron states. In this alignment each matching pair of nucleotides scores 2, each mismatch scores —1, and each insertion and deletion scores —2. A simple calculation shows that $score(f)=r-j+h-i+2-3d$, where d is the edit distance of the match.

The *score* of a chain C is

$$score(C) := \sum_{a=1}^{q} score(f_a) - \sum_{a=1}^{q-1} ovl(f_a, f_{a+1})$$

The *chaining problem* is to find a chain of maximum score, called an *optimal global chain*. A direct solution to this problem is to construct a weighted directed acyclic graph $G=(V, E)$, the match graph. The set V of vertices consists of all matches in M. The set E of edges is characterized as follows: There is an edge $f\to f'$ with weight $score(f')-ovl(f,f')$ if and only if $f\ll f'$; see Fig. 7(b). An optimal chain of matches corresponds to a path of maximum score in the match graph. Because the graph is acyclic, such a path can be computed as follows: Let $scoremax(f')$ be defined as the maximum score of all chains ending with f'. $scoremax(f')$ can be expressed by the recurrence:

$$score\ \max(f') = score(f') + \max\{score\ \max(f) - ovl(f,f') \mid f \ll f'\}$$

$\max\{scoremax(f') \mid f'\in M\}$ gives the maximum score of any chain, and reconstructing a chain of maximum score is an easy task. A dynamic programming algorithm based on Eq. (1) takes $O(|V|+|E|)$ time. Because $(|V|+|E|)\in O(|M|^2)$, computing an optimal global chain takes $O(|M|^2)$ time. There is a method to compute global chains with overlaps in $O(|M|\log|M|)$ time (see [17]). However, this method utilizes a different scoring scheme for matches and overlaps.

We modified this approach to find all biologically meaningful chains, and not only the one with maximum score. For each fragment $f\in M$ we keep track of the start fragment of an optimal chain ending with f. We divide all fragments into equivalence classes according to their corresponding start fragments. That is, two fragments belong to the same equivalence class, if and only if their corresponding optimal chain share the same start fragment. For every equivalence class, which contains a chain covering a minimum user defined percentage of the EST (default is 50%), we keep the chain with the highest coverage. Thus we avoid reporting multiple chains which differ only slightly. This modification allows identification of chains matching at different loci in the genome (resulting from paralogous genes).

5.3. The cutout step

For each of the stored global chains for a given EST and gDNA, we consider the regions of the gDNA covered by the matches of the chain. Each such region is extended to the right and to the left by some user-defined number of positions. This is to make sure that the splice sites of adjacent introns are kept for the dynamic programming step. Extended regions that overlap or are very close together are merged. Each region obtained in this way is a *DP region*, because it defines a substring of the gDNA to which the dynamic programming algorithm of Section 4 is applied. Technically, we create an artificial gDNA, the *spliced gDNA*, by concatenating the gDNA substrings corresponding to DP regions, in the order of the DP regions. For each border

between concatenated substrings we keep a length value, defined as the distance between the substrings in the original gDNA.

Given the spliced gDNA, the dynamic programming algorithm can be used in exactly the same manner as without the intron cutout technique. The only part which needs to be modified is the backtracking procedure. Whenever it crosses a border between different DP regions in the spliced gDNA, an intron with the length of the border is included into the spliced alignment.

6. Computing consensus spliced alignments

Spliced alignments derived from ESTs often do not cover full genes, because ESTs are usually not longer than 500 nucleotides, whereas genes can be much longer. To resolve the complete gene structure, one has to join more than one compatible spliced alignment occurring in the same region of the gDNA. The result of joining such spliced alignments may not lead to a single gene structure. This is often due to events of alternative splicing, i.e. exons or parts of exons are combined in different ways. As a consequence, simple merging of spliced alignments is not possible. Often one has to compute many different *consensus spliced alignments*. This is typically implemented as a post-processing step after all the spliced alignments have been computed. Fig. 8 shows an example of several spliced alignments occurring in the same region of the gDNA.

To calculate consensus spliced alignments, we use the method of Ref. [11]. While the original description is operational, involving the computation of set sizes, we give a more compact description of this method directly describing how certain sets are constructed.

Suppose we are given a gDNA $g=g[1\ldots n]$, a set of EST sequences, and a set of spliced alignments SA. Recall that a spliced alignment always begins and ends with an exon. Since, we consider more than one spliced alignment here, each spliced alignment in SA refers to some substring $g[j\ldots r]$ of the gDNA. Therefore, in this section, a spliced alignment is denoted by a pair (j, r) of positions in g. Of course, for each spliced alignment (j, r) we store which positions in the interval $[j, r]$ are in an exon and which are in an intron.

Two spliced alignments $(j, r), (j', r') \in SA$ *overlap* if $j \le r'$ and $j' \le r$. Consider the *overlap graph* (V, E) with the node set $V=SA$ and the edge set E defined by $(sa, sa') \in E$ if and only if sa and sa' overlap. We assume that this overlap graph is fully connected, i.e. there is at least one path from each node to each other node. Given an arbitrary set of spliced alignments, we can easily divide this into disjoint subsets such that the overlap graph for the subset is fully connected. Hence, this assumption is not a restriction of generality.

Two spliced alignments $(j, r), (j', r') \in SA$ are *compatible*, if they overlap and for all $i \in [j, r] \cap [j', r']$, i is an exon position in (j, r) if and only if i is an exon position in (j', r'). In other words, spliced alignments are compatible, if the overlapping regions are consistent with respect to exon and intron assignments. Note that compatibility is not transitive; see Fig. 9 for an example.

The *consensus spliced alignment problem* of SA is to find a minimal collection $\{SA_1,\ldots,SA_k\}$ of subsets of SA satisfying:

(1) $SA_1 \cup \ldots \cup SA_k = SA$
(2) For each $p \in [1, k]$ and each $sa, sa' \in SA_p$ sa and sa' do not overlap or sa and sa' are compatible.
(3) For each $p \in [1, k]$, SA_p is maximal with respect to compatibility, i.e. for each $sa \in SA_p$ and $sa' \in SA \backslash SA_p$, sa and sa' are not compatible.

We say that each SA_p represents a *consensus spliced alignment* or a *splice form*. A spliced alignment (j, r) *contains* a spliced alignment (j', r') if (j, r) and (j', r') are compatible and $j \le j' \le r' \le r$. Note that each spliced alignment contains itself.

The spliced alignment problem is solved by iteratively constructing the consensus spliced alignments, with the largest one beeing constructed first. For each spliced alignment (j, r) we define $L(j, r)$ as a maximal subset of SA satisfying the following conditions:

- $L(j, r)$ contains (j, r),
- for each pair $(j', r'), (j'', r'') \in L(j, r)$, (j', r') and (j'', r'') do not overlap or are compatible, and
- $j' < j$ and $r' < r$ for each $(j', r') \in L(j, r) \backslash \{(j, r)\}$.

$R(j, r)$ is defined in an analogous way, with the third condition replaced by

- $j' > j$ and $r' > r$ for each $(j', r') \in L(j, r) \backslash \{(j, r)\}$.

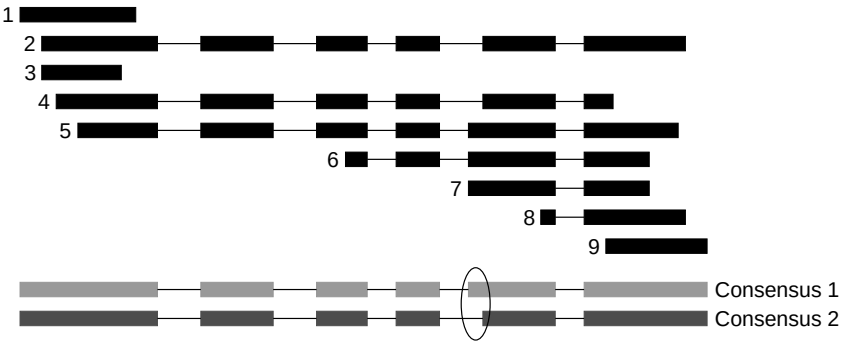

Fig. 8. Adapted from Ref. [11]. An example of consensus spliced alignments. The nine spliced alignments shown in the upper part of the figure have been processed into two consensus spliced alignments. The circled shortened exon suggests that this gene is alternatively spliced.

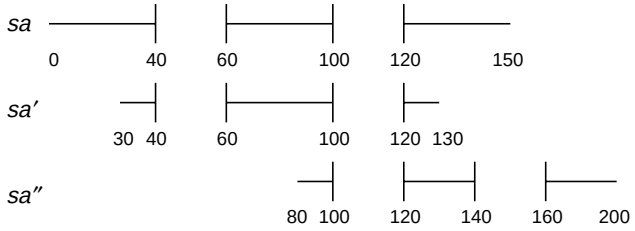

Fig. 9. Adapted from Ref. [11]. Three spliced alignments. *sa* and *sa′* are compatible, as well as *sa′* and *sa″*. *sa* and *sa″* are *not* compatible since the second intron of *sa″* overlaps with the last exon of *sa*. Thus the compatibility relation is not transitive.

The algorithm constructs a sequence of sets U_0, U_1, U_2,…, U_k such that $U_p \neq \emptyset$ for $p \in [0, k-1]$ and $U_k = \emptyset$, and a solution SA_1, SA_2,…,SA_k to the consensus spliced alignment problem as follows:

- $U_0 = SA$,
- $SA_i = L(sa_i) \cup R(sa_i)$ where $sa_i \in U_{i-1}$ satisfies $|L(sa_i) \cup R(sa_i)| \geq |L(sa') \cup R(sa')|$ for all $sa' \in U_{i-1}$,
- $U_i = U_{i-1} \backslash SA_i$.

Since $SA_i \neq \emptyset$ for all $i \geq 1$, the algorithm clearly terminates. It remains to show how to compute the *L*-sets and *R*-sets. To do so, we define

$$left(j, r) = \{(j', r') \in SA \mid j' < j, r' < r, (j', r') \text{ and } (j, r) \text{ are comparible}\}$$

$$right(j, r) = \{(j', r') \in SA \mid j' > j, r' > r, (j', r') \text{ and } (j, r) \text{ are comparible}\}$$

Then $L(sa)$ and $R(sa)$ can be computed by the following recurrences:

$$L(sa) = \begin{cases} C(sa) & \text{if } left(sa) = \emptyset \\ L(sa') \cup C(sa) & \text{if } left(sa) \neq \emptyset \end{cases}$$

where $sa' \in left(sa)$ satisfies

$$|L(sa') \cup C(sa)| \geq |L(sa'') \cup C(sa)| \text{ for all } sa'' \in left(sa)$$

$$R(sa) = \begin{cases} C(sa) & \text{if } right(sa) = \emptyset \\ R(sa') \cup C(sa) & \text{if } right(sa) \neq \emptyset \end{cases}$$

where sa′ ∈ right(sa) satisfies

$$|R(sa') \cup C(sa)| \geq |R(sa'') \cup C(sa)| \text{ for all } sa'' \in right(sa)$$

These recurrences can easily be implemented in a dynamic programming scheme tabulating $|L(sa)|$ and $|R(sa)|$ for each spliced alignment $sa \in SA$. With each *sa* one keeps track of which *sa′* gave rise to the maximum value in the corresponding recurrence. For each $sa \in SA$ one also stores $C(sa)$. A backtrace step then allows to reconstruct the splice forms by joining the appropriate sets $C(sa)$.

Consider the running time of this algorithm. For each pair of spliced alignments, we can decide in constant time if they overlap. By sorting the spliced alignments according to their start position we can also decide in $O(l)$ time if the corresponding overlap graph is fully connected, where $l = |SA|$. For two spliced alignments $sa, sa' \in SA$ assume that the start position of *sa* is smaller or equal to the start position of *sa′*. Then we check compatibility by starting at the first overlapping exon and simultaneously scanning the exons from left to right. For each exon pair we decide the consistency of exon/intron assignment in constant time. Pairs of two internal exons have to be identical. Other pairs of exons only have to have identical left or identical right boundaries. Hence, compatibility can be determined in time proportional to the number of exons in each pair of spliced alignments. Let η be the maximal number of exons in all spliced alignments. Then we can compute an $l \times l$ table storing the compatibility relation using $O(l^2\eta)$ time. Given this table, we can also decide in constant time if one spliced alignment is contained in another. The dominating step in the described algorithm is the computation of SA_1. We have to compute $L(sa)$ for l spliced alignments. For each spliced alignment we have to iterate over all $O(l)$ elements in $left(sa)$ and join it with $C(sa)$. Joining also requires $O(l)$ time. Hence, the total running time is $O(l^3 + l^2\eta)$.

7. Implementation

GenomeThreader is a command line tool with many different options. For a complete description of these options and examples of its application, we refer to the manual at http://www.genomethreader.org/. *GenomeThreader* has a modular structure. Each module implements a certain phase of the data flow, as depicted in Fig. 10. The interface between the different modules are kept small. They exchange information via a small number of different datatypes, which are described in this section. Some of these datatypes are also used in other software tools, and are therefore more general than required for *GenomeThreader*.

7.1. Multiple sequences

A datatype for handling sequences is central to all software for sequence analysis. Because *GenomeThreader* handles many sequences at the same time, we use a datatype *multiseq* for sets of sequences. A set $\{S_1,\ldots,S_k\}$ of $k \geq 1$ sequences is stored in a consecutive memory area of length $k - 1 + \sum_{j=1}^{k} |S_j|$ with a separator symbol between each adjacent pair of sequences. If necessary, we also store the reverse complement of every sequence S_i in another consecutive memory area of the same size and in the same order as the original sequences. Because the datatype *multiseq* handles sequences over alphabets of up to 254 symbols, we use one byte for each sequence character. An additional array stores the positions of the separator symbols. This array allows to map a position in the concatenated string to a position in sequence S_i using $O(\log_2 k)$ time by a binary search. Besides the sequence content, the datatype *multiseq* also stores the description of

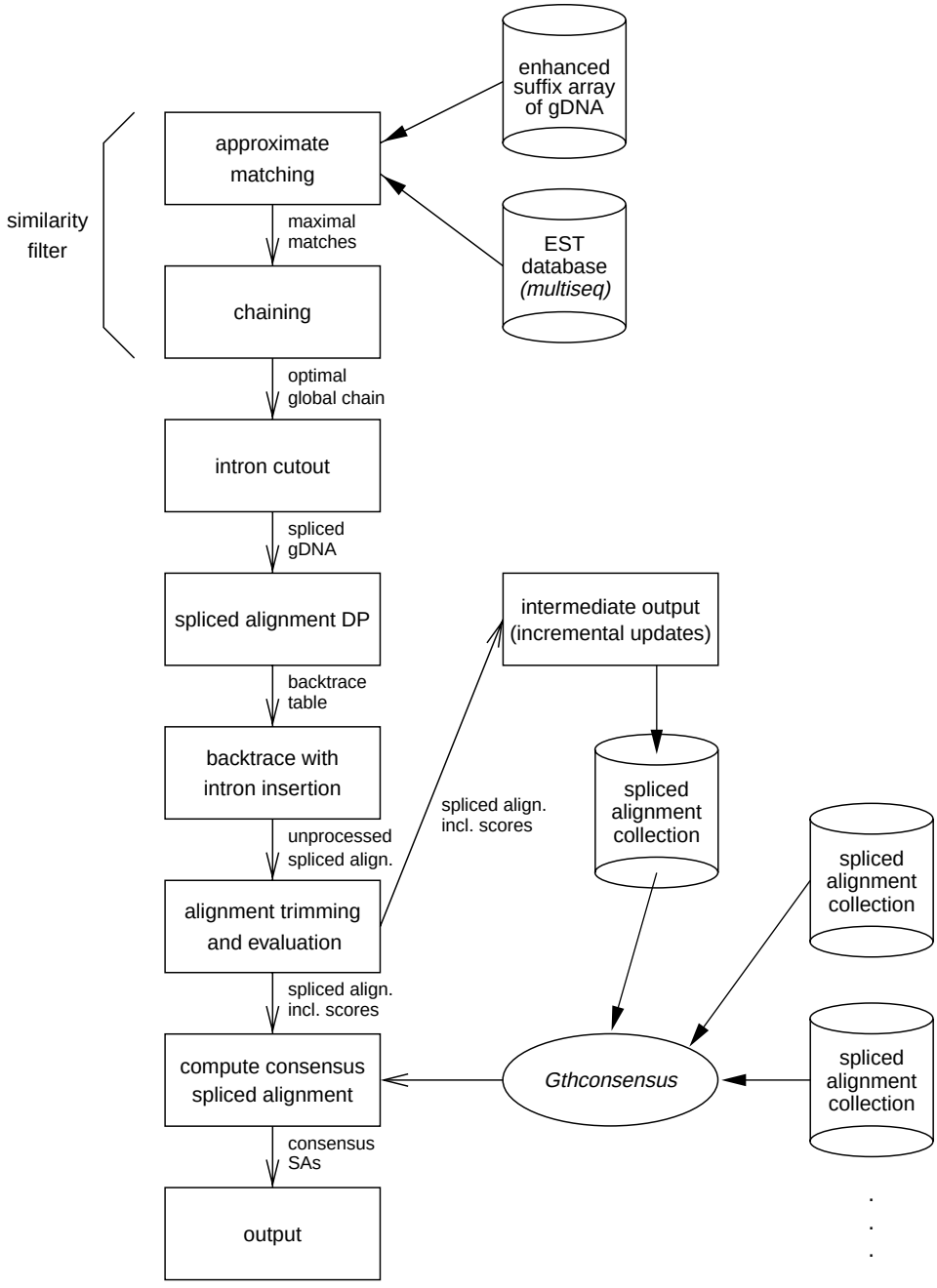

Fig. 10. Overview of the *GenomeThreader*-phases.

each sequence in one large string. The description gives basic information about the origin of the sequence and references to sequence databases. For preparing the final output, an additional array allows accession of each sequence description in constant time, given a sequence number.

7.2. Enhanced suffix arrays

An enhanced suffix array consists of several tables, which encode a tree structure storing all suffixes of a given sequence in linear space. Different algorithms require different tables from the enhanced suffix array, and so the tables are stored in separate files and mapped in memory on demand. Due to its simple structure, an enhanced suffix array is thus represented by a record of pointers which refer to the corresponding table, if this is mapped. To minimize the risk of accessing corrupted tables, we perform several simple consistency checks when mapping a table.

The construction of enhanced suffix arrays mainly consists of sorting the suffixes in lexicographic order to obtain the suffix array. In a first sorting phase, we use the counting sort

algorithm [9] to lexicographically sort all suffixes by their prefix of length d, where $d \leq \log_\sigma n$, n is the total length of all sequences, and $\sigma = |\Sigma|$ is the alphabet size. This step requires $O(n + \sigma^d) = O(n)$ time and n bytes in addition to the array storing the start positions of the suffixes (the suffix array). In the second step, we adapt the string sorting algorithm of Ref. [4] to independently sort sets of suffixes with the same prefix of length d. This algorithm is a variant of the *quicksort*-algorithm, which apart from the space for the suffix array, only requires space for a stack to store intervals left to be sorted.

7.3. Chaining

We collect all approximate matches between the EST and the genomic sequence in an array. The matches are sorted according to their start position in the genomic sequence. Then for two matches f and f', the relation $f \ll f'$ implies that f occurs to the left of f' in the sorted array. Hence, we scan the array of matches from left to right, evaluate Eq. (1) for each match f', and keep a reference to the match f which maximizes Eq. (1). We call f the previous match. Furthermore, for each match we keep a reference to the start fragment of its chain and the coverage of the chain up to this match. This allows us to divide all end fragments (these are the only one we have to consider) into equivalence classes easily. For each equivalence class, which contains a chain with sufficient coverage, we can retrieve the chain with the highest coverage by following the reference to the previous match, until we reach the first match of a chain. Each chain is represented as an array of references to the matches of a chain in the order they occur in the chain.

7.4. Dynamic programming

For a given spliced genomic sequence without cut out regions, we first calculate the BSSM parameters $P_{D(t)}$ and $P_{A(t)}$, for $t \in [1, n]$, see Section 3.2. We use an array of length $m+1$ of pairs of floating point numbers for storing a column of matrix E and matrix I. For each entry E_i^j and I_i^j, we have to store the cases of the corresponding recurrence that gave rise to the maximum value. There are six different cases for E_i^j and two different cases for I_i^j to store. Because we only have to store one case at a time, we need $\lceil \log_2(6) + \log_2(2) \rceil = 4$ bits for each index pair $(i, j) \in [0, n] \times [0, m]$. Hence, a backtrace table B of $4 \cdot (m+1) \cdot (n+1)$ bits suffices. Let $B[j][i]$ denote the 4-bit block storing the backtrace information for E_i^j an I_i^j. After computing the weights column by column and filling table B, a backtracing procedure recovers the spliced alignment encoded in B. The backtrace procedure starts at $B[m][n]$. In each step, it jumps to a value in the previous row and/or previous column, until it reaches $B[0][0]$. Each step generates an exon or intron state and an output column, making up the spliced alignment. In Section 7.5, we describe how to efficiently represent the spliced alignment.

7.5. Representation of spliced alignments

As a result of the spliced alignment phase, we obtain a collection of spliced alignments for different EST sequences and the same gDNA. We efficiently represent a spliced alignment by references to the substrings of the EST and the gDNA being aligned, and by a sequence of *multi edit operations*. An edit operation represents an output column of a spliced alignment, ignoring the symbols. This is possible because we access the columns of a spliced alignment in sequential order, and thus, the symbols are implicitly represented by the two substring references. As there are five different kinds of output columns, there are five edit operations: *match*, *mismatch*, *insertion*, *deletion*, and *intron*. Large stretches of a spliced alignment consist of consecutive columns of the same kind of output columns. Thus, with the exception of deletion columns, we aggregate each such sequence of output columns of the same kind into a corresponding multi edit operation. Each multi edit operation has an *iteration flag*, telling how many output columns it represents. Technically, a multi edit operation is represented by a 16-bit integer. The first two bits store a flag identifying the edit operation. The remaining 14 bits store the iteration flag. We use the same identification flag for deletion and intron. A deletion always has iteration flag 0, while an intron has iteration flag larger than 0. A sequence of l output columns of the same kind is thus represented by $\lceil l/2^{14} \rceil$ multi edit operations. As a result, a spliced alignment usually does not require more than 2 kb. This allows for feasible storage of hundreds of thousands of spliced alignments in main memory, as often required when processing large data sets.

Each spliced alignment is processed in several different ways, each requiring a sequential scan over aligned sequences and the sequence of multi edit operations:

- The spliced alignment has to be shortened on both sides, to get rid of deletion columns resulting from the symmetric extension of regions stemming from matches of a chain projected on the gDNA, see Section 5.3.
- From the shortened spliced alignment the exact exon/intron boundaries are determined.
- Additionally, score values are computed: the shortened spliced alignment is assigned an overall score, which is different from the optimal weight computed in the dynamic programming algorithm. Each exon is assigned an exon score. For the donor site and the acceptor site of each intron, probability values and scores are determined.

To simplify the implementation of these evaluation steps we have implemented the spliced alignment as an abstract datatype with a few generic functions to decode the edit operations in forward or backward order, and apply appropriate functions to the encoded output column.

All spliced alignments exceeding some user defined minimum score are collected into a balanced binary search tree. The spliced alignments are ordered by their start position in the gDNA. Large collections of ESTs often contain the same ESTs, which lead to identical spliced alignments. When inserting a spliced alignment into the search tree, such a situation is detected, and the identical spliced alignment is not stored in the tree. Once all ESTs are processed, the spliced

alignments are output or they are processed into a consensus spliced alignment.

7.6. Output of spliced alignments

GenomeThreader provides two output formats. The first format is text-based, intended to be read by users. It shows spliced alignments as in Fig. 2, with additional information about alignment scores, exon and intron boundaries, splice site scores, and probabilities, see http://www.genomethreader.org/ for an example.

Alternatively, output is in XML conforming to a specification implemented in the RELAX NG schema language, available at http://www.genomethreader.org/GenomeThreader.rng.txt. The benefit of an XML-based approach is that any program intercepting *GenomeThreader* output can expect a standards conforming, monomorphic data structure that can be validated using a tool such as *jing* (http://www.thaiopensource.com/relaxng/jing.html). Given a static, universally accepted schema standard, such otherwise brittle tools should never break, greatly diminishing code maintenance overhead.

We have implemented an assortment of software to utilize the XML output, including a Perl script to parse the data into a MySQL database of our design (*GthDB*, which is optimized for warehousing and querying high volumes of spliced alignment information in a multitude of ways), and a Python program for converting results to the GFF format used by GMOD's Generic Genome Browser (http://www.gmod.org/) [19]. These are distributed both with the *GenomeThreader* package and independently at the *GenomeThreader* web site.

7.7. Incremental updates

We have also defined an XML output schema for the spliced alignment data structure that allows *GenomeThreader* to dump alignments held in main memory into a string representation. This output can be validated against the provided schema specification, facilitating safe incremental updates of spliced alignment results.

There is an extra program *Gthconsensus*, which imports these XML data and runs the consensus spliced alignment algorithm, as described in Section 6. Because the phase generating consensus spliced alignments requires much less resources than the phase calculating the spliced alignments, one can incrementally compute the spliced alignments for a growing collection of ESTs, store these on file, and quickly recompute the consensus for the entire set of spliced alignments. This is of great importance because in practice, genome sequences are often already stable while additional EST and full-length cDNA collections are being generated. Thus, the *GenomeThreader* design allows quick cycles of incorporation of new data.

7.8. Software development tools

GenomeThreader is implemented in C using an object-oriented style. The source code is single threaded and it is written in such a way that it can be compiled without any changes on 32-bit and 64-bit platforms.

We used the GNU C compiler with high levels of optimization. *splint* (http://www.splint.org/) is regularly run to statically check the source code. We use *gdb* for debugging and *valgrind* (http://valgrind.org/) to track memory errors and leaks. The code is portable for different Unix platforms. In fact, we have compiled and tested it on eight different Unix platforms.

7.9. Test strategy

Systematic and automatic testing plays an important role in the entire software development process for *GenomeThreader*. Test data is abundant, as there are many genomes for which to predict gene structures, and many ESTs which can help with this. To check for the consistency of the data structures, we systematically implemented assertions in the program code. The assertions help to catch unexpected cases in the code very early in the development phase. Although the assertions slightly slow the program down, we leave them in production versions of *GenomeThreader*. Besides the code level testing, we employ output level testing, supported by *autotest*, a GNU-tool. In particular, we compare the results produced by older versions of it, or to the output of *GeneSeqer* [20], a program implementing the same spliced alignment algorithm, but with a different similarity filter and without the intron cutout technique.

8. Preliminary evaluation and performance benchmarks

The *GenomeThreader* program is now being distributed and is being used in several projects, including areas of application for which the earlier *GeneSeqer* program proved largely successful. For example, *GenomeThreader* is currently used at the Munich Information Center for Protein Sequences (MIPS, http://mips.gsf.de/) in their annotation pipeline (personal communication G. Haberer, 2005). Detailed performance evaluation of *GenomeThreader* for different applications is beyond the scope of this paper. Our discussion here is limited to a representative application of plant genome annotation and comparison with *GeneSeqer*, which was previously shown to be the most sensitive spliced alignment program for plant genome annotation [6].

We examined the first 600,000 nucleotides of the current assembly of rice chromosome 10 by aligning a set of more than 32,000 full-length rice cDNAs (of unknown chromosomal origin) using *GenomeThreader* and *GeneSeqer*. We selected a set of 52 full-length rice cDNAs that were aligned over at least 90% of their length from this output and re-aligned these sequences using *GeneSeqer*, *GenomeThreader* (without intron cutout), and *GenomeThreader* (with intron cutout). While a detailed analysis of how various combinations of parameters impact the results of the programs is also outside the scope of this report, we based comparisons on a set of options for *GenomeThreader* (without intron cutout) that closely match *GeneSeqer* settings and gave roughly equal numbers of spliced

alignments. Notably, *GenomeThreader* showed about fivefold speed-up compared to *GeneSeqer* under these conditions.

In terms of evaluating the comparative quality of various cDNA alignments, the three algorithms tested produced identical alignments in most cases. As expected, high-quality alignments are invariably consistent between *GenomeThreader* and *GenerSeqer*. Differences occur for lower quality alignments. Here, quality refers to the degree of sequence similarity between genomic DNA and cDNA. Because of natural sequence variation as well as unavoidable sequencing errors, less than perfect matching is not unusual, and this presents the more challenging alignment task. We identified four types of relations between alignments made using the algorithms: the alignments are identical; the alignments cover the same region, but are reported on opposite strands of the genomic DNA; the alignments map to the same locus, but differ in predicted gene structures (this includes the particular case of missing small exons); the same cDNA is mapped to different genomic regions by the programs. Examples are displayed at http://www.genomethreader.org/. Notably, the examined chromosomal region contained one locus for which a high-quality *GenomeThreader* cDNA alignment contained 15 exons and spanned about 17,000 nucleotides, the large span resulting mostly from two long introns. *GeneSeqer* could only partially resolve this gene structure because its length is beyond the default maximal gene length specified in the program in order to restrict memory use.

9. Discussion

This paper describes a new technique that permits gene structure predictions in the presence of long introns. The technique is a building block of a new software tool *GenomeThreader*, which was developed with adherence to strict software engineering principles. *GenomeThreader* implements several datatypes in a reusable manner. Compared to its predecessor *GeneSeqer*, it is considerably faster, easier to maintain, and extensible. Besides the description of the most important algorithms, we have focused on implementation aspects, which are often neglected in the development of bioinformatics software. With about two years of development time, *GenomeThreader* has become a robust software tool. However, there are still several aspects of the software to improve. (i) We want to improve the running time of the chaining phase from $O(k^2)$ to $O(k \log k)$, where k is the number of approximate matches to chain. It might be possible to adapt the $O(k \log k)$ method described in Ref. [17], where a different scoring function is used. (ii) The main bottleneck of *GenomeThreader* is still the computation of spliced alignments. We are planning to implement a linear space spliced alignment algorithm utilizing techniques similar to Ref. [16]. Furthermore, we hope to improve the speed of the spliced alignment algorithm by careful code level optimization. (iii) *GenomeThreader* provides many different options to influence the different phases of the computation and thus often trade running time and space requirement for quality of gene structure predictions. Careful selection of default parameters depending on the specific organism and the quality of the ESTs is very important to balance the resource requirements and quality of the gene structure predictions.

GenomeThreader is available free of charge for non-commercial research institutions. For details see http://www.genomethreader.org/.

References

[1] M.I. Abouelhoda, S. Kurtz, E. Ohlebusch, Replacing suffix trees with enhanced suffix arrays, Journal of Discrete Algorithms 2 (2004) 53–86.
[2] B. Alberts, A. Johnson, J. Lewis, M. Raff, K. Roberts, P. Walter, Molecular Biology of the Cell, Garland Science, 2002.
[3] R. Bellman, Dynamic Programming, Princeton University Press, 1957.
[4] J. Bentley, R. Sedgewick, Fast Algorithms for Sorting and Searching Strings, in: Proceedings of the ACM-SIAM Symposium on Discrete Algorithms, 1997, pp. 360–369.
[5] E. Birney, M. Clamp, R. Durbin, GeneWise and Genomewise, Genome Research 14 (5) (2004) 988–995.
[6] V. Brendel, L. Xing, W. Zhu, Gene structure prediction from consensus spliced alignment of multiple ESTs matching the same genomic locus, Bioinformatics 20 (7) (2004) 1157–1169.
[7] International Human Genome Sequencing Consortium, Finishing the euchromatic sequence of the human genome, Nature 431 (7011) (2004) 931–945.
[8] The ENCODE Project Consortium, The ENCODE (ENCyclopedia Of DNA Elements) Project, Science 306 (5696) (2004) 636–640.
[9] T.H. Cormen, C.E. Leiserson, R.L. Rivest, Introduction to Algorithms, MIT Press, Cambridge, MA, 1990.
[10] L. Florea, G. Hartzell, Z. Zhang, G.M. Rubin, W. Miller, A computer program for aligning a cDNA sequence with a genomic DNA sequence, Genome Research 8 (1998) 967–974.
[11] B.J. Haas, A.L. Delcher, S.M. Mount, J.R. Wortman, R.K. Smith Jr., L.I. Hannick, R. Maiti, C.M. Ronning, D.B. Rusch, C.D. Town, S.L. Salzberg, O. White, Improving the *Arabidopsis* genome annotation using maximal transcript alignment assemblies, Nucleic Acids Research 31 (19) (2003) 5654–5666.
[12] C. Kanz, P. Aldebert, N. Althorpe, W. Baker, A. Baldwin, K. Bates, P. Browne, A. van den Broek, M. Castro, G. Cochrane, K. Duggan, R. Eberhardt, N. Faruque, J. Gamble, F.G. Diez, N. Harte, T. Kulikova, Q. Lin, V. Lombard, R. Lopez, R. Mancuso, M. McHale, F. Nardone, V. Silventoinen, S. Sobhany, P. Stoehr, M.A. Tuli, K. Tzouvara, R. Vaughan, D. Wu, W Zhu, R. Apweiler, The EMBL nucleotide sequence database, Nucleic Acids Research 33 (Database Issue) (2005) 29–33.
[13] W.J. Kent, BLAT—The BLAST-Like Alignment Tool, Genome Research 12 (4) (2002) 656–664.
[14] S. Kurtz, J.V. Choudhuri, E. Ohlebusch, C. Schleiermacher, J. Stoye, R. Giegerich, REPuter: the manifold applications of repeat analysis on a genomic scale, Nucleic Acids Research 29 (22) (2001) 4633–4642.
[15] B. Lewin, Genes VIII, Prentice Hall, 2004.
[16] D.R. Powell, L. Allison, I. Dix, A versatile divide and conquer technique for optimal string alignment, Information Processing Letters 70 (1999) 127–139.
[17] T. Shibuya, I. Kurochkin, Match Chaining Algorithms for cDNA Mapping, in: Proceedings of the Third Workshop on Algorithms in Bioinformatics (WABI 2003), number 2812 in Lecture Notes in Bioinformatics, Springer, 2003, pp. 462–475.
[18] M.E. Sparks, V. Brendel, Incorporation of splice site probability models for non-canonical introns improves gene structure prediction in plants. Bioinformatics, 21 (Suppl. 3), iii1–iii11 (2005).
[19] L.D. Stein, C. Mungall, S. Shu, M. Caudy, M. Mangone, A. Day, E. Nickerson, J.E. Stajich, T.W. Harris, A. Arva, S. Lewis, The generic genome browser: a building block for a model organism system database, Genome Research 12 (10) (2002) 1599–1610.

[20] J. Usuka, W. Zhu, V. Brendel, Optimal spliced alignment of homologous cDNA to a genomic DNA template, Bioinformatics 16 (3) (2000) 203–211.
[21] S.J. Wheelan, D.M. Church, M. Ostell, Spidey: a tool for mRNAto-genomic alignments, Genome Research 11 (11) (2001) 1952–1957.
[22] D.L. Wheeler, T. Barrett, D.A. Benson, S.H. Bryant, K. Canese, D.M. Church, M. DiCuccio, R. Edgar, S. Federhen, W. Helmberg, D.L. Kenton, O. Khovayko, D.J. Lipman, T.L. Madden, D.R. Maglott, J. Ostell, J.U. Pontius, K.D. Pruitt, G.D. Schuler, L.M. Schriml, E. Sequeira, S.T. Sherry, K. Sirotkin, G. Starchenko, T.O. Suzek, R. Tatusov, T.A. Tatusova, L. Wagner, E. Yaschenko, Database resources of the National Center for Biotechnology Information, Nucleic Acids Research 33 (Database Issue) (2005) 39–45.
[23] T.D. Wu, K. Watanabe, GMAP: a genomic mapping and alignment program for mRNA and EST sequences, Bioinformatics 21 (9) (2005) 1859–1875.