Computational Gene Structure Prediction

Dissertation

zur Erlangung des akademischen Grades
Dr. rer. nat.
an der Fakultät für Mathematik, Informatik und Naturwissenschaften
der Universität Hamburg

eingereicht beim Fach-Promotionsausschuss Informatik von

Gordon Gremme

aus Wesel

August 2012

Gutachter:
Prof. Dr. Stefan Kurtz, Universität Hamburg
Prof. Dr. Wolfgang Menzel, Universität Hamburg
Prof. Dr. Volker Brendel, Indiana University Bloomington

Tag der Disputation:
15. Mai 2013

Dedicated to my mother. I miss you.

Acknowledgments

I want to thank Stefan Kurtz for his guidance and encouraging support over the years. He never lost his patience with me, for what I am very grateful. Furthermore, I want to thank Volker Brendel for his helpful explanations concerning gene structure prediction and the nice stays with him in Ames, Iowa. I also want to thank Volker's group for their hospitality and interesting discussions, in particular Michael Sparks for writing an XML parser for *GenomeThreader* and showing me around Ames.

At the Center for Bioinformatics (ZBH) I want to thank all my colleagues for the great time I had while being there, it was an inspiring experience. In particular, I want to thank Ute Willhöft for many helpful discussions on biology, Sascha Steinbiß for the good and fruitful collaboration, and Karin Lundt for her uplifting support.

From Matthias Rarey's group I especially want to thank Patrick Maaß for our countless discussions and for what he taught me about programming. It meant a lot to me. I also want to thank him and Jörg Degen, Axel Griewel, Juri Pärn, Ingo Reulecke, Ingo Schellhammer, Jochen Schlosser, and Katrin Stierand for the fun times playing tabletop soccer together.

Last but not least I want to thank my friends, my family, and my loved ones who helped writing this dissertation with their knowledge, their patience, and their love. You are too many to thank individually here, but I hope you know who you are. I will be forever in your debt and all I can offer is a sincere "Thank you!".

Abstract

Modern molecular biology research is characterized by the availability of an increasing amount of biological data which is often fuzzy due to the nature of the experimental methods used to derive it. Bioinformatics, a branch of computer science, deals with the storage, retrieval, and analysis of this data. DNA, the basic information carrier of life, is now sequenced industrially in large quantities and assembled to complete genomes. The automatic annotation of genes in these genomes, a process called computational gene structure prediction, is the scope of this thesis.

This dissertation describes the computational gene structure prediction software *GenomeThreader* which uses homologous biological sequences (so-called cDNAs/ESTs and/or protein sequences) to predict gene structures by computing spliced alignments. *GenomeThreader* uses a multi-phase approach, filtering the possibly very large sequence data sets in early phases to obtain promising gene candidates which are then refined by more computationally expensive algorithms in later phases. The results of this gene structure predictions, genome annotations, can become quite large and cumbersome to process. To deal with such annotations easily and efficiently, the *GenomeTools* genome analysis system has been developed, which is also described in this thesis.

The prediction quality of *GenomeThreader* was evaluated on a variety of datasets and the results show that the software performs very well on common gene structure prediction tasks. The quality of the results is comparable with the results of the best other programs and in some cases it is even better. The software is very easy to use due to its integrated nature, a feature which distinguishes it from its competitors. *GenomeThreader* has been adopted widely in the scientific community, it has approx. 150 users world-wide and over 30 publications cite the scientific article which describes an earlier version of the software. The open source package *GenomeTools* was used as a foundation for 10 published sequence and annotation processing tools.

Kurzfassung

Moderne molekularbiologische Forschung ist durch die Verfügbarkeit stetig wachsender Datenmengen charakterisiert. Diese Daten sind auf Grund der experimentellen Methoden, die sie erzeugen, oftmals fehlerbehaftet. Die Bioinformatik, ein Teilbereich der Informatik, beschäftigt sich damit molekularbiologische Daten zu speichern, abzurufen und zu analysieren. DNA, der grundlegende Informationsträger des Lebens, wird heutzutage im industriellen Maßstab sequenziert und zu kompletten Genomen zusammengefügt. Diese Disseration befasst sich mit der automatische Annotation von Genen in vollständig sequenzierten Genomen, ein Prozess der rechnergestützte Genstrukturvorhersage genannt wird.

Diese Dissertation beschreibt die Methoden und Techniken, die die Grundlage der Genstrukturvorhersagesoftware *GenomeThreader* bilden. *GenomeThreader* benutzt homologe biologische Sequenzen (sogenannte cDNA/EST und/oder Proteinsequenzen) und berechnet Spliced Alignments, die Genstrukturen beschreiben. Zur Vorhersage der Genstrukturen wird ein mehrphasiger Ansatz benutzt. Dabei werden die unter Umständen sehr großen Sequenzdatenmengen in frühen Phasen auf vielversprechende Genkandidaten reduziert, die dann in späteren Phasen durch rechenaufwendigere Algorithmen verfeinert werden. Die Resultate dieser Genstrukturvorhersagen, die Genomannotationen, können sehr umfangreich werden und aufwendige Schritte der Weiterverarbeitung erfordern. Um mit solchen Annotationen einfach und effizient umgehen zu können, wurde das *GenomeTools* Genomanalysesystem entwickelt, das ebenfalls in dieser Arbeit beschrieben wird.

Die Vorhersagequalität von *GenomeThreader* wurde auf verschiedenen Datensätzen evaluiert. Es zeigt sich, dass *GenomeThreader* für die üblichen Genvorhersageaufgaben sehr gute Ergebnisse liefert. Die Qualität der Ergebnisse ist vergleichbar mit den Ergebnissen der besten anderen Programme und teilweise sogar besser. Durch die gelungene Integration der einzelnen Phasen ist die Software sehr einfach zu benutzen, eine Eigenschaft, die sie von ihren Wettbewerbern unterscheidet. *GenomeThreader* hat weite Verbreitung in der Wissenschaftsgemeinde gefunden. Es gibt ca. 150 Nutzer weltweit und 30 Publikationen zitieren den wissenschaftlichen Artikel, der eine frühe Version der Software beschreibt. Das quelloffene Softwarepaket *GenomeTools* diente als Grundlage für 10 weitere publizierte Werkzeuge zur Sequenz- und Annotationsverarbeitung.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

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.

Modern molecular biology research is characterized by the ability to study questions from a genome-wide perspective. Whereas only 15 years 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 [Bre08]. And with the ever increasing amount of DNA information produced by classical Sanger sequencing [SNC77] and next-generation DNA sequencing (NGS) methods [SJ08] at decreasing cost, the number of genomes awaiting annotation [PLJ+12] and accompanying transcriptome (the set of all messenger RNA molecules in a cell) [BLT93] and proteome (the entire set of proteins expressed by a genome) information [Con12] is rising sharply.

Whereas obtaining the genetic blueprint, or, more technically, genomic DNA sequencing, is a mostly technological process, gene annotation is largely computational, involving both statistically based prediction methods and integration of various sources of experimental and knowledge-based evidence.

The automatic process of gene annotation with computational methods is called *computational gene structure prediction*. This thesis describes the *GenomeThreader* gene structure prediction software which is well-suited for this task.

1.2 Contributions

GenomeThreader uses a similarity-based approach with cDNA/EST and/or protein sequences to predict gene structures. *GenomeThreader* computes so-called *spliced alignments* (for every given cDNA/EST/protein sequence) and combines them into *consensus spliced alignments* which represent the predicted gene structures.

The main algorithmic contributions of this thesis are the *intron cutout technique*, the accompanying *chain enrichment*, and *jump tables*. The main conceptual contributions are *easy-to-use BSSMs* (Bayesian Splice Site Models) and *incremental updates*. The main software engineering contributions are the *combination of both cDNA/EST-based spliced alignments **and** protein-based spliced alignments into consensus spliced alignments* and the development of the *GenomeTools* genome analysis system. The seven main contributions in more detail:

1. The *intron cutout technique* allows to predict 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 these steps to compute spliced alignments in an efficient manner.

2. The accompanying *chain enrichment*, although algorithmically simple, can mitigate some problems of the intron cutout technique on genomes containing very large introns. It improves the prediction results, but increases the running time.

3. *Jump tables* allow to significantly reduce the amount of necessary calculations in the dynamic programming step which computes spliced alignments. The intron cutout technique can be combined with jump tables, resulting in a compound speedup.

4. BSSMs assign splice site probabilities to genomic DNA bases. Splice site probabilities are a very important component for gene prediction methods, but such models have to be trained for the organism in question. The *easy-to-use BSSMs* in *GenomeThreader* make the training and application of custom (that is, organism specific) BSSMs very easy.

5. The *GenomeThreader* phase which computes consensus spliced alignments requires much less resources than the phase calculating the spliced alignments and with *incremental updates* one can incrementally compute the spliced alignments for a growing collection of ESTs and proteins, store these on file, and quickly recompute the consensus for the entire set of spliced alignments. This is of great importance in practice, because genome sequences are often already stable while additional EST and full-length cDNA collections are being generated.

6. The *combination of both cDNA/EST-based spliced alignments **and** protein-based spliced alignments into consensus spliced alignments* allows to use all available gene product sequences for the prediction. To our knowledge, no other gene prediction program has this feature.

7. The *GenomeTools* genome analysis system allows to process genome annotation data in a very fast and flexible manner, a capacity which proofed to be very useful in performing interactive genome annotations. In practice, genome annotations are usually not obtained in a fully automatic manner, but with manual intervention from computational biologists. Therefore, having a system which allows to test different parameter sets in a fast and easy-to-use way is very important.

We discuss not only the algorithmic details of *GenomeThreader* and *GenomeTools* in this thesis, but also important software engineering aspects, because we believe that for bioinformatics software the engineering should not be an afterthought or left aside, but instead should be considered from the beginning. In our opinion, the good prediction results and widespread adoption (Section 4.8.11 lists many published scientific applications of *GenomeThreader* by other authors) were only made possible, because we strove to a create a modular and robust software tool from the inception of this dissertation.

For example, on the dataset from the nematode genome annotation assessment project, nGASP for short [CFM$^+$08], *GenomeThreader* performs better than all competing programs in its category (see Section 6.1). Performed tests with 454 RNA-Seq reads on the nGASP dataset show that *GenomeThreader* is also well suited to use sequences produced by NGS methods to predict gene structures (see Section 6.2). This is also underlined by the fact that *GenomeThreader* was successfully used to align 454 RNA-Seq reads to the tomato genome and that these results are integrated into the tomato annotation pipeline [Fil10, FTD$^+$12, BAG$^+$10, CDT$^+$08]. The tests on the ENCODE dataset [HDF$^+$06] show that *GenomeThreader* is very robust against sequencing errors and able to align sequences from cognate organisms correctly (see Section 6.3 for details).

This dissertation project was motivated by disabling limitations in an original, ad-hoc, and yet widely popular implementation named *GeneSeqer* [BXZ04, UZB00]. It has led to the robust, highly versatile, and extensible tool *GenomeThreader* that not only overcomes the limitations of the earlier implementation but greatly improves space and time requirements while giving very good prediction results.

1.3 Structure of this Thesis

In Chapter 2 the biological background of the gene structure prediction problem is explained. Chapter 3 introduces the different approaches to solve the gene structure prediction problem, how to measure their accuracy, and related work in this area. Chapter 4 describes the *GenomeThreader* gene structure prediction software in detail. In Chapter 5 the *GenomeTools* genome analysis system is discussed. In Chapter 6 the evaluation of *GenomeThreader* is given. In Chapter 7 the thesis is concluded.

Some parts of this thesis have been previously published in G. Gremme, V. Brendel, M.E. Sparks, and S. Kurtz, Engineering a Software Tool for Gene Structure Prediction in Higher Organisms,

Information and Software Technology, **47**(15):965–978, 2005 (part of Chapter 4) and S. Steinbiss, G. Gremme, C. Schärfer, M. Mader, and S. Kurtz, *AnnotationSketch*: A Genome Annotation Drawing Library, *Bioinformatics*, **25**(4):533–534, 2009 (in Section 5.7).

The author of this thesis also contributed to *LTRdigest* which was published in S. Steinbiss, U. Willhoeft, G. Gremme, and S. Kurtz, Fine-Grained Annotation and Classification of *de novo* Predicted LTR Retrotransposons, *Nucleic Acids Res.*, **37**(21):7002–7013, 2009. A paper on *GenomeTools* is currently being written and will be submitted in Autumn 2012[1].

Some of the definitions in Section 3.1 have been taken from S. Kurtz, *Foundations of Sequence Analysis*, Unpublished Lecture Notes, May 2003.

The manuals of *GenomeThreader* and *GenomeTools* and further information referred in the main body of the thesis are given in the various appendices.

[1] Addendum June 2013: The corresponding paper [GSK13] has been accepted.

Chapter 2

Biology Background

This chapter starts off with a high-level view of biology and then zooms in on the central notion of *gene*. This should give further motivation for this thesis and put it in a wider context. For a more thorough introduction, the reader is referred to textbooks of molecular biology, for example [Lew04, AJL+02].

2.1 The Science of Life

Biology is the science of life. Life is usually defined (e.g. in [NC01]) as something which

1. can reproduce itself,
2. absorbs energy from the environment, in form of nutrition or sunlight,
3. has a higher degree of complexity than the environment.

Viruses are not considered to be life, because they cannot reproduce themselves. The living world of organisms is divided into three *domains*: *Bacteria* (Eubacteria), *Archaea* (Archaebacteria) and *Eukarya* [AJL+02]. Organisms are composed of *cells*: From one in simple bacteria up to several billion in the most complex eukaryotes like humans.

2.2 Prokaryotes and Eukaryotes

The three domains of life can also be differentiated as being either *prokaryotes* or *eukaryotes*. All organisms from the domains Bacteria and Archaea are prokaryotes, whereas all organisms from the domain Eukarya are, as the name implies, eukaryotes. Eukaryotes and prokaryotes differ substantially in the structure of their cells. The main difference between prokaryotic cells and eukaryotic cells is that the former are absent of nuclei. There are many more differences

between these, but in the context of this thesis it is only important to know that the structure of genes is much simpler in prokaryotes and that they lack splicing (the terms gene and splicing are described further below). Because of the fairly different properties of pro- and eukaryotes, gene prediction methods are usually only suited for one of them. From now on in this thesis, we will only deal with eukaryotes, and the methods described here only apply to them.

2.3 Basic Macromolecules

In a cell four types of *macromolecules* occur. Each macromolecule is a composition of smaller *monomers* (see e.g. [AJL+02, p. 59]):

1. nucleic acids (composed of nucleotides)
2. proteins (composed of amino acids)
3. fats and lipids (composed of fatty acids)
4. polysaccharides (composed of sugars)

Only nucleic acids and proteins will be addressed here further. Neither fats and lipids nor polysaccharides are relevant in this context.

2.3.1 Nucleic Acids

The two kinds of nucleic acids, namely DNA and RNA are described in this subsection. After describing the structure of the DNA it is shown how RNA differs from DNA.

DNA

Deoxyribonucleic acid (DNA) is the genetic material of the cell. It is present in the cell in the form of a double helix. The double helix is composed of two *complementary*, *antiparallel* strands. Each strand consists of a sugar-phosphate backbone with nitrogenous *purine* and *pyrimidine* bases attached to it by N-glycosidic linkage. The sugar-phosphate backbone consists of pentose carbon rings, where the $5'$ position of one pentose carbon ring is connected to the $3'$ position of the next pentose carbon ring by a phosphate group (phosphodiester bond). This gives the strand a direction, which is usually written down from $5'$ to $3'$. There are four types of nitrogenous bases in DNA, *adenine* (A), *cytosine* (C), *guanine* (G) and *thymine* (T). Adenine and guanine are purines, whereas cytosine and guanine are pyrimidines. The DNA double helix is formed be hydrogen bonds between the bases on the two antiparallel strands, a process which is called *base paring*. Thereby a purine always pairs with a pyrimidine and vice versa, but only in two combinations: Adenine pairs with thymine ($A - T$) through 2 hydrogen bonds and guanine pairs with cytosine ($G - C$) via 3 hydrogen bonds [Lew04].

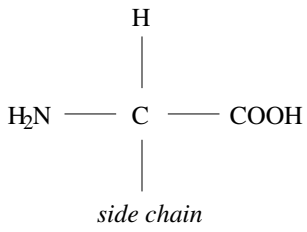


Figure 2.1: General structure of amino acids (nonionized).

RNA

The sugar in RNA is a ribose instead of the 2-deoxyribose in DNA and instead of the pyrimidine thymine in RNA we have a pyrimidine called *uracil* (U). Single-stranded RNA can form *secondary structures*, where the RNA-strand forms base pairs with itself. RNA is generally more unstable than DNA.

The RNA in the cell can be divided into different classes according to its function, like *messenger* RNA (mRNA), *transfer* RNA (tRNA) and *ribosomal* RNA (rRNA).

2.3.2 Proteins

Proteins are linear polymers made up of *amino acids* which are connected by *peptide bonds*. All amino acids consist of an α-carbon linked to a carboxyl group, an amino group and a single H-atom (see Figure 2.1). The difference between the 20 natural amino acids is the fourth group attached to the α-carbon, the so-called *side chain*. The α-carbon is a chirality center which leads to the two optical isomers of every amino acid, namely the L- and the D-form. *"But only L-forms are ever found in proteins"* (page 63 of [AJL+02]).

The peptide bonds link the carboxyl group of one amino acid with the amino group of a second one. The amino acid with a free amino group is called the *N-terminus* and the one with a free carboxyl group the *C-terminus*.

The amino acid sequence of a protein, usually written down from the N-terminus to the C-terminus, is called its *primary structure*. This sequence of amino acids does not occur in the cell in a linear form, instead it is folded in a three dimensional structure, the *tertiary structure*. The *secondary structure* are common structural elements in the tertiary structure, like α*-helices* and β*-sheets*. If a protein consists of more than one polypeptide chain, the overall structure is called *quaternary*.

If proteins catalyze reactions, they are called *enzymes*. If they have structural functions in the cell, we call them *structural proteins*. Usually structural proteins do self-assemble into larger structures.

2.4 Information Carrier DNA

The DNA in the cell nucleus is distributed among *chromosomes*. In a simplified view one can say that each chromosome is a very large DNA double helix associated with proteins. During meiosis and mitosis chromosomes are recognizable as very condensed structures under a light microscope.

The DNA serves as an information carrier: the information necessary for synthesizing the proteins is stored in its *genes*.

Proteins are encoded by the *genetic code*: 3 consecutive bases encode one amino acid, which means there are $4^3 = 64$ possible *codons*. 61 of the codons actually encode for amino acids, whereas 3 are *STOP-codons*. Because there are more codons than amino acids, some codons encode the same amino acids. In these cases the codons differ in the third positions, a fact which is called *third-base degeneracy*.

2.5 Single Pieces of Information: Genes

The total hereditary information contained in the DNA of a cell can be divided into genes. A *gene* is a contiguous piece of DNA, which contains the information for a protein or an RNA molecule.

In the following Section the typical "processing" of a *protein-coding* gene is described. See Section 2.9 for a description of the historical formation of the gene concept and an overview of other gene types (besides protein-coding genes).

2.6 Flow of Information: From a Gene to a Protein

The *Central Dogma* formulated by Francis Crick states that the information in the cell flows from DNA to RNA to protein and that once information got into a protein it cannot get out again [Cri51]. For a schematic presentation of this Section see Figure 2.2. Here the description of this flow of information will be given in a simplified manner. For a more detailed description see [Lew04] or [AJL$^+$02].

The *RNA-polymerase* binds to the start of a gene at the *promoter* region. From there on it moves along the DNA (from 5′ to 3′) while synthesizing the so-called pre-mRNA until a *terminator* sequence is reached and the RNA-polymerase falls off the DNA strand. The start of the transcribed DNA segment is called the *transcription start site* and the end the *transcription termination site*.

Usually the pre-mRNA contains one gene, what is called *monocistronic*. Transcripts with more than one gene (*polycistronic*) mainly occur in bacteria.

The pre-mRNA is modified at the ends: it gets a *cap* at the 5′ end and a *poly(A)* tail at the 3′ end. The 7-methylguanosine cap is bound by a 5′-5′ triphosphate linkage to the first nucleotide at the

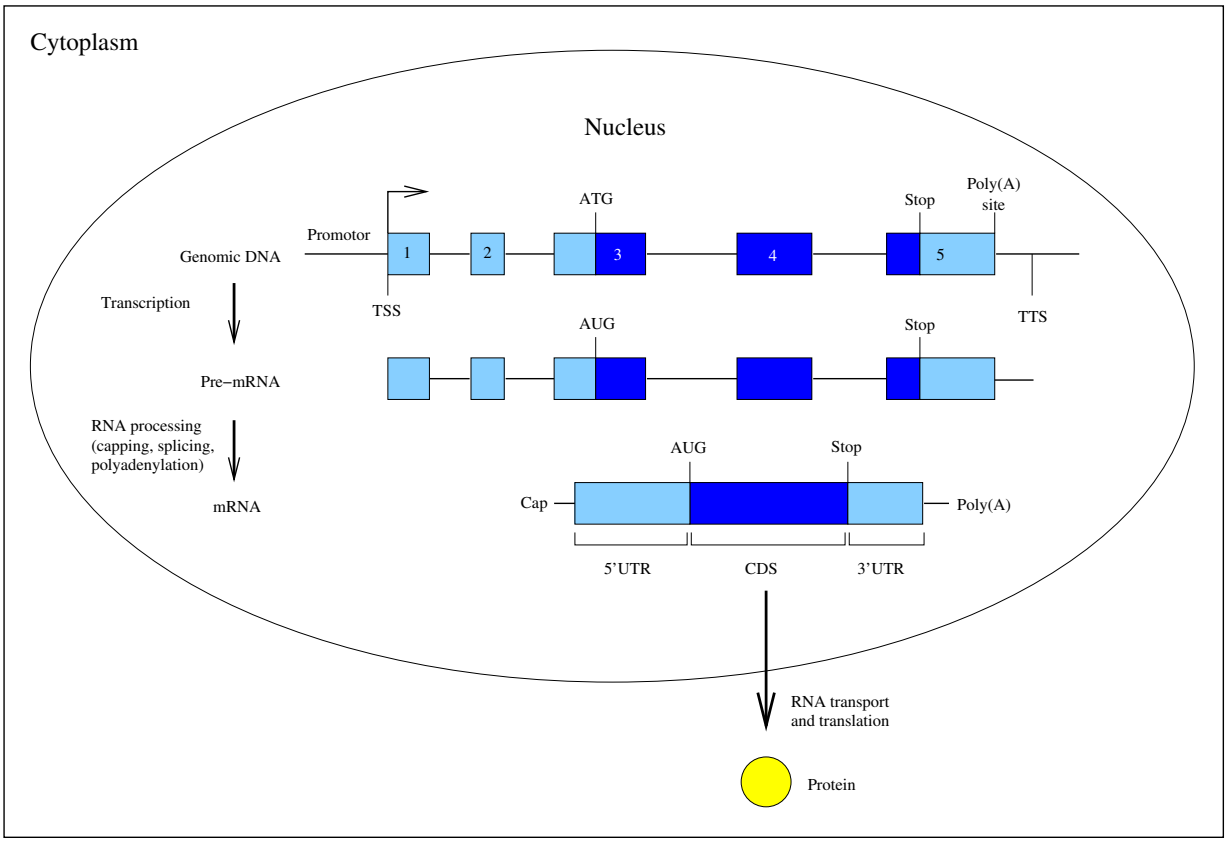


Figure 2.2: Gene expression. See Section 2.6 for an explanation. TSS stands for transcription start site; TTS for transcription termination site. Diagram adapted from [Zha02]. Note that exons in the 5′ UTR are rare.

5′ end of the pre-mRNA. The poly(A) usually consists of 150-250 adenine nucleotides [Kni97, p. 403-404].

2.6.1 Splicing

In the next step, some parts of the pre-mRNA are cut out in a process called *splicing*. The cut out pieces are called *introns*, whereas the remaining ones are called *exons*. Splicing essentially occurs in two forms.

In the first case a large complex of proteins and ribonucleoproteins, the *spliceosome*, is formed. The proteins of the spliceosome recognize conserved regions at the beginning and the end of the intron (so-called *donor* and *acceptor* sites) and one within the intron near the acceptor site (the *branch* site). The spliceosome cuts out the intron in two stages. In the first stage a cut at the donor site is performed and the intron forms a *lariat*. In the second stage it is cut at the acceptor site, the intron is released as a lariat and the remaining exons are joined.

In the second case a pre-mRNA can also have self-splicing properties, where the intron forms a characteristic secondary/tertiary structure, which performs self-splicing activity.

The splice sites (which are totally within the intron) are highly conserved: the generic donor site is GT and the generic acceptor site is AG. This introns are often referred to as $GT - AG$. The branch site lies usually 18-40 nucleotides upstream of the acceptor site. In yeast, the sequence of the branch site is highly conserved, while the conservation is weaker in higher eukaryotes [AJL+02].

The product of the splicing process without the introns is called *mature* mRNA or for short just 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).

2.6.2 Translation

After the mature mRNA has been transported out of the nucleus it is processed by *ribosomes*, large ribonucleoproteins consisting of rRNA and proteins.

The mRNA can be divided into three parts: in the middle is the *coding sequence* (CDS), which will be translated into a protein, while at the $5'$ and $3'$ end *untranslated regions* (UTRs) are found.

A ribosome consists of two subunits, a small and a large one. The small subunit of a ribosome binds to the cap of an eukaryotic RNA and moves along the $5'$ UTR until it reaches the start codon AUG. At this point the whole ribosome assembles and the newly translated polypeptide chain starts by involvement of a special tRNA, the N-formyl-methionyl-tRNA. Afterwards, the starting polypeptide chain is elongated while aminoacyl-tRNAs deliver the new amino acids. The "decision" which amino acid to use is done by base paring between the codons on the mRNA and *anticodons* on the tRNAs. Different codons which encode the same amino acids are sometimes recognized by the same tRNA through *wobbling*. When a stop codon is reached, translation stops and the newly sequenced protein is ready (if no *post-translational modifications* are performed beforehand).

A DNA sequence of triplets which can be translated into protein (that is, which begins with a start codon, ends with a stop codon, and contains no embedded stop codons) is called an *open reading frame* (ORF).

2.7 What is an EST?

EST stands for *expressed sequence tag*. Because ESTs are very important for the gene prediction approach incorporated into *GenomeThreader*, it is described here in greater detail how ESTs are constructed. This is done following [WL01]. A summary can be found in Figure 2.3.

Platform	Read length (bases)	Run time (days)	GB per run	Machine cost (US$)
Roche/454's GS FLX Titanium	330 (avg.)	0.35	0.45	500,000
Illumina/ Solexa's GA	75 or 100	4 (fragment run) 9 (mate-pair run)	18 35	540,000
Life/APG's SOLiD 3	50	7 (fragment run) 14 (mate-pair run)	30 50	595,000
Polonator G.007	26	5 (mate-pair run)	12	170,000

Table 2.1: Comparison of next-generation sequencing platforms. Adapted from [Met10, p. 37]. GA stands for genome analyzer and GS for genome sequencer.

2.7.1 Construction of ESTs

First, mRNAs are extracted from a specific tissue or cell line. Afterwards the mRNAs are reverse transcribed to so called *complementary* DNAs (cDNAs). For the reverse transcription usually oligo(dT) *primer* are used. They hybridize with the poly(A) tail of the mRNA, whereas random primers bind somewhere to the mRNA.

The cDNAs are cloned into *vectors* and propagated in *E. coli* resulting in a *cDNA library*. The cDNA library consists of the cDNA clones from most of the mRNA population of the corresponding tissue or cell line. The mRNA population is represented quantitative and qualitative.

Individual clones from such a cDNA library are randomly picked out and the $5'$ and $3'$ ends of the cDNA insert are sequenced in a single read. Single reading of the ESTs makes determination fast and cheap, but leads to a rather high error rate compared to "normal" DNA sequencing. The length of single reads is approximately 400 bp. This means that the middle part of longer cDNAs is usually not covered by ESTs.

There is steadily growing number of ESTs available. The dbEST [BLT93] release from the 1th July 2012 contains 73,360,923 ESTs in total (among them are 8,692,773 from *Homo sapiens*).

ESTs are for example useful for discovering unknown protein coding genes. This can be either done biologically via hybridization or computationally by spliced alignment gene structure prediction methods (as described in this thesis).

2.8 Next-Generation Sequencing Methods

Traditionally, Sanger sequencing [SNC77] has been used almost exclusively for DNA sequencing. After more than three decades of refinements, the Sanger biochemistry can be applied to achieve read-length of up to about 1,000 bp, and per-base "raw" accuracies as high as 99.999%

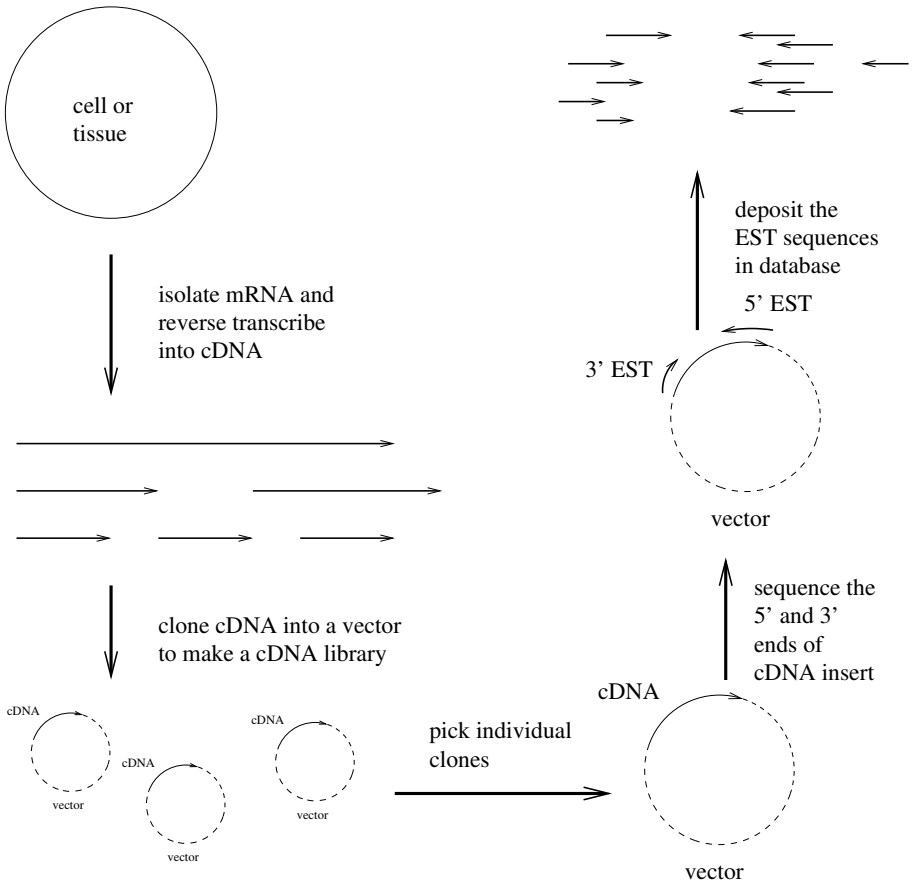


Figure 2.3: This diagram gives an overview of EST construction. It has been adapted from [WL01].

Technology	Tiling microarray	cDNA/EST sequencing	RNA-Seq
Principle	Hybridization	Sanger sequencing	High-throughput
Resolution	From several to 100 bp	Single base	Single base
Throughput	High	Low	High
Reliance on gen. sequence	Yes	No	In some cases
Background noise	High	Low	Low
Required RNA	High	High	Low
Cost	High	High	Relatively low

Table 2.2: Comparison of transcriptomics methods. Adapted from [WGS09, p. 59].

[SJ08, p. 1135]. In high-throughput shotgun genome sequencing, Sanger sequencing costs on the order of $0.50 per kilobase. But the technology has inherent drawbacks, like the necessary *in vivo* cloning of the DNA in *Escherichia coli* and limited scalability.

In recent years, so-called *next-generation sequencing* (NGS) methods have been developed which strive to overcome these limitations. They allow DNA sequencing at a much lower cost than Sanger sequencing, are better to parallelize, and allow *in vitro* preparation of the DNA samples. But NGS methods have shorter read length and lower accuracies than Sanger sequencing, which poses new challenges for bioinformatics. But since these technologies just have been introduced recently, they have not yet seen the refinements which appeared in Sanger sequencing and improvements are likely to expect in the coming years. Table 2.1 gives an overview of various NGS methods and important parameters like read-lengths and the type of errors they produce. As one can see from the table, NGS methods are two to three orders of magnitude cheaper than traditional Sanger sequencing.

The first NGS platform which was available as a commercial product is the so-called 454 pyrosequencing [MEA+05]. This is also the technology which allows the longest read-lengths.

For reviews of next generation sequencing methods see [LCC+12, LMD+12, Met10, SJ08, Mar08a, Mar08b].

2.8.1 RNA-Seq

NGS methods have many applications and an important one is the sequencing of the transcriptome of an organism. A transcriptome is the complete set of transcripts in a cell, and their quantity, for a specific developmental state or physiological condition [WGS09, p. 57].

The high-throughput NGS methods described above allow to determine the transcriptome of an organism better than previous methods. The methods to determine the transcriptome with NGS methods have been coined RNA-Seq (stands for *RNA sequencing*) and they have various advantages over the two older approaches tiling microarrays and cDNA/EST sequencing. Table 2.2 gives a comparisons of the three prevalent approaches for transcriptomics. A review of RNA-Seq is given in [WGS09].

2.9 What is a Gene? Attempting a Definition

The etymology of the term *gene* derives from the Greek *genesis* ("birth") or *genos* ("origin") [GBR+07]. Although it is a central notion in (molecular) biology, it's exact meaning changed considerably during it's history. In this Section we will review the evolution of the term from it's first appearance up to it's most recent definition which is used in this thesis, following the timeline given in [GBR+07].

The related term *genetics* was coined by William Bateson in 1905 [Bat].

2.9.1 Gene as a Heredity Unit

In 1866 Gregor Mendel developed the concept of a *gene* (without naming it that way) as a distinct, discrete heredity unit related to certain traits he observed during his famous plant breeding studies [Men66]. The actual term *gene* was first used in 1909 by Wilhelm Johannsen [Joh09].

Mendel postulated that certain *phenotypes* have a *genotype* as their cause without being able to identify the molecular basis of the genotypes in the cells.

2.9.2 Gene as a Distinct Locus

This situation improved in the 1910s when Alfred Henry Sturtevant created the first genetic map [Stu13] and *The Mechanism of Mendelian Inheritance* which studied the segregation of mutations in *Drosophila melanogaster* was published by Thomas Hunt Morgan and his students [MSMB15]. This established that genes are arranged linearly and that their ability to cross-over is proportional to their distance.

But it wasn't until 1929 that Barbara McClintock established that genetic linkage corresponds to actual physical locations on chromosomes [McC29].

2.9.3 Gene as a Protein Blueprint

In 1941 George Wells Beadle and Edward Lawrie Tatum published the results of their study of the *Neurospora* metabolism and stated the "one gene, one enzyme" hypothesis which was later modified to "one gene, one polypeptide" [BT41].

2.9.4 Gene as a Physical Molecule

In the 1950s the view of a gene as a physical molecule was finally established – the *deoxyribonucleic acid* (*DNA*) was identified as the carrier of the genetic information.

Earlier findings by Hermann Joseph Muller and Frederick Griffith supported this view. In 1927 Hermann Joseph Muller showed the mutagenic effects of X-ray radiation [Mul27]. One year later Frederick Griffith discovered the *transforming principle*: He showed that a non-virulent strain of *Streptococcus pneumoniae* can become virulent if mixed with dead *Streptococcus pneumoniae* from a virulent strain [Gri28].

In 1944 Oswald Theodore Avery et al. showed that the substance underlying the transforming principle can be destroyed by the deoxyribonuclease enzyme [AMM44].

On these previous breakthroughs Alfred Day Hershey and Martha Chase could build upon with their famous Hershey-Chase experiment. They conducted two similar experiments. In the first one, T2 phages with radioactive ^{32}P-labeled DNA infected an *Escherichia coli* (*E. coli*) culture. In the second one, T2 phages with radioactive ^{35}S-labeled protein infected an *E. coli* culture. In both experiments they separated the phages from the bacteria after infection with a blender and measured where most of the radioactivity could be found. In the first case, most of the radioactivity could be measured in the *E. coli* cells and in the second case, most of it in the phages. This proved that DNA and not protein, as it was thought at the time, is the carrier of genetic information [HC52].

2.9.5 Gene as Transcribed Code

After DNA has been identified as heredity carrier the structure of this molecule became the focus of scientific concern and was successfully solved by James Dewey Watson and Francis Harry Compton Crick in 1953 [WC53] (partially based upon earlier work by Erwin Chargaff [Cha51]). This was an important starting point for the field of molecular biology which developed rapidly thereafter. Shortly after this discovery the *Central Dogma* of molecular biology was formulated (see Section 2.6) and the *genetic code* was deciphered.

The *genetic code* is a set of rules which describe how triplets of nucleic acids (DNA or RNA) are translated into a protein. It was solved in 1965 by Marshall Nirenberg et al. [NLB$^+$65] and Dieter Söll et al. [SOJ$^+$65] (see also Section 2.4).

2.9.6 Gene as ORF Sequence Pattern

After the genetic code has been solved the actual nucleotide sequence of genes became the focus of scientific attention. The bacteriophage MS2 was the first fully sequenced organism and its sequenced contained the first gene whose sequence was known [FCDW$^+$71, FCD$^+$76].

Furthermore, the first algorithms for the identification of genes based on their sequence characteristics were developed (see Chapter 3 for a detailed introduction to gene prediction methods). These methods effectively identified genes as annotated ORFs.

These developments led to the new concept of a “nominal gene”, which is defined by the predicted sequence in contrast to the genomic locus used in earlier definitions [GS06].

2.9.7 Gene as Annotated Genomic Entity Stored in a DB (pre-ENCODE)

The predominant view today sees a gene as an annotated genomic entity which is stored in the public databases. This was the exclusive view before the ENCODE project [Con04] started (pre-ENCODE). The findings of the ENCODE project have challenged and refined this view (see Section 2.9.10).

This view can be exemplified with the following two recent definitions. The Human Genome Nomenclature Organization defined a Gene as "a DNA segment that contributes to phenotype/ function. In the absence of demonstrated function a gene may be characterized by sequence, transcription or homology" [WBL+02, p. 464]. A couple of years later the Sequence Ontology Consortium defined a gene as "a locatable region of genomic sequence, corresponding to a unit of inheritance, which is associated with regulatory regions, transcribed regions and/or other functional sequence regions" [Pea06, p. 401].

The increasing number of completely sequenced genomes, like the first completely sequenced genome of *Haemophilus influenzae* [FAW+95] and the human genome [Int01, VAM+01], largely increased the amount of sequences this definitions could be applied to. This even led to a large interest of the public in the number of genes of various organisms, especially human.

2.9.8 Problems with the Current Definition

The sequence centric view of a gene which is currently most dominant has a number of problems which will be discussed below.

Gene regulation

Because regulatory elements can be far away (sequence-wise not chromatin structure-wise) from the actual transcribed sequence it is unclear how they fit into the view of a gene as a compact genetic locus. Furthermore, many-to-many relationships between genes and their enhancers have been found [SLTF05].

Overlapping

Genes with overlapping reading frames have been found [CRVdVF77] which weakens the currently used gene concept further. Even genes which are completely contained in the introns of other genes have been reported [HKFF86].

Splicing

When *splicing* was discovered in 1977 [BMS77, CGBR77, GR77] it became clear that a single locus can transcribe multiple distinct mRNA products which complicates possible gene definitions considerably.

Trans-splicing

The ligation of two separate mRNA molecules in a process called *trans*-splicing [Blu05] completely nullifies the concept of a gene as "a locus" for such products (where the actual DNA can be widely separated across the genome).

Tandem chimerism

In a recently discovered phenomenon called *tandem chimerism* two consecutive genes are transcribed into a single mRNA which leads to a fused protein [ATE+06, PRD+06].

Transposons

A *transposon* (transposable element) is a DNA sequence which is able to insert itself at a new location in the genome. Transposons were first discovered in the 1930s in maize and later it was confirmed that they exists in a wide variety of organisms, including human [McC48]. When a gene is part of a transposon, it can no longer be thought of as having a fixed location.

2.9.9 Experience of the ENCODE Project

The experience of the ENCODE project has shown that the exceptions described above are much more common then previously thought [Con07]. These findings stem mainly from the mapping of transcriptional activity and regulation using tiling arrays. They are described below and should make clear why an updated definition of *gene* became necessary.

Unannotated transcription

The results of the ENCODE project confirmed earlier results [BSR+04, CKD+05] that a large amount of DNA which is not annotated as known gene is transcribed into RNA [Con07]. These novel transcribed regions are called *transcriptionally active regions* (TARs). "While the majority of the genome appears to be transcribed at the level of primary transcripts, only about half of the processed (spliced) transcription detected across all the cell lines and conditions mapped is currently annotated as genes" [GBR+07, p. 673].

Unannotated and alternative TSSs

Furthermore, it was found that a large number of unannotated transcription start sites (TSSs) exist and that many known protein genes have alternative TSSs which can be more then 100 kb upstream of the annotated TSSs [Con07].

More alternative splicing

Motivated by these findings the well-curated GENCODE annotation was made at the Sanger Institute [HDF+06]. GENCODE showed that most of the new transcripts do not correspond to previously unknown genes but rather to previously unknown alternative splice forms of known protein-coding genes or to novel ncRNAs. In the current GENCODE annotation a gene contains on average 5.4 transcripts.

Dispersed regulation

The results of the ENCODE project show that regulation is dispersed throughout the genome and does not necessarily lie upstream of the regulated gene. Despite the fact that regulatory elements can be found in the entire genome, evidence has been found that the genome contains regulatory rich "forests" and regulatory poor "deserts" [ZPF+07].

Furthermore, large-scale evidence has been found that some of the regulatory elements themselves are transcribed and hence should be considered as part of a gene which contradicts the traditional gene model [CBN+04, ERB+04, KBZ+05, Con07, ZPF+07].

Noncoding RNAs

A part of the rich transcriptional activity observed by the ENCODE project is due to *non-coding RNAs* (ncRNAs). Non-coding RNA is an umbrella term for all sorts of RNAs which do not code for a protein sequence. This includes micro RNAs (miRNAs) used for gene regulation, small nucleolar RNAs (snoRNAs) used for RNA processing, and transfer RNAs (tRNAs) as well as ribosomal RNAs (rRNAs) used for protein synthesis.

Genes coding for ncRNAs are computationally hard to identify, because they do not necessarily contain ORFs.

Pseudogenes

Another part of the rich transcriptional activity which cannot be attributed to protein-coding genes belongs to pseudogenes. *Pseudogenes* are derived from functional genes through retrotransposition or duplication.

Their prevalence (in the same order of magnitude as protein-coding genes) and their similarity to protein-coding genes makes distinguishing them from protein-coding genes hard. Recent evidence shows that up to 20% of them are transcribed which makes this even worse, because transcription can no longer be taken as good evidence for locating genes [YSY+03, HZZ+05, ZZH+05, ZFB+07, FWF+06]

2.9.10 Gene as Dispersed Genome Activity (ENCODE)

Since many of the found ncRNAs and pseudogenes are located within introns of protein-coding genes it became clear that a gene definition based on overlapping transcripts would coalesce functionally very different entities into single genes which is clearly not desired.

This lead to three aspects which have to be included in an updated gene definition (taken from [GBR+07, p. 676-677]):

1. A gene is a genomic sequence (DNA or RNA) directly encoding functional product molecules, either RNA or protein.
2. In the case that there are several functional products sharing overlapping regions, one takes the **union** of all overlapping genomic sequences coding for them.
3. This union must be **coherent**—i.e., done separately for final protein and RNA products—but does not require that all products necessarily share a common subsequence.

This can be summarized as [GBR+07, p. 677]:

> "*The gene is a union of genomic sequences encoding a coherent set of potentially overlapping functional products.*"

Figure 2.4 gives an example on how to apply the updated gene definition. Under the pre-ENCODE definition the example DNA would have been annotated as a single gene (consisting of exons A, B, C, D, and E), because the various transcripts overlap and would therefore be considered as part of the same gene (despite their different functions).

In the rest of this thesis the prediction of protein-coding genes will be considered in-depth. The different properties of genes coding for ncRNAs (see Section 2.9.9) renders predicting them a different problem which will not be considered here.

The computational approach to gene finding discussed in this thesis consists of aligning cDNAs/ESTs and/or protein sequences to genomic DNA (gDNA, for short) and thereby identifying the exons and 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 and deletions (so-called 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 of the *GenomeThreader* software come from public domain projects. Internationally maintained public databases such as the database resources of the National Center for Biotechnology Information (NCBI) [WBB+05] or the EMBL Nucleotide Sequence Database [KAA+05] provide access to very large numbers of DNA, RNA, and protein sequences. For example, at the time of writing this thesis, the entire genomes of human,

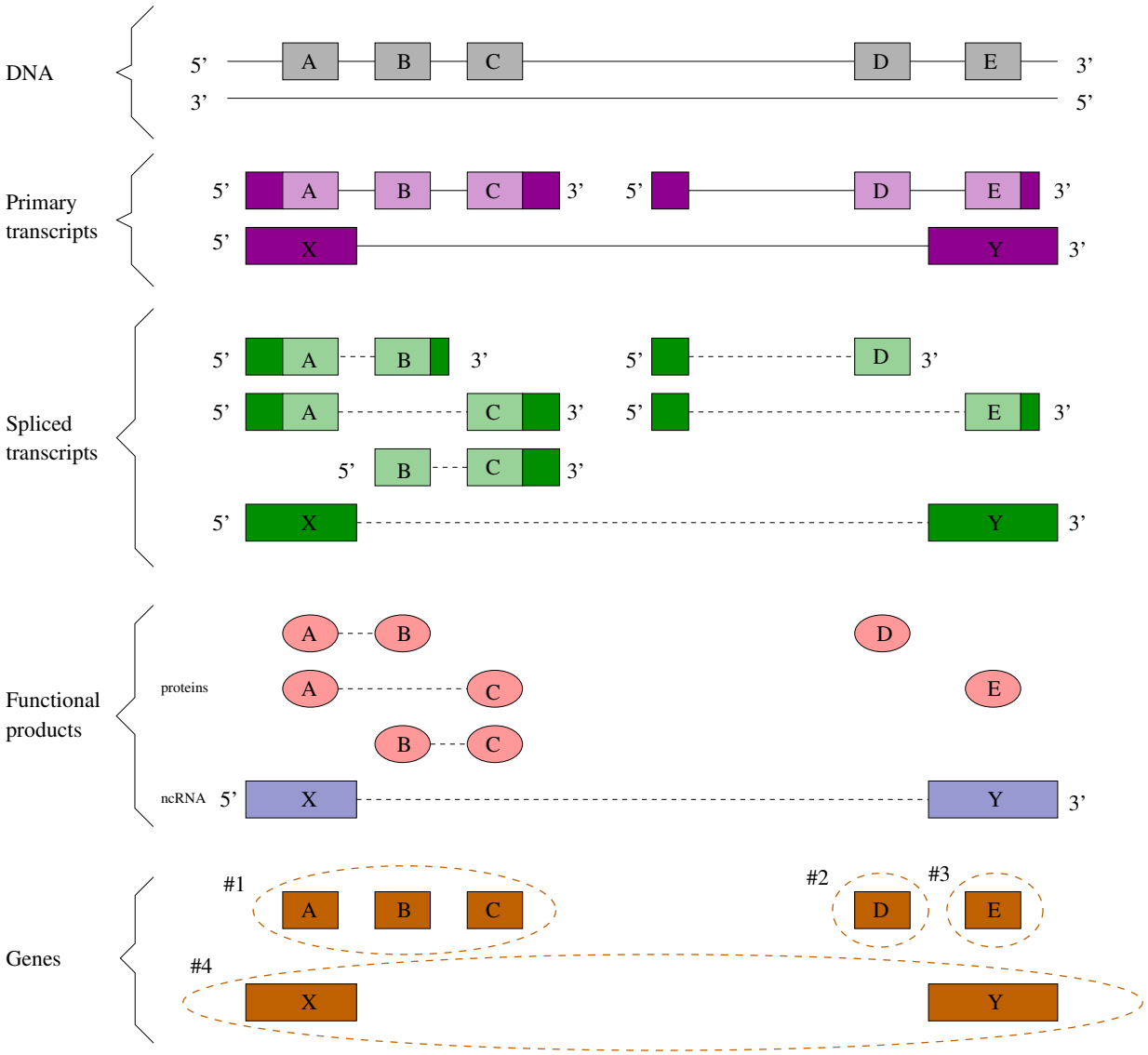


Figure 2.4: Updated gene definition example. Diagram adapted from [GBR+07]. The DNA shown in the example (A-E) produces three primary transcripts which lead to six spliced transcripts. These six transcripts encode five different proteins and one of them is a ncRNA. Because the three protein A, B, and C overlap on the DNA level they are combined to gene #1. The proteins D and E do not overlap on the DNA level and therefore they belong to separate genes #2 and #3. Since the ncRNA consisting of X and Y is a different “product category” it constitutes a separate gene #4.

chimpanzee, mouse, rat, and dog are available for download, as well as several million EST sequences for human and mouse, respectively. The genomes of many plants (for example, *Arabidopsis thaliana*, a laboratory model organism, and rice) are also available, and sequencing of the world-wide most important crop, maize, is finished [SWF+09]. 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.

In the next chapter different approaches for predicting the structure of protein-coding genes which have been described in the scientific literature will be introduced. Furthermore, the measures of how to assess the quality of gene predictions and some example programs in the different gene prediction categories will be shown.

Chapter 3

Computational Background

Four main categories of gene prediction methods exist: *Ab initio* methods, comparative methods, homology methods, and so-called combiners. Methods and programs implementing them will be described below after precisely defining the notion "gene prediction". Afterwards measures for the quality of gene prediction methods will be introduced. Finally, for each category a few programs will be introduced in more detail. For reviews on computational gene prediction methods and their accuracy see for example [CFM$^+$08, GFA$^+$06, Bre05, GR05, BG04, GW03, Bre02, HVT$^+$02, MSSR02, Zha02, RMO01, GAA$^+$00, RHH$^+$00, Sto00, PRD$^+$99, BK98, BG96].

3.1 Basic Definitions

$\mathbb{N}$ denotes the set of positive integers including 0. The symbols $a, b, i, j, k, l, m, n, o, t, u, v, w$ refer to integers if not stated otherwise.

Let $\mathcal{A}$ be a finite set, the *alphabet*. The elements of $\mathcal{A}$ are *characters*. Strings[1] are written by juxtaposition of characters. In particular, ε denotes the *empty sequence*. The set $\mathcal{A}^*$ of *sequences over* $\mathcal{A}$ is defined by

$$\mathcal{A}^* = \bigcup_{i \geq 0} \mathcal{A}^i$$

where $\mathcal{A}^0 = \{\varepsilon\}$ and $\mathcal{A}^{i+1} = \{aw \mid a \in \mathcal{A}, w \in \mathcal{A}^i\}$. $\mathcal{A}^+$ denotes $\mathcal{A}^* \setminus \{\varepsilon\}$. The *length* of a sequence s, denoted by $|s|$, is the number of characters in s. s_i is the i-th character of s. That is, if $|s| = n$, then $s = s_1 \dots s_n$ where $s_i \in \mathcal{A}$. If $i \leq j$, then $s_i \dots s_j$ is the substring of s beginning with the i-th character and ending with the j-th character. These notions have been taken from [Kur03] (with slight modifications).

A substring $s_i \dots s_j$ is also denoted as $s(i, j)$. From now on let $\mathcal{A} = \{A, C, G, T, N\}$, where A refers to adenine, C to cytosine, G to guanine, T to thymine and N denotes an *undetermined*

[1] The term *string* is equivalent to the term *sequence*. But the term *substring* is not equivalent to the term *subsequence*: consecutive characters in substrings have to be consecutive in the originating string, whereas this is not mandatory in subsequences.

character. An element $b \in \mathcal{A}$ is also called a *base*. Let $g \in \mathcal{A}^+$ be a *genomic sequence* ($|g| = n$) and $c \in \mathcal{A}^+$ be a *cDNA/EST sequence* ($|c| = m$).

Definition 1 Let $g = g_1 g_2 \dots g_n$ be a genomic sequence. The total function $\varrho : [1, n] \to \{c, nc\}$ is called a *status function*, where *c* stands for *coding* and *nc* stands for *non-coding*. That is, by means of ϱ every position t, $1 \leq t \leq n$, has a status $\varrho(t)$ assigned. Note that ϱ is a boolean function. We however use the two symbols *c* and *nc* for ease of readability. □

Definition 2 Given a genomic sequence $g = g_1 g_2 \dots g_n$ and a status function ϱ. An *exon* according to ϱ is a pair (i, j) of positions such that $1 \leq i \leq j \leq n$ where

- $\varrho(t) = c$ for all $t \in [i, j]$,
- $i = 1$ or $\varrho(i - 1) = nc$,
- $j = n$ or $\varrho(j + 1) = nc$.

That is, an exon denotes a maximal substring $g_i \dots g_j$ such that each position is a coding position. A predicate $exon$ is defined as follows:

$$exon(g, \varrho, i, j) = \begin{cases} \textit{True} & \text{if } (i, j) \text{ is an exon according to } \varrho \\ \textit{False} & \text{otherwise} \end{cases}$$

□

Definition 3 Let g be a genomic sequence and ϱ be a status function. Assume all exons of g (namely $(i_1, j_1), \dots, (i_k, j_k)$) are numbered from 1 to k from left to right. A partition of the set $\{1, \dots, k\}$ into intervals $[a_1, b_1], [a_2, b_2], \dots, [a_l, b_l]$ is called a *grouping* if

- $a_1 = 1$,
- $b_t + 1 = a_{t+1}$ for $t \in [1, l - 1]$, and
- $b_l = k$.

A single interval of a grouping is called a *gene*.

The *exon mapping* function $\varphi : [1, k] \to \mathbb{N} \times \mathbb{N}$ maps the set $\{1, \dots, k\}$ given above on the according exons $(i_1, j_1), \dots, (i_k, j_k)$. □

Definition 4 Given a genomic sequence g, a status function ϱ, and a grouping (of the resulting exons) $[a_1, b_1], [a_2, b_2], \dots, [a_l, b_l]$. We map the intervals (of the grouping) on genomic substrings by means of the exon mapping function φ:

$$\begin{aligned} & [\varphi(a_1), \varphi(b_1)], [\varphi(a_2), \varphi(b_2)], \dots, [\varphi(a_l), \varphi(b_l)] \\ = \; & [g(i_{a_1}, j_{a_1}), g(i_{b_1}, j_{b_1})], [g(i_{a_2}, j_{a_2}), g(i_{b_2}, j_{b_2})], \dots, [g(i_{a_l}, j_{a_l}), g(i_{b_l}, j_{b_l})] \end{aligned}$$

The substrings $g(0, i_{a_1} - 1)$, $g(j_{b_t} + 1, i_{a_t+1} - 1)$ for $t \in [1, l-1]$, and $g(j_{b_l} + 1, n)$ are called *intergenic regions*.

Colloquially, intergenic regions are the genomic regions between genes. □

Definition 5 Given a genomic sequence g, a status function ϱ, and a grouping. In this definition only the intervals (of the grouping) $[a, b]$ where $a \neq b$ are considered. In such intervals $a, a + 1, \ldots, b$ denotes a numerical order of exons. By φ every exon can be expressed as a subsequence of g:

$$\begin{aligned} & \varphi(a), \varphi(a+1), \ldots, \varphi(b) \\ = \; & g(i_a, j_a), g(i_{a+1}, j_{a+1}), \ldots, g(i_b, j_b) \end{aligned}$$

The substrings $g(j_t + 1, i_{t+1} - 1)$ for $t \in [a, b-1]$ are called *introns*.

For all $t \in [a, b-1]$: If $|g(j_t+1, i_{t+1}-1)| \geq 4$ then the substring $g(j_t+1, j_t+2)$ is called *donor site* and the substring $g(i_{t+1} - 2, i_{t+1} - 1)$ is called *acceptor site*. Donor and acceptor sites are also called *splice sites*.

Colloquially, introns are the genomic regions between the exons of a gene. □

Definition 6 Given a genomic sequence g. A *gene prediction method* computes a status function ϱ of g and a grouping of the resulting exons. The result of a gene prediction method is called a *gene prediction*. □

After a gene prediction a genomic sequence g is a sequence of substrings of the type *exon*, *intron*, and *intergenic region* (according to the definitions above).

Remark 1 The definitions above refer to single stranded DNA for simplicity. Gene prediction methods are usually applied to both strands of the genomic DNA, the *forward* strand and the *reverse complementary* strand. □

Now we define the more special case of a gene prediction method via spliced alignment. A definition of an *optimal* spliced alignment is given in Sections 4.1.2 and 4.1.3, where the spliced alignment algorithm of *GeneSeqer2* and *GenomeThreader* is described.

For this purpose we need the definition of an *edit operation* and of an *alignment* (both taken from [Kur03], with slight modifications):

Definition 7 An *edit operation* is a pair $(\alpha, \beta) \in (\mathcal{A}^1 \cup \{\varepsilon\}) \times (\mathcal{A}^1 \cup \{\varepsilon\}) \setminus \{(\varepsilon, \varepsilon)\}$. □

α and β denote *sequences* of length ≤ 1. However, if $\alpha \neq \varepsilon$ and $\beta \neq \varepsilon$, then the edit operation (α, β) is identified with a pair of characters.

An edit operation (α, β) is usually written as $\alpha \to \beta$. This reflects the operational view which considers edit operations as rewrite rules transforming a source sequence into a target sequence, step by step. In particular, there are three kinds of edit operations:

- $\alpha \to \varepsilon$ denotes the *deletion* of the character α,
- $\varepsilon \to \beta$ denotes the *insertion* of the character β,
- $\alpha \to \beta$ denotes the *replacement* of the character α by the character β.

Notice that $\varepsilon \to \varepsilon$ is not an edit operation. Insertions and deletions are sometimes referred to collectively as *indels*.

Sometimes sequence comparison just means to measure how different sequences are. Often it is additionally of interest to analyze the total difference between two sequences into a collection of individual elementary differences. The most important mode of such analyses is an alignment of the sequences.

Definition 8 An *alignment* A of two sequences x and y ($x, y \in \mathcal{A}^*$) is a sequence

$$(\alpha_1 \to \beta_1, \ldots, \alpha_h \to \beta_h)$$

of edit operations such that $x = \alpha_1 \ldots \alpha_h$ and $y = \beta_1 \ldots \beta_h$. □

Example 1 The alignment $A = (\varepsilon \to d, b \to b, c \to a, \varepsilon \to d, a \to a, c \to \varepsilon, d \to d)$ of the sequences $bcacd$ and $dbadad$ is written as follows:

$$\begin{pmatrix} - & b & c & - & a & c & d \\ d & b & a & d & a & - & d \end{pmatrix}$$

□

(Excerpt from [Kur03] ends here.) Now we are able to define a spliced alignment:

Definition 9 Given a genomic sequence g, a cDNA/EST sequence c, and a gene prediction of g containing exactly on gene. The predicted exons are numbered from 1 to k from left to right: $g(i_1, j_1), g(i_2, j_2), \ldots, g(i_k, j_k)$.

A *spliced alignment* is defined as an alignment between $g(i_1, j_1)g(i_2, j_2) \ldots g(i_k, j_k)$ and $c_1 c_2 \ldots c_m$. □

Definition 10 Given a genomic sequence g and a cDNA/EST sequence c. A *gene prediction via spliced alignment method* (or simply a *spliced alignment method*) computes a gene prediction of g containing one gene and a spliced alignment. □

Remark 2 Again, this is a simplified view: The DNA sequences processed in the actual algorithms contain more than one gene. And usually a set of cDNA/EST sequences is used. □

3.2 Gene Prediction Categories

3.2.1 *Ab initio* Methods

Ab initio methods are purely based on two kinds of inputs: The genomic sequence in which the gene(s) shall be predicted. And a statistical model, which captures features which are specific for genes, e.g. exon and intron features.

Early approaches tried to recognize shifts in *codon usage*. The idea is, that coding sequences have a different bias in the use of the 64 possible codons than the noncoding ones. These approaches have not been very successful, mainly due to the short length of coding regions. In eukaryotes the situation becomes even more complicated: Due to exon-intron structure of the genes the coding regions are too short to be recognized. The average length of an exon in vertebrates is 130 bp [Pev00, p. 156]. This led to more realistic modeling trying to capture more biological knowledge, like modeling the splice sites.

For example, the program *GENSCAN* [BK97] merges in a *Hidden Markov Model* the statistics of splice sites, promoters, polyadenylation sites and coding region statistics. It also takes into account the influence of the $G + C$ composition on gene structure and gene density. Beside *GENSCAN* there are many other *ab initio* methods, for example *mGene* [SZZ$^+$09], *EuGÉNE* [FS05], *AUGUSTUS* [SW03], *Fgenesh* [SS00], *GeneGenerator* [KHV$^+$98], *GeneMark.hmm* [LB98], *Morgan* [SDFH98], *HMMgene* [Kro97], *MZEF* [Zha97], and *Genie* [KHRE96].

Ab initio programs use *training sets* to adjust their model parameters. Training sets contain positive or negative test samples. Positive samples contain sequences with known genes, where the complete gene structure is already correctly annotated. Negative samples contain for example *pseudogenes*[2], which shall not be predicted as genes. With the training sets the parameters can be adjusted. For example, it is possible to derive statistics on the splice sites by simply computing frequencies or consensus splice sites. The splice sites statistics can then be used when one tries to recognize splice sites. In *GenomeThreader* splice site statistics are used via Bayesian Splice Site Models (BSSMs, see Section 4.2).

3.2.2 Comparative Methods

The idea of these methods is that exons and certain signals (like promoters, donor- and acceptor sites and so on) are more conserved between related species than other sequence parts, because mutations in these would corrupt functionally important proteins, change expression level or disturb splicing. One wants to detect this stronger conserved regions by means of *comparative genomics* of two or more genomic sequences.

[2] *Pseudogenes* are derived from ancestral genes by duplication events. In contrast to genes they are not functionally active due to inactivating mutations.

Programs of this type are *mGene* [SZZ+09], *N-SCAN* [GB06], *SGP2* [PAA+03], *AGenDa* [TRG+03], *TWINSCAN* [KFDB01] (derived from *GENSCAN* [BK97]), *SLAM* [PAC01], *CEM* [BH00], and *ROSETTA* [BPM+00].

One problem of these approaches is that the rate of evolution varies among species. Therefore it is not trivial to choose appropriate species for these methods. The rate of evolution can also be different between genes in a single species, which makes the situation even more difficult. Ideally one would like to have a method of this kind, which takes a single genomic sequence and finds suitable sequences to compare it with by a similarity search in a database.

The publication [GW03] comprehensively reviews the comparative gene prediction field.

3.2.3 Homology methods

Homology methods are also called the similarity-based approach. The idea of the similarity-based approach is to use additional sequence information, either ESTs or Proteins to predict gene structures.

We call these sequences *reference* sequences in contrast to the *genomic* sequence (or template). Appropriate means that the ESTs or Proteins are from the same species or (if not available) from closely related species.

ESTs contain, beyond question, important information because they are derived from mature mRNAs (as described in Section 2.7), which are spliced transcripts of genes. Therefore, if a genomic region shows similarity to a reference sequence from a cDNA/EST (or protein) database, it is more likely that this genomic region contains a coding sequence. Similar regions can be found with programs like *BLAST* [AGM+90], *FASTA* [PL88, Pea00] or *Vmatch* (`http://vmatch.de/`).

Similarity does not reveal the exon-intron structure of a putative gene, though. Therefore, in the matching regions a *spliced alignment* is computed, which reveals the exon-intron-structure. Figure 4.1 shows an example of a spliced alignment. Methods of the similarity-based approach using ESTs or Proteins are also called *spliced alignment methods*. Known programs which compute a spliced alignment of a cDNA/EST with a homologous genomic sequence segment are *GMAP* [WW05] and *sim4* [FHZ+98].

Programs which use protein sequences for a similarity-based approach are, for example, *GENEWISE* [BD97] and *PROCRUSTES* [GMP96].

Further programs which should be considered as homology methods, because they use reference sequence information, are *mGene* [SZZ+09], *AUGUSTUS* [SDBH08] and *EuGÉNE* [FS05].

GeneSeqer can use both, cDNAs/ESTs (described in [UZB00]) and proteins (described in [UB00]) as reference sequences. *GeneSeqer* is not a pure spliced alignment method, because it also takes the splice site strength into consideration. This makes it a hybrid approach. Since *GenomeThreader* [GBSK05] extends upon ideas from *GeneSeqer*, it can also use cDNA/EST and protein sequences at the same time to predict gene structures. But *GenomeThreader* is the only

program known to us which can combine *both* cDNA/EST and protein based spliced alignments into consensus spliced alignments.

3.2.4 Combiners

Combiners are a special breed of gene prediction programs, they use the gene models created by other gene prediction methods and possibly additional data and *combine* them into their own gene models, thereby refining the input models.

They are usually among the best gene prediction methods available, but it is not fair to compare them with the other three categories, because combiners would not exist without the other programs from which they draw their initial gene models.

Example programs of this category are *Evigan* [LMRP08], *GENOMIX* [CD07], *GLEAN* [EMR+07], *JIGSAW* [AS05], and *GeneID* [PBG00].

3.3 Measures of Prediction Accuracy

To be able to make statements about the quality of gene prediction methods, one has to compare the predicted gene structure with the real, known gene structure. This can be done by measures at different levels of detail. There are at least three reasonable levels of detail: The nucleotide level, the exon level and the whole-gene level. On the nucleotide level it is measured how many of the coding nucleotides have been predicted to be coding. On the exon level it is measured how many complete exons have been predicted entirely correctly and on the whole-gene level how many complete gene-structures have been entirely correctly predicted. Below some widely used measures for the different levels will be defined following the "classical" paper from Burset and Guigó [BG96] on gene prediction evaluation, the review in [GW03], and the human ENCODE Genome Annotation Assessment Project (EGASP) [GFA+06].

3.3.1 Nucleotide Level

The definition of the quality measures are based on the formal definitions of Section 3.1. We are comparing prediction and reference, which both can be seen as status functions of the same genomic sequence g: ϱ_p and ϱ_r, respectively. The prediction status function ϱ_p is derived from a gene prediction, whereas the reference status function ϱ_r comes from an annotation of g which (ideally) reflects the reality.

Let us consider two bases at the same sequence position t, one from the prediction, one from the reference: Each base can be either *coding* or *non-coding*, as said in the definitions. When alternative splicing exists where a base exists that is coding in one splice form as well as non-coding in another, that base is defined as *coding*. That is, coding bases take precedence over non-coding bases in the same genomic region. See Figure 3.2 for a graphical explanation of this

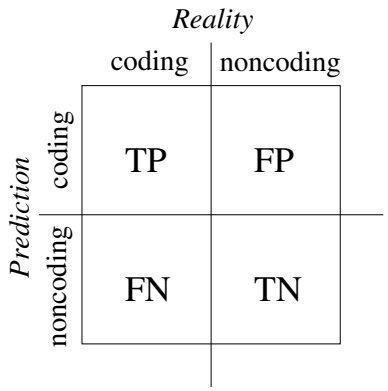


Figure 3.1: 2×2 table representing the possible status combinations from *prediction* and *reference*. Adapted from [BG96].

idea. This means four possible combinations (see Figure 3.1 for visualization). Counting the frequency of each combination along the sequences leads to four measures:

- TP: Number of positions, where both bases are *coding*.
  Formally, $TP = |\{t \in [1, n] \mid \varrho_p(t) = \varrho_r(t) = c\}|$.
- TN: Number of positions, where both bases are *non-coding*.
  Formally, $TN = |\{t \in [1, n] \mid \varrho_p(t) = \varrho_r(t) = nc\}|$.
- FP: Number of positions, where the prediction base is labeled *coding*, but in reality the base is *non-coding*. Formally, $FP = |\{t \in [1, n] \mid \varrho_p(t) = c \text{ and } \varrho_r(t) = nc\}|$.
- FN: Number of positions, where the prediction base is labeled *non-coding*, but in reality the base is *coding*. Formally, $FN = |\{t \in [1, n] \mid \varrho_p(t) = nc \text{ and } \varrho_r(t) = c\}|$.

Hereof the two well known measures *sensitivity* (denoted as M^{nuc}_{sens}) and *specificity* (denoted as M^{nuc}_{spec}) can be derived:

$$M^{nuc}_{sens} = \frac{TP}{TP + FN} \qquad M^{nuc}_{spec} = \frac{TP}{TP + FP}$$

The sensitivity M^{nuc}_{sens} denotes the proportion of nucleotides correctly predicted as coding from all coding nucleotides and the specificity M^{nuc}_{spec} the proportion of nucleotides correctly predicted as coding from all as coding predicted nucleotides.

One can also define the "negated" measures for sensitivity (denoted as $\neg M^{nuc}_{sens}$) and specificity (denoted as $\neg M^{nuc}_{spec}$), which focus on *non-coding*[3] instead of coding nucleotides:

$$\neg M^{nuc}_{sens} = \frac{TN}{TN + FP} \qquad \neg M^{nuc}_{spec} = \frac{TN}{TN + FN}$$

[3]That is, the sensitivity $\neg M^{nuc}_{sens}$ denotes the proportion of nucleotides correctly predicted as *non-coding* from all *non-coding* nucleotides and the specificity $\neg M^{nuc}_{spec}$ the proportion of nucleotides correctly predicted as *non-coding* from all as *non-coding* predicted nucleotides.

Neither sensitivity nor specificity alone represent a good measure about prediction quality, because it is possible to gain high values for each of them without having a meaningful prediction: Simply predicting all values as coding would yield in an M_{sens}^{nuc} value of 1 and a very low M_{spec}^{nuc} value. Predicting only very few nucleotides as coding (correctly) would yield in a very high M_{spec}^{nuc} value but in a very low M_{sens}^{nuc} value. That means one has always to consider both values. This is somewhat inconvenient, and therefore there are approaches to combine these two measures into a single one. A very common one is the *correlation coefficient* (CC) (see [BG96, p. 357]):

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

The correlation gives a combined value of sensitivity and specificity ranging from -1 to 1 but has the disadvantage of being not defined if one of the sums $TP + FN$, $TN + FP$, $TP + FP$ and $TN + FN$ equals zero. This led to the development of other measures, which overcome this disadvantage. The *approximate correlation* (AC) ranges from -1 to 1 and is defined by:

$$AC = (ACP - 0.5) \times 2$$

where the *average conditional probability* ACP (ranging from 0 to 1) is defined by:

$$ACP = \frac{1}{4}\left[M_{sens}^{nuc} + M_{spec}^{nuc} + \neg M_{sens}^{nuc} + \neg M_{spec}^{nuc}\right]$$

But these combined values are not used widely in practice. For example, EGASP [GFA+06] uses only CC and nGASP [CFM+08] uses no combined value at all (see Chapter 6 for more information on these evaluation projects).

Remark on notation

Usually the sensitivity is denoted as S_n and the specificity as S_p. However, this leads to ambiguity when defining sensitivity and specificity on exon and gene level, which are usually denoted the same. Therefore here we use M_{sens} for the sensitivity measure and M_{spec} for the specificity measure with superscript strings denoting the level of the prediction. For nucleotide level, we use nuc (i.e. M_{sens}^{nuc} and M_{spec}^{nuc}), for the exon level ex (i.e. M_{sens}^{ex} and M_{spec}^{ex}), and for the gene level gen (i.e. M_{sens}^{gen} and M_{spec}^{gen}).

3.3.2 Exon Level

An exon is considered to be predicted correctly, if its borders are exactly the same as in the annotation. TE denotes the number of such *true exons* and AE the number of *actual exons* in the annotation. PE is the total number of *predicted exons*. Formally:

Definition 11 Given a genomic sequence g and two status functions ϱ_p and ϱ_r reflecting prediction and reference. We define:

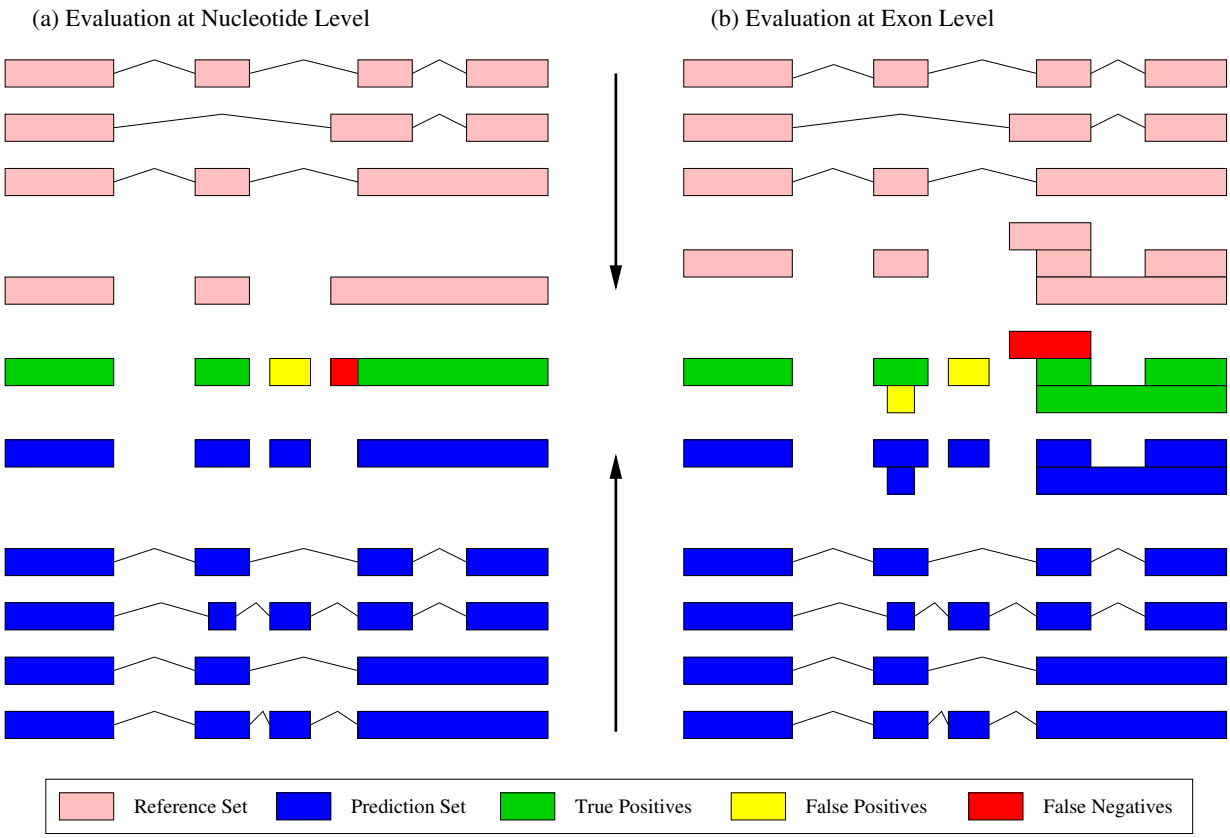


Figure 3.2: Gene feature projection for evaluation, adapted from [GFA$^+$06]. This figure shows the process of projecting genic features into unique nucleotide (a) and exon (b) coordinates in order to compute the accuracy values (see Sections 3.3.1 and 3.3.2).

- $TE = |\{(i,j) \mid 1 \leq i \leq j \leq n, exon(g, \varrho_p, i, j) = exon(g, \varrho_r, i, j) = \textbf{\textit{True}}\}|$
- Let the exons of g according to ϱ_p be numbered from 1 to k_p. Then $PE = k_p$.
- Let the exons of g according to ϱ_r be numbered from 1 to k_r. Then $AE = k_r$.

Definition 12 With Definition 11 *sensitivity* M^{ex}_{sens} and *specificity* M^{ex}_{spec} on exon level are defined:

$$M^{ex}_{sens} = \frac{TE}{AE} \qquad M^{ex}_{spec} = \frac{TE}{PE}$$

On the exon level one can combine sensitivity and specificity by evaluating their average. Two other important measures are the number of actual exons without any overlap with a prediction (the *missing exons* ME) and the number of predicted exons without overlap to any actual exons (the *wrong exons* WE).

In the case of alternative splicing, the procedure to evaluate the exon level is similar to the nucleotide level. That is, all exons from different splice forms with the same borders are *collapsed* into one. Each (possibly collapsed) exon from the reference set has to be predicted in at least one splice form from the prediction set to be considered correctly predicted. See Figure 3.2 for a graphical explanation of this idea.

3.3.3 Gene Level

The overall prediction accuracy on the gene level is the most important one. Considered are complete genes. A gene is predicted correctly if the same exons (as in the annotation) are assigned to that gene *and* all of these exons are predicted correctly (as defined above).

Based on this the *sensitivity* (denoted as M^{gen}_{sens}) on the gene level can be defined as the fraction of all genes which are predicted correct and the *specificity* on the gene level (denoted as M^{gen}_{spec}) as the fraction of all predicted genes which are correct.

Definition 13 Given a genomic sequence g, two status functions ϱ_p and ϱ_r, and two groupings of the resulting exons. The set of intervals of the grouping for ϱ_p is denoted I_p. Accordingly, the set of intervals of the grouping for ϱ_r is denoted I_r. We define:

- $TG = |I_p \cap I_r|$
- Let the intervals of the groupings for ϱ_p be numbered from 1 to l_p. Then $PG = l_p$.
- Let the intervals of the groupings for ϱ_r be numbered from 1 to l_r. Then $AG = l_r$.

Definition 14 With Definition 13 *sensitivity* M^{gen}_{sens} and *specificity* M^{gen}_{spec} on gene level are defined:

$$M^{gen}_{sens} = \frac{TG}{AG} \qquad M^{gen}_{spec} = \frac{TG}{PG}$$

Also common is the measure of the *missing genes* (MG), i.e. genes which are totally missed in the prediction. And *wrong genes* (WG), i.e. predicted genes with no overlap to any existing gene.

3.4 Prediction Accuracy

In this section some actual values for the measures and programs introduced above will be given.

Much effort was put into computational gene prediction methods, but the results are still dissatisfying. The best methods are the ones designed for specific organisms.

Generally speaking the best recognition rates in *C. elegans* are as follows (according to the nGASP project [CFM+08]):

- Nucleotide level: $> 99\%$ sensitivity and $> 93\%$ specificity (best programs).
- Exon level: $> 91\%$ sensitivity and $> 83\%$ specificity (best programs).
- Gene level: a median of 68% (sequence-based) and 54% (*ab initio*) sensitivity, as well as a median of 39% (sequence-based) and 32% (*ab initio*) specificity.

Accuracy values for many different gene finders on the nGASP dataset can be found in Section 6.1. In human the recognition rates are as follows (according to the EGASP project [GFA+06]):

- Nucleotide level: $> 90\%$ sensitivity and specificity.
- Exon level: $> 80\%$ sensitivity and nearly 90% specificity.
- Gene level: $> 70\%$ sensitivity and $> 65\%$ specificity.

One has to be careful when looking at quality values like this. It is important to differentiate between prediction accuracy for short genomic sequences containing exactly one gene and large, more realistic, genomic sequences, which contain many genes and possibly also pseudogenes. For short sequences containing exactly one gene, the problem is significantly easier to solve. But in practice, we are now usually confronted with complete genomes. This makes the gene prediction task harder to solve. A fact which is reflected in the accuracy numbers given above.

3.5 Related Work: *Ab initio* Methods

3.5.1 *GENSCAN*

GENSCAN [BK97] was one of the earliest *ab initio* gene structure prediction programs which achieved good prediction results (for the time). It is based on a general probabilistic model. Namely, a Hidden Markov Model (HMM), see Section 4.1.1 for a definition. The model *GENSCAN* uses represent the functional units of a gene as different HMM states, at which each state has a duration function. *GENSCAN* was quite influential in the gene prediction field and many later programs draw from ideas first introduced there.

3.5.2 *AUGUSTUS*

The *ab initio* gene prediction software *AUGUSTUS* [SW03] also used ideas from *GENSCAN* , but extended them with a new way of modeling intron length, new splice site models, and better parameter estimation based on the GC-content of genomic sequences. With this techniques it was able to predict genes on longer genomic sequences better than previous programs. *AUGUSTUS* [SDBH08] was later extended to also take informations from cDNA alignments into account in order to improve its gene finding capabilities.

3.5.3 *mGene*

mGene [SZZ+09] combines the well studied HMMs used in earlier programs with modern machine learning methods, namely Support Vector Machines (SVMs). *mGene* is also capable of taking other genomic sequences (like comparative methods) or other transcriptomic information (like homology methods) into account, which makes it in fact a hybrid approach to gene prediction. It was among the best prediction programs in nGASP [CFM+08], see Section 6.1 for details.

3.6 Related Work: Comparative Methods

3.6.1 *TWINSCAN*

TWINSCAN [KFDB01] was based on ideas from *GENSCAN* , but uses the homology between two related genomes to improve upon the results from *GENSCAN* which makes it a comparative method. It was specifically designed for the analysis of high-throughput genomic sequences.

3.7 Related Work: Homology Methods

3.7.1 *GMAP*

GMAP [WW05] is a widely used similarity-based gene prediction program which aligns cDNA sequences to a genome (*GMAP* stands for Genome Alignment and Mapping Program). It is a stand-alone program which can map single sequences with minimal startup time and allows for fast batch processing of large cDNA sets. It does *not* employ probabilistic splice site models. The *GMAP* methodology consists of the following four phases:

1. A minimal sampling strategy for genomic mapping.
2. Oligomer chaining for approximate alignment.
3. Sandwich DP for splice site detection.
4. Microexon identification with statistical significance testing.

Sequence Mutations

The *GMAP* paper also describes a procedure to mutate sequences which works as follows. For each position in the given sequences it is randomly determined with probability (mutation rate / 100) if the given position is mutated. If so, in 80% of the cases a substitution is performed, in 10% an insertion, and in 10% a deletion, respectively. For substitution and insertion events, the nucleotide is generated randomly without regard to the original nucleotide [WW05, p. 1867].

This procedure was implemented in the `seqmutate` tool which is documented in Section B.17 and was used for the ENCODE evaluation described in Section 6.3.

3.7.2 *EuGÉNE*

EuGÉNE [FS05] is another gene prediction program which uses the similarity-based approach. It uses multiple homologous sequences to predict gene structures. In the nGASP [CFM+08] competition (see Section 6.1 for details) it performed in the mid-range.

3.8 Related Work: Combiners

3.8.1 *JIGSAW*

JIGSAW [AS05] was one of the early programs which successfully automated the integration of multiple sources of evidence (from other gene prediction programs) into gene structures. It usually outperforms *ab initio* gene prediction programs. It is also based on a Hidden Markov Model.

3.8.2 *Evigan*

Evigan [LMRP08] is another combiner which uses a dynamic Bayes network whose parameters are adjusted to maximize the probability of observed evidence instead of a HMM to integrate multiple sources of evidence. It has been successfully validated on the ENCODE regions.

Chapter 4

GenomeThreader Gene Prediction Software

This chapter describes the following six major contributions of *GenomeThreader*, which were already mentioned in the introduction given in Chapter 1, in detail:

1. The intron cutout technique, see Section 4.5.
2. The accompanying chain enrichment, see Section 4.5.4.
3. Jump tables, see Section 4.6.
4. Easy-to-use BSSMs (Bayesian Splice Site Models), see Section 4.2.
5. Incremental updates, see Section 4.8.8.
6. The combination of both cDNA/EST-based spliced alignments **and** protein-based spliced alignments into consensus spliced alignments, see Section 4.7.

4.1 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 and/or protein sequences (depending on the application, this could be a single, for example newly experimentally 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 (the EST lengths were taken from dbEST [BLT93]) . Protein sequences are typically a few hundred amino acids long, but can reach lengths of up to multiple ten thousand amino acids (numbers based on sequences from the curated protein knowledge base UniProtKB/Swiss-Prot [Con12]). The cDNA/EST/protein sequences are

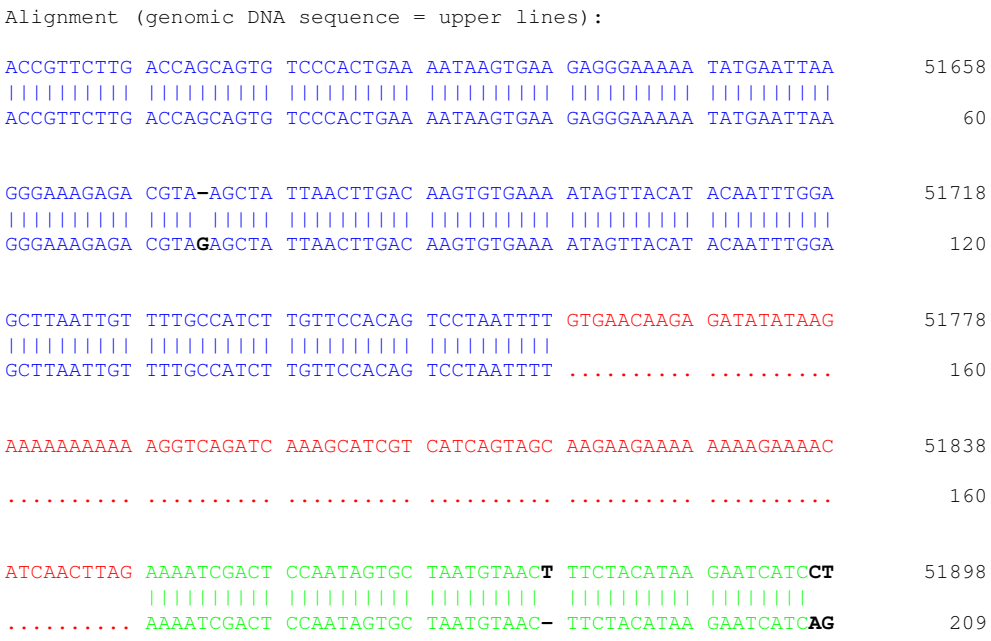


Figure 4.1: Example of a spliced alignment consisting of two exons enclosing one intron. The intron is shown as a sequence of dots in the lower cDNA/EST sequence. A few indels and mismatches are shown in bold.

also called *reference* sequences, see Section 3.2.3 for details. It is unknown how many of the cDNA/EST/protein 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/protein 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 [AKO04], with a subsequent chaining phase combining several consistent matches. The second task involves application of classical dynamic programming [Bel57]. 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 Figure 2.2. In the so-called *splicing* (see Section 2.6.1), the *introns* are cut out and only the enclosing *exons* remain (which comprise the cDNA/EST/protein sequences). 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*. See Section 2.9 for a general introduction into *genes*.

When computing a spliced alignment, one also has to allow for errors (insertions, deletions, and mismatches) because of sequencing errors. Furthermore, this allows to use non-cognate (from a different organism), but homologous (still similar) sequences for gene prediction.

Figure 4.1 shows an example of a spliced alignment consisting of two exons enclosing one intron.

The spliced alignment problem has been extensively considered over the last ten years. Our al-

gorithms are closely related to the *GeneSeqer* spliced alignment algorithm based on cDNA/EST sequences [BXZ04, UZB00] and the one based on protein sequences [UB00]. Other recent programs with similar capabilities are *mGene* [SZZ+09], *Augustus* [SDBH08], *MAKER* [CKR+08], *EuGène* [FS05], *ExonHunter* [BBLV05], *GMAP* [WW05], *Genomewise* [BCD04], *BLAT* [Ken02], *Spidey* [WCO01], *Fgenesh++* [SS00], and *sim4* [FHZ+98]. Some of these programs are explained in more detail in Section 3.5.

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/protein alignments in that region. This *consensus spliced alignment* computation (see Section 4.7) is done by a method described in [HDM+03]. *GenomeThreader* is capable to simultaneously combine spliced alignments based on cDNA/EST sequences with spliced alignments based on protein sequences (located in the same region, of course) into consensus spliced alignments. To our best knowledge, no other published software tool can do this.

4.1.1 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_i is the i-th symbol of s. If $i \leq j$, then $s_i \dots s_j$ is the substring of s beginning with the i-th symbol and ending with the j-th symbol. If $i > j$, then $s_i \dots s_j$ 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'.

The following definition was taken from [Kur12].

Definition 15 A *HMM* (Hidden Markov Model) is a quadruple $M = (\Sigma, S, A, e)$ consisting of

- an alphabet Σ,
- a set of states S,
- a matrix $A = (a_{s,t})_{s,t \in S}$ of transition probabilities $a_{s,t}$ for all $s, t \in S$, and
- an emission probability $e_k(b)$ for every $k \in S$ and $b \in \Sigma$.

4.1.2 The Spliced Alignment Problem for cDNA/EST Sequences

We consider the problem of computing an optimal spliced alignment of a gDNA $g = g_1 \dots g_n$ and a cDNA/EST sequence $c = c_1 \dots c_m$, both over the alphabet $\Sigma = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}, \mathtt{N}\}$, where $\mathtt{N}$ is the undetermined symbol.

```
ACCGTCAAGTT-CG
|  |       ||  ||
AGC......TTACG
```

Figure 4.2: A spliced alignment between a gDNA and an EST sequence where the sequence of intron and exon states is left implicit. The gDNA sequence is shown in the upper line and the EST sequence in the lower line. Each column of the form $\left[\begin{smallmatrix}\alpha\\ \cdot\end{smallmatrix}\right]$ 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.

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. That is, the set of states S is defined as $S = \{ex_1, ex_2, \ldots, ex_n, in_1, in_2, \ldots, in_n\}$. In each exon state an output column $\left[\begin{smallmatrix}\alpha\\ \beta\end{smallmatrix}\right]$ for $\alpha, \beta \in \Sigma \cup \{-\}$ is generated. In each intron state an output column $\left[\begin{smallmatrix}\alpha\\ \cdot\end{smallmatrix}\right]$ for $\alpha \in \Sigma$ is generated. We use the symbol "–" for denoting deletions. That is, $\left[\begin{smallmatrix}-\\ \beta\end{smallmatrix}\right]$ denotes the deletion of symbol β from sequence g, while $\left[\begin{smallmatrix}\alpha\\ -\end{smallmatrix}\right]$ denotes the deletion of symbol α from c. The symbol "." stands for a symbol spliced out of the gDNA. That is, the emissions are somewhat different from a "normal" HMM defined in Section 4.1.1, because we have more complex outputs (the output is not just defined over Σ).

Consider a sequence $Q = q_1, q_2, \ldots, q_k$ of intron and exon states, and let $A = \left[\begin{smallmatrix}\alpha_1\alpha_2\cdots\alpha_k\\ \beta_1\beta_2\cdots\beta_k\end{smallmatrix}\right]$ be the corresponding sequence of column outputs in these states, that is $\left[\begin{smallmatrix}\alpha_i\\ \beta_i\end{smallmatrix}\right]$ 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 occurrences of the symbols "–" and ".". Figure 4.2 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 Figure 4.3 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 collective term for donor and acceptor site is *splice site*. The terms donor and acceptor site are biologically motivated. For the discussion here, it suffices to know that a

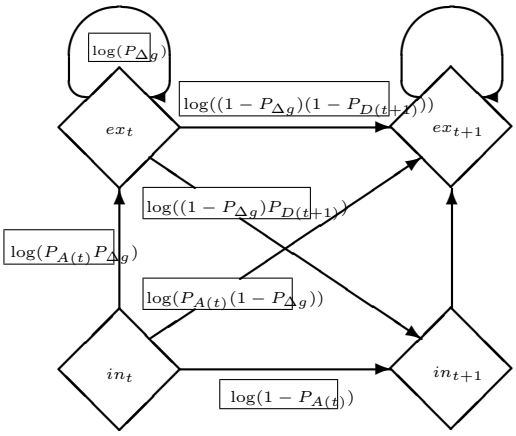


Figure 4.3: States and state transitions of a spliced alignment. Adapted from [UZB00].

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 [BXZ04, SB05] (see Section 4.2). Therefore, $P_{D(t)}$ and $P_{A(t)}$ are called *BSSM parameters*. That is, the function w defines the transition probability matrix A introduced in Section 4.1.1.

Note the systematics in assigning the probabilities:

- A transition from a state for position t to a state for position $t+1$ involving at least one intron-state always means that g_t is bound in some edit operation, that is, it is not deleted. This expressed by using the multiplier $(1-P_{\Delta g})$. Note that the transition from in_t or e_{t+1} corresponds to a base which is part of an intron. This is not considered a deletion and hence the term $P_{\Delta g}$ is not used for defining the probability of this transition. If we would use it, then probability would be proportional to this score, which contradicts the fact that an intron can become very large.

- A transition from in_t or e_t to e_t means that g_t is not bound in the emitted edit operation, that is, g_t is deleted. This expressed by using the multiplier $P_{\Delta g}$.

- The transition from e_t to e_{t+1} means that position $t+1$ is not the start of an intron, and so we use the multiplier $1-P_{D(t+1)}$.

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 $\left[\begin{smallmatrix}\alpha\\ \beta\end{smallmatrix}\right]$ generated in an exon state is assigned a weight $w\left(\left[\begin{smallmatrix}\alpha\\ \beta\end{smallmatrix}\right]\right)$ as follows:

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	-5.0
splice site parameter	$P_{D(t)}$, $P_{A(t)}$	from BSSM

Table 4.1: Parameters determining the weight of a spliced alignment. The BSSM parameters are explained in Section 4.2.

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

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 emission probabilities in the HMM are called *output weights*, because output weights do not necessarily add to one.

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 for cDNA/EST sequences 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 *optimal spliced alignment*.

Table 4.1 gives an overview of the parameters required to determine the weight of a spliced alignment.

4.1.3 The Spliced Alignment Problem for Protein Sequences

We consider the problem of computing an optimal spliced alignment of a gDNA $g = g_1 \dots g_n$ and a protein sequence sequence $p = p_1 \dots p_m$, where g is defined over the alphabet $\Sigma = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}, \mathtt{N}\}$ (N denotes the undetermined symbol) and p is defined over the alphabet $\mathcal{A} = \{L, V, I, F, K, R, E, D, A, G, S, T, N, Q, Y, W, P, H, M, C\}$ (the 20-letter alphabet of naturally

```
GAACACTTGTGTAAGCCTAGGCAAAACG
 E   H   L             C   K   T
 |   |   |             |   |   |
 E   H   L  ..........C   K   T
```

Figure 4.4: A spliced alignment between a gDNA and a protein sequence where the sequence of intron and exon states is left implicit. The gDNA sequence is shown in the upper line and the protein sequence in the lower line. The second row shows the translated codons of the gDNA in their corresponding positions. Each column of the form $\left[\begin{smallmatrix}\alpha\\ \cdot\end{smallmatrix}\right]$ corresponds to an intron state (in_1 states in this example). All other columns correspond to exon states. Matching symbols are denoted in the third row with the symbol "|". The intron is an in_1 intron, because it starts after the first codon base.

occurring amino acids). An alignment of the sequences g and p may include gaps in either sequence, denoted by the gap symbol "–". The symbol "." stands for a symbol spliced out of the gDNA. Such an alignment can be viewed as the output of a Hidden Markov Model (HMM) . That is, the set S of states is defined as $S = \{ex_1, ex_2, \ldots, ex_n, in_{0,1}, in_{0,2}, \ldots, in_{0,n}, in_{1,1}, in_{1,2}, \ldots, in_{1,n}, in_{2,1}, in_{2,2}, \ldots, in_{2,n}\}$. The HMM defines a probability space consisting of all possible "threadings" of protein sequences of length m onto the given genomic sequence of length n [UB00, p. 1077].

Consider a sequence $Q = q_1, q_2, \ldots, q_l$ of intron states (of type in_0, in_1, and in_2, respectively) and exon states ex, where $\max(3m, n) \leq l \leq 3m + n$. The three intron states represent introns in three different coding phases: in_0 introns start after a complete codon, in_1 introns start after codon base one, and in_2 introns start after codon base two. Let

$$A = \left[\begin{smallmatrix}\alpha_{1,1}\alpha_{1,2}\alpha_{1,3}\\ \beta_1\end{smallmatrix}\right]\left[\begin{smallmatrix}\alpha_{2,1}\alpha_{2,2}\alpha_{2,3}\\ \beta_2\end{smallmatrix}\right]\ldots\left[\begin{smallmatrix}\alpha_{l,1}\alpha_{l,2}\alpha_{l,3}\\ \beta_l\end{smallmatrix}\right]$$

be the corresponding sequence of column outputs in these states, that is $\left[\begin{smallmatrix}\alpha_{i,1}\alpha_{i,2}\alpha_{i,3}\\ \beta_i\end{smallmatrix}\right]$ is the output in state q_i. (Q, A) is a spliced alignment of g and p if we obtain g from

$$\alpha_{1,1}\alpha_{1,2}\alpha_{1,3}\alpha_{2,1}\alpha_{2,2}\alpha_{2,3}\ldots\alpha_{l,1}\alpha_{l,2}\alpha_{l,3}$$

and p from $\beta_1\beta_2\ldots\beta_l$ after deleting all occurrences of the symbols "–" and ".". Figure 4.4 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. For the transition weights, the same $P_{D(t)}$ and $P_{A(t)}$ values are used as in the spliced alignment problem for cDNAs/ESTs described in Section 4.1.2 above. The details on how this state transition weights are employed can be seen in the actual recurrences given below in Section 4.4.

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 $\left[\begin{smallmatrix}\alpha_{-,1}\alpha_{-,2}\alpha_{-,3}\\ \vec{\beta}\end{smallmatrix}\right]$ (where $\alpha_{-,j}$ means that the first index of α is not specified) generated in an exon state is assigned a weight $w\left(\left[\begin{smallmatrix}\alpha_{-,1}\alpha_{-,2}\alpha_{-,3}\\ \vec{\beta}\end{smallmatrix}\right]\right)$ by using scaled values

of the BLOSUM62 amino acid substitution matrix [HH92][1]. Thereby, the nucleotide triplet $\alpha_{\_,1}\alpha_{\_,2}\alpha_{\_,3}$ is translated into a amino acid according to the chosen *codon translation table*[2]. All deletions (one to three nucleotide deletions in the gDNA or an amino acid deletion in the protein sequence) are given the same deletion penalty corresponding to twice the lowest mismatch score of the BLOSUM62 matrix [UB00, p. 1078]. In an intron state in_t the column $\left[\begin{smallmatrix} g[t] \\ \cdot \end{smallmatrix}\right]$ 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 for protein sequences is to find a spliced alignment of g and p with maximum weight, denoted by $w(g, p)$. A spliced alignment (Q, A) of g and p satisfying $w(Q, A) = w(g, p)$ is called *optimal spliced alignment*.

4.2 Easy-to-Use Bayesian Splice Site Models (BSSMs)

Bayesian Splice Site Models (BSSMs) were first described in [Sal97] and later adopted for use by *GeneSeqer* in [BXZ04]. BSSMs assign splice site probabilities to genomic DNA bases (for donor sites and acceptor sites, respectively). Briefly, the training of BSSMs consists of tabulating dinucleotide relative frequencies over the 102 positions of interest (a window of length 50 to either side) for the trained splice site type, for seven classes of training data corresponding to seven alternative hypotheses to be evaluated using Bayes rules [SB05]. The training and employment of BSSMs in *GenomeThreader* has been designed in a very *easy-to-use* fashion. More information on the representation of BSSMs and more details on how to train custom (that is, organism specific) BSSMs are given below in Section 4.8.3.

4.3 Computing Optimal Spliced Alignments with ESTs

As with many problems in biological sequence comparison, the spliced alignment problem for cDNA/EST sequences can be solved by a dynamic programming (DP) algorithm. This computes two $(m + 1) \times (n + 1)$-matrices E and I such that the 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 $g_1 \ldots g_j$ and $c_1 \ldots c_i$ such that the state sequence ends with an exon state.
- I_i^j is the maximum weight of any spliced alignment of $g_1 \ldots g_j$ and $c_1 \ldots c_i$ 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

[1] This is the default matrix, it can be changed by using option `-scorematrix` of *GenomeThreader*.

[2] Can be set with the option `-translationtable` of *GenomeThreader*, see manual in Appendix A.

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

Figure 4.5: Adapted from [UZB00]. Hypothetical alignment of a gDNA $g = \texttt{ACCGTCAAGTTCG}$ with an EST sequence $c = \texttt{AGCTTACG}$. The gDNA position j is in the range $[1, 13]$ and the EST sequence position i in the range $[1, 8]$. As one can see from the optimal state sequence Q, positions 4 to 9 of the gDNA have been assigned intron status.

$w(ex_0, in_0) = w(in_0, in_1) = \log(w(in_1)) = \log(1 - w(ex_1))$. Here $w(ex_1)$ denotes the *initial exon state probability*. Now each matrix entry can be computed by the following recurrence:

$$
\begin{aligned}
E_i^j &= \max \left\{ \begin{array}{l} \max\left\{E_i^{j-1} + w(ex_{j-1}, ex_j), I_i^{j-1} + w(in_{j-1}, ex_j)\right\} + w\left(\left[\begin{smallmatrix} g_j \\ - \end{smallmatrix}\right]\right) \\ \max\left\{E_{i-1}^{j-1} + w(ex_{j-1}, ex_j), I_{i-1}^{j-1} + w(in_{j-1}, ex_j)\right\} + w\left(\left[\begin{smallmatrix} g_j \\ c_i \end{smallmatrix}\right]\right) \\ \max\left\{E_{i-1}^{j} + w(ex_j, ex_j), I_{i-1}^{j} + w(in_j, ex_j)\right\} + w\left(\left[\begin{smallmatrix} - \\ c_i \end{smallmatrix}\right]\right) \end{array} \right\} \\
I_i^j &= \max\left\{E_i^{j-1} + w(ex_{j-1}, in_j), I_i^{j-1} + w(in_{j-1}, in_j)\right\}
\end{aligned}
$$

for $i > 0$ and $j > 0$. Additionally, $E_i^j = 0$, for $j = 0$ or $i = 0$, $I_i^j = 0$ for $i = 0$, and $I_i^j = -\infty$ for $j = 0$. The first column of the I matrix (case $j = 0$) is set to $-\infty$, because cDNAs/ESTs do not, theoretically speaking, 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.

Figure 4.5 shows a (hypothetical) optimal spliced alignment including output column weights. The same alignment is shown in Figure 4.6 as a path in the superimposed matrices E and I.

4.4 Computing Optimal Spliced Alignments with Proteins

Similar to the spliced alignment problem for cDNA/EST sequences, the spliced alignment problem for protein sequences can be solved by a dynamic programming (DP) algorithm. In this case,

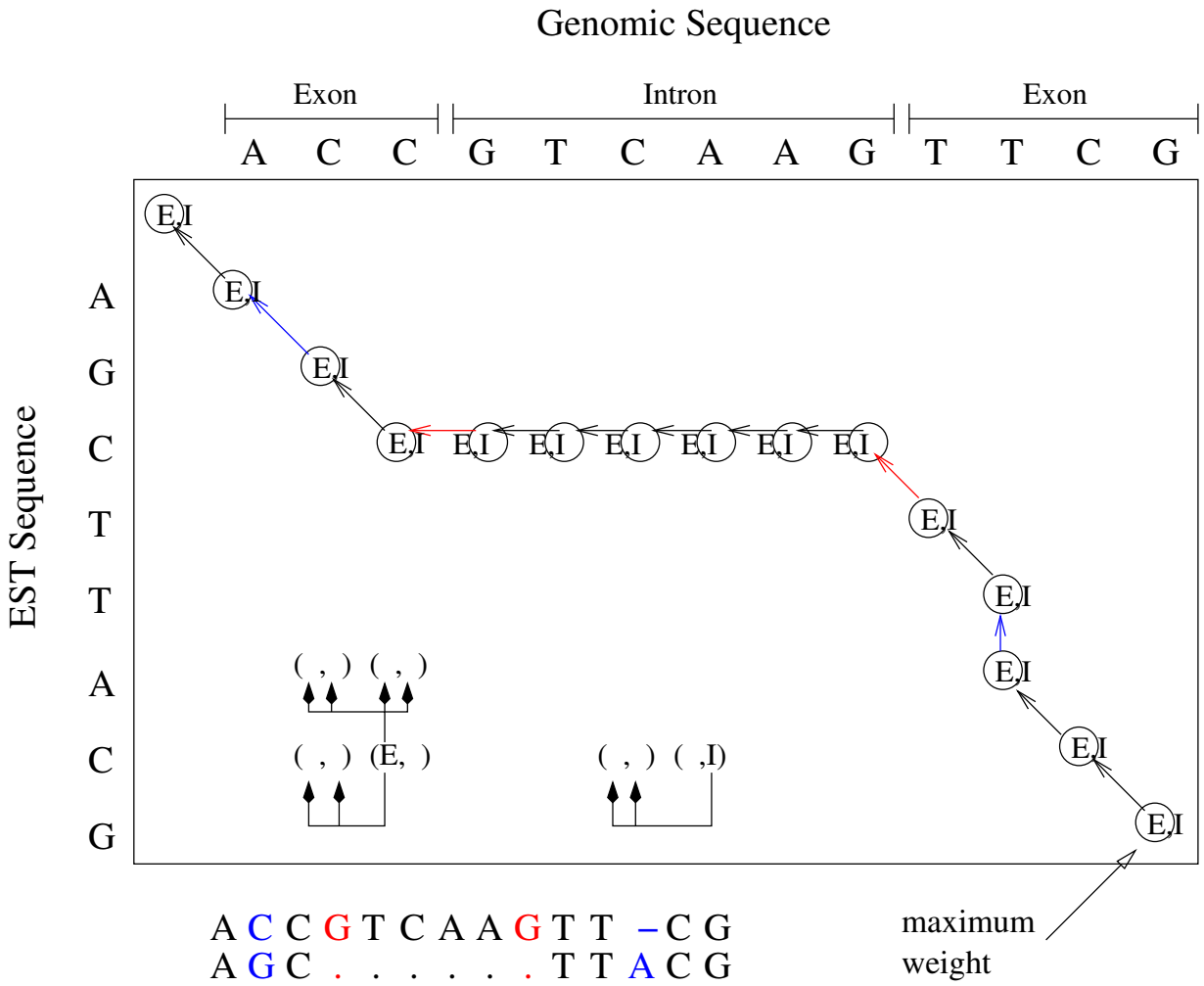


Figure 4.6: An optimal spliced alignment of the sequences $g =$ ACCGTCAAGTTCG and $c =$ AGCTTACG represented by a path in the superimposed matrices E and I. The insert in the lower left part of the matrix shows dependencies of the E and I entries. The optimal path through the matrix is depicted (starting at the position marked with "maximum weight"). Thereby, transitions $E \to I$ and $I \to E$ are marked red and transitions representing indels or mismatches are marked blue. The computed gene structure of the artificial genomic sequence is denoted above the matrix and the resulting spliced alignment with an artificial EST is shown below.

it computes four $(m+1) \times (n+1)$-matrices E, I_0, I_1 and I_2 such that the 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 $g_1 \dots g_j$ and $p_1 \dots p_i$ such that the state sequence ends with an exon state ex.
- $(I_f)_i^j$ is the maximum weight of any spliced alignment of $g_1 \dots g_j$ and $p_1 \dots p_i$ such that the state sequence ends with an intron state in_f, for $f \in \{0, 1, 2\}$.

Obviously, $w(g, p) = \max(E_m^n, (I_0)_m^n, (I_1)_m^n, (I_2)_m^n)$. Now each matrix entry can be computed by the following recurrence:

$$
\begin{aligned}
E_i^j &= \max \left\{ \begin{array}{l}
E_{i-1}^{j-3} + \log(1 - P_{D(j-2)}) + w\left(\left[\begin{smallmatrix} g_{j-2} g_{j-1} g_j \\ p_i \end{smallmatrix}\right]\right), \\
E_{i-1}^{j-2} + \log(1 - P_{D(j-1)}) + w\left(\left[\begin{smallmatrix} g_{j-1} g_j - \\ p_i \end{smallmatrix}\right]\right), \\
E_{i-1}^{j-1} + \log(1 - P_{D(j)}) + w\left(\left[\begin{smallmatrix} g_j - - \\ p_i \end{smallmatrix}\right]\right), \\
E_{i-1}^{j} + \log(P_{\Delta g_j}) + w\left(\left[\begin{smallmatrix} - - - \\ p_i \end{smallmatrix}\right]\right), \\
E_{i}^{j-3} + \log(1 - P_{D(j-2)}) + w\left(\left[\begin{smallmatrix} g_{j-2} g_{j-1} g_j \\ - \end{smallmatrix}\right]\right), \\
E_{i}^{j-2} + \log(1 - P_{D(j-1)}) + w\left(\left[\begin{smallmatrix} g_{j-1} g_j - \\ - \end{smallmatrix}\right]\right), \\
E_{i}^{j-1} + \log(1 - P_{D(j)}) + w\left(\left[\begin{smallmatrix} g_j - - \\ - \end{smallmatrix}\right]\right), \\
(I_0)_{i-1}^{j-3} + \log(P_{A(j-3)}) + w\left(\left[\begin{smallmatrix} g_{j-2} g_{j-1} g_j \\ p_i \end{smallmatrix}\right]\right), \\
(I_1)_{i-1}^{j-2} + \log(P_{A(j-2)}) + w\left(\left[\begin{smallmatrix} g_x g_{j-1} g_j \\ p_i \end{smallmatrix}\right]\right), \\
(I_2)_{i-1}^{j-1} + \log(P_{A(j-1)}) + w\left(\left[\begin{smallmatrix} g_x g_{x+1} g_j \\ p_i \end{smallmatrix}\right]\right)
\end{array} \right\} \\
(I_0)_i^j &= \max\left\{(I_0)_i^{j-1} + \log(1 - P_{A(j-1)}), E_i^{j-1} + \log(P_{D(j)})\right\} \\
(I_1)_i^j &= \max\left\{(I_1)_i^{j-1} + \log(1 - P_{A(j-1)}), E_i^{j-2} + \log(P_{D(j)})\right\} \\
(I_2)_i^j &= \max\left\{(I_2)_i^{j-1} + \log(1 - P_{A(j-1)}), E_i^{j-3} + \log(P_{D(j)})\right\}
\end{aligned}
$$

for $i > 0$ and $j > 2$. Additionally, $E_i^j = (I_0)_i^j = (I_1)_i^j = (I_2)_i^j = 0$ for $i = 0$ and $j \geq 0$, $E_i^j = 0$ for $i > 0$ and $j \leq 2$, and $(I_0)_i^j = (I_1)_i^j = (I_2)_i^j = -\infty$ for $i > 0$ and $j \leq 2$. The probability of a genomic insertion $P_{\Delta g_j}$ is defined as follows:

$$
P_{\Delta g_j} = \left\{ \begin{array}{ll} 1 - P_{D(j+1)} & \text{if } j < n \\ 1 - P_{D(j)} & \text{otherwise} \end{array} \right.
$$

g_x and g_{x+1} denote upstream nucleotides of split codons that are recored at the appropriate intron opening transitions [UB00, p. 1078].

The first three columns of the I_0, I_1 and I_2 matrices (case $i > 0$ and $j \leq 2$) are set to $-\infty$, because proteins, theoretically speaking, do not contain introns. This is also the reason why a value in the matrices I_0, I_1 and I_2 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

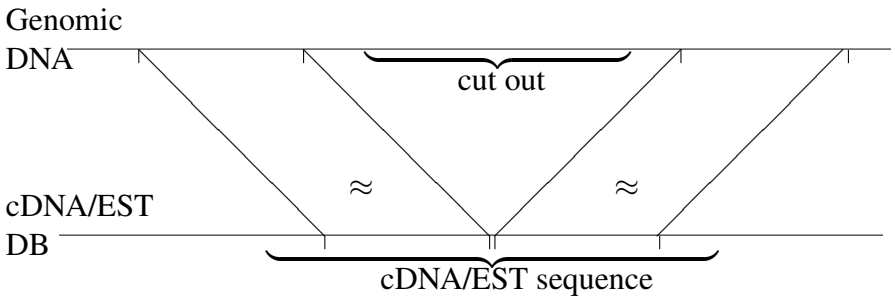


Figure 4.7: A graphical explanation of the intron cutout idea.

row. Each entry can be computed in constant time. Hence all four matrices can be computed in $O(mn)$ time, which also gives the time bound for determining $w(g, p)$. An optimal spliced alignment is recovered by tracing back from the entry $\max(E_m^n, (I_0)_m^n, (I_1)_m^n, (I_2)_m^n)$ 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 p is computed in time proportional to its length.

4.5 The Intron Cutout Technique

When predicting the gene structure for genomic sequences of vertebrates (for example, human or mouse) or plants one is faced with the problem of long introns. Some known introns consist of several 10,000 or even 100,000 bases (see for example, [Lew04, AJL$^+$02]), and thus the dynamic programming algorithm described in the previous section 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/protein sequences derived from this genomic locus, the introns should be devoid of any but chance matches to the EST/protein sequences. Thus, the idea is to apply a *similarity filter*: this first finds approximate matches between the gDNAs and the ESTs/proteins. Several of these matches are combined into a chain if they are compatible with each other, that is, 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 Figure 4.7 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 to recognize the exact exon/intron boundaries. Most important, the cutout technique considerably 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/proteins, and then to chain these. The chain gives regions which possibly contain exons, delineating an alignment. The regions between these exons can be cut out.

4.5.1 Computing cDNA/EST 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 $g_j \dots g_r$ and $c_i \dots c_h$ 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 $g_j \dots g_r$ and $c_i \dots c_h$. 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 $\left\lfloor \frac{\ell_{\min}}{d_{\max}+1} \right\rfloor$ 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 [ZSWM00] for a description of the technical details. This seed-and-extend approach is implemented in the program *Vmatch* (`http://vmatch.de/`), and we utilize *Vmatch* for computing maximal approximate matches between g and c.

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 [AKO04]. 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.

4.5.2 Chaining the cDNA/EST 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 $g_j \dots g_r$ and $c_i \dots c_h$ are denoted by $([j..r], [i..h])$. For any pair of matches $f =$

$([j..r],[i..h])$ and $f' = ([j'..r'],[i'..h'])$ 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 match* and f_q is called *end match*. To obtain the score for a chain, we score matches: Each maximal approximate match $f = ([j..r],[i..h])$ is assigned a positive score. This is defined on the basis of an optimal alignment of $c_i \ldots c_h$ and $g_j \ldots g_r$ 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 *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 Figure 4.8(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:

$$scoremax(f') = score(f') + \max\{scoremax(f) - ovl(f, f') \mid f \ll f'\} \tag{4.1}$$

$\max\{scoremax(f') \mid f' \in M\}$ gives the maximum score of any chain, and reconstructing a chain of maximum score is an easy task. A dynamic programming algorithm based on recurrence (4.1) 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 [SK03]. 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 match $f \in M$ we keep track of the start match of an optimal chain ending with f. We divide all matches into equivalence classes according to their corresponding start matches. That is, two matches belong to the same equivalence class, if and only if their corresponding optimal chain share the same start match. For every equivalence class which

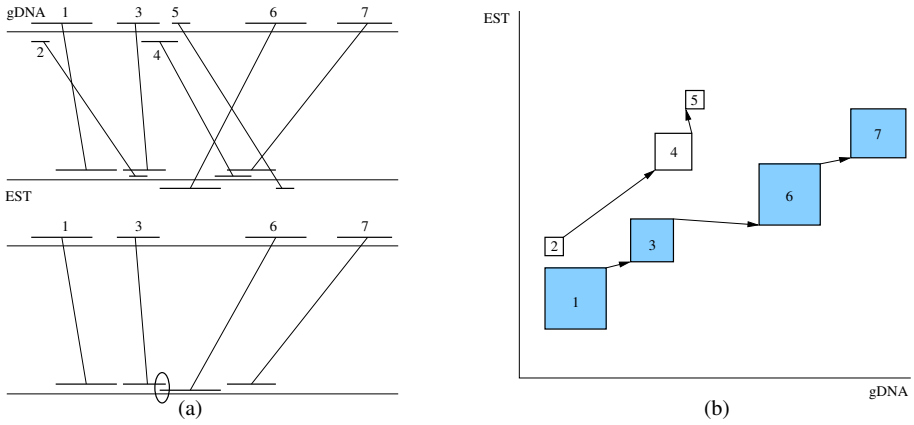


Figure 4.8: Given a set of matches (upper left figure), an optimal global chain of collinear (possibly) overlapping matches (lower left figure) can be computed, for example, by computing an optimal path in the graph in (b) (in which not all edges are shown). The overlapping part of match 3 and match 6 is circled.

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 to report multiple chains which differ only slightly. This modification allows to find chains matching at different loci in the genome (resulting from paralogous genes).

4.5.3 Computing and Chaining Protein Matches

The computation of *maximal approximate matches* between a gDNA g and a protein sequence p, which is necessary to compute optimal spliced alignments with protein sequences, works similar to computing maximal approximate matches between a gDNA g and an EST sequence c (described above in Section 4.5.1).

In this case, a six frame translation of g (three reading frames on the forward strand and three on the reverse strand) is matched against an index of p and the Hamming distance (only mismatches and no insertions or deletions are allowed) is used instead of the edit distance. This approach is also implemented in *Vmatch* (`http://vmatch.de`) and we utilize *Vmatch* for computing the maximal approximate matches between g and p.

The matches between the six frame translation of g and p are mapped back onto the original genomic sequence g. Afterwards, the chaining for maximal approximate matches between g and p is done in exactly the same way as the chaining for maximal approximate matches between g and c (described above in Section 4.5.2).

4.5.4 Chain Enrichment

For each stored global chain for a given cDNA/EST (or protein) and gDNA pair, the regions of the gDNA covered by the chain are considered in the dynamic programming step. Without the intron cutout technique, the complete gDNA between the leftmost and the rightmost base covered by the chain is considered in the DP step (after it has been extended to the right and to the left by 300 bases). On the other hand, if the cutout technique is used, only the regions of the gDNA covered by matches *contained in the chain* are considered during the dynamic programming (after they have been extended, see Section 4.5.5 for details). That is, in the (rather rare) case that a correct exon has been covered by a match which is *not* included in the chain, the corresponding gDNA segment will be removed in the cutout step and a wrong spliced alignment will result.

To prevent this, the *chain enrichment* was introduced. If the chain enrichment is enabled[3], all matches whose genomic region overlaps with the gDNA region covered by the optimal global chain are added to the chain. Formally: We consider a set M of matches and an optimal global chain $C = f_1, f_2, \ldots, f_q$ such that $f_a \in M$ for $a \in [1, q]$ and $f_a \ll f_{a+1}$ for $a \in [1, q-1]$. The $\ll$ relation has been defined in Section 4.5.2 above. Given the start match $f_1 = ([j..r], [i..h])$ and end match $f_q = ([j'..r'], [i'..h'])$ we define the *genomic range* of C by

$$gr(C) := [j..r']$$

A match $m = ([j..r], [i..h])$ and a genomic range $gr(C) = [x..y]$ *overlap*, if and only if $j \leq y$ and $r \geq x$. We define the *enrichment set* of M by

$$M' := \{m \in M \mid m \text{ and } gr(C) \text{ overlap}\}$$

Then the *enriched chain* C' is a sequence of matches $f'_1, f'_2, \ldots, f'_q$ such that $f_a \in M'$ for $a \in [1, q]$, $q = |M'|$ and for every pair of matches $f_a = ([j..r], [i..h])$ and $f_{a+1} = ([j'..r'], [i'..h'])$ for $a \in [1, q-1]$ one of the following two conditions holds:

- $j < j'$
- $j = j'$ and $r \leq r'$

Figure 4.9 gives an example for the chain enrichment.

As is shown in Section 6.3.3 the chain enrichment clearly improves prediction results on datasets containing long introns. Of course, using chain enrichment increases the total running time, because the additional matches lead to less gDNA that is removed in the cutout step. But the time measures also given in Section 6.3.3 show that this runtime increase is within reasonable bounds.

[3]It will be switched on, if the option `-enrichchains` is passed to *GenomeThreader*.

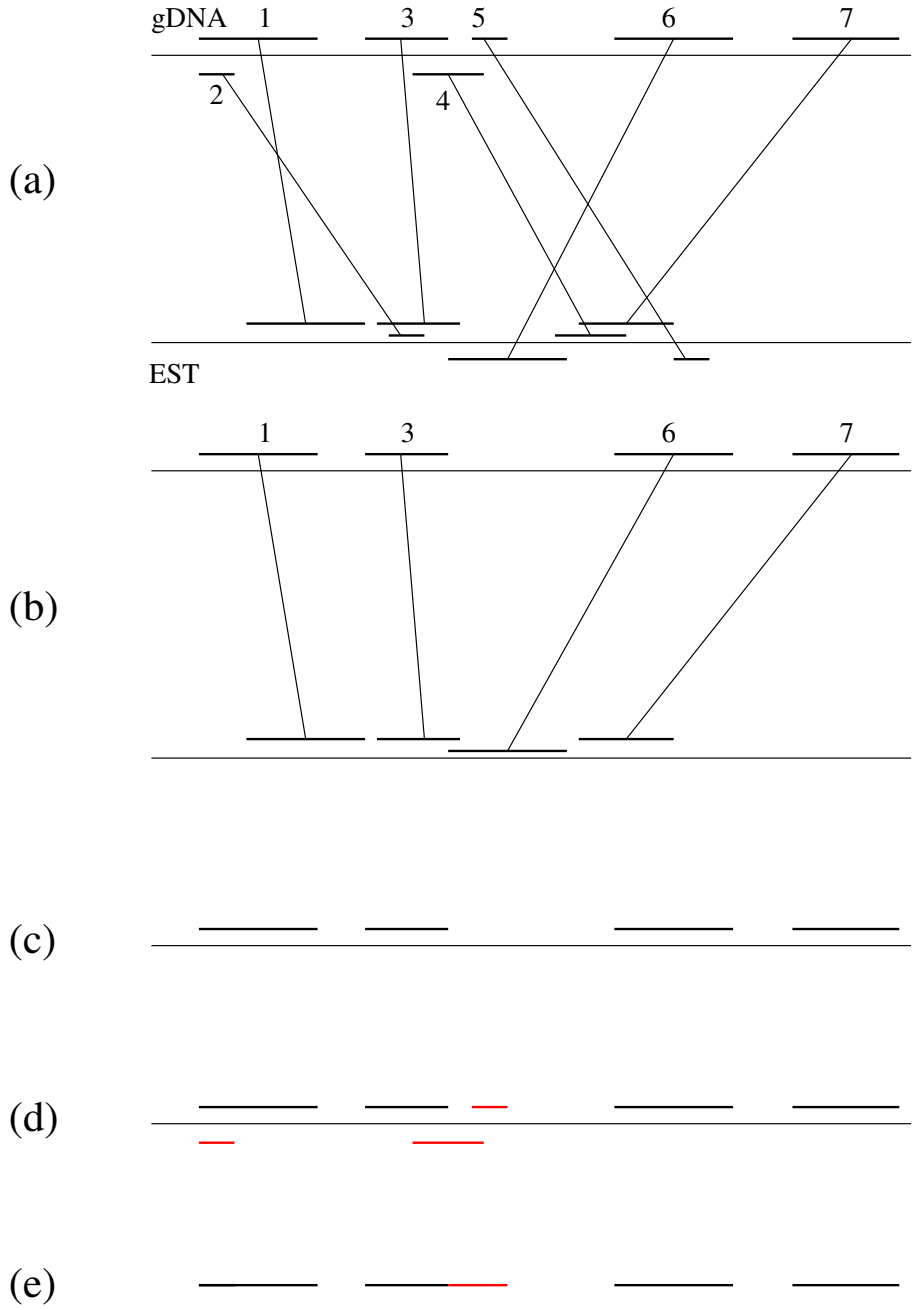


Figure 4.9: Example chain enrichment for the matches given in Figure 4.8, shown in part (a). Part (b) shows the matches comprising the optimal global chain (on both sequences) and part (c) only the parts which lay on the gDNA. The additional matches shown in part (d) (depicted in red) are the ones which overlap with the gDNA region covered by the optimal global chain. These matches are added in the chain enrichment and the resulting covered gDNA region is shown in (e).

4.5.5 The Cutout Step

For each stored global chains for a given EST/protein and gDNA pair, 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 4.3 (for ESTs) or Section 4.4 (for proteins) 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.

4.6 Jump Tables

In the dynamic programming, the complete DP matrix is computed. That means that the information contained in the chained matches from the chaining and chain enrichment is not used, although it is very likely that the optimal path would run through them.

The matches between a cDNA/EST/protein sequence and a gDNA can be divided into three classes:

1. *Straight* matches between both sequences which do not overlap with other matches (on either sequence). Straight matches have the same length on the cDNA/EST/protein sequence as on the gDNA. That is, they contain only matches and mismatches.

2. *Uneven* matches between both sequences which do not overlap with other matches (on either sequence). The length of unbalanced matches on the cDNA/EST/protein sequence differs from their length on the gDNA. That is, they contain insertions and/or deletions.

3. *Overlapping* Matches: Matches which do overlap with other matches.

Figure 4.10 shows examples for the three match classes. This three match classes can be used to reduce the DP computation by assuming the following:

1. The optimal path lies on balanced matches.

2. The optimal path goes through the rectangle outlined by unbalanced matches.

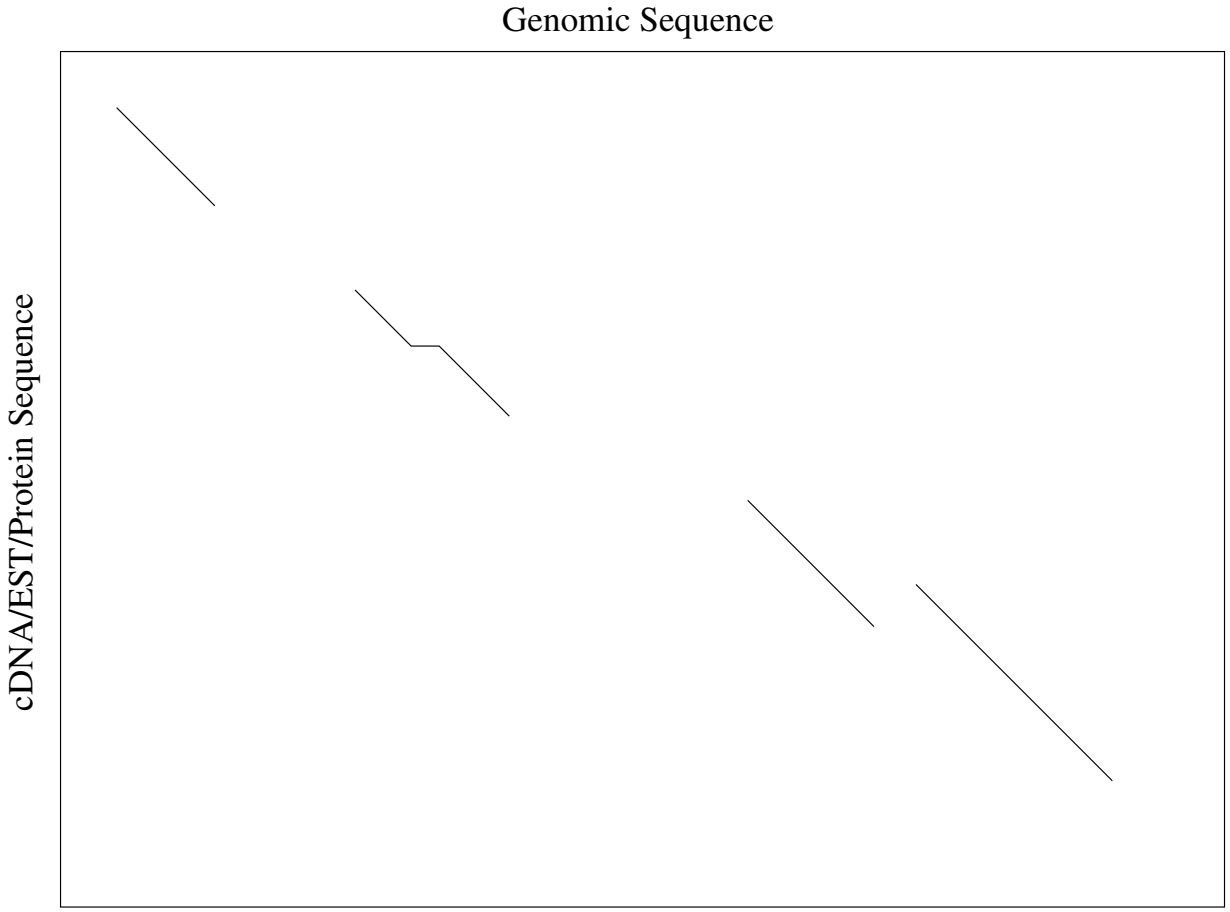


Figure 4.10: Examples for match classes. The single match in the upper left corner is a *balanced* match. The single match in the middle is an *unbalanced* match. The two matches in the lower right corner are *overlapping* matches and are part of the same cluster.

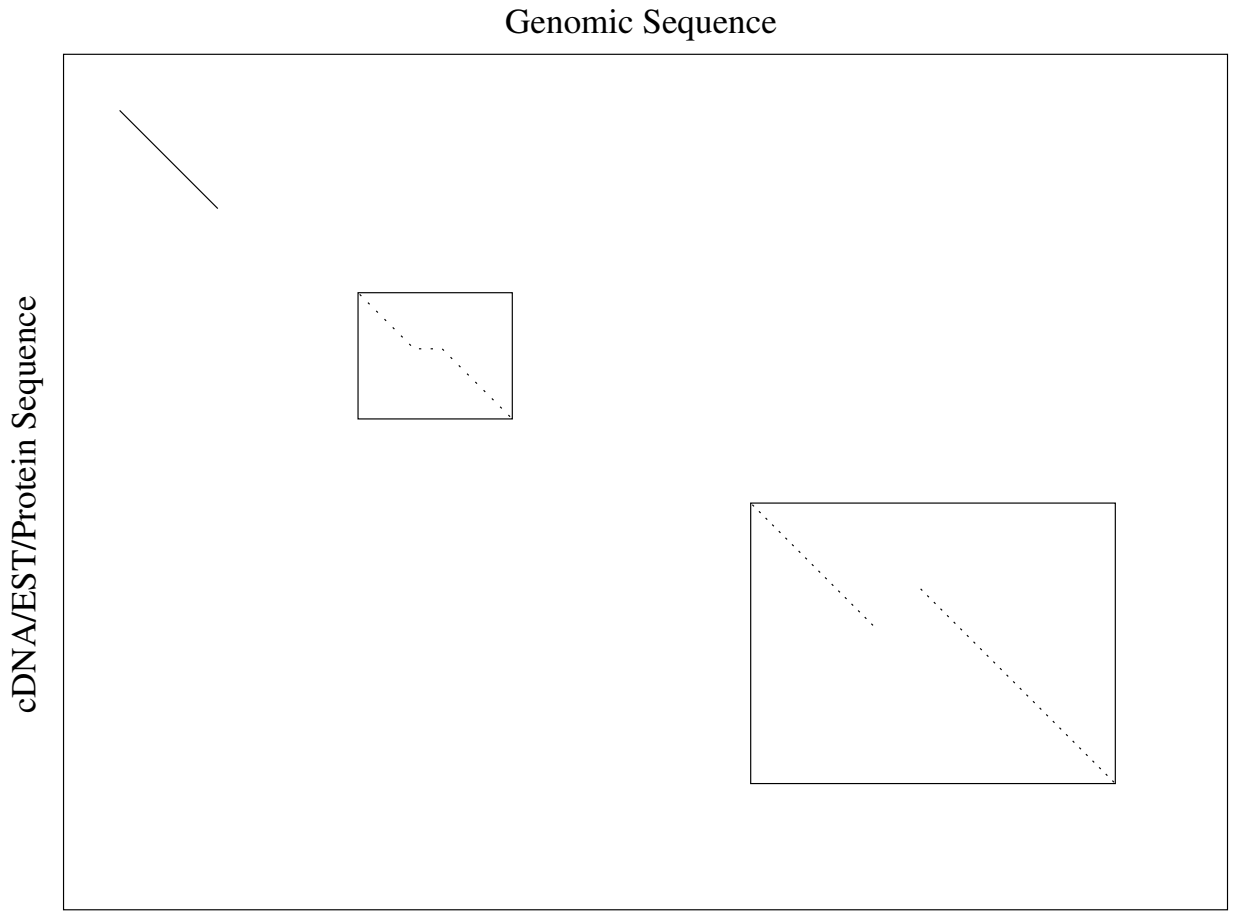


Figure 4.11: Example for Reducing the DP Computation with Match Classes. We assume that the optimal path lies on the balanced match (in the upper left corner), on the two rectangles outlined by the unbalanced match (in the middle), and on the cluster of overlapping matches (in the lower right corner).

3. The optimal path goes through the rectangle outlined by clusters of overlapping matches.

Figure 4.11 shows an example for this. With this assumption we can reduce the parts of the complete DP matrix that have to be computed by connecting the areas outlined by the matches with overlapping rectangles. The rectangles between the match areas should overlap with them to make sure that the borders are aligned correctly (an overlap of 5 base pairs was chosen in the implementation). Figure 4.12 shows an example.

With the information which parts of the matrix have to be computed, there are basically two approaches to do that. The first one is to compute each rectangle as a separate DP matrix (with the appropriate transfer of values between consecutive overlapping matrices). The second one is to allocate just one large DP matrix and compute just the parts of it which are covered by rectangles (as shown in the example in Figure 4.12). The latter approach was chosen in *Genome-*

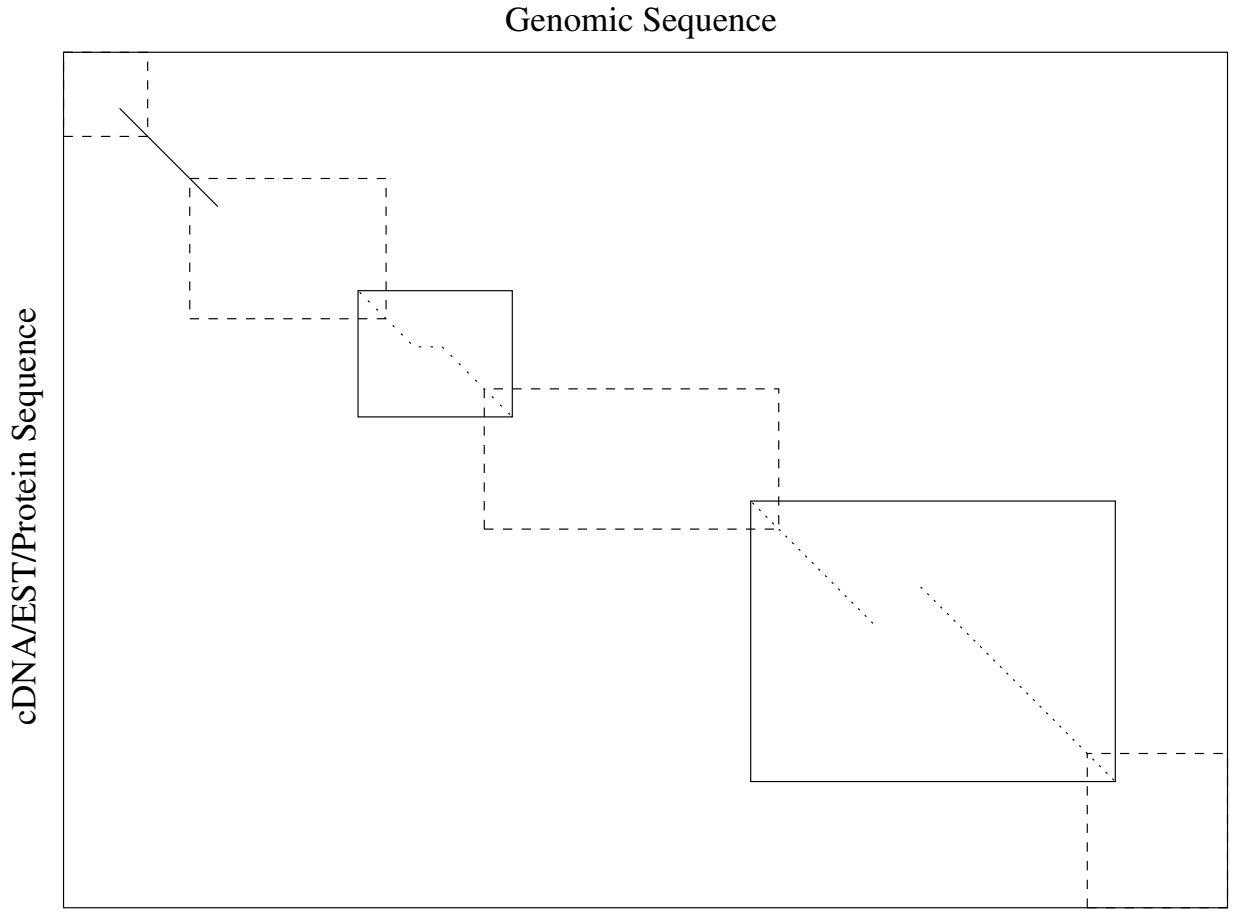


Figure 4.12: Example for reducing the complete DP computation. The complete DP computation can be reduced by connecting the areas defined by the match classes (see Figure 4.11) with overlapping rectangles (drawn with dashed lines). That is, only the outlined parts of the DP matrix have to be computed.

Threader and implemented with a so-called *jump table*[4] and a sparse DP matrix. See Section 4.8.4 for the implementation details concerning jump tables.

In conclusion, with the jump table array and the techniques described above basically the same DP computation routine can be used to speed up the DP matrix calculation considerably, because only the parts of the DP matrix covered by matches and the connecting rectangles have to be computed. Of course, it is still necessary to allocate the memory for the complete matrix (in contrast to the first approach mentioned above), but the implementation is much simpler and the extra memory does not pose a problem in practice.

Since the intron cutout and the jump table technique are orthogonal, both techniques can be applied simultaneously which leads to a combination of their speedups.

Section 6.1.5 gives an example how the intron cutout and the jump table technique influence the run times and prediction accuracies on a real dataset.

4.7 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, that is, 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 spliced alignments have been delivered. In this step, it is not important, if the spliced alignments are based on cDNA/EST sequence, or protein sequences, or both! Fig. 4.13 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 [HDM+03]. 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 $g = g_1 \dots g_n$, 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 $g_j \dots g_r$ 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 $j \leq r'$ and $j' \leq r$. 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 connected, that is, there is at least

[4]This can be activated in the spliced alignment DP of *GenomeThreader* with the option `-fastdp`.

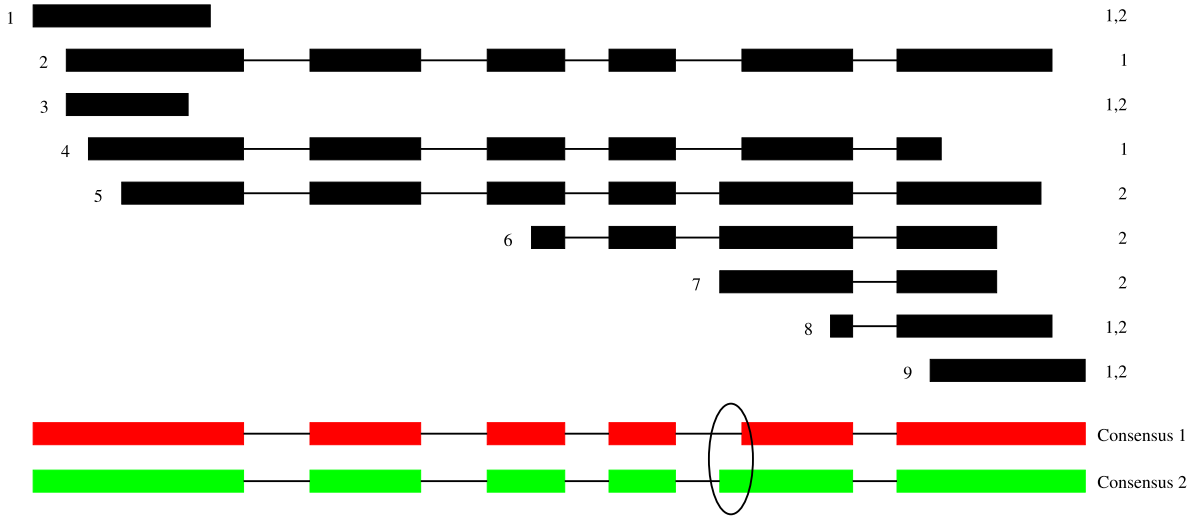


Figure 4.13: An example of consensus spliced alignments, adapted from [HDM+03]. The 9 spliced alignments shown in the upper part of the figure have been processed into two consensus spliced alignments named Consensus 1 and Consensus 2. The rightmost column in the figure shows which consensus the spliced alignments are part of. Spliced alignments 1, 3, 8 and 9 are part of both consensi. The spliced alignments 2 and 4 give rise to consensus 1 while 5, 6 and 7 give rise to consensus 2. The circled shortened exon suggests that this gene is alternatively spliced.

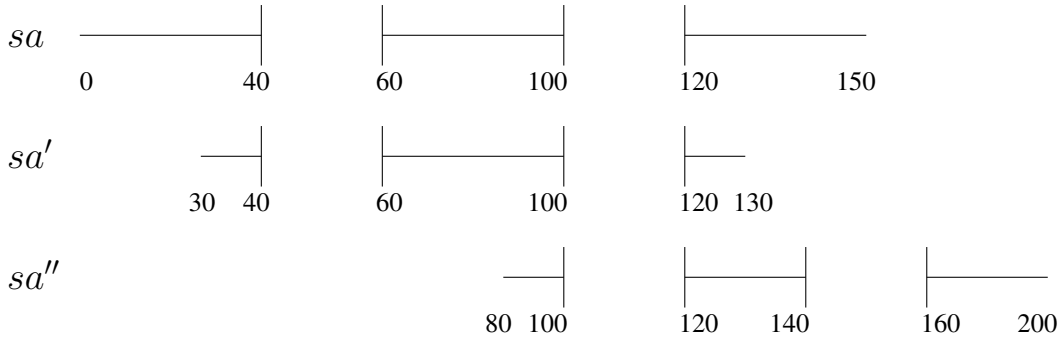


Figure 4.14: Adapted from [HDM+03]. Three spliced alignments. sa and sa' are compatible, as well as sa' and sa''. sa and sa'' are *not* compatible since the second intron of sa'' overlaps with the last exon of sa. Thus the compatibility relation is not transitive.

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 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 w.r.t. exon and intron assignments. Note that compatibility is not transitive, see Figure 4.14 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 w.r.t. to compatibility, i.e., for each $sa' \in SA \setminus SA_p$ a $sa \in SA_p$ exists, such that sa and sa' are not compatible.

The last condition means that additional values in SA_p would violate the compatibility. 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 $j \leq j' \leq r' \leq r$. Note that each spliced alignment contains itself.

The spliced alignment problem is solved by iteratively constructing the consensus spliced alignments, with the largest one being 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),

	1	2	3	4	5
L	$\{1,3\}$	$\{1,2,3,4,8\}$	$\{3\}$	$\{1,3,4\}$	$\{1,3,5,6,7\}$
R	$\{1,3,5,6,7,8,9\}$	$\{2,3,4,8,9\}$	$\{3,5,6,7,8,9\}$	$\{4,8,9\}$	$\{5,6,7,8,9\}$

	6	7	8	9
L	$\{1,3,6,7\}$	$\{1,3,7\}$	$\{1,3,5,6,7,8\}$	$\{1,3,5,6,7,8,9\}$
R	$\{6,7,8,9\}$	$\{7,8,9\}$	$\{8,9\}$	$\{9\}$

Table 4.2: The L-sets and the R-sets for the spliced alignments in Figure 4.13. The numbering is consistent with the numbering of the spliced alignments in Figure 4.13.

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

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

- $r' \geq r$ for each $(j',r') \in R(j,r) \setminus \{(j,r)\}$.

Table 4.2 gives an example for L-sets and R-sets.

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

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

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

$$\begin{aligned} C(sa) &= \{sa' \in SA \mid sa \text{ contains } sa'\} \\ \mathit{left}(j,r) &= \{(j',r') \in SA \mid j' < j, r' < r, (j',r') \text{ and } (j,r) \text{ are compatible}\} \\ \mathit{right}(j,r) &= \{(j',r') \in SA \mid j' > j, r' > r, (j',r') \text{ and } (j,r) \text{ are compatible}\} \end{aligned}$$

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

$$L(sa) = \begin{cases} C(sa) & \text{if } \mathit{left}(sa) = \emptyset \\ L(sa') \cup C(sa) & \text{if } \mathit{left}(sa) \neq \emptyset \end{cases}$$
$$\text{where } sa' \in \mathit{left}(sa) \text{ satisfies}$$
$$|L(sa') \cup C(sa)| \geq |L(sa'') \cup C(sa)| \text{ for all } sa'' \in \mathit{left}(sa)$$

$$R(sa) = \begin{cases} C(sa) & \text{if } \mathit{right}(sa) = \emptyset \\ R(sa') \cup C(sa) & \text{if } \mathit{right}(sa) \neq \emptyset \end{cases}$$
$$\text{where } sa' \in \mathit{right}(sa) \text{ satisfies}$$
$$|R(sa') \cup C(sa)| \geq |R(sa'') \cup C(sa)| \text{ for all } sa'' \in \mathit{right}(sa)$$

These recurrences can easily be implemented in a dynamic programming scheme tabulating $|L(sa)|$ and $|R(sa)|$ for each spliced alignment $sa \in SA$.

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 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 $L(sa)$ and $R(sa)$ for l spliced alignments. For each spliced alignment we have to iterate over all $O(l)$ elements in $\mathit{left}(sa)$ and $\mathit{right}(sa)$ and join it with $C(sa)$. Joining also requires $O(l)$ time. Hence the total running time is $O(l^3 + l^2\eta)$.

4.8 *GenomeThreader* 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 in Appendix A. *GenomeThreader* has a modular structure. Each module implements a certain phase of the data flow, as depicted in Figure 4.15. The interfaces between the different modules are kept small. They exchange information via a small number of different data types, which are described in this section. Some of these data types are also used in other software tools, and are therefore more general than required for *GenomeThreader*.

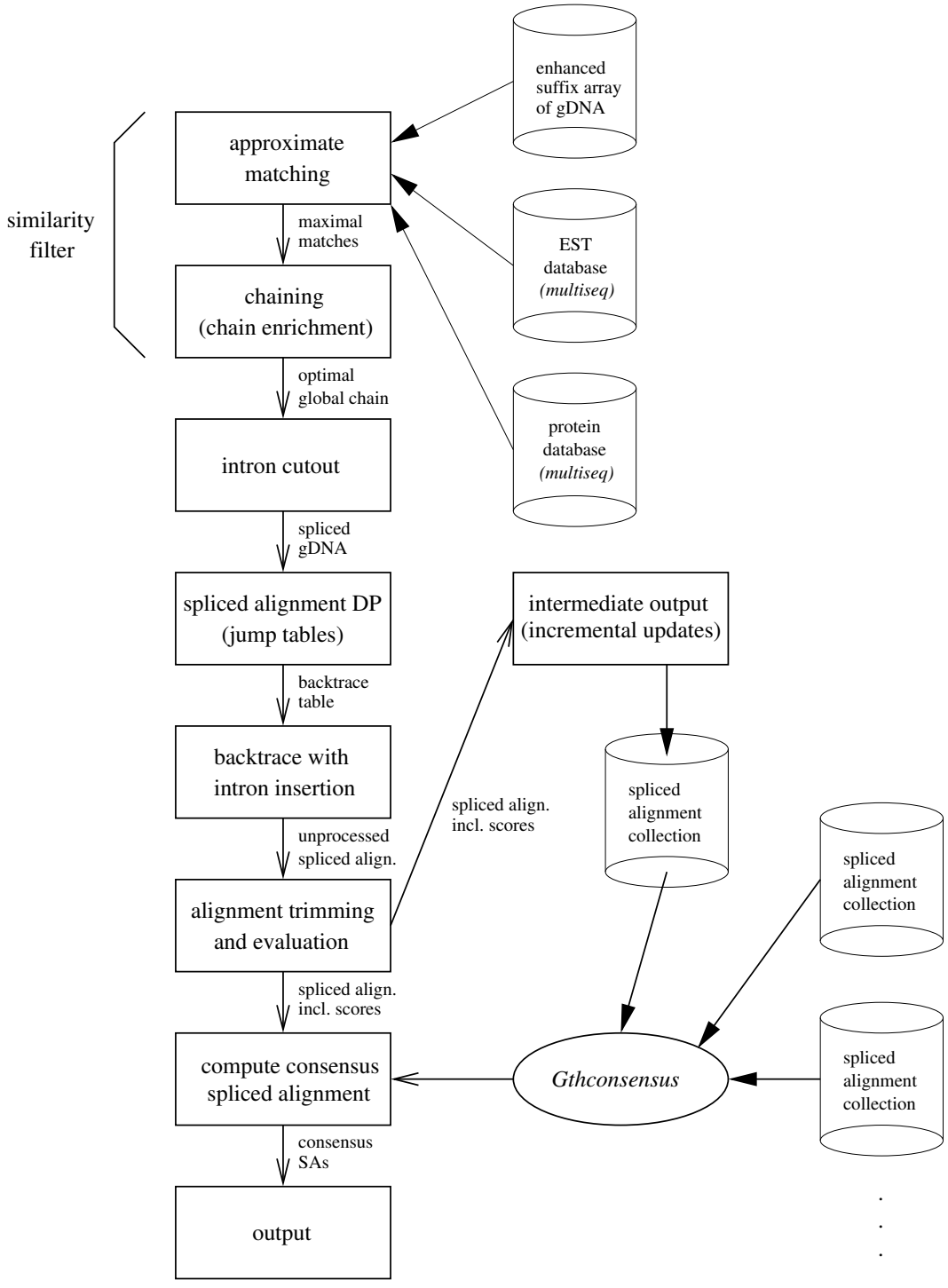


Figure 4.15: Overview of the *GenomeThreader*-phases.

4.8.1 Fast Matching for Filtering of Exon Candidates

The fast matching for the filtering of exon candidates is implemented with *Vmatch* (`http://vmatch.de`) and this is the reason why the *Vmatch* code is necessary for *GenomeThreader*, although it could, in principle, be replaced by other matchers like *BLAST* [AGM+90].

Multiple Sequences

A data type for handling sequences is central to every software for sequence analysis. Because *GenomeThreader* handles many sequences at the same time, we use a data type *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 data type *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 data type *multiseq* also stores the description of 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 to access each sequence description in constant time, given a sequence number.

As a possible future development, this functionality could be implemented with the more efficient `GtEncseq` data structure [SK12].

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 enhanced suffix arrays are necessary to perform the fast matching in *Vmatch*.

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, the counting sort algorithm [CLR90] is used 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 [BS97] 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.

4.8.2 Chaining

We collect all approximate matches between the EST/protein 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 equation (4.1) for each match f', and keep a reference to the match f which maximizes equation (4.1). We call f the previous match. Furthermore, for each match we keep a reference to the start match of its chain and the coverage of the chain up to this match. This allows us to easily divide all end matches (this are the only one we have to consider) into equivalence classes. 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.

4.8.3 Representation of BSSMs

A BSSM can be represented by cascading tables which require a text format that is not trivial to parse. For this reason BSSMs are represented as Lua tables (see `http://www.lua.org/`) which form a textual format that is easy to read and write by users and other programs.

In the implementation of *GenomeThreader* and related tools, the BSSMs are represented by objects of the BSSM class. These objects are created from the textual BSSM files in a simple fashion: The BSSM file is interpreted by a Lua interpreter. Thereby, the interpreter takes care of all the parsing and error checking which simplifies the implementation. Afterwards, the BSSM table is contained in the state of the interpreter from where it can be retrieved easily by the C Code via Lua's C API.

There are various BSSM related tools distributed alongside *GenomeThreader*, the most important ones are *GthBSSMtrain* and *GthBSSMbuild*. *GthBSSMtrain* creates BSSM training data from an annotation given in a GFF3 file (which can, for example, result from an annotation established by *GenomeThreader*). *GthBSSMbuild* can then be used to build a BSSM file from the resulting training data. Details of this process are described in the *GenomeThreader* manual given in Appendix A (in Sections A.9 and A.10). The GFF3 format is described in detail in Section 5.2.

In summary, *GthBSSMtrain* and *GthBSSMbuild* make it very easy for the user to create custom BSSMs which have been shown to improve prediction results (see, for example, Section 6.1.4).

4.8.4 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 4.2.

cDNA/EST Sequences

To solve the spliced alignment problem for cDNA/EST sequences (see Section 4.3) 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 (see page 60) 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 $\lceil \log_2(6) \rceil + \lceil \log_2(2) \rceil = 4$ 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 the following subsection, we describe how to efficiently represent the spliced alignment.

Protein Sequences

To solve the spliced alignment problem for protein sequences (see Section 4.4) we use four arrays of length $m + 1$ of quadruples of floating point numbers for storing a column of matrix E, matrix I_0, matrix I_1, and matrix I_2. For each entry E_i^j, $(I_0)_i^j$, $(I_1)_i^j$, and $(I_2)_i^j$, we have to store the cases of the corresponding recurrence (see page 62) that gave rise to the maximum value. There are 9 different cases for E_i^j and two different cases to store for $(I_0)_i^j$, $(I_1)_i^j$, and $(I_2)_i^j$, respectively. Because we only have to store one case at a time, we need $\lceil \log_2(9) \rceil + \lceil \log_2(2) \rceil + \lceil \log_2(2) \rceil + \lceil \log_2(2) \rceil = 7$ bits for each index pair $(i, j) \in [0, n] \times [0, m]$. Hence a backtrace table B of $7 \cdot (m + 1) \cdot (n + 1)$ bits would suffice, but for practical reasons a backtrace table B of $8 \cdot (m + 1) \cdot (n + 1)$ bits was chosen: We store the 7 bits in a byte to simplify the access to the bits.

The backtracing procedure for protein spliced alignments works analogous to the backtracing procedure for cDNA/EST spliced alignments described in the previous paragraph.

Jump Tables

The jump table is an array of length n (the length of the gDNA) which contains two values per column: the last value "*from*" to be computed in this column and the first value "*to*" to be computed in the next column. That is, it contains a jump pair *from-to*. It is used in combination

with a column-wise calculation of the DP matrix as follows. The first column calculation starts at the upper left position of the matrix and computes all entries of this column until the *from* position in the jump table is reached, the computation of this column is aborted and a jump to the *to* position in the next column is performed where the same process starts again. To make sure a computed matrix entry uses only correctly initialized values of the matrix, two checks have to be performed: The computed matrix entry directly after a jump cannot depend on the cell above it (that is, the corresponding recurrences are not used to compute this entry). Furthermore, a matrix entry cannot depend on the entry to the left of it, if this entry is in a row below the *from* jump table position of its column and therefore has not been computed (that is, the corresponding recurrences are also not used to compute this entry). The entry to the upper left of a computed matrix entry is always properly initialized.

Because in the jump table approach, the used DP entries always are a contiguous part of each row, the DP matrix was implemented as a *sparse matrix*. That is, only the used parts of the matrix are actually allocated, but the access to the matrix still works in the same way, which is possible with pointer arithmetic. This leads to a solution which contains the best of both worlds: A memory efficient matrix (no space wasted for unused matrix entries) and a simple DP implementation (because the matrix access stays the same with and without jump tables).

4.8.5 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 (the representation of protein-gDNA-alignments is described below). 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 $\left\lceil \frac{l}{2^{14}} \right\rceil$ multi edit operations. As a result, a spliced alignment usually does not require more than 2 kilobytes. This allows to store hundred thousands of spliced alignments in main memory, as often required when processing large data sets.

Alignments between a protein sequence and a gDNA are represented in a similar fashion. In addition to the five edit operations needed to represent EST-gDNA-alignments (match, mismatch,

insertion, deletion, intron) protein-gDNA-alignments need the following six edit operations: *mismatch with one gap*, *mismatch with two gaps*, *deletion with one gap*, *deletion with two gaps*, *intron with one base left*, and *intron with two bases left*. This leads to 11 edit operations in total for protein-gDNA-alignments, which means that four bits are needed to represent the edit operations in protein-gDNA-alignments. The remaining 12 bits can be used to store the iteration flag (multi edit operations for protein-gDNA-alignments are also represented as a 16-bit integer).

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 4.5.5.
- 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 data type with 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 alignment collection*. The spliced alignments are ordered by their start position in the gDNA. Large collections of ESTs/proteins often contain the same ESTs/proteins 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/proteins are processed, the spliced alignments are output or they are processed into consensus spliced alignments.

4.8.6 Consensus Spliced Alignments

The spliced alignment in the spliced alignment collection are clustered according to their gDNA location and gDNA strand. The resulting clusters represent connected overlap graphs (see Section 4.7). Each cluster is assembled into a consensus spliced alignment with the method of [HDM+03] (also described in Section 4.7). The resulting consensus spliced alignments are stored in an object of the *consensus spliced alignment collection* class.

The sets used in the consensus spliced alignment computation are represented as bit tables of the size $\frac{x}{w}$, where x is the number of spliced alignments and w is the word size (see Section C.4 for the bit table API).

4.8.7 Output of Spliced Alignments

GenomeThreader provides three output formats. The first format is text-based, intended to be read by users. It shows spliced alignments as in Fig. 4.2, with additional information about alignment scores, exon and intron boundaries, splice site scores, and probabilities, see `http://genomethreader.org/` for an example.

Alternatively, output is in XML conforming to a specification implemented in the RELAX NG schema language, available at `http://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.

Michael Sparks contributed 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/`) [SMS+02]. These are distributed both with the *GenomeThreader* package and independently at the *GenomeThreader* web site.

The third output format is GFF3 (described in detail in Section 5.2), a line-based format which is commonly used in Bioinformatics. This compact format allows for an easy preprocessing of the gene structures predicted by *GenomeThreader* with other tools, for example with *GenomeTools* (described in Chapter 5), and the easy rendering of the results, for example with GMOD's Generic Genome Browser (`http://www.gmod.org/`) [SMS+02] or *AnnotationSketch* [SGS+09] (see Section 5.7 for details).

Technically, the different output formats are implemented with the visitor pattern (see [GHJV94] for an introduction to object-oriented design patterns): A *spliced alignment visitor* can traverse the spliced alignment collection and accept every spliced alignment in it (and showing the corresponding output in the process). Similarly, a *consensus spliced alignment visitor* can traverse a consensus spliced alignment collection. This leads to a clear separation of the representation of (consensus) spliced alignments and the different output routines. It also makes it very easy to implement additional output formats. For that purpose, it is only necessary to implement a spliced alignment visitor and a consensus spliced alignment visitor which shows the desired output format.

4.8.8 Incremental Updates

We have also defined an intermediate 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.

The intermediate XML format differs from the "normal" XML format mentioned above in the way, that it only represents spliced alignments (and no consensus spliced alignments) and does so in a very detailed fashion. It is optimized for the purpose of incremental updates, for postprocessing purposes the normal XML output should be used.

There is an extra program *Gthconsensus* which reads the intermediate XML-format and runs the consensus spliced alignment algorithm, as described in Section 4.7. 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/proteins, store these on file, and quickly recompute the consensus for the entire set of spliced alignments. This is of great practical 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.

If the spliced alignments have been stored in GFF3 format, the role of *Gthconsensus* can be filled by the `csa` tool from *GenomeTools* (documented in Section B.4) which has the same functionality.

4.8.9 Software Development Tools

GenomeThreader is implemented in ANSI 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. 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 8 different UNIX platforms.

4.8.10 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/proteins which can help with this. However, the number of data sets which would allow to evaluate sensitivity and specificity values against a gold standard is still small. Moreover, the results computed by other gene prediction methods are often not reproducible.

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 down the program, 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 *GenomeThreader* to older versions of it, or to the output of *GeneSeqer* [UZB00], a program implementing the same spliced alignment algorithm, but with a different similarity filter and without intron cutout technique.

4.8.11 Practical Applications

GenomeThreader has been used in the following practical applications:

1. *GenomeThreader* was extensively used for the annotation of *Solanum lycopersicum* (tomato): "I just wanted to let you know that I successfully used GenomeThreader for (spliced) mapping of 6.7M 454 RNA-seq reads to tomato genome and results are integrated into the tomato annotation pipeline." [Fil10]. Also cited in [FTD+12], [BAG+10], and [CDT+08].

2. The *Ectocarpus siliculosus* (a filamentous brown alga) genome was annotated with the help of *GenomeThreader* [GBGS+11].

3. *GenomeThreader* was used in the annotation of the *Arabidopsis lyrata* genome [HPB+11].

4. *GenomeThreader* was used to predict the genomic structure and localization of the candidate genes of the bovine genome sequence [PFJ+11].

5. In the project which analyzed the filamentous seaweed *Ectocarpus siliculosus* (Dillwyn) Lyngbye, a model organism for brown algae, *GenomeThreader* was used to compute spliced alignments with the *Ectocarpus* cDNA sequences [Coc10].

6. In [LSH+10] *GenomeThreader* was used in a Soybean genome annotation pipeline for gene prediction with mRNA evidence.

7. *GenomeThreader* is used to compute spliced alignments in EuGène-maize, a web site for maize gene prediction [MJ10].

8. *GenomeThreader* was used to annotate *Physcomitrella patens* [RTL+10, SZB+10, MLH+09].

9. *GenomeThreader* was used in a study of of the *APOSPORY* locus in *Hypericum perforatum* [SAJ+10].

10. *GenomeThreader* was used in the creation of the SolEST database for spliced alignment [DTFC09]. SolEST is a database for the study of *Solanaceae* transcriptomes.

11. In a study on small RNAs in *Medicago truncatula* ESTs were aligned with the help of *GenomeThreader* [LBNS+09].

12. Proteins spliced alignments computed by *GenomeThreader* were used for MaizeGDB [SAS+09].

13. Alignments of assembled barley sequences against protein and EST databases were performed using GenomeThreader in [STG+09].

14. *GenomeThreader* was used to predict genes in a study on perennial flowering in *Arabis alpina* [WFV+09].

15. *GenomeThreader* was used to align available ESTs against the genome of the metazoan plant-parasitic nematode *Meloidogyne incognita* [AGA+08].

16. *GenomeThreader* was used in a comparative genomics study in rice [CBM08].

17. The genomics database PlantGDB uses *GenomeThreader* to spliced-align predicted polypeptides to genomic sequences [DFM+08].

18. *GenomeThreader* was used in a genome survey of the fungus *Moniliophthora perniciosa* to make protein-DNA spliced alignments between the BLAST first hit against and the genomic sequence, serving as a guide to delimit the start and stop codons and exon-intron boundaries of the regions of the contigs containing similarity with GenBank [MCC+08].

19. The MetWAMer system [SB08] can be used to refine gene structure predictions generated by the *GenomeThreader*.

20. Putative exons and open reading frames (ORFs) were predicted by alignment of ESTs and protein sequences using *GenomeThreader* [BJV+07].

21. *GenomeThreader* is used to compute spliced alignments in the *Tracembler* software [DWB07].

22. At the Munich Information Center for Protein Sequences (MIPS) *GenomeThreader* was used for for the annotation of plant genomes, for example *Zea mays* (maize) [BBG+06].

150 scientists from research institutions in 24 different countries acquired a *GenomeThreader* license which is available at `http://genomethreader.org/`.

Chapter 5

GenomeTools Genome Analysis Software

This chapter describes the *GenomeTools* open source genome analysis software. The *GenomeTools* project was started by the author of this thesis in 2006 to solve some problems encountered with *GenomeThreader* at the time (which has become rather monolithic at this point) and other open source bioinformatics projects (which were often hard to use, slow, and had many dependencies which made them hard to set up).

The goal of the project was to create a collection of command-line *tools* which would allow to analyze genomes and their annotations in a flexible manner. That is, the tools should be fast enough to allow interactive usage and allow to set up genome analysis pipelines easily (with UNIX pipelines).

Another goal of the project was to produce reusable software classes and modules which would simplify the development of additional software in the realm of genome informatics.

The *GenomeTools* software package consists of a software library named `libgenometools` and a collection of programs (the *tools*) based on the library which are combined into a single binary named `gt`. A single binary has the advantages that it needs less space on the machine (especially if it is statically linked) and is easier to update. It has the disadvantage that command-line completion is harder to set up.

GenomeTools is written in ANSI C in an object-oriented fashion with a focus on correctness, portability, efficiency and minimalism. Minimalism in this context means, that we strive for simple solutions which tend to have better usability and are easier to maintain in the long run.

To achieve correctness, many classes and modules of `libgenometools` have built-in unit tests and the tools contained in `gt` are tested automatically with an extensive test suite. To achieve high portability, C was chosen as programming language and very few external dependencies (libraries) were used in the project. *GenomeTools* was coded in standard-conformant ANSI C following the C90 standard with a few extensions from the C99 standard.

It is possible to compile most parts of *GenomeTools* on every POSIX compliant UNIX system

with just an ANSI C compiler and the GNU Make build system installed[1]. It is also possible to run `gt` on Windows with the help of Cygwin [Cyg]. To achieve an efficient implementation, C was used as a programming language and care was taken in the coding to get efficient code. Runtime and memory profiling helped to determine the bottlenecks of certain parts of the system an improve them. The Sections 5.5.5 and 5.5.6 show an example which illustrates that these efforts payed off. Minimalism was kept in mind during the development of the *GenomeTools* system to obtain a system which remains maintainable. This of course sometimes means to trade performance of the software for a simpler solution in order to get a maintainable system.

`libgenometools` is written in C, but with script bindings it is possible to use the library from languages like Lua, Python, Ruby, and Java.

The *GenomeTools* project was started by the author of thesis and saw many contributions by other developers during its development. Please refer to Appendix E for details. This chapter focuses on the parts of *GenomeTools* which the author of this thesis developed alone or in which he played a major role.

One of the reasons leading to the development of *GenomeTools* in 2006 where problems with *GenomeThreader*'s – at that time – rather monolithic design. Evaluating different parameter settings required rerunning large parts of the *GenomeThreader* pipeline which took too long to be usable in an interactive setting. This hindered a proper evaluation of *GenomeThreader*. Therefore, the idea was to break the monolithic *GenomeThreader* pipeline up into separate communicating tools based around a common annotation format. This allows for more flexible workflows, because it is much easier to introduce new steps into the annotation pipeline and to rerun parts of the pipeline with different parameter settings.

The Generic Feature Format Version 3 (GFF3) which is described in the Section 5.2 was chosen as an annotation format for the purpose of exchanging data between different parts of the pipeline, because it is generic enough to represent all kind of genome annotations (including the gene structure predictions produced by *GenomeThreader*) and at the same time compact enough to use it for large datasets in an interactive setting.

Instead of using the actual annotation files (as we do in *GenomeTools*) other toolkits (like BioPerl [SBB+02]) represent the contents of annotation files as relational databases (for example, Chado [MEC07]). This has the advantage, that the annotations are stored in a single location and consistency is easier to enforce. On the other hand, this approach is much harder to set up (the user has to install and populate the corresponding database first), which makes it inconvenient for small jobs. Furthermore, having a database means having a possible bottleneck, because the operations of retrieving and storing large annotation sets if often quite costly. For this reason, we chose to stay with simple annotation files (where large jobs are easier to parallelize).

This approach worked out nicely, as can be seen in Chapter 6 which describes the evaluation of *GenomeThreader* and the corresponding Appendix H which gives all the details on how the different tools comprising the *GenomeThreader* pipeline can be combined (and used to try out

[1] Some parts of *GenomeTools* have external dependencies. For example *AnnotationSketch* requires the *Cairo* 2D library [WP03], but *GenomeTools* can also be compiled without *AnnotationSketch*.

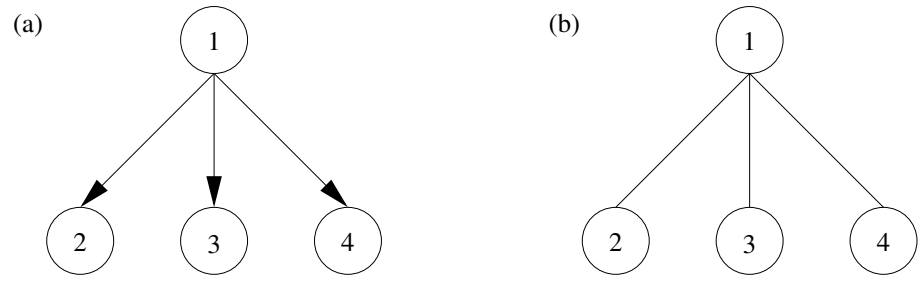


Figure 5.1: Example of an directed and undirected graph. Part (a) shows a directed graph $G = (V, E)$, where $V = \{1, 2, 3, 4\}$ and $E = \{(1, 2), (1, 3), (1, 4)\}$. Part (b) shows an undirected graph $G' = (V', E')$ (which is the undirected version of G), where $V' = V$ and $E' = \{\{1, 2\}, \{1, 3\}, \{1, 4\}\}$. G is also a DAG and a tree.

different parameter settings very easily). Most code which comprises *GenomeThreader* (except for the *Vmatch* code) is now open source and part of `libgenometools`. *GenomeTools* has been published under the ISC license which is documented in Appendix F.

5.1 Basic Notions

A *directed graph* G is a pair (V, E), where V is a finite set and E is a binary relation on V. The set V contains the *vertices* of the graph and the set E the *edges*. Edges from a vertex to itself are called *self-loops*. In an *undirected graph* $G = (V, E)$, the edge set E consists of unordered pairs of vertices, rather than ordered pairs. That is, an edge is a set $\{u, v\}$, where $u, v \in V$ and $u \neq v$. In an undirected graph, self-loops are forbidden, and so every edge consists of exactly two distinct entries.

A *path* of *length* k from a vertex u to a vertex u' in a graph $G = (V, E)$ is a sequence $\langle v_0, v_1, v_2, \ldots, v_k \rangle$ of vertices such that $u = v_0$, $u' = v_k$, and $(v_{i-1}, v_i) \in E$ for $i = 1, 2, \ldots, k$. The length of the path is the number of edges in the path. The path *contains* the vertices $v_0, v_1, \ldots, v_k$ and the edges $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$.

A path $\langle v_0, v_1, \ldots, v_k \rangle$ forms a *cycle* if $v_0 = v_k$ and the path contains at least one edge. A self-loop is a cycle of length 1. A graph with no cycles is *acyclic*. A directed acyclic graph is abbreviated as *DAG*. A *tree* is a DAG where no vertex has an in-degree larger than 1.

An undirected graph is *connected* if every pair of vertices is connected by a path. Given a directed graph $G = (V, E)$, the *undirected version* of G is the undirected graph $G' = (V, E')$, where $\{u, v\} \in E'$ if and only if $u \neq v$ and $(u, v) \in E$. A directed graph G is *weakly connected* if the undirected version of G is connected.

A topologically sorted depth-first search [CLR90, pp. 485-487] is necessary to show GFF3 DAGs (see the following Section) in a linear order without forward references.

5.2 The Generic Feature Format Version 3 (GFF3)

The Generic Feature Format Version 3 (GFF3) is a file format commonly used in bioinformatics to describe genomic features in a generic way [Ste11]. That is, it allows to describe features on a genomic sequence and relations between them. GFF3 was defined as an attempt to unify diverging earlier definitions like GFF2 [GFF] and GTF [GTF] in a backward compatible manner. GFF3 is a line based format and a GFF3 file has to start with the following line:

```
##gff-version   3
```

which denotes that this files is a GFF file confirming to version 3 of the specification. GFF3 defines four different kinds of lines: meta, comment, feature and termination lines.

5.2.1 Meta Lines

A meta line starts with "##" and defines important meta information about the annotation given in the file. For example, the common sequence-region meta directive looks like this:

```
##sequence-region seqid start end
```

It defines that the genomic sequence (segment) *seqid* is annotated by this GFF3 file (from position *start* up to position *end*). The sequence-region lines are not mandatory but strongly encouraged, because they allow GFF3 parsers to perform feature bound checking. A GFF3 file can contain multiple sequence-region lines. Here is an example for a single sequence-region meta line:

```
##sequence-region   ctg123 1 1497228
```

This states that in this file the genomic sequence segment named "ctg123" is annotated from position 1 up to position 1497228. In GFF3 genomic sequences start with position 1.
Another meta directive is the FASTA directive which looks like this:

```
##FASTA
```

It denotes that the annotation part of the GFF3 file is completed and a list of sequence entries in FASTA format follows. This allows to bundle annotations and sequences together in a single file. Example that shows the last part of a GFF3 file:

```
##FASTA
>ctg123
cttctgggcgtacccgattctcggagaacttgccgcaccattccgccttg
tgttcattgctgcctgcatgttcattgtctacctcggctacgtgtggcta
tctttcctcggtgccctcgtgcacggagtcgagaaaccaaagaacaaaaa
aagaaattaaaatatttatttgctgtggtttttgatgtgtgttttttat
aatgatttttgatgtgaccaattgtacttttccttttaaatgaaatgtaat
cttaaatgtatttccgacgaattcgaggcctgaaaagtgtgacgccattc
gtatttgatttgggtttactatcgaataatgagaattttcaggcttaggc
ttaggcttaggcttaggcttaggcttaggcttaggcttaggcttaggctt
```

```
aggcttaggcttaggcttaggcttaggcttaggcttaggcttaggcttag
aatctagctagctatccgaaattcgaggcctgaaaagtgtgacgccattc
>cnda0123
ttcaagtgctcagtcaatgtgattcacagtatgtcaccaaatattttggc
agctttctcaagggatcaaaattatggatcattatggaatacctcggtgg
aggctcagcgctcgatttaactaaaagtggaaagctggacgaaagtcata
tcgctgtgattcttcgcgaaattttgaaaggtctcgagtatctgcatagt
gaaagaaaaatccacagagatattaaaggagccaacgttttgttggaccg
tcaaacagcggctgtaaaaatttgtgattatggttaaagg
```

5.2.2 Comment Lines

Comment lines start with a single # and are usually ignored by GFF3 parsers. Example:

```
# example comment
```

5.2.3 Feature Lines

The actual annotations in GFF3 files are defined by feature lines. They do not start with a "#" character and contain exactly 9 tabulator separated columns. Undefined columns are replaced with the "." character. The columns have the following meanings:

1. *seqid*: The sequence ID of the defined feature. Usually corresponds to the sequence ID defined by a sequence-region meta line earlier in the GFF3 file.
2. *source*: The source of the feature. This is usually the name of a software program (for example, *GenomeThreader*) or a database (for example, *GenBank*).
3. *type*: The type of the feature, which has to be from one of the following three sources:
   (a) A term from the Sequence Ontology Feature Annotation (SOFA). SOFA is a condensed version of the Sequence Ontology (SO) that is especially well suited for labeling the outputs of automated or semi-automated sequence annotation pipelines [ELM+05].
   (b) A term from the (full) SO.
   (c) A SOFA or SO accession number written in the following way: `SO:000000`.
4. *start*: The start of the feature (relative to the sequence ID defined in the first column).
5. *end*: The end of the feature (relative to the sequence ID defined in the first column). *end* has to be greater or equal than *start*.
6. *score*: The score of the feature (a floating point number). The exact semantics is not defined, but it is recommended to use E-Values for sequence similarity values and P-values for *ab initio* gene prediction features.

7. *strand*: The strand character of the feature (on the sequence ID defined in the first column). "+" denotes features on the forward strand, "-" denotes features on the reverse strand, "." denotes features without a defined strand, and "?" denotes features where the strand is relevant but unknown.

8. *phase*: The phase of features with the type "CDS" (for all other feature types, this column has to be set to "."). The phase can be one of the integers "0", "1", or "2" and it denotes the number of bases which have to be removed from the start of the currently defined feature to get the next codon start. A correct phase entry is mandatory for CDS features.

9. *attributes*: The attribute column defines the list of feature attributes in the form `tag=value` which are separated by semicolons.

   Some attribute tags have predefined meanings, for example:

   - *ID*: The ID indicates the name of the feature and must be unique within the scope of the GFF3 file. IDs can be auto-generated. The main purpose of ID attributes is to define an anchor used in part-of relationships defined by the Parent attribute described below.
   - *Name*: The name of the features which is shown to the user. In contrast to IDs the names do not have to be unique.
   - *Parent*: Denotes the parent of the feature with the ID of an earlier feature as attribute value. Thereby a part-of relationship between the feature and its parent is established. That is, the feature with the parent attribute is part-of the referred feature (with the corresponding ID). A feature can have multiple parents. Because the parent attributes denote a strict part-of relationship, features linked together by parent and ID attributes form a directed acyclic graph (DAG): The features are the nodes in the graph, the parent and ID attributes define directed links, and the strict part-of requirement prevents cycles.
   - *Note*: A note defined in free text.

   Multiple attributes of the same type are separated by commas (","). For example:

   ```
   Parent=mRNA1,mRNA2,mRNA3
   ```

   Attribute names are case sensitive. Attributes that start with an uppercase letter are reserved for GFF3 (that is, they have a predefined meaning already or might have it in the future). Attributes that start with an lowercase letter can be freely used by users and applications (that is, the meaning can be defined by and is depended on the corresponding application).

5.2.4 Termination Lines

A termination line consists solely of three "#" characters in a row:

```
###
```

It denotes that all features introduced up to this point are completely defined. That is, a features with a given ID defined before a termination line cannot be referenced by a feature after the termination line (via a "Parent" attribute). This allows to process a GFF3 file in sections, because if a termination line is encountered all features before it are complete and can be processed by the software that parses the GFF3 file. The use of termination lines is recommended in order to allow processing the file section by section.

5.2.5 Example GFF3 File

The middle part of Figure 5.2 shows an example of a GFF3 file which annotates the sequence "ctg123" with a gene named "EDEN" comprised of three alternative splice forms. A graphical representation is shown in the upper part of Figure 5.2. Since some exons have multiple parents, the features described in the GFF3 feature lines of the file form a DAG . The same annotation can also be described as a tree (that is, no feature has multiple parents), as shown in the lower part of Figure 5.2.

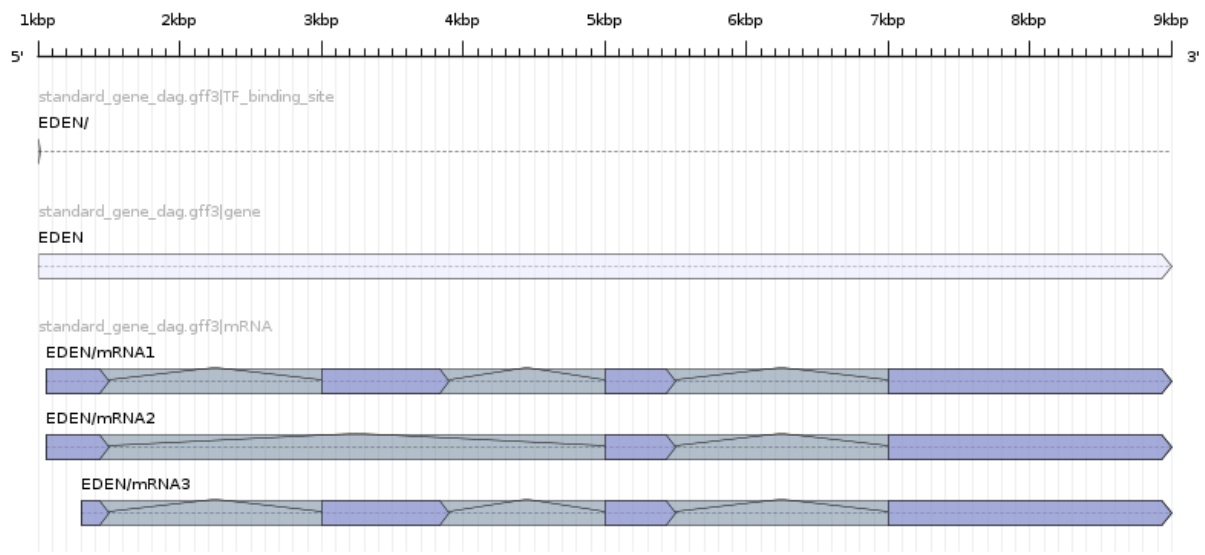
5.3 *GenomeTools* Overview

Due to the flexibility of GFF3 it is well suited to represent genome annotations of all kinds including gene structure predictions. But in order to store and manipulate such annotations in a software program, they have to be represented in an abstract way independent of the GFF3 notation. We chose to represent a GFF3 file as a collection of *genome nodes* as described in Section 5.4 below.

Since genome annotations and their corresponding GFF3 files can become rather large (on the order of Gigabytes) it is advisable to allow processing section by section for performance reasons. Annotations are usually created and modified locally (only in a small portion of the annotated genomic sequence at the same time) and therefore it is possible to treat a genome annotation as a stream of *genome nodes*, a so-called *node stream* which can be processed with so-called *node visitors*. Node streams and visitors are described in Section 5.5 below.

Treating genome annotations as streams also allows to easily combine multiple tools by chaining them, analogous to UNIX pipelines.

Hence the name *GenomeTools*, a collection of *tools* which create or modify *genome* annotations. Due to the well-known advantages of object-oriented programming (OOP) all parts of the *GenomeTools* system have been implemented in an object-oriented fashion. It is assumed that the reader is familiar with the object-oriented terminology and common design patterns. If necessary, please refer to [GHJV94] for more details on OOP and design patterns.




```
##gff-version   3
##sequence-region   ctg123 1 1497228
ctg123 . gene 1000 9000 . + . ID=gene1;Name=EDEN
ctg123 . TF_binding_site 1000 1012 . + . Parent=gene1
ctg123 . mRNA 1050 9000 . + . ID=mRNA1;Parent=gene1
ctg123 . mRNA 1050 9000 . + . ID=mRNA2;Parent=gene1
ctg123 . exon 1050 1500 . + . Parent=mRNA1,mRNA2
ctg123 . mRNA 1300 9000 . + . ID=mRNA3;Parent=gene1
ctg123 . exon 1300 1500 . + . Parent=mRNA3
ctg123 . exon 3000 3902 . + . Parent=mRNA1,mRNA3
ctg123 . exon 5000 5500 . + . Parent=mRNA1,mRNA2,mRNA3
ctg123 . exon 7000 9000 . + . Parent=mRNA1,mRNA2,mRNA3
###
```

```
##gff-version   3
##sequence-region   ctg123 1 1497228
ctg123 . gene 1000 9000 . + . ID=gene1;Name=EDEN
ctg123 . TF_binding_site 1000 1012 . + . Parent=gene1
ctg123 . mRNA 1050 9000 . + . ID=mRNA1;Parent=gene1
ctg123 . exon 1050 1500 . + . Parent=mRNA1
ctg123 . exon 3000 3902 . + . Parent=mRNA1
ctg123 . exon 5000 5500 . + . Parent=mRNA1
ctg123 . exon 7000 9000 . + . Parent=mRNA1
ctg123 . mRNA 1050 9000 . + . ID=mRNA2;Parent=gene1
ctg123 . exon 1050 1500 . + . Parent=mRNA2
ctg123 . exon 5000 5500 . + . Parent=mRNA2
ctg123 . exon 7000 9000 . + . Parent=mRNA2
ctg123 . mRNA 1300 9000 . + . ID=mRNA3;Parent=gene1
ctg123 . exon 1300 1500 . + . Parent=mRNA3
ctg123 . exon 3000 3902 . + . Parent=mRNA3
ctg123 . exon 5000 5500 . + . Parent=mRNA3
ctg123 . exon 7000 9000 . + . Parent=mRNA3
###
```

Figure 5.2: Drawing of example GFF3 file. The drawing was done with *AnnotationSketch* (described in Section 5.7), namely with the `sketch` tool (documented in Section B.19). The two shown GFF3 files describe the same gene structure, with the first beeing more compact.

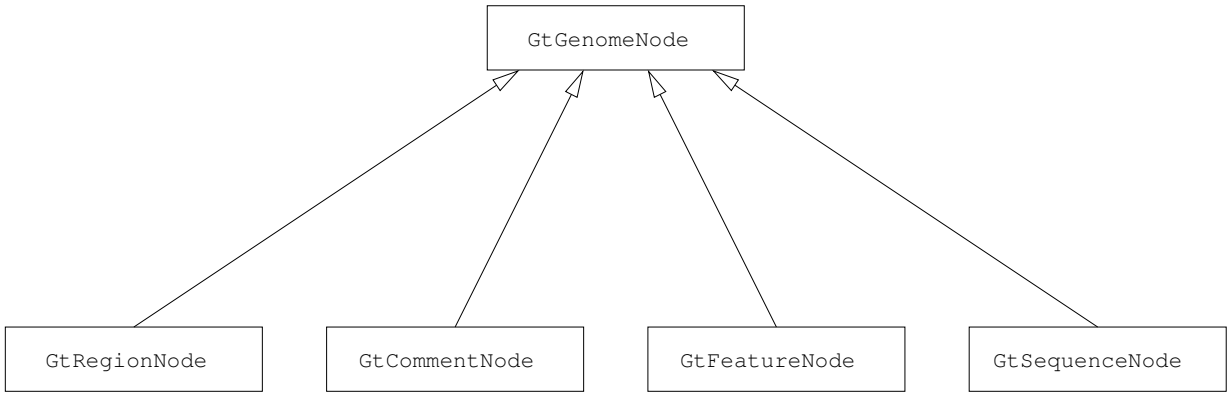


Figure 5.3: Implementations of the genome node interface (the arcs denote the implementors of the abstract genome node interface).

5.4 Representing GFF3 Files with Genome Nodes

The different kinds of GFF3 lines have been represented as different implementations of a common *interface*, the *genome node interface*. This allowed to design node streams in a way that they handle only one kind of object (namely, genome nodes). The following four implementations exist (an overview is given in Figure 5.3):

1. *region node*: a region node represents a sequence-region meta line and contains the sequence ID and the corresponding range (genomic start and stop position).

2. *comment node*: a comment node represents a comment line and contains the actual comment.

3. *sequence node*: a sequence node represents a singular embedded FASTA sequence (that is, each sequence after the FASTA meta directive described above is represented as a separate sequence node).

4. *feature node*: a feature node represents a GFF3 feature line and contains all information of such a line (sequence ID, source, type, genomic range, score, strand, phase and attributes). The parent attribute is handled in a unique fashion: It is used to link different feature nodes together into connected DAGs. For each feature DAG a "representative" is determined which can be used to pass the DAG around. Usually, this is the unique top-level feature. A top-level feature is a feature that is not the child of another feature (it has no parent attribute) but has one or multiple children (its ID attribute is listed as a parent attribute value of at least one other feature). In the example shown in Figure 5.2 the feature line describing the feature with type "gene" is the unique top-level feature and all other feature lines are part of its connected DAG (see Figure 5.4).

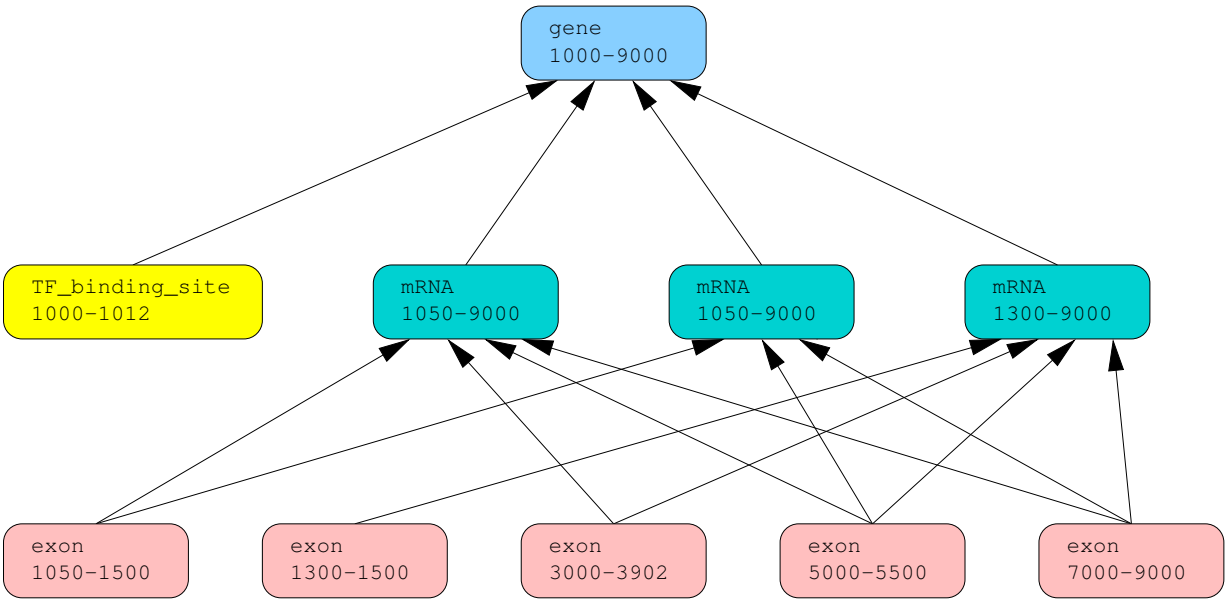


Figure 5.4: Feature node DAG for the GFF3 file given in Figure 5.2.

A special kind of feature is the *multi-feature*, a feature spanning multiple feature lines. It is denoted by the same ID attribute. For each part of a multi-feature a separate node in the feature DAG is introduced (with a special multi-feature flag set). Furthermore, one of the nodes comprising the multi-feature is distinguished as a *representative*. Figure 5.5 shows an example GFF3 file containing one multi-feature comprised of three feature lines.

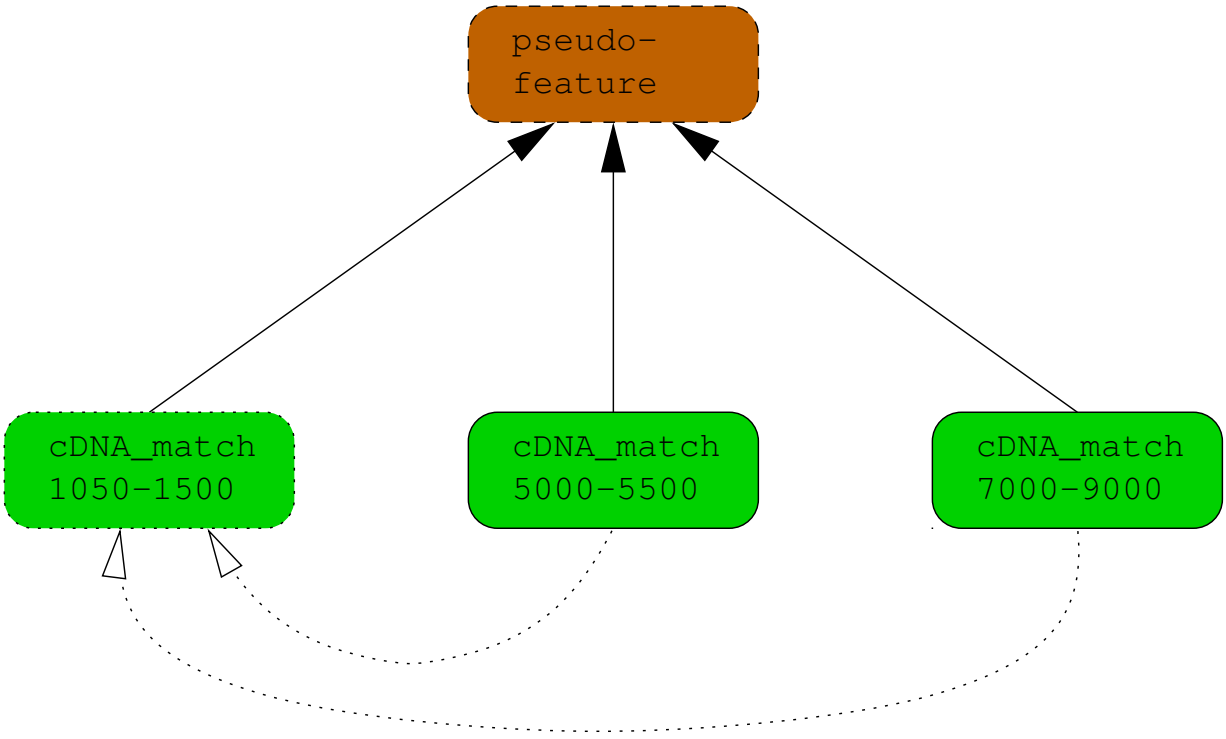
In the rare case that a connected DAG has no unique top-level feature, an artificial *pseudo-feature* is introduced which has all top-level features of the connected DAG as children (and thus becomes the unique top-level feature). This technique allows to have a unique representative for each connected DAG which can be used to easily pass connected DAGs through the streaming machinery (described below in Section 5.5). There are two cases in which a pseudo-feature has to be introduced.

First, if a multi-feature has no parent. Then all features which comprise the multi-feature become the children of a pseudo-feature.

Second, if two or more top-level features have the same children (and are thereby connected). Then all these top-level features become the children of a pseudo-feature.

The upper part of Figure 5.5 shows the internal representation of a multi-feature comprised of three feature line (as shown in the lower part of the Figure 5.5) with a top-level pseudo-feature.

Termination lines do not have to be represented, their only purpose is to hint the GFF3 parser that all feature nodes up to this point are completed. When genome nodes are output as GFF3 (after the processing of them in a tool is done), termination lines can be introduced after each




```
##gff-version   3
##sequence-region   ctg123 1 1497228
ctg123 . cDNA_match 1050 1500 5.8e-42 + . ID=cDNA_match1;Target=cdna0123 12 462
ctg123 . cDNA_match 5000 5500 8.1e-43 + . ID=cDNA_match1;Target=cdna0123 463 963
ctg123 . cDNA_match 7000 9000 1.4e-40 + . ID=cDNA_match1;Target=cdna0123 964 2964
###
```

Figure 5.5: Pseudo-feature and multi-feature example. The upper part of the figure shows the internal representation of the multi-feature given in the lower part. The lower "line" (of the upper part) shows the three feature nodes which comprise the multi-feature. The first one has a dotted border to denote that this node is the representative. The other two nodes have dotted links to their multi-feature representative. All three of them are children of the pseudo-feature top-level shown in the upper "line" with a dashed border.

(connected DAG) feature node — producing a GFF3 file which is optimally divided into the smallest possible parts.

5.5 Processing Genome Nodes with Node Streams and Node Visitors

The *node stream interface* (see Section C.29) allows to process genome annotations piecewise. Its *next* method returns the next available genome node in the stream. This means that the node stream architecture is *pull-based*: In a chain of streams, nodes are *pulled* from the last one which in turn pulls from the second last and so forth up to the first. The first stream is a *node source* which produces the genome nodes.

This is basically a lazy evaluation strategy [Wad71, HM76] for feature nodes. The feature nodes are pulled (and generated) from the previous stream only when they are actually required.

Node sources usually generate the genome nodes by reading them from a file. For example, this can be a file in GFF3 (see Section C.16), GTF (see Section C.20) or BED (see Section C.3) format. Tools like *GenomeThreader* [GBSK05] and *LTRharvest* [EKW08] are also node sources which generate genome annotations in the first place.

The GFF3 input stream employs a GFF3 parser (see Section C.18) to construct the corresponding genome nodes. After reading a proper termination line, the GFF3 parser can process the current section of the GFF3 file to return genome nodes. Thus the GFF3 input stream doesn't have to store all genome nodes resulting from the GFF3 file internally before returning them upon its next method calls. This makes it very memory efficient. If the GFF3 file does not contain the sequence-region meta directives, the GFF3 parser introduces the corresponding region nodes automatically.

All the other streams (which are not node sources) are called *node processors*. They take one or more node streams, process the nodes they retrieve from them, and pass them along. For example, the GFF3 output stream (see Section C.17) takes the nodes from its input stream, shows them as GFF3, and passes them along unmodified. The GFF3 output stream automatically inserts appropriate termination lines.

In combination with the different node sources this makes the construction of *converters* rather easy. One just has to combine a node source with an output stream. The `bed_to_gff3` tool (see Section B.1), `gtf_to_gff3` tool (see Section B.10), and `gff3_to_gtf` tool (see Section B.8) are examples of this technique. The `bed_to_gff3` tool chains the BED input stream (see Section C.3) with the GFF3 output stream (see Section C.17). Correspondingly, the `gtf_to_gff3` tool chains the GTF input stream (see Section C.20) with the GFF3 output stream and `gff3_to_gtf` tool the GFF3 input stream with the GTF output stream (see Section C.21).

This approach allows to construct $s \cdot o$ converters using s node sources and o output streams.

5.5.1 Sorted Streams

For many applications it is important that the genome nodes returned by a stream are *sorted*, according to a specific order. The *order* of genome nodes is defined as follows[2]:

- Region nodes come first. They are ordered by their sequence ID (lexicographically). Region nodes with the name sequence ID are ordered by their genomic range (similar to the order used for feature nodes, see below).
- Comment nodes are shown after region nodes. They are kept in the same order as they have been found in the input file
- Feature nodes come in the middle (that is after region and comment nodes, but before sequence nodes). They are lexicographically ordered by their sequence ID. Feature nodes with the same sequence ID are ordered by their genomic range as follows:
  - Two ranges a and b are the same, if their start and end points are exactly the same.
  - Range a is before the range b, if and only if the start point of a is before the start point of a or the start points are equal and if the end point of a is before the end point of b.

  This range comparison it implemented in the method `gt_range_compare()` which is documented in Section C.36.
- Sequence nodes are last. They are lexicographically ordered by their descriptions.

The sort stream (see Section C.41) allows to order genome nodes. It pulls all genome nodes from its input stream (storing them in memory), sorts them (according to the order described above) in a stable way and returns them one after the other upon calls of its *next* method. To sort GFF3 files on the command line, the `gff3` tool (see Section B.7) can be used with option `-sort`.

Appendix D shows how a simple GFF3 sorter could be implemented using the `libgenometools` and illustrates how the chaining of streams works on the source code level.

The sorting performed by the sort stream only refers to top-level feature nodes. Children of top-level feature nodes are sorted according to the same order used for the construction of the feature DAGs (described below in Section 5.5.4).

The ordering of input streams happens right at the interface level (see method `gt_node_stream_create()` documented in Section C.29) which allows to get this functionality in all implementing classes without any code duplications.

[2]the method which implements this is `gt_genome_node_cmp()` documented in Section C.22

5.5.2 Merging Sorted Stream

Because sorting genome nodes requires large amounts or memory (all genome nodes have to be held in main memory at the same time), for very large input sets a multiway merge strategy can be employed:

1. Split the input data set into *bins*.
2. Sort the bins separately.
3. Merge the sorted bins in a sorted fashion.

To implement this strategy, the merge stream (see C.28) was implemented. It takes a list of sorted input streams and merges them to deliver a single sorted output stream — without storing all genome nodes in main memory. A sorted input stream is a GFF3 input stream which enforces that the GFF3 file it reads is already sorted[3]. At the command line level, the merge functionality is available with the `merge` tool (see Section B.14). Of course, the `merge` tool was implemented with the merge stream.

5.5.3 The Case for Sorted Streams

The sortedness of streams and feature DAGs allows to simplify the implementation of many tools. It also allows for memory efficient implementations, because the corresponding tool needs only "local" knowledge of the annotation, allowing for a piecewise processing.

For example, the `uniq` tool (described in Section B.22) which filters out repeated feature node graphs uses a sorted input stream. This allows the uniq stream (which is used to implement the `uniq` tool, see Section C.49) to look only at adjacent nodes when is looks for repetitions. Without a sorted input it would have to hold all feature nodes in main memory before looking for repetitions, a huge difference in memory consumption.

Another example is the `csa` tool (described in Section B.4) which transforms spliced alignments into consensus spliced alignments (CSAs). A sorted input stream allows the CSA stream (which is used to implement the `csa` tool, see Section C.6) to slide a window over the annotation, to determine which sets of spliced alignments must possibly be combined into consensus spliced alignments. This considerably reduces the running time compared to an approach that process an unsorted stream and thus has to store all features in main memory.

The automatic sorting of children of feature nodes (described below in Section 5.5.4) also simplifies the implementations of several tools.

An example for this is the `interfeat` tool (described in Section B.12) which adds intermediary features between outside features (outside features are features which are supposed to enclose the intermediary features on the genomic sequence). The sortedness of children during the

[3] See the constructor `gt_gff3_in_stream_new_sorted()` documented in Section C.16

traversal of a feature node DAG allows the inter feature stream (which is used to implement the `interfeat` tool, see Section C.25) to keep a single pointer to the last encountered feature node and use this pointer to determine if an intermediary feature needs to be introduced. Otherwise it would have to look at all children of a feature node, sort them itself, and figure out where to introduce intermediary features.

Another example is the `mergefeat` tool (described in Section B.15) which merges adjacent features of the same type (which might have been introduced by other tools or are artifacts in GFF3 files not conforming to the notational conventions). The sortedness of children reduces the amount of feature node pointers the merge feature stream (which is used to implement the `mergefeat` tool, see Section C.27) has to keep in memory to determine if a merge is necessary.

Similar advantages of the automatic sorting of children can be found in the `cds` tool (see Section B.3, implemented with the CDS stream documented in Section C.5) and the `extractfeat` tool (see Section B.6, implemented with the extract feature stream documented in Section C.12).

5.5.4 Memory Efficient Representation of Genome Nodes

Because it is sometimes still necessary to hold large amounts of genome nodes in main memory at the same time, great care has been taken to represent them efficiently. Thereby, the efficient representation of feature nodes is most important, because GFF3 files are usually dominated by feature lines. Each implementation of genome nodes shares the instance variables defined in the genome node interface.

For some string classes (for example, filenames and feature node types), the following holds: The number of different strings is small compared to the number of nodes, so that this number and the space used for the strings can be considered a constant.

Common Representation of Genome Nodes

The common representation of all implementation of the genome node interface is as follows:

- A pointer to the class implementing the genome node interface. This is where the method pointer of the implementation are stored. The class is only held once in memory (for each implementation), therefore the class is represented by a single pointer and the space consumption for each object can be considered constant.
- A pointer to a string (see Section C.43) containing the filename from which this genome node originates. This is just a new reference to a string containing the filename (that is, each filename is only stored once in memory). Therefore the string is represented by a pointer and the space consumption for each object can be considered constant.
- A pointer to a hash map (see Section C.23) containing the user data attached to this genome node (user data means auxiliary data which can be added to the genome node by the user

of the genome node interface). The hash map is only created on demand (that is, if user data is attached to the node) and therefore usually a null pointer is stored. For example, user data is used in *LTRdigest* [SWGK09] to attach related alignments to genome nodes which are output at a later stage in the streaming machinery.

- A 32-bit integer storing the line number of the file where this genome node originated from.
- A 32-bit integer storing the reference count to this object.
- A 32-bit integer storing the number of items in the user data hash map mentioned above. This bookkeeping allows to free the hash map, if it is no longer needed (that is, if it does not contain any more user data items).

Representation of Region Nodes

- A string (see Section C.43) storing the sequence ID of this region node. The GFF3 parser (see Section C.18) reuses the sequence ID string it passes to the constructor of region and feature nodes, therefore the the sequence ID string of each object is represented by a single pointer and the space consumption for each object can be considered constant.
- A genomic range (see Section C.36) to store the range which this region node refers to. It consists of two word length integers storing the start and end positions of the region node.

Representation of Comment Nodes

A comment node just contains a null terminated ("\0") array of characters (C string) representing the comment.

Representation of Feature Nodes

- A string (see Section C.43) storing the sequence ID of this feature node. The GFF3 parser (see Section C.18) reuses the sequence ID string it passes to the constructor of region and feature nodes, therefore the the sequence ID string of each object is represented by a single pointer and the space consumption for each object can be considered constant.
- A string storing the source of this feature node. The GFF3 parser caches the source strings it uses for feature nodes. Therefore the source is represented by a single pointer and the space consumption for each object can be considered constant (usually only a few different sources can be found in a GFF3 file).

- A C string storing the type. Symbols (see Section C.54) are used for types, which makes the space consumption basically just this pointer, because the actual string is only stored once for every type (and the set of types is small compared to number of type strings). Note that the set of types is not known at compile time, therefore the type cannot be represented with an enum.
- A genomic range (see Section C.36) consisting of two word length integers which stores the start and the end of the feature node.
- A floating point number storing the score of this feature node.
- A tag/value map (see Section C.46) used to store the attribute tags and values. The tag/-value map is optimized for space (that is, memory consumption) at the cost of time. Basically, each read/write access requires $O(n)$ time, where n denotes the total length of all tags and values contained in the map. The space requirement for a tag/value map is $n + 2t + 1$ bytes (where t is the number of tags) plus the pointer to the memory area.
- A 32-bit integer is used as a bit field storing certain properties of the feature node in a space efficient manner. For example, the strand, the phase, the multi-feature status, and the pseudo-feature status are all stored here.
- A double-linked list storing the children of the feature node. The double-linked list is sorted according to the order described in Section 5.5.1 above. The double-linked list is created on demand (that is, once the first child is added to a feature node).
- A pointer to the multi-feature representative (described above in Section 5.4) of this feature, if it has one.

Representation of Sequence Nodes

A sequence node contains pointers to two strings (see Section C.43): one stores the description and the other the actual sequence.

5.5.5 Memory Footprint of Parsed GFF3 Files

The memory representation of genome nodes has some space advantages over the representation in the GFF3 file, because duplicated things are only stored once with pointers referring to them for each of their occurrences: this holds for sequence IDs, sources, and types.

On the other hand the additional information stored in genome nodes (filename, userdata and its count, reference count, line number) and the linking between feature nodes has its cost. Especially on 64-bit system with pointers requiring 8 bytes main memory to hold a GFF3 file is larger than the actual file size. This can be seen in the test runs given below[4].

[4]Details on how the tests were performed are given in Appendix G.

File	file size	# genome node DAGs	# Genes	# mRNAs	# Exons
`tair.gff3`	40 MB	33616	28775	35386	215909
`fruitfly.gff3`	88 MB	290238	-	-	-
`ensembl.gff3`	157 MB	51859	51715	113694	1036805

Table 5.1: Properties of GFF3 test files. The `fruitfly.gff3` file contains only EST based spliced alignments and no annotations.

Program	`tair`	`fruitfly`	`ensembl`
`gt gff3 -show no` (32-bit)	0.15	0.08	0.60
`gt gff3 -show no` (64-bit)	0.28	0.15	1.01
`gt gff3 -show no -sort` (32-bit)	87.82	151.40	289.10
`gt gff3 -show no -sort` (64-bit)	141.83	222.05	447.42
`parse.pl`	10.53	10.53	10.63
`parse_and_store.pl`	1161.29	2047.49	4293.83

Table 5.2: Memory consumption of different GFF3 parsers on test files (the memory peak in MB is shown).

The memory footprint was determined on three test files with the properties described in Table 5.1. The test files have been retrieved from The Arabidopsis Information Resource (TAIR) [RBB+03], the Berkeley Drosophila Genome Project (BDGP) [MCM+02] and Ensembl [FAB+11] (see Appendix G for the exact sources). To determine the actual space requirement of the GFF3 parsing machinery from the *GenomeTools*, we applied the `gff3` tool with the option `-show no` (to test the streamed memory consumption) and with the additional option `-sort` (to test the memory consumption, if all genome nodes are held in main memory). To test the influence of the word size on the memory footprint, both the 32-bit and the 64-bit binary were used. The memory consumption was compared with two Perl script `parse.pl` and `parse_and_store.pl` which employ the *Bio::Tools::GFF* class from BioPerl [SBB+02] to do the same job as the `gff3` tool calls (details are given in Appendix G). The results are given in Table 5.2: As one can see, the memory consumption of the streaming calls is very low. For the calls which store the features, the memory consumption is a multiple of the file size (see Figure 5.3): for the `gff3` tool (32-bit) the memory consumption is around 2 times the file size and for the 64-bit binary around 3 times the file size. In comparison, the memory consumption for the BioPerl script is around 27 times the files size.

5.5.6 Efficient GFF3 Parsing

As one can see from the memory peaks shown in Figure 5.2, the *GenomeTools* GFF3 processing machinery is well suited to process very large GFF3 sets, especially if they are streamed in a sorted fashion. But to achieve interactive usage, short runtimes are important. Figure 5.4 shows

Program	tair	fruitfly	ensembl
gt gff3 -show no -sort (32-bit)	2.2×	1.7×	1.8×
gt gff3 -show no -sort (64-bit)	3.4×	2.5×	2.9×
parse_and_store.pl	29.0×	23.3×	27.3×

Table 5.3: Memory consumption ratios (the memory peak in relation to GFF3 file size). parse_and_store.pl employs the *Bio::Tools::GFF* class from BioPerl [SBB+02].

Program	tair	fruitfly	ensembl	1.5GB.gff3
gt gff3 -show no (64-bit)	3s	5s	8s	89s
parse.pl	88s	160s	338s	3500s

Table 5.4: Runtime of different GFF3 parsers.

the runtime of the *GenomeTools* GFF3 parser in comparison to the one from BioPerl (employed in the parse.pl script). The results clearly show, that the GFF3 parser from *GenomeTools* allows interactive usage for all but the largest GFF3 files and the BioPerl parser is not well suited for interactive usage, because the runtime is quite long even for GFF3 files of moderate size.

5.5.7 Node Visitors

The *node visitor* interface (see Section C.31) is an interface following the *visitor pattern* (see [GHJV94]). This allows implementing classes to differentiate between different implementations of the genome node interface in an elegant manner. Each node visitor implements a method for each genome node implementation it wants to process and the genome node interface takes care of calling the correct method.

The following example illustrates the process. Let's look at a node stream that pulls a genome node from its node source and wants to process it with its node visitor. It just has to call the method gt_genome_node_accept() (which is documented in Section C.22) with the node visitor passed as an argument. gt_genome_node_accept() simply calls the *accept* method of its implementing class (this differentiates between the different genome node implementations). Suppose we have a feature node. That means the accept method of the feature node class is called which in turn dispatches the call to gt_node_visitor_visit_feature_node() (which is documented in section C.31) in which the feature node is processed.

That is, the visitor pattern allows to differentiate between different implementations of an interface (the genome node interface in our case) in the visitor class — without the need to do that in every class which processes genome nodes. With the visitor pattern, it is easy to add a new implementation of the genome node interface without much change in all the classes which process them.

For many node streams, most of the action happens in the corresponding node visitor class. If everything happens inside the visitor class, the visitor stream (documented in Section C.50) can be used. This applies its node visitor to every genome node passed through it.

5.6 *LTRdigest*

The program *LTRdigest* [SWGK09] which is also part of *GenomeTools* profits from the versatility of the node stream machinery. It uses a GFF3 input stream to read LTR retrotransposon candidates given in a GFF3 file, passes them through an *LTRdigest* stream to add annotations of internal features of the putative LTR retrotransposons, and shows the enriched feature nodes with the help of the GFF3 output stream

Although the whole genome node and node stream machinery was primarily developed to process gene structure annotations, *LTRdigest* shows that it is generic enough to process completely different genomic features like LTR retrotransposons (which, of course, is also due to the flexibility of the GFF3 format).

GenomeTools also contains the *LTRharvest* software [EKW08] which can be used to predict LTR retrotransposon candidates possibly further processed by *LTRdigest*. Instead of using GFF3 input files, *LTRdigest* can directly process nodes directly from *LTRharvest*.

Despite the fact that *GenomeTools* was very useful for the implementation of *LTRdigest*, most parts of the *GenomeTools* are not yet published. A publication is in preparation[5].

5.7 *AnnotationSketch* Genome Annotation Drawing Library

5.7.1 Introduction

Genome annotations are often provided in the GFF3 format [Ste11] using the vocabulary of the Sequence Ontology [ELM+05]. It is not uncommon for annotations to contain tens of thousands of features. This makes it difficult to obtain an overview of the structure and hierarchy of the features in a particular genomic location by looking at tabular data. For this reason, annotation browsers like the *UCSC Genome Browser* [KSF+02] or *GBrowse* [SMS+02] as well as curation tools like *Apollo* [LSH+02] provide an intuitive graphical representation of annotated features, allowing, for example, to jump to a specific feature. However, such drawing components are often tied to the particular tool's data model and programming language, limiting their reusability in other contexts. While the BioPerl toolkit [SBB+02] includes the *Bio::Graphics* module as an established reusable and extensible solution for genome annotation drawing, it has the disadvantage to be conveniently usable in Perl applications only. Furthermore, its output is limited to files,

[5] Addendum June 2013: The corresponding paper [GSK13] has been accepted.

which is inefficient in desktop GUI applications because temporary files must be created. Another disadvantage is the need for a database backend and the explicit definition of *aggregators* to visualize feature relationships.

5.7.2 Design and Implementation

AnnotationSketch is designed to be a small and efficient drawing library for genome annotations with a focus on simplicity, allowing to draw any given annotation in a wide variety of application fields while automatically considering feature relationships. *AnnotationSketch* directly uses annotation graphs as its underlying data model. They can be created and manipulated dynamically by user code (for example in custom gene prediction software) via library functions available in *GenomeTools*. Alternatively, they can be imported from GFF3, GTF or BED files by using the respective *GenomeTools* parser. The actual drawing process is divided into three separate phases:

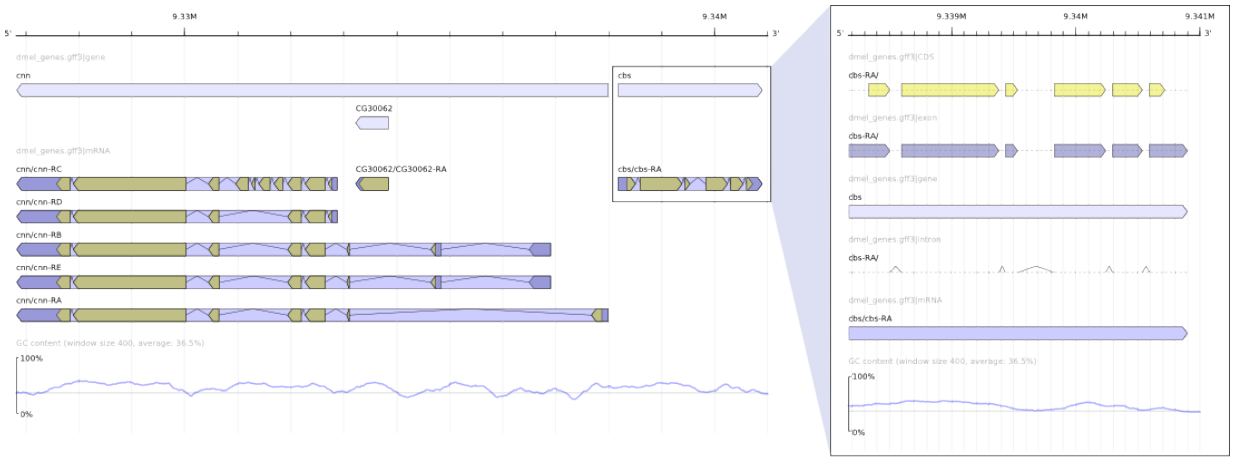
1. *Feature selection phase.* Obtain a collection of features, either by retrieving, from an efficiently searchable *feature index*, all features overlapping the range of sequence positions to draw, or by supplying an array of features. Based on user preferences and feature relationships, group single features into *blocks*, the smallest units which can be laid out.

2. *Layout phase.* Distribute the blocks into a hierarchical structure representing vertical *tracks* (containing all blocks with a common feature, for example type) and *lines* (each containing non-overlapping blocks) such that the obtained packing in the 2D representation is most compact.

3. *Rendering phase.* Use the track and line structure as a blueprint for drawing a specific output format.

While some concepts (such as the use of tracks) are shared with *Bio::Graphics*, tracks need not be explicitly created by the programmer but are determined from the feature types encountered in the input data. This minimizes programming overhead. Nevertheless, user-defined tracks can be created according to arbitrary block properties. Each feature can also optionally be drawn transparently on top of its parent feature (for example, all *exon* and *intron* features are placed into their parent *mRNA* or *gene* track). Relationships are implicitly given by the annotation graph. This approach, called *collapsing*, can significantly improve visual clarity in renderings of annotations with many levels of hierarchy.The *AnnotationSketch* library is implemented using ANSI C in an object-oriented style. This approach makes it straightforward to create bindings to other object-oriented languages. Bindings for the Ruby, Python and Lua scripting languages are included with the software. We also provide an *AnnotationSketch*-based command line tool named `sketch` (documented in Section B.19), allowing to draw GFF3, GTF and BED annotation data.

An image is represented by one class per structural component (*Element*, *Block*, *Line*, *Track* and *Diagram*) or processing result (*Layout*). Additionally, *custom tracks* – special classes implementing a common interface – can be added to a *Diagram* and allow development of user-defined



(a) collapsed view

(b) uncollapsed view

Figure 5.6: Two images drawn by *AnnotationSketch*, showing the *cnn* and *cbs* gene region from the Ensembl *Drosophila melanogaster* annotation (release 51, (a) 2R:9326816–9341000, (b) 2R:9338169–9341000). At the bottom, the GC content of the respective sequence (calculated on the fly) is drawn via an example custom track attached to the diagram. Image (a) shows the *exon* (dark blue), *CDS* (yellow) and *intron* type features collapsed into their *mRNA* type parent features (medium blue) for a more concise view. Image (b) shows an uncollapsed rendering of the *cbs* gene.

drawing functionality, for example to display arbitrary plots along the annotation (see Figure 5.6 for example output). On user request, the *Layout*'s *sketch* method invokes rendering methods for each component in a drawing surface abstraction called *Canvas*, which in turn calls primitive shape drawing methods of a *Graphics* object wrapping a graphics back-end. The currently used *Graphics* implementation uses the *Cairo* 2D graphics library [WP03], allowing output to PNG bitmaps as well as PDF, SVG and PostScript vector formats. *Cairo* also facilitates integration into GUI-based applications by providing native rendering surfaces for windowing systems like the X Window System or Mac OS X *Quartz*. Attaching a rendered image to a user interface is possible by mapping 2D image coordinates to the respective feature. This enables *AnnotationSketch* to be used, for example, in a genome annotation browser or editor. Recent examples are the FISH Oracle web server [MSSK11] for flexible visualization of DNA copy number data in a genomic context and the CASSys software system [AKB11] which is an integrated solution for the interactive analysis of ChIP-Seq data.

User preferences are stored in instances of the *Style* class. Configuration options (colors, borders, collapsing flags etc.) can be set and retrieved both globally and for specific feature types. Additionally, it is possible to supply callback functions to make colors or captions dependent on individual feature properties.

To evaluate the performance, 100 random regions of 500 kb length from the *Drosophila melanogaster* Ensembl release 50 GTF gene annotations were drawn to a 800 pixels wide PNG image from a *FeatureIndex* held in memory, resulting in an average rendering time of 0.61 seconds per im-

age. It has to be noted that the time-consuming part appeared to be the bitmap rastering process. Using SVG or PDF for output reduced the time to 0.05 seconds (SVG) and 0.04 seconds (PDF) per image. In contrast, creating a comparable output with a Perl script using *Bio::Graphics* took 3.98 seconds on average per PNG image and 4.58 seconds per SVG file. This makes *AnnotationSketch* favorable in SVG-based web applications to reduce server load. Memory usage of the *AnnotationSketch*-based program peaked at 34.1 MB for a single run, of which 33.3 MB were occupied by the feature index for the 15.6 MB GFF3 file. The Perl script's average peak memory usage for a single run was 15.45 MB. However, the memory usage of the Perl script did not include the MySQL database storing the features.

5.7.3 Conclusion

AnnotationSketch provides a fast and easy to use library for drawing annotations to be used in any application in which a light-weight visualization of annotation data compatible with an annotation graph format is desired. By implementing all functionality in C and using foreign function interfaces to add high-level bindings to a variety of other languages afterwards, applications can benefit from both portability and interface consistency across all bindings. For further details on *AnnotationSketch*, please refer to the *AnnotationSketch* documentation which can be found at `http://genometools.org/annotationsketch.html`.

Chapter 6

Evaluation of Gene Prediction Methods

6.1 nGASP Evaluation

The paper [CFM+08] describes the nematode genome annotation assessment project (nGASP), an experiment to test the accuracy of different gene finders on a *C. elegans* data set. All test and training data which have been used for the gene predictions have been published, which makes this assessment well suited to compare *GenomeThreader* with the results of other similarity-based gene prediction programs of the assessment published in the paper.

6.1.1 nGASP Dataset

The nGASP dataset is comprised of the following components:

- 10 Mb of the *C. elegans* genome (WormBase release WS160 [WB06]) as a training set and 10 Mb as a test set. The 10 MB of the training and test set are both comprised of non-overlapping 1 Mb regions and they represent $\approx 10\%$ of the *C. elegans* genome. See Table 6.1 for further details about the regions.

- Two gene reference sets **ref1** and **ref2** for the test and the training regions, respectively. **ref1** contains all genes from Wormbase release WS160 that were supported by full-length cDNAs across their entire coding region. **ref2** additionally contains manually curated genes, based on gene predictions and experimental data [CFM+08]. Table 6.2 shows the number of features contained in the sets.

  ref1 was used to compute the sensitivity and **ref2** to compute the specificity values. The reason for this is that true-positive and false-negative values computed in relation to **ref1** are more dependable, because the gene models in **ref1** are of higher quality than the ones in **ref2** (because the gene models in **ref1** are all supported by full-length cDNAs). On the other hand, the false-positive values computed in relation to **ref2** are more dependable, because a higher fraction of true genes are contained in **ref2** [CFM+08].

Type of nGASP region	Criterion used for selecting region	Coordinates in the *C. elegans* WS160 genome
Training	High conservation, high gene density, autosomal	II: 2000001–3000000
Training	High conservation, high gene density, autosomal	V: 9000001–10000000
Training	High conservation, low gene density, autosomal	III: 1000001–2000000
Training	High conservation, low gene density, autosomal	IV: 2000001–3000000
Training	Low conservation, high gene density, autosomal	I: 12000001–13000000
Training	Low conservation, high gene density, autosomal	V: 4000001–5000000
Training	Low conservation, low gene density, autosomal	I: 2000001–3000000
Training	Low conservation, low gene density, autosomal	II: 13000001–14000000
Training	High conservation, low gene density, X-chromosome	X: 3000001–4000000
Training	High conservation, low gene density, X-chromosome	X: 2000001–3000000
Test	High conservation, high gene density, autosomal	IV: 7000001–8000000
Test	High conservation, high gene density, autosomal	V: 12000001–13000000
Test	High conservation, low gene density, autosomal	IV: 1–1000000
Test	High conservation, low gene density, autosomal	I: 14000001–15000000
Test	Low conservation, high gene density, autosomal	V: 16000001–17000000
Test	Low conservation, high gene density, autosomal	II: 1–1000000
Test	Low conservation, low gene density, autosomal	IV: 14000001–15000000
Test	Low conservation, low gene density, autosomal	I: 1000001–2000000
Test	High conservation, low gene density, X-chromosome	X: 4000001–5000000
Test	High conservation, low gene density, X-chromosome	X: 8000001–9000000

Table 6.1: The genomic nGASP test and training regions. The table was taken from [CFM+08]

.

- 42496 protein sequences with *BLAST* [AMS+97] matches in the training or test regions, but excluding matches to proteins encoded by genes in the test regions. For details on the *BLAST* run see [CFM+08]. The average length of the protein sequences is 688.35 bp.
- 44820 cDNA/EST sequences with *BLAT* [Ken02] matches in the training or test regions. The average length of the cDNA/EST sequences is 479.64 bp.

The coordinates of the *BLAST* and *BLAT* runs mentioned above are also available, but they have not been used to run *GenomeThreader* on the nGASP dataset, because *GenomeThreader* has its own matching phase (which is described in Section 4.5.1).

6.1.2 nGASP Gene Finder Categories

The gene prediction programs assessed for nGASP have been divided into four categories:

File	# Genes	# mRNAs	# Exons
`training_ref1.gff3`	432	780	4348
`training_ref2.gff3`	1907	2429	14375
`test_ref1.gff3`	493	894	5294
`test_ref2.gff3`	1948	2549	15558

Table 6.2: Number of features in nGASP annotation files. The numbers refer to the number of protein-coding features. That is *# genes* refers to the number of protein-coding genes, *# mRNAs* to the number of protein-coding isoforms, and *# exons* to the number of protein-coding exons.

1. *Ab initio* gene finders, programs which predict genes solely based on the genomic sequence.
2. Multi-genome alignment programs.
3. Similarity-based methods which predict genes based on cDNA/EST and/or protein sequences.
4. So-called *combiners*, software tools which combine the prediction results from other predictors and potentially refine them.

GenomeThreader uses cDNAs/ESTs to predict gene structures, therefore it belongs into category 3 and to assess it's prediction quality it has to be compared against other programs in this category.

6.1.3 *GenomeThreader* Assessment

A step-by-step outline on how this was done is given in Appendix H. This makes the results shown in Table 6.3 easy to reproduce. To compare the results of the different programs, the average of the nucleotide sensitivity, nucleotide specificity, exon sensitivity, and exon specificity has been used as a metric (these measures are explained in Section 3.3).

As one can see in Table 6.3, *GenomeThreader* is slightly better than all the other competing gene prediction programs in the categories 1, 2, and 3.

6.1.4 Influence of BSSMs on Prediction Accuracy

It was also tested how different BSSMs influence the prediction accuracy of *GenomeThreader* on the nGASP dataset.

For this purpose, *GenomeThreader* was used with different parameterizations as follows:

- For *gth_bssm*, *GenomeThreader* was called with the options described in Appendix H.5.

Program	Cat	NSn	NSp	NAvg	ExSn	ExSp	ExAvg	Average
gth_bssm	3	94.70	**95.83**	95.27	90.39	79.67	85.03	**90.15**
mgene_v1	3	98.13	91.37	94.75	89.37	80.41	84.89	89.82
mgene_v3	3	98.14	91.36	94.75	89.40	80.35	84.88	89.81
augustus	3	98.46	89.69	94.07	**90.98**	79.98	**85.48**	89.78
fgenesh++	3	96.84	89.17	93.00	88.64	**80.68**	84.66	88.83
mgene_v2	3	**98.53**	87.36	92.94	90.19	75.64	82.91	87.93
gramene_v1	3	97.53	94.94	**96.23**	87.03	71.63	79.33	87.78
mgene_v1	2	97.31	90.42	93.87	84.20	78.08	81.14	87.50
mgene_v2	2	97.31	90.41	93.86	84.20	78.03	81.12	87.49
mgene_v1	1	96.64	90.97	93.81	83.11	78.30	80.70	87.25
mgene_v2	1	96.36	91.08	93.72	82.75	78.42	80.59	87.15
mgene_v3	1	96.36	91.07	93.72	82.75	78.35	80.55	87.13
gramene_v2	3	97.98	94.17	96.08	87.03	67.63	77.33	86.70
eugene_v2	3	98.28	84.08	91.18	**90.98**	70.11	80.55	85.86
craig	1	95.15	90.36	92.75	79.07	77.89	78.48	85.62
fgenesh	1	97.87	86.56	92.22	84.73	73.29	79.01	85.61
augustus_v2	1	96.43	88.73	92.58	83.15	74.10	78.62	85.60
augustus_v1	1	96.64	88.28	92.46	84.60	72.32	78.46	85.46
eugene_v1	3	96.88	84.79	90.84	87.06	72.00	79.53	85.18
glimmerhmm	1	97.05	87.11	92.08	82.88	71.14	77.01	84.55
nscan	2	97.08	87.51	92.30	82.06	70.60	76.33	84.31
eugene	2	96.00	86.94	91.47	81.34	72.58	76.96	84.22
exonhunter_v1	3	97.17	86.77	91.97	82.49	69.10	75.79	83.88
eugene	1	93.71	88.92	91.31	78.87	72.72	75.80	83.55
maker_v2	3	91.09	90.55	90.82	78.44	69.36	73.90	82.36
sgp2	2	92.95	89.37	91.16	76.10	70.00	73.05	82.11
genemarkhmm	1	98.03	82.52	90.28	82.06	65.32	73.69	81.98
exonhunter_v2	3	97.59	83.47	90.53	83.34	61.62	72.48	81.50
geneid	1	93.40	87.65	90.53	75.81	68.36	72.09	81.31
maker_v1	3	91.70	87.97	89.84	78.70	66.10	72.40	81.12
exonhunter	1	95.21	85.48	90.34	71.76	62.30	67.03	78.69
snap	1	93.67	83.92	88.80	73.11	61.05	67.08	77.94
agene	1	93.52	82.86	88.19	68.01	60.86	64.44	76.31

Table 6.3: The nGASP gene prediction results for all gene prediction programs in categories 1-3. *Cat* stands for the category of the the gene prediction program. *NSn* denotes the nucleotide sensitivity, *NSp* the nucleotide specificity, *NAvg* the average of *NSn* and *NSp*, *ExSn* the exon sensitivity, *ExSp* the exon specificity, *ExAvg* the average of ExSn and ExSp, and *Average* the average of NSn, NSp, ExSn, and ExSp.

BSSM parameter	NSn	NSp	NAvg	ExSn	ExSp	ExAvg	Average
gth_bssm_testset	94.67	95.86	95.27	90.36	**79.80**	**85.08**	**90.17**
gth_species	**94.70**	**95.90**	**95.30**	90.26	79.78	85.02	90.16
gth_bssm	**94.70**	95.83	95.27	**90.39**	79.67	85.03	90.15
gth_generic	94.66	95.62	95.14	90.26	76.76	83.51	89.33

Table 6.4: BSSM comparison.

- For *gth_species* the nematode BSSM distributed with *GenomeThreader* was used (option `-species nematode`).
- For *gth_bssm_testset* the nGASP test set was used to create the custom BSSM instead of the training set.
- For *gth_generic*, the generic splice site model of *GenomeThreader* was used (that is, the option `-bssm` was removed).

The results are shown in Table 6.4. As one can see, the three runs where a BSSM was used have quite similar results. *gth_bssm_testset* has the best results, because it was trained with the test data. *gth_species* gives slightly better results than *gth_bssm*, probably because it is more generic (that is, not so biased for the training set as *gth_bssm*).

The results for *gth_generic* are worse than the other three runs with BSSMs. Especially the exon specificity suffers. This is an expected effect, because without a BSSM it is "harder" for the spliced alignment algorithm to predict the exact exon-intron boundaries. This shows that BSSMs clearly improve prediction accuracy and it is therefore worthwhile for the user to take the time to create them, if he wants to achieve good prediction results.

6.1.5 Intron Cutout Technique and Jump Table Prediction Accuracy

It was tested how the intron cutout technique (described in Section 4.5) and the jump table (described in Section 4.6) impact the prediction accuracy on the nGASP dataset.

For this purpose, *GenomeThreader* was called with the options described in Appendix H.5 (*gth_bssm*). For *gth_bssm_ic* the option `-introncutout` was added to activate the intron cutout technique. For *gth_bssm_fast* the option `-fastdp` was added to activate the jump table. For *gth_bssm_fast_ic* both options `-introncutout` and `-fastdp` were used. The results are shown in Table 6.5.

As one can see, this heuristic still leads to good results and the sensitivity/specificity values differ not much, despite considerable speed improvements which can be seen in Table 6.6. Instead of being the best gene prediction program on the nGASP dataset (for the *gth_bssm* parameter set), the other parameter sets (*gth_bssm_ic*, *gth_bssm_fast*, and *gth_bssm_fast_ic*) land in fourth.

Parameter Set	NSn	NSp	NAvg	ExSn	ExSp	ExAvg	Average
gth_bssm	**94.70**	**95.83**	**95.27**	**90.39**	**79.67**	**85.03**	**90.15**
gth_bssm_ic	94.28	95.81	95.05	89.47	79.10	84.29	89.67
gth_bssm_fast	94.50	95.78	95.14	89.99	78.08	84.04	89.59
gth_bssm_fast_ic	94.08	95.76	94.92	89.14	77.53	83.34	89.13

Table 6.5: Intron cutout technique and jump table prediction accuracy on the nGASP dataset.

Program	Time	Speedup
gth_bssm	2188s	—
gth_bssm_ic	1772s	$\approx 19\%$
gth_bssm_fast	1947s	$\approx 11\%$
gth_bssm_fast_ic	1440s	$\approx 34\%$

Table 6.6: Intron cutout technique and jump table running times on the nGASP dataset. Appendix K gives an overview of the used hardware and software setup.

Because the introns in the nGASP dataset are relatively short, the speed improvements are much smaller than what was measured on organisms with larger introns.

6.1.6 Chain Enrichment

It was tested how the chain enrichment (described in Section 4.5.4) impact the prediction accuracy on the nGASP dataset.

For this purpose, *GenomeThreader* was called with the options described in Appendix H.5 (*gth_bssm*). For *gth_bssm_ec* the option `-enrichchains` was added to activate the chain enrichment. For *gth_bssm_ic* the option `-intron cutout` was added to activate the intron cutout technique. For *gth_bssm_ec_ic* both options `-enrichchains` and `-intron cutout` were used. The results are shown in Table 6.7.

As one can see, the chain enrichment does not improve the results significantly on the nGASP dataset (in contrast to the chain enrichment on the ENCODE dataset described below in Section 6.3.3). This is due to the fact that the introns in the nGASP dataset are relatively short (in contrast to the introns contained in the ENCODE dataset). The running times of the runs with and without chain enrichment do not differ significantly.

6.1.7 Comparison with *GMAP*

GMAP (see Section 3.7.1) was also tested on the nGASP dataset and it performs rather poorly on this dataset (results with default parameter were named *gmap*, results were option `-n 1` was

Parameter Set	NSn	NSp	NAvg	ExSn	ExSp	ExAvg	Average
gth_bssm	**94.70**	95.83	**95.27**	**90.39**	79.67	85.03	**90.15**
gth_bssm_ec	**94.70**	**95.84**	**95.27**	**90.39**	**79.68**	**85.04**	**90.15**
gth_bssm_ic	94.28	95.81	95.05	89.47	79.10	84.29	89.67
gth_bssm_ec_ic	94.28	95.86	95.07	89.47	79.27	84.37	89.72

Table 6.7: Chain enrichment results on nGASP dataset.

Parameter Set	NSn	NSp	NAvg	ExSn	ExSp	ExAvg	Average
gth_bssm	**94.70**	95.83	**95.27**	**90.39**	79.67	**85.03**	**90.15**
gth_cdna	92.19	94.58	93.39	86.54	69.42	77.98	85.68
gth_prot	49.88	**98.24**	74.06	44.57	**88.09**	66.33	70.20
gmap_n1	92.32	78.65	85.49	66.59	11.24	38.92	62.20
gmap	92.39	78.13	85.26	66.69	11.09	38.89	62.08

Table 6.8: *GMAP* nGASP results.

used *gmap_n1*). Namely, *GMAP* performed worse than *all* the other gene finders which were tested. This is probably due to the fact that *GMAP* does not have a consensus phase built in. To have a fair comparison, *GenomeThreader* was used with the cDNAs only (*gth_cdna*) which puts *GenomeThreader* still in middle of all gene finders tested on the nGASP dataset. The run *gth_prot* shows the *GenomeThreader* results with proteins only. The specificity is rather high (which is to be expected with protein sequences), but the low sensitivity suggests that the protein dataset does not contain enough sequences to perform a successful gene prediction with them alone. *GMAP* is about twice as fast as *GenomeThreader* on the cDNA set (see running times in Table 6.9).

Program	Time
gth_bssm	2188s
gth_cdna	1384s
gth_prot	705s
gmap	752s
gmap_n1	732s

Table 6.9: Running times on nGASP dataset for *GenomeThreader* and *GMAP*.

x-fold	cov. ref1	cov. ref2	NSn	NSp	NAvg	ExSn	ExSp	ExAvg	Average
5	1.76	1.64	74.72	97.33	86.03	59.12	66.89	63.01	74.52
10	3.55	3.29	90.34	96.97	93.66	76.70	76.67	76.69	85.17
25	8.80	8.23	96.83	96.07	96.45	87.43	78.83	83.13	89.79
50	17.63	16.45	98.27	95.44	96.86	90.78	75.78	83.28	90.07
100	35.29	32.95	98.41	94.95	96.68	92.76	68.88	80.82	88.75
200	70.43	65.80	98.99	94.42	96.71	95.00	61.92	78.46	87.58

Table 6.10: *GenomeThreader* Mapping 454 sequences results for different numbers of reads (x-fold for $x \in \{5, 10, 25, 50, 100, 200\}$). The different number of reads result in the shown coverages for the gene reference sets **ref1** and **ref2**. *NSn* denotes the nucleotide sensitivity, *NSp* the nucleotide specificity, *NAvg* the average of *NSn* and *NSp*, *ExSn* the exon sensitivity, *ExSp* the exon specificity, *ExAvg* the average of ExSn and ExSp, and *Average* the average of NSn, NSp, ExSn, and ExSp.

x-fold	cov. ref1	cov. ref2	NSn	NSp	NAvg	ExSn	ExSp	ExAvg	Average
5	1.76	1.64	78.32	92.17	85.25	42.66	11.78	27.22	56.23
10	3.55	3.29	92.92	89.81	91.37	60.17	9.22	34.70	63.03
25	8.80	8.23	99.05	85.33	92.19	76.93	5.54	41.24	66.71
50	17.63	16.45	99.63	81.62	90.63	84.40	3.38	43.89	67.26
100	35.29	32.95	99.75	78.10	88.93	87.85	1.96	44.91	66.92
200	70.43	65.80	99.77	75.52	87.65	89.80	1.13	45.47	66.56

Table 6.11: *GMAP* Mapping 454 sequences results for different numbers of reads (*GMAP* was used with option `-n 1`). The column names have the same meaning as in Figure 6.10.

6.2 Mapping 454 Sequences

Next-generation sequencing (NGS) methods have become an important source of transcriptomic data (see Section 2.8 for an overview of NGS methods). To show that *GenomeThreader* is also suited to predict genes with transcriptomic sequences from NGS methods, we simulated 454 reads on the nGASP dataset and aligned them with *GenomeThreader* as follows:

1. Extract mRNA sequences from the **ref1** and **ref2** gene reference sets with the `extractfeat` tool (see Section B.6).
2. Simulate 454 reads with *Flowsim* [BML+10] with different numbers of reads (x-fold for $x \in \{5, 10, 25, 50, 100, 200\}$).
3. Align simulated 454 reads with *GenomeThreader*.

The results are shown in Table 6.10, at which the sensitivity values have been computed with the alignment of ref1 reads and the specificity values with the alignment of ref2 reads. A step-by-step outline on how this was done is given in Appendix I.

As was already mentioned in Section 4.8.11, *GenomeThreader* was used successfully to align 454 RNA-Seq reads in the tomato annotation project [Fil10], which shows that *GenomeThreader* also works well on real 454 reads.

To compare the performance on 454 reads, *GMAP* was also run on the simulated 454 reads (with option `-n 1`). The results are given in Table 6.11. Figure 6.1 compares the sensitivity and specificity results on the nucleotide and exon level between *GenomeThreader* and *GMAP* for different x-fold values. As one can see, the nucleotide sensitivity for *GMAP* is a little bit better than for *GenomeThreader*, but *GenomeThreader* is better for the exon sensitivity and much better for the nucleotide and exon specificity. This is probably due to the fact that *GMAP* does not contain a consensus phase and therefore produces many incomplete gene structure, because the 454 reads usually do not cover the complete gene. In contrast, *GenomeThreader* is able to successfully reconstruct the genes in the consensus phase (which is reflected in the much higher specificity values). The decreasing specificity for large read numbers is due to the fact that wrong alignments become more likely with larger number of reads. On this dataset *GenomeThreader* is about 10% faster than *GMAP*.

6.3 ENCODE Evaluation

EGASP, the human ENCODE Genome Annotation Assessment Project [GFA+06] was a community experiment to assess the state-of-the-art in genome annotation within the ENCODE regions. The ENCODE regions span 1% of the human genome and a reference annotation exists which was generated as part of the related GENCODE project. EGASP followed closely the model of its predecessor GASP1 [RHH+00].

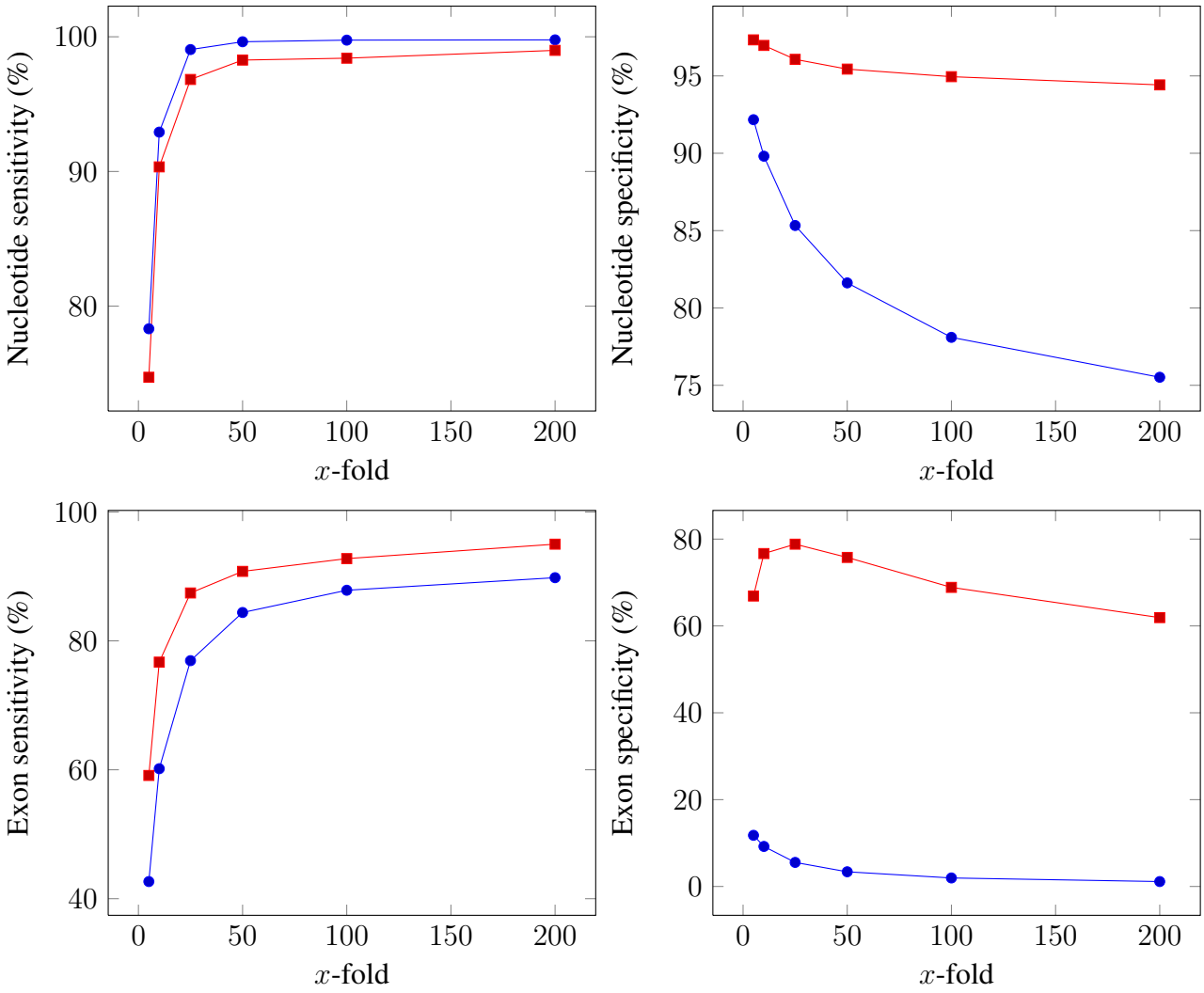


Figure 6.1: Result comparison for 454 sequence mapping. *GenomeThreader* results are shown in red and *GMAP* results in blue.

6.3.1 The ENCODE Dataset

The ENCODE dataset is comprised of the following components:

- The 44 ENCODE regions with a total length of ≈ 30 million base pairs. 33 of the regions have a size of 500.000 base pairs and the remaining 11 regions have a size from 600.000 up to 1.9 million base pairs.
- The GENCODE annotation gives all protein-coding genes in the 44 ENCODE regions (40 regions contain protein-coding genes). In total, 1332 protein-coding genes are annotated which are comprised of 10938 exons (10072 of those are actually translated). The longest annotated gene spans 321157 base pairs.

6.3.2 Comparing *GenomeThreader* with *GMAP*

To test *GenomeThreader* on the ENCODE dataset and compare it against *GMAP*, the following evaluation was performed:

1. Preparing the GENCODE annotation files for later processing.
2. Extracting the mRNA sequences annotated in the GENCODE files and mutate them with different mutation rates (with the `seqmutate` tool, documented in Section B.17).
3. Aligning the resulting mRNA sequences for different mutation rates against the ENCODE regions with *GenomeThreader* and *GMAP*.

This procedure shows how good *GenomeThreader* and *GMAP* are capable to reproduce an annotation when the complete mRNA is given (for mutation rate 0) and how good they perform under sequencing errors and when non-homologous mRNAs are given (for higher mutation rates). Appendix J describes the procedure in more detail to make it reproducible.

The result names used in the tables were contructed as follows: The prefix *gth* was used for *GenomeThreader* and the prefix *gmap* for *GMAP* results. After that the corresponding mutation rate was denoted in the form *r*x, where x is the mutation rate. If the option `-n 1` was used with *GMAP*, the suffix *n_1* was added. If the option `-enrichchains` was used with *GenomeThreader*, the string *ec* was added, for option `-intronicutout` the string *ic*, for a match length of 12 the string *m12* and for a match length of 15 the string *m15*.

Table 6.15 and Table 6.16 show the results for *GMAP* on this dataset (see Figure 6.2 for a graphical comparison). On average, *GMAP* performs better with option `-n 1` on this dataset. Therefore, the *GenomeThreader* results were compared against these *GMAP* results.

The results of *GenomeThreader* and *GMAP* on the ENCODE dataset are very similar. Table 6.12 shows the best results for *GenomeThreader* (for match length 12) and Table 6.15 for *GMAP*.

Parameter Set	NSn	NSp	NAvg	ExSn	ExSp	ExAvg	Average
gth_r0_ec_ic_m12	99.18	99.84	99.51	97.82	99.46	98.64	99.08
gth_r1_ec_ic_m12	99.18	99.84	99.51	97.59	98.85	98.22	98.87
gth_r3_ec_ic_m12	98.99	99.82	99.41	96.72	98.01	97.37	98.39
gth_r5_ec_ic_m12	97.75	99.77	98.76	95.29	96.78	96.04	97.40
gth_r10_ec_ic_m12	95.02	99.62	97.32	90.35	93.92	92.14	94.73

Table 6.12: *GenomeThreader* ENCODE results for match length 12. The average of the averages is 97.69.

Parameter Set	NSn	NSp	NAvg	ExSn	ExSp	ExAvg	Average
gth_r0_ec_ic_m15	99.21	99.83	99.52	97.77	99.30	98.54	99.03
gth_r1_ec_ic_m15	99.19	99.74	99.47	97.45	98.51	97.98	98.72
gth_r3_ec_ic_m15	99.20	99.63	99.42	96.40	97.28	96.84	98.13
gth_r5_ec_ic_m15	98.18	99.60	98.89	94.88	96.25	95.57	97.23
gth_r10_ec_ic_m15	94.59	99.24	96.92	86.96	91.81	89.39	93.15

Table 6.13: *GenomeThreader* ENCODE results for match length 15. The average of the averages is 97.25.

Parameter Set	NSn	NSp	NAvg	ExSn	ExSp	ExAvg	Average
gth_r0_ec_ic	99.21	99.82	99.52	97.54	99.14	98.34	98.93
gth_r1_ec_ic	99.19	99.71	99.45	97.10	98.17	97.64	98.54
gth_r3_ec_ic	99.19	99.54	99.37	95.68	96.65	96.17	97.77
gth_r5_ec_ic	98.22	99.38	98.80	93.75	94.92	94.34	96.57
gth_r10_ec_ic	89.52	98.70	94.11	75.77	88.90	82.34	88.22

Table 6.14: *GenomeThreader* ENCODE results with default match parameters. The average of the averages is 96.01.

Parameter Set	NSn	NSp	NAvg	ExSn	ExSp	ExAvg	Average
gmap_r0_n1	99.74	99.84	99.79	98.88	99.20	99.04	99.42
gmap_r1_n1	99.73	99.83	99.78	98.07	97.97	98.02	98.90
gmap_r3_n1	99.76	99.82	99.79	96.05	95.44	95.75	97.77
gmap_r5_n1	99.47	99.80	99.64	94.40	93.31	93.86	96.75
gmap_r10_n1	99.55	99.72	99.64	89.71	86.64	88.18	93.91

Table 6.15: *GMAP* ENCODE results (with option `-n 1`). The average of the averages is 97.35.

Parameter Set	NSn	NSp	NAvg	ExSn	ExSp	ExAvg	Average
gmap_r0	99.83	99.54	99.69	99.06	97.92	98.49	99.09
gmap_r1	99.82	99.55	99.69	98.26	96.87	97.57	98.63
gmap_r3	99.77	99.59	99.68	96.21	94.09	95.15	97.42
gmap_r5	99.76	99.43	99.60	94.72	91.98	93.35	96.47
gmap_r10	99.60	99.19	99.40	89.89	85.47	87.68	93.54

Table 6.16: *GMAP* ENCODE results. The average of the averages is 97.03.

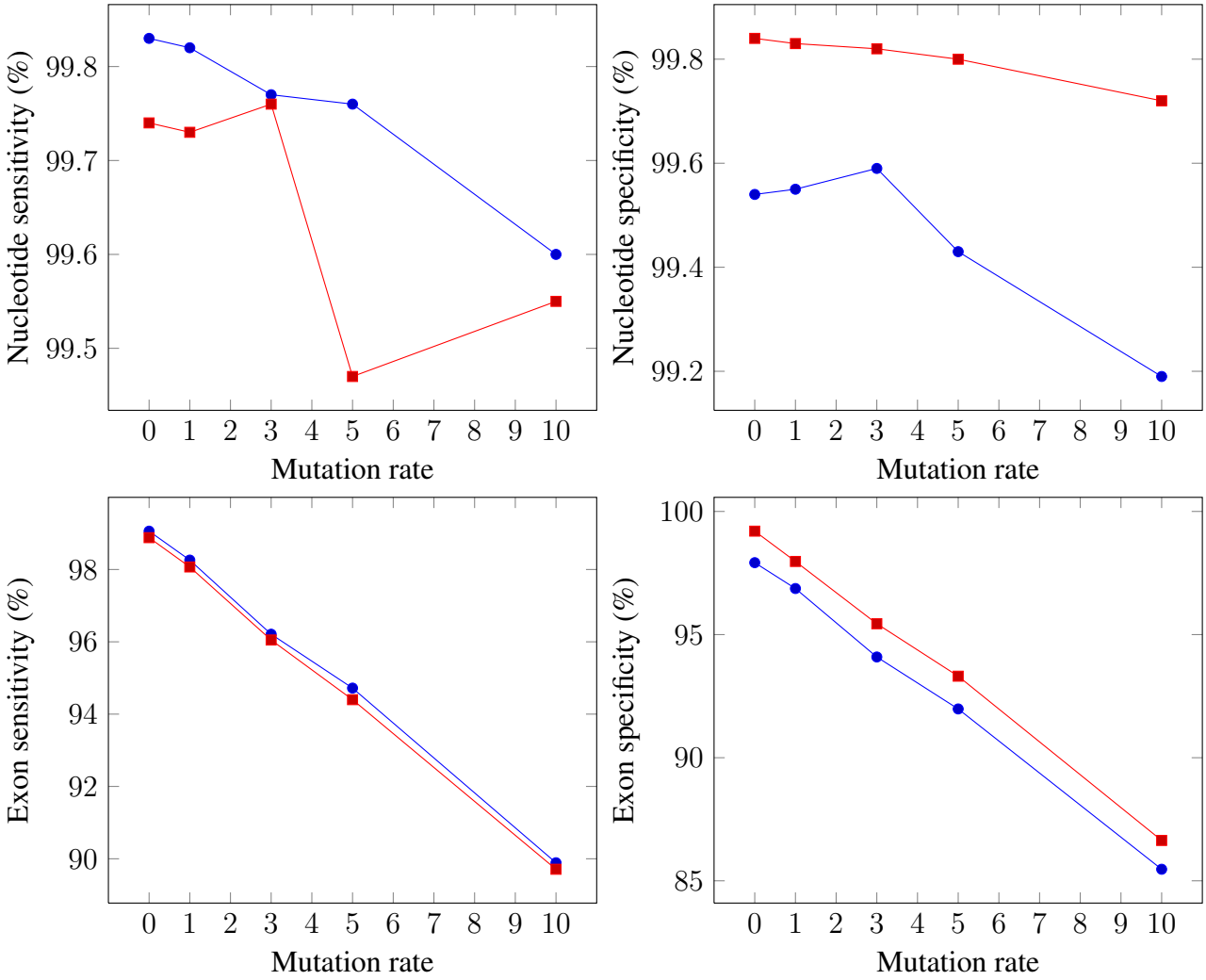


Figure 6.2: ENCODE result comparison between *GMAP* used with option `-n 1` (in red) and without (in blue).

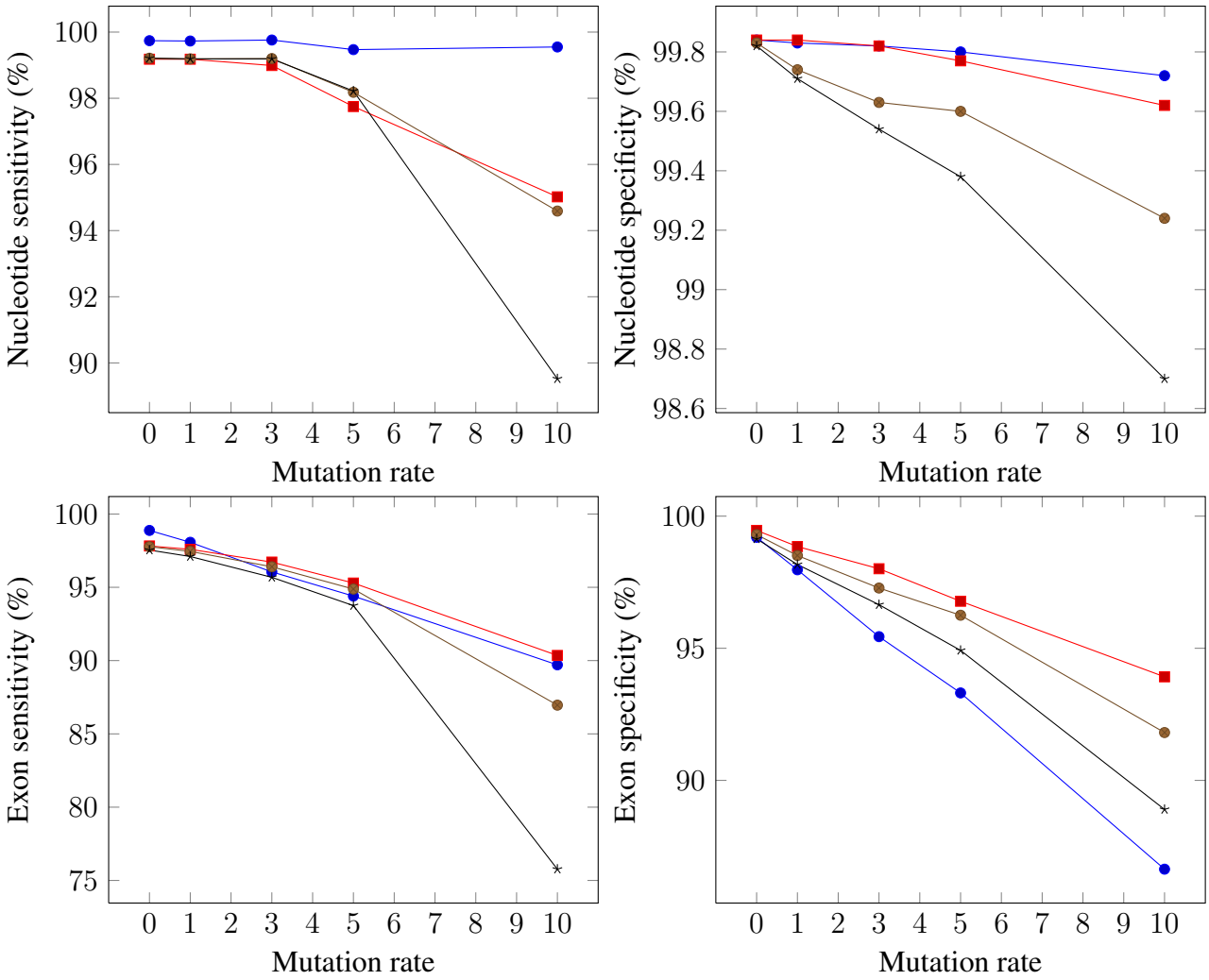


Figure 6.3: ENCODE result comparison between *GenomeThreader* for different matching parameter and *GMAP* with option `-n 1` (in blue). *GenomeThreader* results for match length 12 are shown in red, for match length 15 in brown, and for default matching parameter in black.

Parameter Set	NSn	NSp	NAvg	ExSn	ExSp	ExAvg	Average
gth_r0_ic_m15	99.29	99.68	99.49	97.02	98.28	97.65	98.57
gth_r1_ic_m15	99.26	99.59	99.43	96.72	97.55	97.14	98.28
gth_r3_ic_m15	99.22	99.47	99.35	95.75	96.24	96.00	97.67
gth_r5_ic_m15	97.59	99.42	98.51	94.23	95.12	94.68	96.59
gth_r10_ic_m15	91.89	98.70	95.30	85.74	90.37	88.06	91.68

Table 6.17: *GenomeThreader* ENCODE results without chain enrichment (for match length 15). The average of the averages is 96.56.

The average of the averages for the five different mutation rates is slightly higher for *GenomeThreader* (97.69 vs. 97.35). *GMAP* is better in terms of exon sensitivity while *GenomeThreader* is better in terms of exon specificity. The *GenomeThreader* results for a match length of 15 and for default matching parameter are shown in Table 6.13 and Table 6.14, respectively. Figure 6.3 shows a graphical comparison of this results.

6.3.3 Chain Enrichment Improves Prediction Results

The chain enrichment (described in Section 4.5.4) clearly improves prediction results on the ENCODE dataset. Table 6.17 shows the results for the same parameters as in Table 6.13, except that the chain enrichment was disabled. As one can see, the chain enrichment improves all results significantly: The exon sensitivity *and* the exon specificity is better for all mutation rates and the average of the averages for the different mutation rates is 0.69% better with chain enrichment than without. Figure 6.4 compares the results graphically. From Table 6.18 one can see that this comes at a price of about 50% increased runtime, because of the chain enrichment fewer parts of the dynamic programming matrix can be cutout in the intron cutout step (see Section 4.5.4 for algorithmical details).

This result shows that the chain enrichment improves prediction results on datasets with long introns (in contrast to datasets with relatively short introns, see example given in Section 6.1.6).

6.3.4 Influence of Match Size on Prediction Accuracy

The Tables 6.12, 6.13 and 6.14 and Figure 6.3 show that small matches improve the prediction accuracy. With shorter matches it is less likely that exons are "missed" in the dynamic programming. This comes at a cost in terms of increased runtimes (see Table 6.18 and Figure 6.5), especially for a match length of 12. If the match length is shorter, more matches are computed in the matching phase and have to be chained in the chaining phase. *GMAP* is much faster on this dataset, because it contains only cDNAs which match to the genomic region and the actual DP computation is faster in *GMAP*.

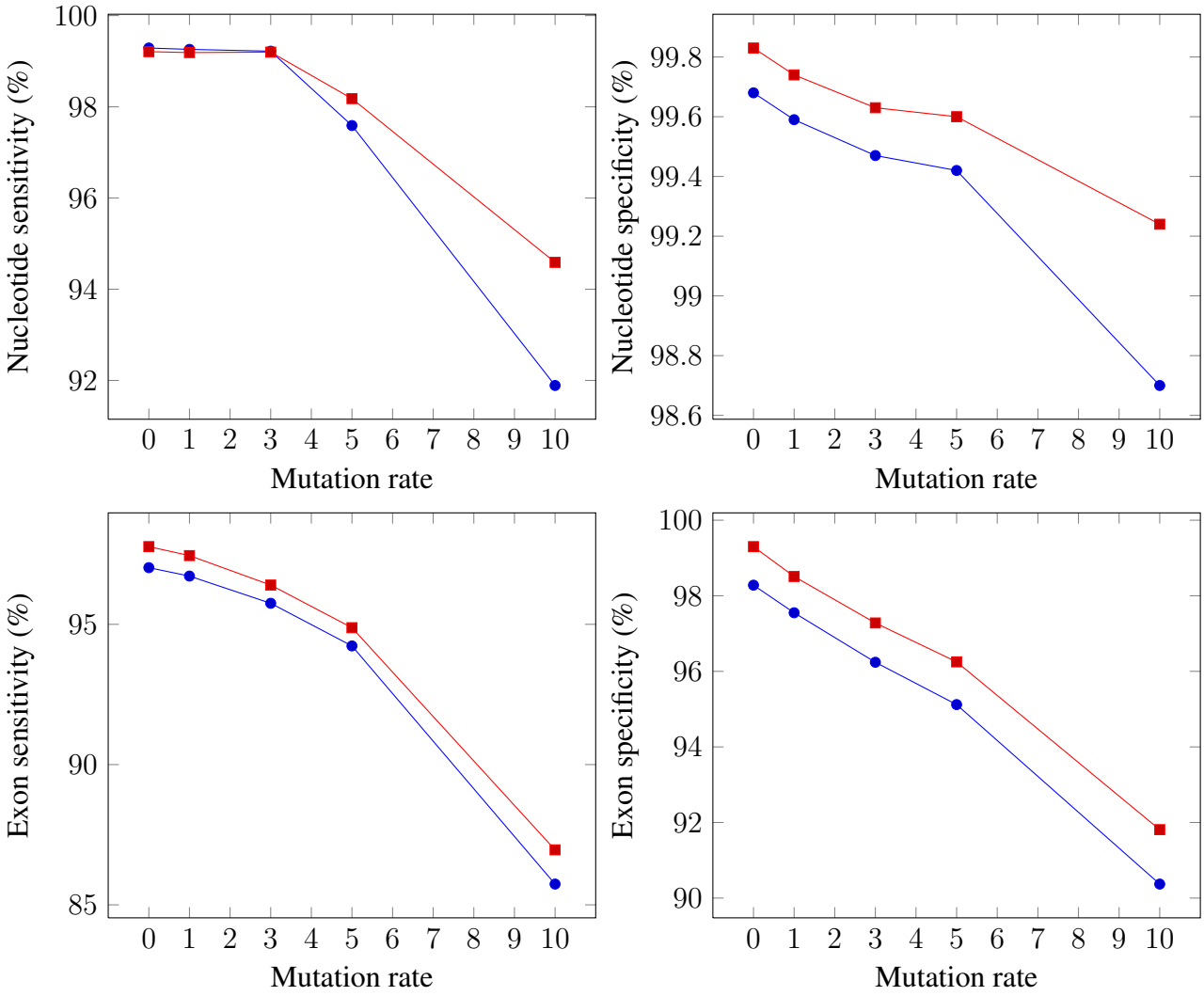


Figure 6.4: *GenomeThreader* ENCODE result comparison with chain enrichment (shown in red) and without (shown in blue). The runs were performed with a match length of 15.

Program	0	1	3	5	10
gth_ec_ic_m12	7858s	7700s	6957s	6884s	4975s
gth_ec_ic_m15	1535s	1565s	1521s	1674s	1308s
gth_ec_ic	1222s	1267s	1215s	1420s	1196s
gth_ic_m15	806s	835s	814s	832s	931s
gmap_n1	30s	51s	123s	138s	284s

Table 6.18: Runtimes on ENCODE Dataset for *GenomeThreader* and *GMAP* (for mutation rates 0, 1, 3, 5, and 10).

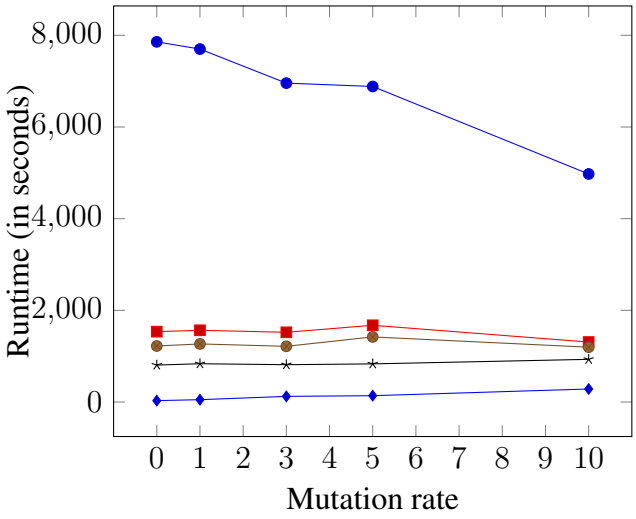


Figure 6.5: Runtimes on ENCODE dataset. *GenomeThreader* results for a match length of 12 are shown in blue (with circles), for a match length of 15 with chain enrichment in red, for default matching parameter in brown, and for a match length of 15 without chain enrichment in black. *GMAP* results are shown in blue (with diamonds).

6.4 General Discussion of *GenomeThreader* vs. *GMAP*

The conclusions for the three *GenomeThreader* vs. *GMAP* comparisons are as follows. *GenomeThreader* has a slower dynamic programming, but usually produces better prediction results, especially if multiple overlapping gene predictions have to be combined into a consensus. *GenomeThreader* has the better similarity filter (matching and chaining), which allow to efficiently cut out cDNA/EST sequences which do not match the genomic region. The following use case exemplifies this.

A 50 MB EST file from dbEST [BLT93] (which contains 148.385 ESTs) was aligned to human chromosome 21 (with masked repeats). On this task *GenomeThreader* was nearly 8 times as fast as *GMAP* (index construction times have not been considered for both programs, because they are usually performed only once for every genomic file). This is because only a fraction of the ESTs from the file are actually aligned to the human chromosome. The ESTs which show no similarities with the reference are discarded by the similarity filter in *GenomeThreader* before the DP. In contrast, *GMAP* tries to align all of them which leads to the higher runtime.

Chapter 7

Conclusion

This thesis presented the *GenomeThreader* gene structure prediction software, which was developed with adherence to strict software engineering principles, in the context of *seven* main contributions:

1. The *intron cutout technique*, which allows to predict gene structures stretching over large regions of a genome efficiently, was presented in Section 4.5. It speeds up the computation without effecting the prediction accuracy much (see Section 6.1.5).

2. The accompanying *chain enrichment* was described in Section 4.5.4 and it was shown that it significantly improves the prediction results on datasets with long introns (see Section 6.3.3).

3. *Jump tables* were explained in Section 4.6 and it was shown that they speed up the computation without a large effect on the accuracy of the prediction (see Section 6.1.5). They can be combined with the intron cutout technique, resulting in a compound speedup.

4. *Easy-to-use BSSMs* were introduced in Section 4.2 and it was experimentally confirmed that an organism specific BSSM improves the prediction compared to a generic splice site model (see Section 6.1.4).

5. The concept of *incremental updates* was given in Section 4.8.8. It allows to grow the used EST/protein collection without the need to recompute all spliced alignments. This has been proven to be very useful in practice. For example, incremental updates are used in PlantGDB [DFM+08].

6. The *combination of both cDNA/EST-based spliced alignments **and** protein-based spliced alignments into consensus spliced alignments* was described in Section 4.7. To our knowledge, *GenomeThreader* is the only gene prediction program with this feature. This is also used in practice, for example in PlantGDB.

7. The *GenomeTools* genome analysis system was described in Chapter 5. It allows to process genome annotations in a fast an flexible manner. For example, this capability is demonstrated in the workflow described in Appendix H which was used for the nGASP evaluation in Section 6.1.

GenomeThreader implements several datatypes in 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.

7.1 Future Developments

With several years of development time, *GenomeThreader* has become a robust and well-tested software tool which is widely used in practice (see Section 4.8.11 for a list of practical applications). However, as software is never finished, there are still several aspects of the software to improve:

- 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 [SK03], where a different scoring function is used.
- *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 EST and protein sequences is very important to balance the resource requirements and quality of the gene structure predictions. Furthermore, BSSMs (see Section 4.2) for a wider collection of organisms could be provided with the software.
- The single-threaded *GenomeThreader* binary can easily be parallelized on the command line by splitting up the input dataset for the spliced alignment computation (which is computationally the most expensive) and combining the results in the consensus spliced alignment phase. Although this approach is successfully used in practice, it would be convenient to parallelize *GenomeThreader* in such a way that it is possible to employ the ever more common multi-core CPUs without manually splitting the input dataset. This could be achieved by computing the spliced alignments in parallel via multi-threading. The spliced alignment computation itself could also be accelerated by leveraging highly-parallel GPUs (graphical processing units) or employing acceleration heuristics similar to the ones which are successfully used in HMMER3 [Edd11].

Despite this possible improvements we believe that *GenomeThreader* is a well integrated and easy-to-use gene prediction software package which delivers good prediction results in practice and that the algorithms, software architecture, and evaluation results which have been presented in this thesis successfully back up this claim.

Appendix A

Manual of *GenomeThreader*

A.1 Introduction

This manual describes how to use *GenomeThreader*, a software tool to compute gene structure predictions. The gene structure predictions are calculated using a similarity-based approach where additional cDNA/EST and/or protein sequences are used to predict gene structures via spliced alignments.

The algorithms, the phases, and the software engineering of *GenomeThreader*[1] are described in [GBSK05]. More details on the core dynamic programming (DP) algorithms used to computed spliced alignments via cDNAs/ESTs are given in [UZB00]. The DP algorithms used to compute spliced alignments via protein sequences are described in [UB00]. Here are the most important features of *GenomeThreader*.

Intron Cutout Technique

The core DP algorithms have been extended by the *intron cutout technique* [GBSK05] which allows to apply *GenomeThreader* to organisms with long introns. This overcomes the time and space limitations the algorithms described in [UZB00] and [UB00] have when applied to genomic sequences containing long introns.

Incremental Updates

With the help of *incremental updates* a lot of duplicated computations can be avoided, when the used cDNA/EST and/or protein databases have been updated. See Section A.4 for details.

[1] The name was suggested by Volker Brendel, because the method can be seen as a spliced "threading" of ESTs with a genomic template, allowing gene structure predictions to be computed.

Highly Parameterized

GenomeThreader is highly parameterized. That is, you can set many of the internal parameters via command line option to adjust the program to your personal gene prediction needs.

A.1.1 The Parts of *GenomeThreader*

When referring to *GenomeThreader* we mean the collection of gene prediction tools with this name. Whereas `gth` (in typewriter font) denotes the most important tool in this collection which computes the gene structure predictions. Besides `gth`, there are the following tools available:

1. `gthconsensus` computes consensus spliced alignments using intermediate files.
2. `gthsplit` splits intermediate files.
3. `gthgetseq` gets FASTA sequences from intermediate files.
4. `gthfilestat` show statistics of spliced alignments contained in intermediate files.
5. `gthbssmfileinfo` prints information about BSSM [2].
6. `gthbssmtrain` trains a BSSM.
7. `gthbssmbuild` builds a BSSM file.
8. `gthclean.sh` removes all indices.

A.1.2 Structure of the Manual

In Section A.3 it is shown how to use the various options of `gth` to perform gene predictions and in Section A.4 it is described how to use `gthconsensus` to process intermediate files produced by `gth`. In the following three sections some tools are explained which are helpful in handling intermediate files. In Section A.8 the small tool `gthbssmfileinfo` is introduced. In Section A.9 the BSSM training tool `gthbssmtrain` is documented. In Section A.10 the tool `gthbssmbuild` to build BSSMs is explained. In Section A.11 the shell script `gthclean.sh` is described. In Section A.12 it is explained how to construct the indices used by *GenomeThreader*. If you are new to *GenomeThreader* and want to use the program as fast as possible skip this sections in the first run and go directly to the tutorial given in Section A.13. At the end of the manual you can find the acknowledgments, the recent changes, the references, and the index.

[2] BSSM stands for *Bayesian Splice Site Model*

A.2 Installation

Extracting the *GenomeThreader* distribution for your platform gives you a directory named after your distribution. For your convenience you should extend your `PATH` variable by its `bin` subdirectory.

To be able to use all features of *GenomeThreader*, you have to set two environment variables, namely `BSSMDIR` and `GTHDATADIR`, pointing to the subdirectories `bssm` and `gthdata` of your distribution.

If one uses the `csh` or the `tcsh` shell, the definition of the environment variables could look like this:

```
$ setenv BSSMDIR "$HOME/gth-0.9.36-sparc-sun-solaris-64bit/bssm"
$ setenv GTHDATADIR "$HOME/gth-0.9.36-sparc-sun-solaris-64bit/gthdata"
```

For the `bash` or the `sh` the definitions could look like:

```
$ export BSSMDIR="$HOME/gth-0.9.36-sparc-sun-solaris-64bit/bssm"
$ export GTHDATADIR="$HOME/gth-0.9.36-sparc-sun-solaris-64bit/gthdata"
```

One can also specify more then one directory. In this case, they have to be separated by a colon in the according environment variable definition.

To disable file locking in *GenomeThreader* (not recommended), set the environment variable `GTHNOFLOCK` to any value. In `csh` or `tcsh`, this would look like this:

```
$ setenv GTHNOFLOCK "yes"
```

In `bash` or `sh` like this:

```
$ export GTHNOFLOCK="yes"
```

The `GTHNOFLOCK` environment variable should only be used, if one experiences problems with file locking. This may happen if a Network File System (NFS) is used.

A.3 `gth`: Computing Gene Predictions

`gth` is called as follows:

`gth` [*options*] `-genomic` *genseqfiles* `-cdna` *cdnafiles* `-protein` *proteinfiles*

Here *genseqfiles* denotes the input files containing the genomic template, *cdnafiles* denotes the input files containing cDNAs/ESTs sequences, and *proteinfiles* the input files containing protein sequences. `-cdna` and `-protein` do not have to be used simultaneously. For the input files indices

have to be constructed, either by `mkvtree` or automatically, as described in Section A.12. If an error occurs during the computation of *GenomeThreader*, the program exits with error code 1. Otherwise, the exit code is 0. All available *options* are explained below. They have been divided into different categories for clarity. An overview of the option categories with a short one-line description of each option is given in Table A.1.

Table A.1: Overview of the `gth`-Options Sorted by Categories

	Input Options
`-genomic`	specify input files containing genomic sequences
`-cdna`	specify input files containing cDNA/EST sequences
`-protein`	specify input files containing protein sequences
	Parameter Files
`-species`	specify species to select splice site model
`-bssm`	read bssm parameter from file
`-scorematrix`	read amino acid substitution scoring matrix from file
`-translationtable`	set the codon translation table
	Strand Direction
`-f`	analyze only forward strand of genomic sequences
`-r`	analyze only reverse strand of genomic sequences
	Genomic Sequence Positions
`-frompos`	analyze genomic sequence from this position
`-topos`	analyze genomic sequence to this position
`-width`	analyze only this width of genomic sequence
	Output
`-v`	be verbose
`-xmlout`	show output in XML format
`-gff3out`	show output in GFF3 format
`-md5ids`	show MD5 fingerprints as sequence IDs
`-o`	redirect output to specified file
`-gzip`	gzip compressed output file
`-bzip2`	bzip2 compressed output file
`-force`	force writing to output file
`-skipalignmentout`	skip output of spliced alignments
`-mincutoffs`	show full spliced alignments
`-showintronmaxlen`	set the maximum length of a fully shown intron
`-minorflen`	set the minimum length of an ORF to be shown
`-startcodon`	require than an ORF must begin with a start codon
`-finalstopcodon`	require that the final ORF must end with a stop codon
`-showseqnums`	show sequence numbers in output
`-pglgentemplate`	show genomic template in PGL lines
`-gs2out`	output in old GeneSeqer2 format
	Data Preprocessing
`-maskpolyatails`	mask poly(A) tails in cDNA/EST files
`-proteinsmap`	specify smap file used for protein files
`-noautoindex`	do not create indices automatically
`-createindicesonly`	stop program flow after the indices have been created
`-skipindexcheck`	skip index check (in preprocessing phase)
	Similarity Filter
`-minmatchlen`	specify minimum match length (cDNA matching)
`-seedlength`	specify the seed length (cDNA matching)
`-exdrop`	specify the Xdrop value for edit distance
`-prminmatchlen`	specify minimum match length (protein matches)
`-prseedlength`	specify seed length (protein matching)
`-prhdist`	specify Hamming distance (protein matching)
`-online`	run the similarity filter online

-inverse	invert query and index in vmatch call
-exact	use exact matches
-gcmaxgapwidth	set the maximum gap width for global chains
-gcmincoverage	set the minimum coverage of global chains
-paralogs	compute paralogous genes (different chaining procedure)
-enrichchains	enrich genomic sequence part of global chains with additional matches
	Intron Cutout Technique
-intron cutout	enable the intron cutout technique
-fastdp	use jump table to increase speed of DP calculation
-autointroncutout	set the automatic intron cutout matrix size
-icinitialdelta	set the initial delta used for intron cutouts
-iciterations	set the number of intron cutout iterations
-icdeltaincrease	set the delta increase during every iteration
-icminremintronlen	set the minimum remaining intron length
	U12-type Intron Model
-nou12intronmodel	disable the U12-type intron model
-u12donorprob	set the probability for perfect U12-type donor
-u12donorprob1mism	set the prob. for U12-type donor w. 1 mismatch
	Basic DP Algorithm
-probies	set the initial exon state probability
-probdelgen	set the genomic sequence deletion probability
-identityweight	set the pairs of identical characters weight
-mismatchweight	set the weight for mismatching characters
-undetcharweight	set the weight for undetermined characters
-deletionweight	set the weight for deletions
	Short Exon/Intron Parameters
-dpminexonlen	set the minimum exon length for the DP
-dpminintronlen	set the minimum intron length for the DP
-shortexonpenal	set the short exon penalty
-shortintronpenal	set the short intron penalty
	Special Parameters DP Algorithm
-wzerotransition	set the zero transition weights window size
-wdecreasedoutput	set the decreased output weights window size
	Processing of "raw" spliced alignments
-leadcutoffsmode	set the cutoffs mode for leading bases
-termcutoffsmode	set the cutoffs mode for terminal bases
-cutoffsminexonlen	set the cutoffs minimum exon length
-scoreminexonlen	set the score minimum exon length
	Advanced Similarity Filter Option
-minaveragessp	set the minimum average splice site prob.
	Spliced Alignment Filter
-minalignmentscore	set the minimum alignment score
-maxalignmentscore	set the maximum alignment score
-mincoverage	set the minimum coverage
-maxcoverage	set the maximum coverage
-intermediate	stop after calc. of spliced alignments
-sortags	sort alternative gene structures
-sortagswf	set the weight factor for the sorting of AGSs
-exondistri	show the exon length distribution
-introndistri	show the intron length distribution
-refseqcovdistri	show the reference sequence coverage distribution
-first	set the maximum number of spliced alignments
-help	show basic options and exit
-help+	show all options and exit
-version	display version information and exit

A.3.1 Input Options

-genomic *genseqfiles*

genseqfiles denotes the input files containing the genomic sequences, for which the gene prediction is to be computed.

`-cdna` *cdnafiles*

cdnafiles denotes the input files containing the cDNAs/ESTs which are spliced aligned to the genomic sequences.

`-protein` *proteinfiles*

proteinfiles denotes the input files containing the protein sequences which are spliced aligned to the genomic sequences.

The option `-genomic` is mandatory. Furthermore, at least one of the options `-cdna` and `-protein` has to be used.

The names of the input files are separated by white spaces. We support the following formats for the input files. They are recognized according to the first non-white space symbol in the file.

multiple FASTA format If the file begins with the symbol >, then this file is considered to be a file in multiple FASTA format (i.e. it contains one or more sequences). Each line starting with the symbol > contains the *description* of the sequence following it.

multiple EMBL/SWISSPROT format If the file begins with the string `ID`, then this file is considered to be a file in multiple EMBL format (i.e. containing one or more sequences, each in EMBL-format). The information contained in the `ID` and `DE`-lines is taken as the *description* of the corresponding sequence. The EMBL format is identical to the SWISSPROT format (w.r.t. the information we need to extract from such entries). So one can also use files in multiple SWISSPROT format as input.

multiple GENBANK format If the file begins with the string `LOCUS`, then this file is considered to be a file in multiple GENBANK format (i.e. containing one or more entries in GENBANK-format). The information contained in the `LOCUS` and the `DEFINITION`-lines is taken as the *description* of the corresponding sequence.

plain format If the file does not begin with the symbol > or the strings `ID` or `LOCUS`, then the file is taken verbatim. That is, the entire file is considered to be the input sequence (white spaces are *not* ignored).

A.3.2 Parameter File Options

If no BSSM parameter file is specified by one of the following two options, the generic splice site model is used. BSSM stands for *Bayesian Splice Site Model*.

`-species` *speciesname*

Use precomputed BSSM parameter file for species with name *speciesname*, according to Table A.3. `gth` searches for the file in the directory where it is run and in the directory specified by the environment variable `BSSMDIR`. How to set `BSSMDIR` is explained in Section A.2.

1	Standard
2	Vertebrate Mitochondrial
3	Yeast Mitochondrial
4	Mold Mitochondrial; Protozoan Mitochondrial; Coelenterate Mitochondrial; Mycoplasma; Spiroplasma
5	Invertebrate Mitochondrial
6	Ciliate Nuclear; Dasycladacean Nuclear; Hexamita Nuclear
9	Echinoderm Mitochondrial
10	Euplotid Nuclear
11	Bacterial
12	Alternative Yeast Nuclear
13	Ascidian Mitochondrial
14	Flatworm Mitochondrial
15	Blepharisma Macronuclear
16	Chlorophycean Mitochondrial
21	Trematode Mitochondrial
22	Scenedesmus Obliquus Mitochondrial
23	Thraustochytrium Mitochondrial

Table A.2: The possible codon translation table numbers and table names.

`-bssm` *paramfileprefix*

Load the BSSM parameter file *paramfileprefix*`.bssm`. If *paramfileprefix*`.bssm` is not in the current directory, the directories given by `BSSMDIR` are searched for *paramfile*`.bssm`.

`-scorematrix` *scorematrixname*

Use the amino acid substitution matrix given in the file *scorematrixname* for spliced alignments with protein sequences. The default score matrix file is `BLOSUM62`. `gth` searches for the file *scorematrixname* in the directory where it is run and in the directory specified by the environment variable `GTHDATADIR`. How one can set `GTHDATADIR` is explained in Section A.2.

`-translationtable` t

Set the codon translation table t used in the matching, DP, and output phase. t must be a number in the range $[1, 23]$ except for 7, 8, 17, 18, 19 and 20. Table A.2 gives the possible numbers and their names. The codon translation tables were taken from the website `ftp://ftp.ncbi.nih.gov/entrez/misc/data/gc.prt`.

Option `-species` and option `-bssm` exclude each other. If two excluding options are used together, an error is thrown.

All BSSM parameter files have the extension `.bssm`.

It is highly recommended to use an adequate BSSM parameter file via the option `-species` or `-bssm`, if one is available. Because it is most likely that using a BSSM parameter file for the

#	Species	Filename	Class	GT	GC	AG
1	*Homo sapiens*	`human`	2	$\surd$	–	$\surd$
2	*Mus musculus*	`mouse`	2	$\surd$	–	$\surd$
3	*Rattus norvegicus*	`rat`	7	$\surd$	–	$\surd$
4	*Gallus gallus*	`chicken`	7	$\surd$	–	$\surd$
5	*Drosophila*	`drosophila`	7	$\surd$	–	$\surd$
6	*Caenorhabditis elegans*	`nematode`	7	$\surd$	–	$\surd$
7	*Schizosaccharomyces pombe*	`fission_yeast`	7	$\surd$	–	$\surd$
8	*Aspergillus*	`aspergillus`	7	$\surd$	–	$\surd$
9	*Arabidopsis thaliana*	`arabidopsis`	7	$\surd$	$\surd$	$\surd$
10	*Zea mays*	`maize`	7	$\surd$	$\surd$	$\surd$
11	*Medicago truncatula*	`medicago`	7	$\surd$	–	$\surd$
12	*Oryza sativa*	`rice`	7	$\surd$	$\surd$	$\surd$

Table A.3: Available BSSM parameter files: *GenomeThreader* comes with BSSM parameter files for twelve different species. All parameter files have the extension `.bssm`. *Class* denotes the Bayesian classification model: Either a Bayesian two classification model or a Bayesian seven classification model was used for calculation of the parameter (as described in [BXZ04]). A $\surd$ in the GT, GC, and AG columns means that the corresponding BSSM file contains a model for GT donor, GC donor, or AG acceptor sites, respectively.

species under consideration (or a cognate one) will yield in better gene predictions. If no such file is available, feel free to contact us.

If possible, one should prefer `-species` over `-bssm`, because in this case additional internal parameters can be adjusted for the given species.

A.3.3 Strand Direction Options

`-f`
: Analyze only the forward strand of the genomic sequences.

`-r`
: Analyze only the reverse strand of the genomic sequences.

Option `-f` and option `-r` exclude each other. If neither `-f` nor `-r` is used, both strands of the genomic sequences are analyzed.

A.3.4 Genomic Sequence Positions Options

Positions in the genomic sequence begin with 1. The following options are available to specify a region of the genomic sequence for which the gene prediction should be computed. They are

only applicable if exactly one genomic sequence is given.

`-frompos` i
: Specify first position i of a genomic region to analyze. The parameter i must be a positive integer.

`-topos` j
: Analyze genomic sequence up to (and including) position j, whereas j must be a positive integer.

`-width` w
: Analyze only width w of genomic sequence, whereas w must be a positive integer.

If option `-topos` or `-width` are used, the option `-frompos` is required and the other way round, that is, `-topos` and `-width` exclude each other. If the parameters do not specify a substring of the genomic sequence, then an error is thrown.

Furthermore, if option `-frompos` is used the option `-inverse` is set automatically. This may lead to a large memory consumption which can be avoided by putting the desired part of the genomic sequence in a separate file and running `gth` without the options `-frompos` and `-inverse`.

A.3.5 Output Options

The following options concern the output of *GenomeThreader*:

`-v`
: Be verbose, that is, give reports about the different steps as well as the resource requirements of the computation.

`-xmlout`
: Shows the output in XML format. This can be useful, if one wants to postprocess the output files.

`-gff3out`
: Shows the output in GFF3 format. Either the spliced alignments (if option `-intermediate` is used) or the consensus spliced alignments (if option `-skipalignmentout` is used) are shown.

`-md5ids`
: Show MD5 fingerprints as sequence IDs. This makes subsequent mapping of the annotation to the actual sequences less error-prone.

`-o` *outputfile*
: Redirect output to specified file *outputfile*.

`-gzip`
: Compress the output file given by `-o` with `gzip`.

`-bzip2`
: Compress the output file given by `-o` with `bzip2`.

`-force`
: Forces writing to the output file *outputfile* given by `-o`. By default, writing to *outputfile* is only performed if the file does not exist already.

`-skipalignmentout`
: Skip the output of spliced alignments.

`-mincutoffs`
: The complete spliced alignments are shown. That is, on either side of the alignment only insertions and introns are cut off. This option has the same effect as using `-leadcutoffsmode MINIMAL -termcutoffsmode MINIMAL`.

`-showintronmaxlen` *maxintronlen*
: Sets the maximal length of an intron to be shown completely. If an intron is larger than *maxintronlen*, it is shown in an abbreviated form. Set to 0 to show all introns completely regardless of their lengths.

`-minorflength` *o*
: Sets the minimum length of an open reading frame to be shown. Thereby, o denotes the number of amino acids which must be an integer value greater 0. The default value is 64.

`-startcodon`
: Require than an ORF must begin with a start codon.

`-finalstopcodon`
: Require that the final ORF must end with a stop codon.

`-showseqnums`
: Show the sequence numbers in output. That is, in the output lines describing the sequences used in a spliced alignment an additional tag is added giving the number of the sequence in the corresponding sequence file. Sequences are numbered from 0 on. This may be useful when *GenomeThreader* output is postprocessed.

`-pglgentemplate`
: Show genomic template in PGL lines. The default is `yes`. Switch off with `-pglgentemplate no` for backward compatibility.

`-gs2out`

Output is shown in the format of the program *GeneSeqer2* , which is a predecessor of *GenomeThreader*. We do not recommend to use this option. It is available for compatibility. For example, all wildcards (S, Y, W, R, K, V, B, D, H and M) of the input sequences are replaced by the wildcard N.

If the options `-o` and `-v` are invoked together: The additional output produced by the option `-v` is not redirected to the file *outputfile* given by the option `-o`. This allows to save the results of the computation in a file while watching its progress on stdout.

A.3.6 Data Preprocessing

This options affect the preprocessing of the input data, for a detailed discussion see Section A.12. This is done by calling `mkvtree`, which is part of the *Vmatch* package (see `http://vmatch.de/`), internally.

`-maskpolyatails`
: When this option is used, all poly(A) tails and poly(T) heads in cDNA/EST reference sequences are masked automatically. To be able to predict gene structures correctly it is very important that poly(A) tails are masked!

`-proteinsmap` smapfile
: Specifies the smap file *smapfile* used for the (automatic) index construction of protein files. That is, internally `mkvtree` is called with option `-smap` *smapfile* (see the *Vmatch* manual for details). If this option is not used, `mkvtree` is called internally with its option `-protein`.

`-noautoindex`
: This option disables the automatic construction of the necessary indices. That is, it is assumed that they have been created manually beforehand using `mkvtree`. See Section A.12 for details.

  If you use `gth` with option `-createindicesonly` (instead of `mkvtree`) to construct the indices, use the option `-skipindexcheck` and not this option.

`-createindicesonly`
: This option stops the program flow after the indices have been created (that is, after the preprocessing phase). This is useful if one wants to let multiple instances of *GenomeThreader* run on the same data set simultaneously without interfering each other during the index construction.

`-skipindexcheck`
: This option skips the index checks. That is, it is not checked anymore if an index exists or is corrupted. This is useful if one lets multiple instances of *GenomeThreader* run on indices created with `-createindicesonly`. This option speeds up the preprocessing

phase, especially if one uses multiple indices over a network file system. To ensure the correct functioning of this option, one has to use the same type of input data (that is, using `-cdna` and/or `-protein` in a similar way) and use `-maskpolyatails`, `-online`, and `-inverse` similarly in both index construction and splice alignment computation. If you want to use this option together with `-frompos`, pass option `-inverse` to the corresponding `-createindicesonly` call.

The options `-maskpolyatails` and `-noautoindex` as well as `-proteinsmap` and `-noautoindex` exclude each other. Furthermore, the option `-createindicesonly` excludes using the option `-noautoindex` or `-skipindexcheck` (and the other way around).

A.3.7 Options of the Similarity Filter

The following options are used to compute the similarity regions, that is, the regions in the genomic sequence which are similar to the cDNAs/ESTs and/or proteins. This is done by calling *Vmatch* (`http://vmatch.de/`) internally.

Vmatch matches a sequence file called *query* against a persistent index named *subject* of another sequence file. When matching a cDNA/EST file against a genomic file, the default is to use the cDNAs/ESTs as query and the genomic sequences as subject. For proteins it is the other way round.

`-minmatchlen` ℓ
: Specify the length value ℓ for the initial matches used in the similarity filter for cDNA/EST matching. The default value is 20.

`-seedlength` m
: Set the length m of the exact seeds used for cDNA/EST matching. m must be a positive integer. The default value is 18.

`-exdrop` x
: Specify the $Xdrop$-score x when extending a seed in both directions allowing for matches, mismatches, insertions, and deletions. The argument x must be a positive integer smaller or equal to 255. Matches are scored 2, mismatches are scored -1, and indels are scored -2. The default value is 2. The extension procedure is further explained in the *Vmatch* manual which can be found at `http://vmatch.de/`. The minimum length of the seeds is specified by the argument to option `-seedlength`.

`-prminmatchlen` ℓ
: Specify the length value ℓ for the initial matches used in the similarity filter for protein matching. The default value is 24.

`-prseedlength` m
: Set the length m of the exact seeds used for protein matching. m must be a positive integer. The default value is 10.

`-prhdist` h

Set the maximum Hamming distance h a protein match is allowed to have. The default value is 4.

`-online`

Run online algorithms to compute initial matches in the similarity filter. In this case the complete index is not needed, except for the original and the transformed input sequences plus the descriptions. Therefore, this option is more space efficient. However, the online algorithms usually run not as fast as the indexed based algorithms.

`-inverse`

This option only affects cDNA/EST files. Invert subject and query, i.e., use the genomic files as query and the cDNA/EST files as subject.

`-exact`

Use exact matches in the similarity filter. If this option is invoked it is not possible to use the options `-seedlength` and `-exdrop`, because it would make no sense.

`-gcmaxgapwidth` *gw*

Set the maximum gap width *gw* for global chains. This width is approximately the same as the maximum intron length in the studies organism. The default is 1000000 which might be a good guess for human. It is very important to set this parameter appropriately!

`-gcmincoverage` *mc*

Set the minimum coverage a global chain must achieve to be kept. That is, at least this proportion of the according cDNA/EST/protein sequence must be covered by the global chain. Thereby, *mc* is an integer denoting the minimum coverage in percent. The default is 50. This option has a great influences on the number of computed spliced alignments!

`-paralogs`

By default, the chaining returns all global chains with maximal score (usually 1), if their coverage is higher than the argument *mc* to option `-gcmincoverage`. If this option is used, all non-overlapping global chains with a coverage higher than *mc* are reported. This can be helpful to detect paralogous genes. The total run time usually increases, because more spliced alignments need to be computed.

`-enrichchains`

Enrich genomic sequence part of global chains with additional matches.

Using option `-inverse` without using option `-online` can lead to a large main memory consumption.

A.3.8 Intron Cutout Technique Options

In this section the options used for the intron cutout technique are described.

`-intron cutout`
: Enable the intron cutout technique.

`-fastdp`
: Use jump table to increase speed of DP calculation.

`-autointroncutout` *s*
: Enables the automatic intron cutout technique. That is, the intron cutout technique is only used if the dynamic programming matrix in the "normal" DP call would be larger than *s* megabytes. Increasing *s* increases the computation time and decreases the probability of wrong gene predictions.

`-icinitialdelta` *d*
: Set the initial delta *d* used for intron cutouts. The parameter *d* must be a positive integer. The default value is 50.

`-iciterations` *i*
: Set the number of iterations *i* the initial delta *d* is increased by the increase delta *di*. Thereby, the first iteration counts, too. That is, in the default setting of 2 *d* is increased once by *di*, resulting in (up to) two iterations. The incrementation is only triggered if the intron cutout technique did not succeed.

`-icdeltaincrease` *di*
: Set the increment *di* for the delta used in the intron cutout technique.

`-icminremintronlen` *r*
: Set the minimum remaining intron length *r* for an intron to be cut out. The parameter *r* must be a positive integer. The default value is 10.

The options `-intron cutout` and `-autointroncutout` exclude each other.

A.3.9 Advanced Options

This section describes the advanced options. To understand their semantics, a basic understanding of the underlying spliced alignment algorithms is required. For the initial use of *GenomeThreader*, this section can safely be skipped.

Options for U12-type introns

GenomeThreader has a model for U12-type introns built in. That is, donor sites which match the consensus /[AG]TATCCTT (where / denotes the exon end and [AG] indicates A or G) [ZB03] perfectly or with one mismatch get a high probability. The mismatch is only allowed in the last 6 bases of the consensus. See [ZB03] for details on U12-type introns and additional pointers to the literature.

`-nou12intronmodel`
: If this option is used, the U12-type intron model is disabled.

`-u12donorprob` p
: Sets the probability p for perfectly matching U12-type donor sites. The default value is 0.99.

`-u12donorprob1mism` p
: Sets the probability p for U12-type donor sites with exactly one mismatch. The default value is 0.9.

In both cases, p must be a positive floating point value smaller or equal than 1.0.

Basic DP Algorithm Options

With these options all DP parameters can be changed.

`-probies` τ_{e_1}
: Sets the probability that the initial state is an exon state. τ_{e_1} must be a positive floating point value smaller or equal than 1.0. The default value is 0.5.

`-probdelgen` $P_{\Delta g}$
: Sets the probability of a nucleotide deletion in the genomic sequence. τ_{e_1} must be a positive floating point value smaller or equal than 1.0. The default value is 0.03.

`-identityweight` σ
: Sets the weight for pairs of identical characters. σ must be a floating point value. The default value is 2.0.

`-mismatchweight` μ
: Sets the weight for mismatching characters. μ must be a floating point value. The default value is -2.0.

`-undetcharweight` ν
: Sets the weight for alignment positions involving undetermined characters, i.e., involving character N. ν must be a floating point value. The default value is 0.0.

`-deletionweight` δ
: Sets the weight for deletions. δ must be a floating point value. The default value is -5.0.

Short Exon/Intron Parameters

The following options allow to change the parameters for short exons and introns. If an exon is shorter than the minimum exon length ξ, then a penalty for short exons χ is *subtracted* from the actual weight. Analog, if an intron is shorter than the minimum intron length η, then a penalty for short introns ψ is *subtracted* from the actual weight.

`-dpminexonlength` ξ
: Sets the minimum exon length. ξ must be an integer value greater 0. The default value is 5.

`-dpminintronlength` η
: Sets the minimum intron length. η must be an integer value greater 0. The default value is 50.

`-shortexonpenal` χ
: Sets the short exon penalty. χ must be a floating point value greater or equal 0.0. The default value is 100.0.

`-shortintronpenal` ψ
: Sets the short intron penalty. ψ must be a floating point value greater or equal 0.0. The default value is 100.0.

Special Parameters for the DP Algorithm

With these options, the more special DP Parameter can be changed.

`-wzerotransition` ϑ
: Sets the window size for zero transition weights. ϑ must be an integer value greater or equal to 0. The default value is 80.

`-wdecreasedoutput` ω
: Sets the window size for decreased output weights. ω must be an integer value greater or equal to 0. The default value is 80.

Options for Processing of "raw" Spliced Alignments

The following options affect the processing of "raw" spliced alignments after the dynamic programming.

`-leadcutoffsmode` *leadingmode*
: Set the mode for determination of the leading cutoffs. The mandatory argument *leadingmode* can be `RELAXED`, `STRICT`, or `MINIMAL`. The default is `RELAXED`.

`-termcutoffsmode` *terminalmode*
: Set the mode for determination of the terminal cutoffs. The mandatory argument *terminalmode* can be `RELAXED`, `STRICT`, or `MINIMAL`. The default is `STRICT`.

`-cutoffsminexonlen` κ
: Set the minimum length κ an exon must have, such that it is not cut off when the corresponding cut off mode is `STRICT`. The default is 5.

`-scoreminexonlen` λ
: Set the minimum length λ an exon must have, such that it is included in the computation of the overall similarity score of a spliced alignment. The default is 50.

Spliced Alignment Filter

The spliced alignment filter controls which spliced alignments are stored in the internal set of spliced alignment. That is, only these spliced alignments are shown or written to an output file and are used to compute consensus spliced alignments.

When a spliced alignment is computed, its *coverage* is also determined. The coverage of a spliced alignment is the maximum of the the *genomic coverage* and the *reference coverage*. Thereby, the genomic coverage denotes the cumulative exon length divided by the length of the genomic sequence part used for the dynamic programming. The reference coverage denotes the cumulative exon length divided by the length of the reference sequence. If the genomic coverage was maximal a `G` in shown in the output[3]. If the reference coverage was maximal a `C` (for cDNA/EST based spliced alignments) or `P` (for protein based spliced alignments) is shown in the output.

`-minalignmentscore` s
: A spliced alignment must at least an alignment score of s.

`-maxalignmentscore` s
: A spliced alignment is not allowed to have an alignment score higher than s.

`-mincoverage` c
: A spliced alignment must at least a coverage of c.

`-maxcoverage` c
: A spliced alignment is not allowed to have a coverage higher than c.

By default all computed spliced alignments are used.

[3] In the textual output as last part of the `MATCH` line and in the final XML output as the `high_type` attribute.

Advanced Similarity Filter Option

`-minaveragessp` ζ
: Sets the minimum average splice site probability. ζ must be a floating point value in the interval $[0.0, 1.0]$. The default value is 0.5.

`-duplicatecheck` mode
: Set the criterion used to check for spliced alignment duplicates. The mandatory argument *mode* can be `none`, `id`, `desc`, `seq`, or `both`. The default is `both`.

With the option `-duplicatecheck` the "sameness" criterion of the duplicate check is set. The duplicate check is designed to prevent duplicate alignments (that is, alignments from the "same" cDNA/EST or protein sequence to the same genomic region on the same strand of a particular genomic sequence). Duplicate alignments can occur for two reasons:

1. The chaining algorithm (which chains the matches before the actual spliced alignment computation) produced two overlapping chains in the same genomic region which leads to the same spliced alignments. This is a rather rare event, but it can happen from time to time. For this reason alone, a duplicate check is needed, if we don't want to see spliced alignment duplications.
2. The same cDNA/EST or protein sequence was fed to GenomeThreader more than once. The duplicate check should prevent that it appears in the output multiple times and the "sameness" criterion used to compare the sequences depends on the *mode* supplied to `-duplicatecheck`.

The different duplicate check modes work as follows:

`none`: No duplicate check is done whatsoever. Only useful for testing purposes.

`id`: The duplicate check is based on the "ID" of the cDNA/EST or protein sequence. The "ID" is parsed from the sequence description as follows: leading '`gi|`', '`SQ;`', '`(gi|`', '`ref|`' are dropped and the part until the first '`:`', '`|`', ' ' or '`\t`' is stored. This can cause problems if the description is empty or starts with a prefix that is not recognized. This was the behavior up to (and including) version 1.4.6.

`desc`: In contrast to the `ID` mode, the complete description is used to compare sequences. This can cause problem if the description is empty or ambiguous.

`seq`: The actual sequences is used to determine, if two cDNA/EST or protein sequences are the same. This would lead to the exclusion of cDNA/EST or protein sequences where the actual sequence equals but the descriptions differ (for example, equal ESTs sequenced independently from another and therefore with different descriptions).

`both`: This mode combines the `desc` and `seq` modes. That is, a cDNA/EST or protein sequences is only considered equal to another one, if the complete descriptions and the actual sequences are equal. This is the default (from version 1.4.7 onwards).

Interrupt Option

With the following option it is possible to perform the incremental updates which are described in Section A.4.

`-intermediate`
: If this option is invoked, the dataflow of `gth` is interrupted after the output of the spliced alignments.

This option implies option `-xmlout` since the intermediate results are stored in an XML format. You should save the intermediate results using option `-o` instead of redirecting them to a file, because this allows for an internal check which ensures that the intermediate output reflects the spliced alignments stored in main memory. Do not process the intermediate XML output yourself, use the "normal" XML output instead!

Options for Postprocessing of Predicted Gene Locations

With the following options it is possible to postprocess the *predicted gene locations* (PGLs), also referred to as *consensus spliced alignments*.

`-sortags`
: If this option is invoked, the alternative gene structures (AGSs) of every PGL are sorted according to the so-called *overall score*. Every AGS has an exon score for every contained exon and a donor and a acceptor site probability for every contained intron (if any). The overall score o of an AGS is computed as follows:

  - If the AGS consists of exactly one exon, the overall score simply equals the exon score.
  - Otherwise, the average exon score e and the average spliced site probability s is calculated. Then
    $$o = \frac{w \cdot e + s}{w + 1.0},$$
    whereas w is the *weight factor* (see option below).

`-sortagswf` *wf*
: Set the weight factor *wf* for the calculation of the overall score (see option above). *wf* must be a floating point value larger than 0.0. If *wf* is set to a value larger than 1.0, the average exon score gets more weight in the calculation of the overall score. If *wf* is set to a value smaller than 1.0, the average splice site probability gets more weight. The default value is 1.0.

If option `-sortagswf` is used, option `-sortags` is required.

Statistical Options

In this section the options are described which yield in the output of additional statistical information at the end of a program run.

`-exondistri`
: Show the exon length distribution at the end of the `gth` output.

`-introndistri`
: Show the intron length distribution at the end of the `gth` output. This option might be useful to get a feeling for a good setting of the option `-gcmaxgapwidth`.

`-refseqcovdistri`
: Show the reference sequence coverage distribution at the end of the `gth` output. This option might be useful to get a feeling for a good setting of the option `-gcmincoverage`.

Miscellaneous Options

`-first` n
: The positive integer n specifies the maximum number of computed spliced alignments per genomic DNA input. The default value is 0, which leads to the computation of all sensible spliced alignments.

`-help`
: Show a summary of the basic options. That is, only the most common options. Afterwards terminate.

`-help+`
: Show a summary of all options and terminate.

`-version`
: Show version number and built-date of *GenomeThreader*. Afterwards terminate.

If one of the last three options is used, `gth` terminates with exit code 0 after showing the corresponding output.

A.4 `gthconsensus`: Incremental Updates

With the help of `gthconsensus` one can perform so-called *incremental updates* of cDNA/EST and protein databases[4]:

[4]To simplify the explanation, in the following we mention only cDNA/EST databases, but everything said also applies to protein databases.

In a typical application of `gth` one uses a cDNA/EST database to annotate a genomic sequence. As a result of the ongoing sequencing efforts it is very likely that a few weeks after such an annotation the used cDNA/EST database is not up-to-date anymore, because new suitable cDNAs/ESTs are available. The conventional approach was to rerun the whole annotation process using an updated database. This has the drawback that one repeats a lot of spliced alignment calculations of the cDNAs/ESTs which have already been in the database before the update while computing typically only a few new spliced alignments of the added cDNAs/ESTs.

This observation leads to the idea to split the annotation process into two parts, such that one can reuse the already computed spliced alignments (that is, to perform incremental updates):

1. In one part one performs the computations which are independent of each other only once and stores the intermediate results.
2. In the other part the stored results are used to perform the calculations which depend on one another.

The first part refers to the calculation of spliced alignments (or predicted gene structures) and is realized in `gth` by the option `-intermediate` (see Section A.3.9). If these results are stored in a file, we call it an *intermediate file*.

The second part refers to the calculation of *consensus* spliced alignments (or predicted gene locations) and is realized by the program `gthconsensus`, which processes a set of intermediate files. In the following section the application of `gthconsensus` is explained. An example session using `gth -intermediate` and `gthconsensus` is shown in Section A.13.

A.4.1 The Options of `gthconsensus`

`gthconsensus` is called as follows:

`gthconsensus` [*options*] *intermediate_files*

Here *intermediate_files* denotes a list of one or more files containing XML intermediate output produced by `gth` invocations using the option `-intermediate`. The *options* available for `gthconsensus` are a subset of the options available for `gth`. To find out which options are included in the subset try:

```
gthconsensus -help
```

Basically all options are included into `gthconsensus` which make sense for the purpose of it. If you think a useful option is missing in `gthconsensus`, feel free to contact the author.

A.5 `gthsplit`: Split Intermediate Files

With the help of `gthsplit` one can split *GenomeThreader* output files containing intermediate results into multiple sets according to different criteria. `gthsplit` is called as follows:

`gthsplit` [*options*] *intermediate_files*

If no intermediate file is given as input, stdin is used instead. This allows to use `gthsplit` in a UNIX pipe.

`gthsplit` offers the following options:

`-alignmentscore`
: If this option is used, the input files are split according to the overall alignment score (scr).

`-coverage`
: If this option is used, the input files are split according to the coverage (cov).

`-range` $\Re$
: Set the percentage range $\Re$ used to create the sets. Each set contains the spliced alignments where the corresponding percentage (i.e., the alignment score or the coverage) is greater or equal then the lower set bound and lower then the higher bound. Spliced alignments with a percentage of 100% go into the last set. $\Re$ must divide 100 without rest. The default range is 5.

Furthermore, the options `-v`, `-gzip`, `-bzip2`, `-force`, and `-help` are available and have the same semantic as in `gth`.

The spliced alignment filter options described in Section A.3.9 can also be used to reduce the set of spliced alignment accordingly before splitting it.

A.5.1 Applying `gthsplit`

The following examples show how to use `gthsplit` and its output:

```
$ gthsplit -alignmentscore -v -gzip U89959.inter.gz
$ process all intermediate output files
$ process file 1/1: U89959.inter.gz
$ split file created: U89959.inter.scr70-75.gz (size=1)
$ split file created: U89959.inter.scr80-85.gz (size=11)
$ split file created: U89959.inter.scr85-90.gz (size=18)
$ split file created: U89959.inter.scr90-95.gz (size=25)
$ split file created: U89959.inter.scr95-100.gz (size=144)
```

A.5.2 The Script `gthsplit2dim.sh`

With the shell script `gthsplit2dim.sh` on can split up an intermediate file in both dimensions (i.e., along the alignment score and the coverage) at the same time. It offers the following options:

`-r` ℜ
: Same as `-range` when `gthsplit` is used.

`-f`
: Same as `-force` when `gthsplit` is used.

`-v`
: Same as `-v` when `gthsplit` is used.

`-g`
: Same as `-gzip` when `gthsplit` is used.

`-b`
: Same as `-bzip2` when `gthsplit` is used.

A.5.3 Applying `gthsplit2dim.sh`

The following example shows how to use `gthsplit2dim.sh` and its output:

```
$ gthsplit2dim.sh -r 10 -v -g ceres_full.inter.gz
$ split according to alignment score
$ process all intermediate output files
$ process file 1/1: ceres_full.inter.gz
$ split file created: ceres_full.inter.scr70-80.gz (size=30)
$ split file created: ceres_full.inter.scr80-90.gz (size=102)
$ split file created: ceres_full.inter.scr90-100.gz (size=1310)
$ split according to coverage
```

A.6 `gthgetseq`: Get FASTA Sequences

With the help of `gthgetseq` one can get the used FASTA sequences from *GenomeThreader* output files containing intermediate results. That is, all sequences which are represented in the intermediate files are printed in FASTA format on stdout. Strictly speaking, the sequences are not extracted from the intermediate file, but from the corresponding input sequence files. `gthgetseq` is called as follows:

`gthsplit` [*options*] `-getcdna` | `-getprotein` | `-getgenomic` *intermediate_files*

If no intermediate file is given as input, stdin is used instead. This allows to use `gthgetseq` in a UNIX pipe. `gthgetseq` offers the following options:

`-getcdna`
: Get cDNA/EST sequences used in the spliced alignments contained in the given intermediate files.

`-getcdnacomp`
: Get cDNA/EST sequences *not* used in the spliced alignments contained in the given intermediate files. That is, the *complement* of `-getcdna`.

`-getprotein`
: Get protein sequences used in the spliced alignments contained in the given intermediate files.

`-getproteincomp`
: Get protein sequences *not* used in the spliced alignments contained in the given intermediate files. That is, the *complement* of `-getprotein`.

`-getgenomic`
: Get genomic sequences used in the spliced alignments contained in the given intermediate files.

`-getgenomiccomp`
: Get genomic sequences *not* used in the spliced alignments contained in the given intermediate files. That is, the *complement* of `-getgenomic`.

Furthermore, the options `-gzip`, `-bzip2`, and `-help` are available and have the same semantic as in `gth`.

The spliced alignment filter options described in Section A.3.9 can also be used to extract sequences only from spliced alignments which pass the filter.

At least one of the options `-getcdna`, `-getcdnacomp`, `-getprotein`, `-getproteincomp`, `-getgenomic`, `-getgenomiccomp` is mandatory.

A.6.1 Applying `gthgetseq`

Lets assume we have an intermediate file called `ceres_full.inter.gz`. To get all cDNAs which led to a spliced alignment with a maximum alignment score of 0.75, we would issue the following command:

```
$ gthgetseq -getcdna -gzip -maxalignmentscore 0.75 ceres_full.inter.gz
>7894
accactacaaccaccgcaacaaccaccaaaaaccctctcaaagaaatttctttttttttct
```

```
tactttcttggtttgtcaaatatggtcagccatccaatggagaaagctgcaaatggtgcg
tctgcgttggaaacgcagacgggtgagttagatcagccggaacggcttcgtaagatcata
tcggtgtcttccattgccgccggtgtacagttcggttgggctttacagttatctctgttg
actccttacgtgcagctactcggaatcccacataaatgggcttctctgatttggctctgt
ggtccaatctccggtatgcttgttcagcctatcgtcggttaccacagtgaccgttgcacc
tcaagattcggccgtcgtcgtcccttcatcgtcgctggagctggtttagtcaccgttgct
gttttccttatcggttacgctgccgatataggtcacagcatgggcgatcagcttgacaaa
ccgccgaaaacgcgagccatagcgatattcgctctcgggttttggattcttgacgtggct
aacaacaccttacaaggaccctgcagagctttcttggctgatttatcagcagggaacgct
aagaaaacgcgaaccgcaaacgcgtttttctcgtttttcatggcggttggaaacgttttg
ggttacgctgcgggatcttacagaaatctctacaaagttgtgcctttcacgatgactgag
tcatgcgatctctactgcgcaaacctcaaaacgtgttttttcctatccataacgcttctc
ctcatagtcactttcgtatctctctgttacgtgaaggagaagccatggacgccagagcca
acagccgatggaaaagcctccaacgttccgtttttcggagaaatcttcggagctttcaag
gaactaaaaagacccatgtggatgcttcttatagtcactgcactaaactggatcgcttgg
ttccctttccttctcttcgacactgattggatgggccgtgaggtgtacggaggaaactca
gacgcaaccgcaaccgcagcctctaagaagctttacaacgacggagtcagagctggtgct
ttggggcttatgcttaacgctattgttcttggtttcatgtctcttggtgttgaatggatt
ggtcggaaattgggaggagctaaaaggctttggggtattgttaacttcatcctcgccatt
tgcttggccatgacggttgtggttacgaaacaagctgagaatcaccgacgagatcacggc
ggcgctaaaacaggtccacctggtaacgtcacagctggtgctttaactctcttcgccatc
ctcggtatccccccaagccattacgtttagcattccttttgcactagcttccatattttca
accaattccggtgccggccaaggactttccctaggtgttctgaatctagccattgtcgtc
cctcagatggtaatatctgtgggaggtggaccattcgacgaactattcggtggtggaaac
attccagcatttgtgttaggagcgattgcggcagcggtaagtggtgtattggcgttgacg
gtgttgccttcaccgcctccggatgctcctgccttcaaagctactatgggatttcattga
attttagcagtggttgtttggctctctttctctcataaaacagtagtgttgtgcaaatcc
tacataaagaaaaaagaaaaggaaattaaactcattgggttggtttgtattttacctaaa
cccacgaagttccttttttctttttgtaactcaatttaaatttggagtatattttacttttt
tgc
```

To store the output in a file `cdna`, one can redirect the output like this:

```
$ gthgetseq -getcdna -gzip -maxalignmentscore 0.75 ceres_full.inter.gz > cdna
```

A.7 `gthfilestat`: Show Statistics

`gthfilestat` shows statistics about spliced alignments in *GenomeThreader* output files containing intermediate results. This might be helpful in getting an overview of a set of intermediate files, before they are processed further, for example, with `gthsplit`.

`gthfilestat` [*options*] *intermediate_files*

If no intermediate file is given as input, stdin is used instead. This allows to use `gthfilestat` in a UNIX pipe.

Besides the options `-v` and `-help`, the spliced alignment filter options described in Section A.3.9 are available. The semantic is the same as in `gth`.

A.7.1 Applying `gthfilestat`

The following example shows how to use `gthfilestat` and its output:

```
$ gthfilestat ceres_sub.inter.gz
spliced alignment alignment score distribution:

spliced alignment coverage distribution:

memory statistics:
5 spliced alignments have been stored
5 predicted gene locations have been stored

date finished: 2011-07-30 22:13:04
```

A.8 `gthbssmfileinfo`: BSSM File Information

With `gthbssmfileinfo` one can print out information about BSSM files. It is called as follows:

`gthbssmfileinfo` *bssm_file*

`gthbssmfileinfo` uses the directories specified by the environment variable `BSSMDIR` to look for the specified *bssm_file*. For example, printing information about the human BSSM file works like this:

```
$ gthbssmfileinfo human
$ the specified BSSM parameter file contains the following models:
$ GT donor sites   = True (two-class)
$ GC donor sites   = False
$ AG acceptor sites= True (two-class)
```

A.9 `gthbssmtrain`: Train BSSMs

`gthbssmtrain` is called as follows:

`gthbssmtrain` [*options*] `-seqfile(s)|-regionmapping` *arg GFF3_file*

`-outdir` *dir*
: Specify the name of the output directory to which the training files are written. The default is `training_data`.

`-gcdonor` *arg*
: Extract training data for GC donor sites. The default is `yes`.

`-filtertype` *type*
: Set type of features used for filtering (usually `exon` or `CDS`). The default is `exon`.

`-goodexoncount` *n*
: Set the minimum number *n* of good exons a feature must have to be in the training data. The default is 1.

`-cutoff` *s*
: Set the minimum score *s* an exon must have to count towards the "good exon count" (exons without a score count as good). The default is 1.0.

`-extracttype` *type*
: Set type of features to to be extracted as exons (usually `exon` or `CDS`). The default is `CDS`.

`-seqfile` *file*
: Set the sequence file from which to extract the features.

`-seqfiles` *file ...*
: Set the sequence files from which to extract the features. The list of sequence files can be terminated with `--`.

`-matchdesc` *arg*
: Match the sequence descriptions for the desired sequence IDs. The default is `no`.

`-usedesc` *arg*
: Use sequence descriptions to map the sequence IDs (in the GFF3 file) to actual sequence entries. If a description contains a sequence range position (for example, `III:1000001..2000000`), the first part is used as sequence ID ('`III`') and the first range as offset ('`1000001`'). The default is `no`.

`-regionmapping` *file*
: Set file containing sequence-region to sequence file mapping.

`-seed` *s*
: Set seed for random generator manually. 0 generates a seed from the current time and the process id. The default is 0.

One of the options `-seqfile`, `-seqfiles`, or `-regionmapping` is mandatory. The mandatory argument *GFF3_file* must contain the annotation used to train the BSSM in GFF3 format.

A.9.1 Applying `gthbssmtrain`

The following example shows how to use `gthbssmtrain`.

At first we have to create the annotation data which can be used to train a BSSM. For this purpose, `gth` or `gthconsensus` is called with the options `-gff3out` and `-skipalignmentout`.

Furthermore, the option `-md5ids` should be used to make the subsequent sequence ID mapping of the GFF3 file used in `gthbssmtrain` easier and less error prone. The annotation should be of high quality (that is, all gene predicted gene structures seem to be correct) and contain at least a couple of hundred splice sites. A `gth` call could look like this (we store the result in a file called `arab.gff3`):

```
$ gth -genomic U89959_genomic.fas -cdna U89959_ests.fas -gff3out -skipalignmentout -md5ids
```

Afterwards we can train the BSSM with `gthbssmtrain` using the following command. Thereby, we supply the genomic sequence file used in the annotation to the option `-seqfile`. If more than one genomic sequence file was used, the option `-seqfiles` can be used. Because we used `-md5ids` when we produced the GFF3 file, the mapping from sequence ID works automatically:

```
$ gthbssmtrain -seqfile U89959_genomic.fas arab.gff3
gt-ag:  94.33% (n=133)
gc-ag:   0.71% (n=1)
```

The output of `gthbssmtrain` shows you how many `gt-ag` and `gc-ag` splice sites have been processed. If this number looks wrong, probably something is wrong with the sequence ID mapping. Because we did not set option `-outdir`, the training data is stored in the default output directory `training_data`.

We can now use `gthbssmbuild` (which is described in detail in the next Section) to build the actual BSSM file `arab.bssm`. We use only the options `-gtdonor` and `-agacceptor`, because there were not enough donor sites that it would make sense to build the GC donor model into the BSSM file with `-gcdonor`:

```
$ gthbssmbuild -gtdonor -agacceptor -datapath training_data -bssmfile arab.bssm
```

We now have a new BSSM file named `arab.bssm` which can be used in subsequent `gth` or `gthconsensus` calls with the option `-bssm` (without the `.bssm` suffix), like this:

```
$ gth -bssm arab -genomic U89959_genomic.fas -cdna U89959_ests.fas
```

A.10 `gthbssmbuild`: Build BSSM files

With the help of `gthbssmbuild` one can build a BSSM file from a directory tree containing the training data. The training data is usually generated with `gthbssmtrain` which is described in the previous section.

`gthbssmbuild` is called as follows:

`gthsplit` [*options*] `-databasedir` *dir* `-bssmfile` *file*

`-bssmfile` *file*
: Specify the name of the BSSM file *file* to store parameters in.

`-datapath` *dir*
: Specify root of species-specific training data directory tree *dir*.

`-gtdonor`
: Train the `GT` donor model.

`-gcdonor`
: Train the `GC` donor model.

`-agacceptor`
: Train the `AG` acceptor model.

Furthermore, the options `-gzip` and `-help` are available and have the same semantic as in `gth`.

The options `-bssmfile` and `-datapath` are mandatory. Furthermore, at least one of the options `-gtdonor`, `-gcdonor`, and `-agacceptor` is required.

A.10.1 The BSSM Training Data Directory

A BSSM training data directory has up to three subdirectories named `GT_donor`, `GC_donor`, and `AG_acceptor` which contain the training data files for the corresponding models. Each such directory must contain seven files named `F0`, `F1`, `F2`, `Fi`, `T0`, `T1`, and `T2` (with an additional suffix `.gz` if they are compressed). For example, the directory tree containing the compressed training data for *rice* looks like this:

```
$ ls -R rice
AG_acceptor/ GC_donor/    GT_donor/

rice/AG_acceptor:
F0.gz  F1.gz  F2.gz  Fi.gz  T0.gz  T1.gz  T2.gz

rice/GC_donor:
F0.gz  F1.gz  F2.gz  Fi.gz  T0.gz  T1.gz  T2.gz

rice/GT_donor:
F0.gz  F1.gz  F2.gz  Fi.gz  T0.gz  T1.gz  T2.gz
```

A.11 `gthclean.sh`: Remove Indices

The script `gthclean.sh` removes all automatically and manually constructed indices in the directory where it is called. Furthermore, all files ending with `.polya` and `.polya.info`

created by using the option `-maskpolyatails` of `gth` are removed. Do not use the endings removed by the script, otherwise the corresponding files will be deleted when `gthclean.sh` is called! The script looks as follows:

A.12 Construction of the Indices

GenomeThreader uses an persistent index to determine which spliced alignments have to be computed during an initial matching and chaining phase called *similarity filter*.

If `gth` is called the first time on a set of input files (without using the option `-noautoindex`), these indices are constructed automatically. In subsequent calls it is recognized that these indices already exist and the construction phase is skipped. That is, the index construction is completely transparent to the user, the only difference is the reduced execution time when the index already exists.

If the option `-noautoindex` is used, it is assumed that the indices already exist. They have to be created by `mkvtree` beforehand. It is part of the *Vmatch* software suite, which is available on the website `http://vmatch.de`. On this website you can also find a manual for `mkvtree`. For DNA files, indices are constructed as follows (the file is called `dna.fasta`):

```
$ mkvtree -v -dna -allout -pl -db dna.fasta
```

And for protein files like this (the filename is `protein.fasta`):

```
$ mkvtree -v -protein -allout -pl -db protein.fasta
```

If you use DNA reference files, you have to construct the index for the genomic files, if option `-inverse` is not used. And for the reference files, if option `-inverse` is used. If you use protein reference files, the index has to be constructed for the reference files, no matter if option `-inverse` is used or not. You can mix both types of input files.

Input files can be compressed with `gzip`. In this case they must end with `.gz`.

A.13 Tutorial

In this section we give an tutorial on how to use *GenomeThreader* by presenting some typical uses of *GenomeThreader*. On our walk through the different examples, the most important features and options of *GenomeThreader* are introduced.

A.13.1 Mapping a Single EST on the *A. thaliana* Chromosome 1

We invoke `gth` in the simplest possible way, just using two mandatory options `-genomic` and `-cdna` to map the EST with gi number 19875482 against the first chromosome of *Arabidopsis thaliana* (gi 42592260). The EST is supplied as file `AU236313.gbk.gz` in GENBANK format and the chromosome in FASTA format (file `NC_003070.fna.gz`). Both files have been compressed with the program `gzip` beforehand, which is indicated by the ending `.gz`.

```
$ gth -genomic NC_003070.fna.gz -cdna AU236313.gbk.gz
$ GenomeThreader 1.4.9 (2011-07-02 14:50:10)
$ Date run: 2011-07-30 22:13:35
$ Arguments: -genomic NC_003070.fna.gz -cdna AU236313.gbk.gz
********************************************************************************
EST Sequence: file=AU236313.gbk.gz, strand=+, description=AU236313 AU236313 RAFL14 Arabidopsis thaliana cDNA c

    1  GCGTGTTGTT AAAGAGCTCA CTAGTTGGGT GATTTATTCA GAGGAGGATC CGGAAGCTCA
   61  ACAAAGATAT TACTATTGGT CTTATCCAGC GTGAGTTGCT TAGCCTAGCG GAGTACAATG
  121  TCCACATGGC GAAGCATCTT GATGGAGGGA GAAACAAGAC CGCAACTGAC TTTGCTATTT
  181  CTCTACTCCA ATCCTTGGTC ACTGAGGAGT CNAGTGTCAT TTNNTAG

Genomic Template: file=NC_003070.fna.gz, strand=+, from=382240, to=383395, description=gi|42592260|ref|NC_0030

Predicted gene structure:

 Exon  1   382540   382567 (  28 n);  cDNA      1     28 (  28 n); score: 1.000
  Intron  1   382568   382802 ( 235 n); Pd: 0.050 (s:    0), Pa: 0.050 (s: 0.95)
 Exon  2   382803   382929 ( 127 n);  cDNA     29    156 ( 128 n); score: 0.980
  Intron  2   382930   383029 ( 100 n); Pd: 0.050 (s: 1.00), Pa: 0.050 (s: 1.00)
 Exon  3   383030   383100 (  71 n);  cDNA    157    227 (  71 n); score: 0.944

MATCH 42592260+ AU236313+ 0.967 226 0.996 C
PGS_42592260+_AU236313+ (382540  382567,382803  382929,383030  383100)

Alignment (genomic DNA sequence = upper lines):

GCGTGTTGTT AAAGAGCTCA CTAGTTGGGT ATGTTTACAA CCTTTTCAAG ATTTCACTTC      382599
|||||||||| |||||||||| ||||||||
GCGTGTTGTT AAAGAGCTCA CTAGTTGG.. .......... .......... ..........          28


GCTGATGTGC TTTGTTCAGT TACTTCTCCA TTAATTTGTC ACTATTTCTG TCAGAACACA      382659

.......... .......... .......... .......... .......... ..........          28


TAGGATAACA CATATCATAT AAGTGCTAGG TCGAGTCTGT TTCCTGTAGT TGGAGCCTAT      382719

.......... .......... .......... .......... .......... ..........          28


CCTCAACTGG TTATAGATAC TAGATTTGTT TCTTTGGTAT TTTTAGTTAT AATTAATTAT      382779

.......... .......... .......... .......... .......... ..........          28


CTTTCTTCAA ACTTTTGACA CAGGTGATTT ATTCAGAGGA GGAT-CGGAA GCTCAACAAA      382838
                        ||||||| |||||||||| |||| ||||| ||||||||||
.......... .......... ...GTGATTT ATTCAGAGGA GGATCCGGAA GCTCAACAAA          65


GATATTACTA TTGGTCTTAT CCAGCGTGAG TTGCTTAGCC TAGCGGAGTA CAATGTCCAC      382898
```

```
|||||||||| |||||||||| |||||||||| |||||||||| |||||||||| ||||||||||
GATATTACTA TTGGTCTTAT CCAGCGTGAG TTGCTTAGCC TAGCGGAGTA CAATGTCCAC         125


ATGGCGAAGC ATCTTGATGG AGGGAGAAAC AGTATGCTGA ATTGCTTAAC CTTTGTTGAT      382958
|||||||||| |||||||||| |||||||||| |
ATGGCGAAGC ATCTTGATGG AGGGAGAAAC A......... .......... ..........         156


GTCCTTGTGT GGTAACATTC TTTTTTTGTT TTGGCTAGTG AACTTGTTTT AACAAGTTGT      383018

.......... .......... .......... .......... .......... ..........         156


TTGTTACGCA GAGACCGCAA CTGACTTTGC TATTTCTCTA CTCCAATCCT TGGTCACTGA      383078
           ||||||||| |||||||||| |||||||||| |||||||||| ||||||||||
.......... .AGACCGCAA CTGACTTTGC TATTTCTCTA CTCCAATCCT TGGTCACTGA         205


GGAGTCGAGT GTCATTTCAG AG                                               383100
|||||| ||| |||||||    ||
GGAGTCNAGT GTCATTTNNT AG                                                  227
```

Predicted gene locations (1):

```
PGL   1 (+ strand):      382540     383100
AGS-1 (382540  382567,382803  382929,383030  383100)
SCR   (e 1.000  d 0.050 a 0.050,e 0.980  d 0.050 a 0.050,e 0.944)

  Exon  1   382540   382567 (  28 n); score: 1.000
    Intron  1   382568   382802 ( 235 n);          Pd: 0.050  Pa: 0.050
  Exon  2   382803   382929 ( 127 n); score: 0.980
    Intron  2   382930   383029 ( 100 n);          Pd: 0.050  Pa: 0.050
  Exon  3   383030   383100 (  71 n); score: 0.944

  PGS (382540  382567,382803  382929,383030  383100) AU236313+
```

3-phase translation of AGS-1 (+strand):

```
          .        .        .       :   .        .        .
  382540  GCGTGTTGTTAAAGAGCTCACTAGTTGG : GTGATTTATTCAGAGGAGGATCGGAAGCTCAA
           A  C  C  *  R  A  H  *  L   : G  D  L  F  R  G  G  S  E  A  Q
            R  V  V  K  E  L  T  S  W  :  V  I  Y  S  E  E  D  R  K  L  N
             V  L  L  K  S  S  L  V  G :  *  F  I  Q  R  R  I  G  S  S


          .        .        .        .        .        .
  382835  CAAAGATATTACTATTGGTCTTATCCAGCGTGAGTTGCTTAGCCTAGCGGAGTACAATGT
           Q  R  Y  Y  Y  W  S  Y  P  A  *  V  A  *  P  S  G  V  Q  C
            K  D  I  T  I  G  L  I  Q  R  E  L  L  S  L  A  E  Y  N  V
          T  K  I  L  L  L  V  L  S  S  V  S  C  L  A  *  R  S  T  M


          .        .        .        .    :     .        .
  382895  CCACATGGCGAAGCATCTTGATGGAGGGAGAAACA : AGACCGCAACTGACTTTGCTATTTC
           P  H  G  E  A  S  *  W  R  E  K  Q :   D  R  N  *  L  C  Y  F
            H  M  A  K  H  L  D  G  G  R  N   : K  T  A  T  D  F  A  I  S
          S  T  W  R  S  I  L  M  E  G  E  T  :  R  P  Q  L  T  L  L  F


          .        .        .        .        .
```

```
  383055  TCTACTCCAATCCTTGGTCACTGAGGAGTCGAGTGTCATTTCAGAG
           S  T  P  I  L  G  H  *  G  V  E  C  H  F  R
            L  L  Q  S  L  V  T  E  E  S  S  V  I  S  E
           L  Y  S  N  P  W  S  L  R  S  R  V  S  F  Q


Maximal non-overlapping open reading frames (>= 64 codons)

>42592260+_PGL-1_AGS-1_PPS_1 (382541  382567,382803  382929,383030  383100) (frame '1';   225 bp,   75 residue
       1  RVVKELTSWV IYSEEDRKLN KDITIGLIQR ELLSLAEYNV HMAKHLDGGR NKTATDFAIS
      61  LLQSLVTEES SVISE



$ general statistics:
$ 1 chain has been computed
$
$ memory statistics:
$ 1 spliced alignments have been stored
$ 1 predicted gene locations have been stored
$ 0 megabytes was the average size of the backtrace matrix
$ 2 backtrace matrices have been allocated
$
$ date finished: 2011-07-30 22:13:58
```

In the first line (starting with symbol `$`) of all examples, the user input consisting of the called program plus the arguments are shown. Output lines starting with the symbol `$` give you useful information about the job — which parameters have been used is shown in the beginning and various statistical information is output at the end. In our example call an additional warning was issued after the parameter section complaining about the missing usage of the option `-species` which specify a so-called BSSM[5] parameter file. These files contain statistical information which makes it possible to "get the splice sites right" more easily (this is an oversimplification, [BXZ04] provides a good starting point to the theory behind this). So we execute the job again with the correct splice site file and the option `-v` which gives us additional information about the run:

```
$ gth -species arabidopsis -genomic NC_003070.fna.gz -cdna AU236313.gbk.gz -v
$ GenomeThreader 1.4.9 (2011-07-02 14:50:10)
$ Date run: 2011-07-30 22:13:58
$ Arguments: -species arabidopsis -genomic NC_003070.fna.gz -cdna AU236313.gbk.gz -v
$ make sure all necessary indices exist
$ make sure the necessary indices of all genomic input files exist
$ check the following file for index:
$ NC_003070.fna.gz
$ index exists
$ make sure the necessary indices off all reference input files exist
$ check the following file for index:
$ AU236313.gbk.gz
$ index exists
$ invoking similarity filter
$ compute direct matches
$ call vmatch to compute matches
# args=-d -v -l 20 -seedlength 18 -exdrop 2 -q AU236313.gbk.gz.dna /Users/gordon/work/dissertation/NC_003070.f
$ file=NC_003070.fna.gz 30867397 30432563
$ databaselength=30432562
$ alphabet "aAcCgGtTuUnsywrkvbdhmNSYWRKVBDHM" (size 32) mapped to "acgtn" (size 5)
```

[5] BSSM stands for *Bayesian Splice Site Model*

```
$ NC_003070.fna.gz.dna.tis read
$ NC_003070.fna.gz.dna.suf read
$ NC_003070.fna.gz.dna.lcp read
$ NC_003070.fna.gz.dna.llv read
$ NC_003070.fna.gz.dna.sti1 read
$ NC_003070.fna.gz.dna.bck read
$ file=AU236313.gbk.gz 2488 227
$ databaselength=226
$ alphabet "aAcCgGtTuUnsywrkvbdhmNSYWRKVBDHM" (size 32) mapped to "acgtn" (size 5)
$ AU236313.gbk.gz.dna.tis read
$ AU236313.gbk.gz.dna.des read
$ AU236313.gbk.gz.dna.sds read
$ AU236313.gbk.gz.dna.ssp read
# matches are reported in the following way
# l(S) n(S) r(S) t l(Q) n(Q) r(Q) d e s i
# where:
# l = length
# n = sequence number
# r = relative position
# t = type (D=direct, P=palindromic)
# d = distance value (negative=hamming distance, 0=exact, positive=edit distance)
# e = E-value
# s = score value (negative=hamming score, positive=edit score)
# i = percent identity
# (S) = in Subject
# (Q) = in Query
$ find direct substring matches against query (maximal exact matches)
$ overall space peak: main=0.02 MB (0.00 bytes/symbol), secondary=214.19 MB (7.38 bytes/symbol)
$ d=+, compute chains for bucket 1/1 (matches in bucket=4)
$ sort global chains according to reference sequence coverage
$ calculate spliced alignment for every chain
$ d=+, compute spliced alignment, genseq=+, chain=1/1, refseq=+
$ compute palindromic matches
$ call vmatch to compute matches
# args=-p -v -l 20 -seedlength 18 -exdrop 2 -q AU236313.gbk.gz.dna /Users/gordon/work/dissertation/NC_003070.f
$ file=NC_003070.fna.gz 30867397 30432563
$ databaselength=30432562
$ alphabet "aAcCgGtTuUnsywrkvbdhmNSYWRKVBDHM" (size 32) mapped to "acgtn" (size 5)
$ NC_003070.fna.gz.dna.tis read
$ NC_003070.fna.gz.dna.suf read
$ NC_003070.fna.gz.dna.lcp read
$ NC_003070.fna.gz.dna.llv read
$ NC_003070.fna.gz.dna.sti1 read
$ NC_003070.fna.gz.dna.bck read
$ file=AU236313.gbk.gz 2488 227
$ databaselength=226
$ alphabet "aAcCgGtTuUnsywrkvbdhmNSYWRKVBDHM" (size 32) mapped to "acgtn" (size 5)
$ AU236313.gbk.gz.dna.tis read
$ AU236313.gbk.gz.dna.des read
$ AU236313.gbk.gz.dna.sds read
$ AU236313.gbk.gz.dna.ssp read
# matches are reported in the following way
# l(S) n(S) r(S) t l(Q) n(Q) r(Q) d e s i
# where:
# l = length
# n = sequence number
# r = relative position
# t = type (D=direct, P=palindromic)
# d = distance value (negative=hamming distance, 0=exact, positive=edit distance)
# e = E-value
# s = score value (negative=hamming score, positive=edit score)
# i = percent identity
# (S) = in Subject
# (Q) = in Query
$ find palindromic substring matches against query
```

```
$ overall space peak: main=29.04 MB (1.00 bytes/symbol), secondary=214.19 MB (7.38 bytes/symbol)
$ calculate spliced alignment for every chain
$ output spliced alignments
********************************************************************************
EST Sequence: file=AU236313.gbk.gz, strand=+, description=AU236313 AU236313 RAFL14 Arabidopsis thaliana cDNA c

    1  GCGTGTTGTT AAAGAGCTCA CTAGTTGGGT GATTTATTCA GAGGAGGATC CGGAAGCTCA
   61  ACAAAGATAT TACTATTGGT CTTATCCAGC GTGAGTTGCT TAGCCTAGCG GAGTACAATG
  121  TCCACATGGC GAAGCATCTT GATGGAGGGA GAAACAAGAC CGCAACTGAC TTTGCTATTT
  181  CTCTACTCCA ATCCTTGGTC ACTGAGGAGT CNAGTGTCAT TTNNTAG

Genomic Template: file=NC_003070.fna.gz, strand=+, from=382240, to=383395, description=gi|42592260|ref|NC_0030

Predicted gene structure:

 Exon  1   382540   382567 (  28 n);  cDNA      1     28 (  28 n); score: 1.000
  Intron  1   382568   382802 ( 235 n); Pd: 0.985 (s:    0), Pa: 0.999 (s: 0.95)
 Exon  2   382803   382929 ( 127 n);  cDNA     29    156 ( 128 n); score: 0.980
  Intron  2   382930   383029 ( 100 n); Pd: 0.956 (s: 1.00), Pa: 0.983 (s: 1.00)
 Exon  3   383030   383100 (  71 n);  cDNA    157    227 (  71 n); score: 0.944

MATCH 42592260+ AU236313+ 0.967 226 0.996 C
PGS_42592260+_AU236313+ (382540  382567,382803  382929,383030  383100)

Alignment (genomic DNA sequence = upper lines):

GCGTGTTGTT AAAGAGCTCA CTAGTTGGGT ATGTTTACAA CCTTTTCAAG ATTTCACTTC      382599
|||||||||| |||||||||| ||||||||
GCGTGTTGTT AAAGAGCTCA CTAGTTGG.. .......... .......... ..........          28


GCTGATGTGC TTTGTTCAGT TACTTCTCCA TTAATTTGTC ACTATTTCTG TCAGAACACA      382659

.......... .......... .......... .......... .......... ..........          28


TAGGATAACA CATATCATAT AAGTGCTAGG TCGAGTCTGT TTCCTGTAGT TGGAGCCTAT      382719

.......... .......... .......... .......... .......... ..........          28


CCTCAACTGG TTATAGATAC TAGATTTGTT TCTTTGGTAT TTTTAGTTAT AATTAATTAT      382779

.......... .......... .......... .......... .......... ..........          28


CTTTCTTCAA ACTTTTGACA CAGGTGATTT ATTCAGAGGA GGAT-CGGAA GCTCAACAAA      382838
                        ||||||| |||||||||| |||| ||||| ||||||||||
.......... .......... ...GTGATTT ATTCAGAGGA GGATCCGGAA GCTCAACAAA          65


GATATTACTA TTGGTCTTAT CCAGCGTGAG TTGCTTAGCC TAGCGGAGTA CAATGTCCAC      382898
|||||||||| |||||||||| |||||||||| |||||||||| |||||||||| ||||||||||
GATATTACTA TTGGTCTTAT CCAGCGTGAG TTGCTTAGCC TAGCGGAGTA CAATGTCCAC         125


ATGGCGAAGC ATCTTGATGG AGGGAGAAAC AGTATGCTGA ATTGCTTAAC CTTTGTTGAT      382958
|||||||||| |||||||||| |||||||||| |
ATGGCGAAGC ATCTTGATGG AGGGAGAAAC A......... .......... ..........         156


GTCCTTGTGT GGTAACATTC TTTTTTTGTT TTGGCTAGTG AACTTGTTTT AACAAGTTGT      383018

.......... .......... .......... .......... .......... ..........         156
```

```
TTGTTACGCA GAGACCGCAA CTGACTTTGC TATTTCTCTA CTCCAATCCT TGGTCACTGA      383078
           ||||||||| |||||||||| |||||||||| |||||||||| ||||||||||
.......... .AGACCGCAA CTGACTTTGC TATTTCTCTA CTCCAATCCT TGGTCACTGA         205


GGAGTCGAGT GTCATTTCAG AG                                              383100
|||||| ||| |||||||    ||
GGAGTCNAGT GTCATTTNNT AG                                                 227


$ compute predicted gene locations
$ output predicted gene locations
--------------------------------------------------------------------------------

Predicted gene locations (1):


PGL   1 (+ strand):      382540     383100
AGS-1 (382540  382567,382803  382929,383030  383100)
SCR   (e 1.000  d 0.985 a 0.999,e 0.980  d 0.956 a 0.983,e 0.944)

  Exon  1   382540   382567 (  28 n); score: 1.000
    Intron  1   382568   382802 ( 235 n);          Pd: 0.985  Pa: 0.999
  Exon  2   382803   382929 ( 127 n); score: 0.980
    Intron  2   382930   383029 ( 100 n);          Pd: 0.956  Pa: 0.983
  Exon  3   383030   383100 (  71 n); score: 0.944

  PGS (382540  382567,382803  382929,383030  383100) AU236313+

3-phase translation of AGS-1 (+strand):

         .        .        .       :   .        .        .
  382540  GCGTGTTGTTAAAGAGCTCACTAGTTGG : GTGATTTATTCAGAGGAGGATCGGAAGCTCAA
           A  C  C  *  R  A  H  *  L   : G  D  L  F  R  G  G  S  E  A  Q
            R  V  V  K  E  L  T  S  W  :  V  I  Y  S  E  E  D  R  K  L  N
             V  L  L  K  S  S  L  V  G :   *  F  I  Q  R  R  I  G  S  S


         .        .        .        .        .        .
  382835  CAAAGATATTACTATTGGTCTTATCCAGCGTGAGTTGCTTAGCCTAGCGGAGTACAATGT
           Q  R  Y  Y  Y  W  S  Y  P  A  *  V  A  *  P  S  G  V  Q  C
            K  D  I  T  I  G  L  I  Q  R  E  L  L  S  L  A  E  Y  N  V
          T  K  I  L  L  L  V  L  S  S  V  S  C  L  A  *  R  S  T  M


         .        .        .        .    :     .        .
  382895  CCACATGGCGAAGCATCTTGATGGAGGGAGAAACA : AGACCGCAACTGACTTTGCTATTTC
           P  H  G  E  A  S  *  W  R  E  K  Q :   D  R  N  *  L  C  Y  F
            H  M  A  K  H  L  D  G  G  R  N   : K  T  A  T  D  F  A  I  S
          S  T  W  R  S  I  L  M  E  G  E  T  :  R  P  Q  L  T  L  L  F


         .        .        .        .        .
  383055  TCTACTCCAATCCTTGGTCACTGAGGAGTCGAGTGTCATTTCAGAG
           S  T  P  I  L  G  H  *  G  V  E  C  H  F  R
            L  L  Q  S  L  V  T  E  E  S  S  V  I  S  E
          L  Y  S  N  P  W  S  L  R  S  R  V  S  F  Q


Maximal non-overlapping open reading frames (>= 64 codons)

>42592260+_PGL-1_AGS-1_PPS_1 (382541  382567,382803  382929,383030  383100) (frame '1';   225 bp,   75 residue
       1  RVVKELTSWV IYSEEDRKLN KDITIGLIQR ELLSLAEYNV HMAKHLDGGR NKTATDFAIS
      61  LLQSLVTEES SVISE
```

```
$ general statistics:
$ 1 chain has been computed
$
$ memory statistics:
$ 1 spliced alignments have been stored
$ 1 predicted gene locations have been stored
$ 0 megabytes was the average size of the backtrace matrix
$ 1 backtrace matrix has been allocated
$
$ date finished: 2011-07-30 22:13:59
```

As one can see, the output grew quite a bit. In lines starting with the symbol # output from internal `vmatch` and `mkvtree` calls is shown. There is no `mkvtree` output in our example, because the necessary indices did already exist (see Section A.12 for details). To combine the option `-v` with option `-o` which saves the output in the supplied output file is quite useful when performing larger tasks, because this gives us progress information on stdout while the actual output is saved in the output file. For example, to match 5000 full-length cDNAs against the first 200000 bases of the *A. thaliana* chromosome 1, we issue the following command (only the last 20 lines of progress information is shown):

```
$ gth -v -frompos 1 -topos 200000 -inverse -force -o arab_mapping.txt -cdna CeresTigr.gz -species arabidopsis
$ d=-, compute chains for bucket 368/377 (matches in bucket=1)
$ d=-, compute chains for bucket 369/377 (matches in bucket=1)
$ d=-, compute chains for bucket 370/377 (matches in bucket=1)
$ d=-, compute chains for bucket 371/377 (matches in bucket=1)
$ d=-, compute chains for bucket 372/377 (matches in bucket=1)
$ d=-, compute chains for bucket 373/377 (matches in bucket=9)
$ d=-, compute chains for bucket 374/377 (matches in bucket=1)
$ d=-, compute chains for bucket 375/377 (matches in bucket=2)
$ d=-, compute chains for bucket 376/377 (matches in bucket=2)
$ d=-, compute chains for bucket 377/377 (matches in bucket=1)
$ sort global chains according to reference sequence coverage
$ calculate spliced alignment for every chain
$ d=-, compute spliced alignment, genseq=-, chain=1/5, refseq=+
$ d=-, compute spliced alignment, genseq=-, chain=2/5, refseq=+
$ d=-, compute spliced alignment, genseq=-, chain=3/5, refseq=+
$ d=-, compute spliced alignment, genseq=-, chain=4/5, refseq=+
$ d=-, compute spliced alignment, genseq=-, chain=5/5, refseq=+
$ output spliced alignments
$ compute predicted gene locations
$ output predicted gene locations
```

After this run, the output can be found in the file `arab_mapping.txt`. To get better results, we would also add the option `-gcmaxgapwidth 5000`, limiting the maximal intron size in *A. thaliana* to 5000 bases.

A.13.2 Using the Intron Cutout Technique

Assume we want to map a single mRNA against human chromosome 21. The obvious call would not succeed (only last 3 lines of output shown):

```
$ gth -species human -genomic hs_ref_chr21.fa.gz -cdna NM_003253.gbk
$ important messages:
$ 1 matrix allocations failed
$ 1 undetermined spliced alignments
```

This is due to the fact that human genes can contain very long introns. Therefore, we have to use the option `-intron cutout`. Furthermore, we use the option `-introndistri` which gives us a distribution of the intron sizes (only this distribution is shown from the output):

```
$ gth -intron cutout -introndistri -species human -genomic hs_ref_chr21.fa.gz -cdna NM_003253.gbk
```

```
$ length distribution of all introns:
88: 1 (prob=0.0357,cumulative=0.0357)
304: 1 (prob=0.0357,cumulative=0.0714)
574: 1 (prob=0.0357,cumulative=0.1071)
1112: 1 (prob=0.0357,cumulative=0.1429)
2134: 1 (prob=0.0357,cumulative=0.1786)
2370: 1 (prob=0.0357,cumulative=0.2143)
3060: 1 (prob=0.0357,cumulative=0.2500)
3182: 1 (prob=0.0357,cumulative=0.2857)
3684: 1 (prob=0.0357,cumulative=0.3214)
4080: 1 (prob=0.0357,cumulative=0.3571)
4354: 1 (prob=0.0357,cumulative=0.3929)
4984: 1 (prob=0.0357,cumulative=0.4286)
5076: 1 (prob=0.0357,cumulative=0.4643)
5390: 1 (prob=0.0357,cumulative=0.5000)
5632: 1 (prob=0.0357,cumulative=0.5357)
5706: 1 (prob=0.0357,cumulative=0.5714)
6081: 1 (prob=0.0357,cumulative=0.6071)
7032: 1 (prob=0.0357,cumulative=0.6429)
7602: 1 (prob=0.0357,cumulative=0.6786)
8136: 1 (prob=0.0357,cumulative=0.7143)
9748: 1 (prob=0.0357,cumulative=0.7500)
10534: 1 (prob=0.0357,cumulative=0.7857)
13820: 1 (prob=0.0357,cumulative=0.8214)
17355: 1 (prob=0.0357,cumulative=0.8571)
```

A.13.3 Employing `gthconsensus`

Assume we want to perform *incremental updates*, i.e., we want to annotate a genomic sequence with a set of cDNAs/ESTs and/or proteins and update the annotation as soon as new sequences become available.

Because we do not want to redo the complete computation every time new sequences become available, we save the *intermediate results* containing the spliced alignments and only recompute the *consensus spliced alignments* using `gthconsensus`.

For explanatory purposes, we use the same files as in the examples above. We produce the intermediate results of the mapping of 5000 cDNAs using the option `-intermediate` and store these in a file. The option `-intermediate` requires that we also use the option `-xmlout`, because the intermediate results are stored in an XML format. To save disk space we compress the output file on the fly using the option `-gzip`. The output is stored in the file `ceres.inter.gz`.

```
$ gth -intermediate -xmlout -gzip -o ceres.inter.gz -genomic NC_003070.fna.gz -cdna CeresTigr.gz -frompos 3000
```

To look at the results stored in the file `ceres.inter.gz`, we employ `gthconsensus` (with `-gzip`, because the file is compressed). The output was piped through `grep ^PGS` to shorten it:

```
$ gthconsensus ceres.inter.gz
PGS_42592260+_24737-(319898  320145,320252  320296,320380  320457,320769  321417,321745  321865,321958  322406
PGS_42592260+_116833+ (345866  347527)
PGS_42592260-_1693+ (370967  370859,370751  370258)
PGS_42592260-_40042+ (389646  389410,389301  389226,389138  389050,388940  388851,388760  388716,388609  3885
PGS_42592260-_35733+ (404439  404186,403947  403771,403526  403195)
```

We can also print some information about the intermediate file with the help of `gthfilestat`:

```
$ gthfilestat ceres.inter.gz
spliced alignment alignment score distribution:

spliced alignment coverage distribution:

memory statistics:
5 spliced alignments have been stored
5 predicted gene locations have been stored

date finished: 2011-07-30 22:15:10
```

Now we map an additional EST and store the result in `new.inter.gz`.

```
$ gth -intermediate -xmlout -gzip -o new.inter.gz -genomic NC_003070.fna.gz -cdna AU236313.gbk.gz -frompos 300
```

Afterwards we combine the two results (the output was also shortened):

```
$ gthconsensus ceres.inter.gz new.inter.gz
PGS_42592260+_24737-(319898  320145,320252  320296,320380  320457,320769  321417,321745  321865,321958  322406
PGS_42592260+_116833+ (345866  347527)
PGS_42592260-_1693+ (370967  370859,370751  370258)
PGS_42592260+_AU236313+ (382540  382567,382803  382929,383030  383100)
PGS_42592260-_40042+ (389646  389410,389301  389226,389138  389050,388940  388851,388760  388716,388609  3885
PGS_42592260-_35733+ (404439  404186,403947  403771,403526  403195)
```

Appendix B

Manual of *GenomeTools*

This appendix describes the tools of the *GenomeTools* package which are relevant for this dissertation.

B.1 The `bed_to_gff3` Tool

```
$ gt bed_to_gff3 -help
Usage: gt bed_to_gff3 [BED_file]
Parse BED file and convert it to GFF3.

-featuretype Set type of parsed BED features
             default: BED_feature
-thicktype   Set type of parsed thick BED features
             default: BED_thick_feature
-blocktype   Set type of parsed BED blocks
             default: BED_block
-o           redirect output to specified file
             default: undefined
-gzip        write gzip compressed output file
             default: no
-bzip2       write bzip2 compressed output file
             default: no
-force       force writing to output file
             default: no
-help        display help and exit
-version     display version information and exit

Report bugs to <gt-users@genometools.org>.
```

B.2 The `chseqids` Tool

```
$ gt chseqids -help
Usage: gt chseqids [option ...] mapping_file [GFF3_file]
Change sequence ids by the mapping given in mapping_file.

-sort    sort the GFF3 features after changing the sequence ids
         (memory consumption is proportional to the input file size)
         default: no
-v       be verbose
         default: no
-o       redirect output to specified file
         default: undefined
-gzip    write gzip compressed output file
         default: no
-bzip2   write bzip2 compressed output file
         default: no
-force   force writing to output file
         default: no
-help    display help and exit
-version display version information and exit

File format for mapping_file:

The supplied mapping file defines a mapping table named ``chseqids''. It maps
the sequence-regions given in the GFF3_file to other names.  It can be defined
as follows:

chseqids = {
  chr1  = "seq1",
  chr2  = "seq2"
}

When this example is used, all sequence ids ``chr1'' will be changed to ``seq1''
and all sequence ids ``chr2'' to ``seq2''.

Report bugs to <gt-users@genometools.org>.
```

B.3 The cds Tool

```
$ gt cds -help
Usage: gt cds [option ...] [GFF3_file]
Add CDS (coding sequence) features to exon features given in GFF3 file.

-minorflen      set the minimum length an open reading frame (ORF) must have to
                be added as a CDS feature (measured in amino acids)
                default: 64
-startcodon     require than an ORF must begin with a start codon
                default: no
-finalstopcodon require that the final ORF must end with a stop codon
                default: no
-seqfile        set the sequence file from which to extract the features
                default: undefined
-seqfiles       set the sequence files from which to extract the features
                use '--' to terminate the list of sequence files
-matchdesc      match the sequence descriptions from the input files for the
                desired sequence IDs (in GFF3)
                default: no
-usedesc        use sequence descriptions to map the sequence IDs (in GFF3) to
                actual sequence entries.
                If a description contains a sequence range (e.g.,
                III:1000001..2000000), the first  part is used as sequence ID
                ('III') and the first range position as offset ('1000001')
                default: no
-regionmapping  set file containing sequence-region to sequence file mapping
                default: undefined
-v              be verbose
                default: no
-o              redirect output to specified file
                default: undefined
-gzip           write gzip compressed output file
                default: no
-bzip2          write bzip2 compressed output file
                default: no
-force          force writing to output file
                default: no
-help           display help and exit
-version        display version information and exit

File format for option -regionmapping:

The file supplied to option -regionmapping defines a ``mapping''.  A mapping
maps the sequence-regions given in the GFF3_file to a sequence file containing
the corresponding sequence. Mappings can be defined in one of the following two
forms:

mapping = {
  chr1  = "hs_ref_chr1.fa.gz",
  chr2  = "hs_ref_chr2.fa.gz"
}

or

function mapping(sequence_region)
  return "hs_ref_"..sequence_region..".fa.gz"
end

The first form defines a Lua (http://www.lua.org/) table named ``mapping''
which maps each sequence region to the corresponding sequence file.
The second one defines a Lua function ``mapping'', which has to return the
sequence file name when it is called with the sequence_region as argument.
```

B.4 The `csa` Tool

```
$ gt csa -help
Usage: gt csa [option ...] [GFF3_file]
Transform spliced alignments from GFF3 file into consensus spliced alignments.

-join-length set join length for the spliced alignment clustering
             default: 300
-v           be verbose
             default: no
-o           redirect output to specified file
             default: undefined
-gzip        write gzip compressed output file
             default: no
-bzip2       write bzip2 compressed output file
             default: no
-force       force writing to output file
             default: no
-help        display help and exit
-version     display version information and exit

Example:
--------

Let's assume we have a GFF3 file 'csa_example_spliced_alignments.gff3'
containing the following four overlapping spliced alignments (represented as
genes with exons as children):

##gff-version   3
##sequence-region   seq 1 290
seq . gene 1 209 . + . ID=gene1
seq . exon 1 90 . + . Parent=gene1
seq . exon 110 190 . + . Parent=gene1
seq . exon 201 209 . + . Parent=gene1
###
seq . gene 1 290 . + . ID=gene2
seq . exon 1 90 . + . Parent=gene2
seq . exon 101 190 . + . Parent=gene2
seq . exon 201 290 . + . Parent=gene2
###
seq . gene 10 290 . + . ID=gene3
seq . exon 10 90 . + . Parent=gene3
seq . exon 110 190 . + . Parent=gene3
seq . exon 201 290 . + . Parent=gene3
###
seq . gene 181 290 . + . ID=gene4
seq . exon 181 190 . + . Parent=gene4
seq . exon 201 290 . + . Parent=gene4
###

To compute the consensus spliced alignments we call:

$ gt csa csa_example_spliced_alignments.gff3

Which returns:

##gff-version   3
##sequence-region   seq 1 290
seq gt csa gene 1 290 . + . ID=gene1
seq gt csa mRNA 1 290 . + . ID=mRNA1;Parent=gene1
seq gt csa exon 1 90 . + . Parent=mRNA1
seq gt csa exon 110 190 . + . Parent=mRNA1
seq gt csa exon 201 290 . + . Parent=mRNA1
seq gt csa mRNA 1 290 . + . ID=mRNA2;Parent=gene1
```

```
seq gt csa exon 1 90 . + . Parent=mRNA2
seq gt csa exon 101 190 . + . Parent=mRNA2
seq gt csa exon 201 290 . + . Parent=mRNA2
###
```

As one can see, they have been combined into a consensus spliced alignment (represented as genes with mRNAs as children which in turn have exons as children) with two alternative splice forms. The first and the third spliced alignment have been combined into the first alternative splice form (mRNA1) and the the second and the fourth spliced alignment into the second alternative splice form (mRNA2).

As one can see, the second exon from the first alternative splice form ist shorter than the corresponding exon from the second alternative splice form.

Report bugs to <gt-users@genometools.org>.

B.5 The `eval` Tool

```
$ gt eval -help
Usage: gt eval reference_file prediction_file
Compare annotation files and show accuracy measures (prediction vs. reference).

-nuc      evaluate nucleotide level (memory consumption is proportional to the
          input file sizes)
          default: yes
-ltr      evaluate a LTR retrotransposon prediction instead of a gene prediction
          (all LTR_retrotransposon elements are considered to have an
          undetermined strand)
          default: no
-ltrdelta set allowed delta for LTR borders to be considered equal
          default: 20
-v        be verbose
          default: no
-o        redirect output to specified file
          default: undefined
-gzip     write gzip compressed output file
          default: no
-bzip2    write bzip2 compressed output file
          default: no
-force    force writing to output file
          default: no
-help     display help and exit
-version  display version information and exit

The program shows sensitivity and specificity values for certain feature types
(e.g., gene, mRNA, and exon). For some feature types the number of missing and
wrong features of that type is also shown. Thereby, ``missing'' means the number
of features of that type from the ``reference'' without overlap to a feature of
that type from the ``prediction''. Vice versa, ``wrong'' denotes the number of
features of that type from the ``prediction'' without overlap to a feature of
that type from the ``reference''.

Report bugs to <gt-users@genometools.org>.
```

B.6 The `extractfeat` Tool

```
$ gt extractfeat -help
Usage: gt extractfeat [option ...] [GFF3_file]
Extract features given in GFF3 file from sequence file.

-type          set type of features to extract
               default: undefined
-join          join feature sequences in the same subgraph into a single one
               default: no
-translate     translate the features (of a DNA sequence) into protein
               default: no
-seqid         add sequence ID of extracted features to FASTA descriptions
               default: no
-target        add target ID(s) of extracted features to FASTA descriptions
               default: no
-seqfile       set the sequence file from which to extract the features
               default: undefined
-seqfiles      set the sequence files from which to extract the features
               use '--' to terminate the list of sequence files
-matchdesc     match the sequence descriptions from the input files for the
               desired sequence IDs (in GFF3)
               default: no
-usedesc       use sequence descriptions to map the sequence IDs (in GFF3) to
               actual sequence entries.
               If a description contains a sequence range (e.g.,
               III:1000001..2000000), the first  part is used as sequence ID
               ('III') and the first range position as offset ('1000001')
               default: no
-regionmapping set file containing sequence-region to sequence file mapping
               default: undefined
-v             be verbose
               default: no
-width         set output width for FASTA sequence printing
               (0 disables formatting)
               default: 0
-o             redirect output to specified file
               default: undefined
-gzip          write gzip compressed output file
               default: no
-bzip2         write bzip2 compressed output file
               default: no
-force         force writing to output file
               default: no
-help          display help and exit
-version       display version information and exit

File format for option -regionmapping:

The file supplied to option -regionmapping defines a ``mapping''.  A mapping
maps the sequence-regions given in the GFF3_file to a sequence file containing
the corresponding sequence. Mappings can be defined in one of the following two
forms:

mapping = {
  chr1  = "hs_ref_chr1.fa.gz",
  chr2  = "hs_ref_chr2.fa.gz"
}

or

function mapping(sequence_region)
  return "hs_ref_"..sequence_region..".fa.gz"
end
```

```
The first form defines a Lua (http://www.lua.org/) table named ``mapping''
which maps each sequence region to the corresponding sequence file.
The second one defines a Lua function ``mapping'', which has to return the
sequence file name when it is called with the sequence_region as argument.

Report bugs to <gt-users@genometools.org>.
```

B.7 The `gff3` Tool

```
$ gt gff3 -help
Usage: gt gff3 [option ...] [GFF3_file ...]
Parse, possibly transform, and output GFF3 files.

-sort                sort the GFF3 features (memory consumption is proportional
                     to the input file size(s))
                     default: no
-tidy                try to tidy the GFF3 files up during parsing
                     default: no
-retainids           when available, use the original IDs provided in the source
                     file
                     (memory consumption is proportional to the input file
                     size(s))
                     default: no
-checkids            make sure the ID attributes are unique within the scope of
                     each GFF3_file, as required by GFF3 specification
                     (memory consumption is proportional to the input file
                     size(s))
                     default: no
-addids              add missing "##sequence-region" lines automatically
                     default: yes
-fixregionboundaries automatically adjust "##sequence-region" lines to contain
                     all their features (memory consumption is proportional to
                     the input file size(s))
                     default: no
-addintrons          add intron features between existing exon features
                     default: no
-offset              transform all features by the given offset
                     default: undefined
-offsetfile          transform all features by the offsets given in file
                     default: undefined
-typecheck           check GFF3 types against "id" and "name" tags in given OBO
                     file
                     default: undefined
-show                show GFF3 output
                     default: yes
-v                   be verbose
                     default: no
-width               set output width for FASTA sequence printing
                     (0 disables formatting)
                     default: 0
-o                   redirect output to specified file
                     default: undefined
-gzip                write gzip compressed output file
                     default: no
-bzip2               write bzip2 compressed output file
                     default: no
-force               force writing to output file
                     default: no
-help                display help and exit
-version             display version information and exit

File format for option -offsetfile:

The file supplied to option -offsetfile defines a mapping table named
``offsets''. It maps the sequence-regions given in the GFF3_file to offsets.
It can be defined as follows:

offsets = {
  chr1  = 1000,
  chr2  = 500
}
```

When this example is used, all features with seqid ``chr1'' will be offset by
1000 and all features with seqid ``chr2'' by 500.

If -offsetfile is used, offsets for all sequence-regions contained in the given
GFF3 files must be defined.

Report bugs to <gt-users@genometools.org>.

B.8 The `gff3_to_gtf` Tool

```
$ gt gff3_to_gtf -help
Usage: gt gff3_to_gtf [GFF3_file ...]
Parse GFF3 file(s) and show them as GTF2.2.

-o       redirect output to specified file
         default: undefined
-gzip    write gzip compressed output file
         default: no
-bzip2   write bzip2 compressed output file
         default: no
-force   force writing to output file
         default: no
-help    display help and exit
-version display version information and exit

Report bugs to <gt-users@genometools.org>.
```

B.9 The `gff3validator` Tool

```
$ gt gff3validator -help
Usage: gt gff3validator [option ...] [GFF3_file ...]
Strictly validate given GFF3 files.

-typecheck check GFF3 types against "id" and "name" tags in given OBO file
           default: undefined
-help      display help and exit
-version   display version information and exit

Report bugs to <gt-users@genometools.org>.
```

B.10 The `gtf_to_gff3` Tool

```
$ gt gtf_to_gff3 -help
Usage: gt gtf_to_gff3 [GTF_file]
Parse GTF2.2 file and convert it to GFF3.

-tidy    try to tidy the GTF file up during parsing
         default: no
-o       redirect output to specified file
         default: undefined
-gzip    write gzip compressed output file
         default: no
-bzip2   write bzip2 compressed output file
         default: no
-force   force writing to output file
         default: no
-help    display help and exit
-version display version information and exit

Report bugs to <gt-users@genometools.org>.
```

B.11 The `id_to_md5` Tool

```
$ gt id_to_md5 -help
Usage: gt id_to_md5 [option ...] [GFF3_file ...]
Change sequence IDs in given GFF3 files to MD5 fingerprints of the corresponding sequences.

-seqfile       set the sequence file from which to extract the features
               default: undefined
-seqfiles      set the sequence files from which to extract the features
               use '--' to terminate the list of sequence files
-matchdesc     match the sequence descriptions from the input files for the
               desired sequence IDs (in GFF3)
               default: no
-usedesc       use sequence descriptions to map the sequence IDs (in GFF3) to
               actual sequence entries.
               If a description contains a sequence range (e.g.,
               III:1000001..2000000), the first  part is used as sequence ID
               ('III') and the first range position as offset ('1000001')
               default: no
-regionmapping set file containing sequence-region to sequence file mapping
               default: undefined
-subtargetids  substitute the target IDs with MD5 sums
               default: yes
-v             be verbose
               default: no
-o             redirect output to specified file
               default: undefined
-gzip          write gzip compressed output file
               default: no
-bzip2         write bzip2 compressed output file
               default: no
-force         force writing to output file
               default: no
-help          display help and exit
-version       display version information and exit

File format for option -regionmapping:

The file supplied to option -regionmapping defines a ``mapping''.  A mapping
maps the sequence-regions given in the GFF3_file to a sequence file containing
the corresponding sequence. Mappings can be defined in one of the following two
forms:

mapping = {
  chr1  = "hs_ref_chr1.fa.gz",
  chr2  = "hs_ref_chr2.fa.gz"
}

or

function mapping(sequence_region)
  return "hs_ref_"..sequence_region..".fa.gz"
end

The first form defines a Lua (http://www.lua.org/) table named ``mapping''
which maps each sequence region to the corresponding sequence file.
The second one defines a Lua function ``mapping'', which has to return the
sequence file name when it is called with the sequence_region as argument.

Report bugs to <gt-users@genometools.org>.
```

B.12 The `interfeat` Tool

```
$ gt interfeat -help
Usage: gt interfeat [option ...] [GFF3_file ...]
Add intermediary features between outside features in given GFF3 file(s).

-outside set outside type
         default: exon
-inter   set intermediary type
         default: intron
-o       redirect output to specified file
         default: undefined
-gzip    write gzip compressed output file
         default: no
-bzip2   write bzip2 compressed output file
         default: no
-force   force writing to output file
         default: no
-help    display help and exit
-version display version information and exit

Report bugs to <gt-users@genometools.org>.
```

B.13 The `md5_to_id` Tool

```
$ gt md5_to_id -help
Usage: gt md5_to_id [option ...] [GFF3_file ...]
Change MD5 fingerprints used as sequence IDs in given GFF3 files to ``regular'' ones.

-seqfile       set the sequence file from which to extract the features
               default: undefined
-seqfiles      set the sequence files from which to extract the features
               use '--' to terminate the list of sequence files
-matchdesc     match the sequence descriptions from the input files for the
               desired sequence IDs (in GFF3)
               default: no
-usedesc       use sequence descriptions to map the sequence IDs (in GFF3) to
               actual sequence entries.
               If a description contains a sequence range (e.g.,
               III:1000001..2000000), the first  part is used as sequence ID
               ('III') and the first range position as offset ('1000001')
               default: no
-regionmapping set file containing sequence-region to sequence file mapping
               default: undefined
-v             be verbose
               default: no
-o             redirect output to specified file
               default: undefined
-gzip          write gzip compressed output file
               default: no
-bzip2         write bzip2 compressed output file
               default: no
-force         force writing to output file
               default: no
-help          display help and exit
-version       display version information and exit

File format for option -regionmapping:

The file supplied to option -regionmapping defines a ``mapping''.  A mapping
maps the sequence-regions given in the GFF3_file to a sequence file containing
the corresponding sequence. Mappings can be defined in one of the following two
forms:

mapping = {
  chr1  = "hs_ref_chr1.fa.gz",
  chr2  = "hs_ref_chr2.fa.gz"
}

or

function mapping(sequence_region)
  return "hs_ref_"..sequence_region..".fa.gz"
end

The first form defines a Lua (http://www.lua.org/) table named ``mapping''
which maps each sequence region to the corresponding sequence file.
The second one defines a Lua function ``mapping'', which has to return the
sequence file name when it is called with the sequence_region as argument.

Report bugs to <gt-users@genometools.org>.
```

B.14 The `merge` Tool

```
$ gt merge -help
Usage: gt merge [option ...] [GFF3_file ...]
Merge sorted GFF3 files in sorted fashion.

-o       redirect output to specified file
         default: undefined
-gzip    write gzip compressed output file
         default: no
-bzip2   write bzip2 compressed output file
         default: no
-force   force writing to output file
         default: no
-help    display help and exit
-version display version information and exit

Report bugs to <gt-users@genometools.org>.
```

B.15 The `mergefeat` Tool

```
$ gt mergefeat -help
Usage: gt mergefeat [option ...] [GFF3_file ...]
Merge adjacent features of the same type in given GFF3 file(s).

-o       redirect output to specified file
         default: undefined
-gzip    write gzip compressed output file
         default: no
-bzip2   write bzip2 compressed output file
         default: no
-force   force writing to output file
         default: no
-help    display help and exit
-version display version information and exit

Report bugs to <gt-users@genometools.org>.
```

B.16 The `select` Tool

```
$ gt select -help
Usage: gt select [option ...] [GFF3_file ...]
Select certain features (specified by the used options) from given GFF3 file(s).

-seqid         select feature with the given sequence ID (all comments are
               selected).
               default: undefined
-source        select feature with the given source (the source is column 2 in
               regular GFF3 lines)
               default: undefined
-contain       select all features which are contained in the given range
               default: undefined
-overlap       select all features which do overlap with the given range
               default: undefined
-strand        select all top-level features(i.e., features without parents)
               whose strand equals the given one (must be one of '+-.?')
               default: undefined
-targetstrand  select all top-level features (i.e., features without parents)
               which have exactly one target attribute whose strand equals the
               given one (must be one of '+-.?')
               default: undefined
-targetbest    if multiple top-level features (i.e., features without parents)
               with exactly one target attribute have the same target_id, keep
               only the feature with the best score. If -targetstrand is used at
               the same time, this option is applied after -targetstrand.
               Memory consumption is proportional to the input file size(s).
               default: no
-hascds        select all top-level features which do have a CDS child
               default: no
-maxgenelength select genes up to the given maximum length
               default: undefined
-maxgenenum    select the first genes up to the given maximum number
               default: undefined
-mingenescore  select genes with the given minimum score
               default: undefined
-maxgenescore  select genes with the given maximum score
               default: undefined
-minaveragessp set the minimum average splice site probability
               default: undefined
-v             be verbose
               default: no
-o             redirect output to specified file
               default: undefined
-gzip          write gzip compressed output file
               default: no
-bzip2         write bzip2 compressed output file
               default: no
-force         force writing to output file
               default: no
-help          display help and exit
-version       display version information and exit

Report bugs to <gt-users@genometools.org>.
```

B.17 The `seqmutate` Tool

```
$ gt seqmutate -help
Usage: gt seqmutate [option ...] [sequence_file ...]
Mutate the sequences of the given sequence file(s).

-rate    set the mutation rate
         default: 1
-width   set output width for FASTA sequence printing
         (0 disables formatting)
         default: 0
-o       redirect output to specified file
         default: undefined
-gzip    write gzip compressed output file
         default: no
-bzip2   write bzip2 compressed output file
         default: no
-force   force writing to output file
         default: no
-help    display help and exit
-version display version information and exit

For each position in the given sequences it is randomly determined with
probability (mutation rate / 100) if the given position is mutated. If so, in
80% of the cases a substitution is performed, in 10% an insertion, and in 10% a
deletion, respectively. For substitution and insertion events, the nucleotide is
generated randomly without regard to the original nucleotide. That is,
resubstitutions are possible. This procedure equals the one described on page
1867 of the following paper

T.D. Wu and C.K. Watanabe. GMAP: a genomic mapping and alignment program for
mRNA and EST sequences. Bioinformatics, 21(9):1859-1875, 2005.

Report bugs to <gt-users@genometools.org>.
```

B.18 The `seqtransform` Tool

```
$ gt seqtransform -help
Usage: gt seqtransform [option ...] [sequence_file ...]
Perform simple transformations on the given sequence file(s).

-addstopaminos append stop amino acids ('*') to given protein sequences, if not
               already present
               default: no
-width         set output width for FASTA sequence printing
               (0 disables formatting)
               default: 0
-o             redirect output to specified file
               default: undefined
-gzip          write gzip compressed output file
               default: no
-bzip2         write bzip2 compressed output file
               default: no
-force         force writing to output file
               default: no
-help          display help and exit
-version       display version information and exit

Report bugs to <gt-users@genometools.org>.
```

B.19 The `sketch` Tool

```
$ gt sketch -help
Usage: gt sketch [option ...] image_file [GFF3_file ...]
Create graphical representation of GFF3 annotation files.

-pipe         use pipe mode (i.e., show all gff3 features on stdout)
              default: no
-flattenfiles do not group tracks by source file name and remove file names from
              track description
              default: no
-seqid        sequence region identifier
              default: first one in file
-start        start position
              default: first region start
-end          end position
              default: last region end
-width        target image width (in pixel)
              default: 800
-style        style file to use
              default: /home/gordon/genometools/bin/../gtdata/sketch/default.style
-format       output graphics format
              choose from png|pdf|svg|ps
              default: png
-input        input data format
              choose from gff|bed|gtf
              default: gff
-addintrons   add intron features between existing exon features (before
              drawing)
              default: no
-unsafe       enable unsafe mode for style file
              default: no
-v            be verbose
              default: no
-force        force writing to output file
              default: no
-help         display help and exit
-version      display version information and exit

Report bugs to <gt-users@genometools.org>.
```

B.20 The `splicesiteinfo` Tool

```
$ gt splicesiteinfo -help
Usage: gt splicesiteinfo [option ...] [GFF3_file ...]
Show information about splice sites given in GFF3 files.

-seqfile       set the sequence file from which to extract the features
               default: undefined
-seqfiles      set the sequence files from which to extract the features
               use '--' to terminate the list of sequence files
-matchdesc     match the sequence descriptions from the input files for the
               desired sequence IDs (in GFF3)
               default: no
-usedesc       use sequence descriptions to map the sequence IDs (in GFF3) to
               actual sequence entries.
               If a description contains a sequence range (e.g.,
               III:1000001..2000000), the first  part is used as sequence ID
               ('III') and the first range position as offset ('1000001')
               default: no
-regionmapping set file containing sequence-region to sequence file mapping
               default: undefined
-addintrons    add intron features between existing exon features
               (before computing the information to be shown)
               default: no
-o             redirect output to specified file
               default: undefined
-gzip          write gzip compressed output file
               default: no
-bzip2         write bzip2 compressed output file
               default: no
-force         force writing to output file
               default: no
-help          display help and exit
-version       display version information and exit

File format for option -regionmapping:

The file supplied to option -regionmapping defines a ``mapping''.  A mapping
maps the sequence-regions given in the GFF3_file to a sequence file containing
the corresponding sequence. Mappings can be defined in one of the following two
forms:

mapping = {
  chr1  = "hs_ref_chr1.fa.gz",
  chr2  = "hs_ref_chr2.fa.gz"
}

or

function mapping(sequence_region)
  return "hs_ref_"..sequence_region..".fa.gz"
end

The first form defines a Lua (http://www.lua.org/) table named ``mapping''
which maps each sequence region to the corresponding sequence file.
The second one defines a Lua function ``mapping'', which has to return the
sequence file name when it is called with the sequence_region as argument.

Report bugs to <gt-users@genometools.org>.
```

B.21 The `stat` Tool

```
$ gt stat -help
Usage: gt stat [option ...] [GFF3_file ...]
Show statistics about features contained in GFF3 files.

-genelengthdistri   show gene length distribution
                    default: no
-genescoredistri    show gene score distribution
                    default: no
-exonlengthdistri   show exon length distribution
                    default: no
-exonnumberdistri   show exon number distribution
                    default: no
-intronlengthdistri show intron length distribution
                    default: no
-cdslengthdistri    show CDS length distribution (measured in amino acids)
                    default: no
-source             show the set of used source tags (column 2 in regular GFF3
                    lines)
                    default: no
-addintrons         add intron features between existing exon features (before
                    computing stats)
                    default: no
-v                  be verbose
                    default: no
-o                  redirect output to specified file
                    default: undefined
-gzip               write gzip compressed output file
                    default: no
-bzip2              write bzip2 compressed output file
                    default: no
-force              force writing to output file
                    default: no
-help               display help and exit
-version            display version information and exit

Report bugs to <gt-users@genometools.org>.
```

B.22 The `uniq` Tool

```
$ gt uniq -help
Usage: gt uniq [option ...] [GFF3_file]
Filter out repeated feature node graphs in a sorted GFF3 file.

-v       be verbose
         default: no
-o       redirect output to specified file
         default: undefined
-gzip    write gzip compressed output file
         default: no
-bzip2   write bzip2 compressed output file
         default: no
-force   force writing to output file
         default: no
-help    display help and exit
-version display version information and exit

A depth-first traversal of a feature node graph starts at the top-level feature
node (or pseudo-node) and explores as far along each branch as possible before
backtracking. Let's assume that the feature nodes are stored in a list in the
order of their traversal (called the ``feature node list'').

Two feature node graphs are considered to be repeated if their feature node list
(from the depth-first traversal) have the same length and each feature node pair
(from both lists at the same position) is ``similar''.

Two feature nodes are ``similar'', if they have the same sequence ID, feature
type, range, strand, and phase.

For such a repeated feature node graph the one with the higher score (of the
top-level feature) is kept. If only one of the feature node graphs has a defined
score, this one is kept.

Report bugs to <gt-users@genometools.org>.
```

Appendix C

GenomeTools API Reference

Table of Classes

Table of Modules

C.1 Class GtAddIntronsStream

Implements the GtNodeStream interface. A GtAddIntronsStream inserts new feature nodes with type *intron* between existing feature nodes with type *exon*. This is a special case of the GtInterFeatureStream.

Methods

`GtNodeStream* gt_add_introns_stream_new(GtNodeStream *in_stream)`

Create a GtAddIntronsStream* which inserts feature nodes of type *intron* between feature nodes of type *exon* it retrieves from in_stream and returns them.

C.2 Class GtArray

GtArray objects are generic arrays for elements of a certain size which grow on demand.

Methods

`GtArray* gt_array_new(size_t size_of_elem)`

Return a new `GtArray` object whose elements have the size `size_of_elem`.

`GtArray* gt_array_ref(GtArray *array)`

Increase the reference count for `array` and return it. If `array` is `NULL`, `NULL` is returned without any side effects.

`GtArray* gt_array_clone(const GtArray *array)`

Return a clone of `array`.

`void* gt_array_get(const GtArray *array, unsigned long index)`

Return pointer to element number `index` of `array`. `index` has to be smaller than `gt_array_size(array)`.

`void* gt_array_get_first(const GtArray *array)`

Return pointer to first element of `array`.

`void* gt_array_get_last(const GtArray *array)`

Return pointer to last element of `array`.

`void* gt_array_pop(GtArray *array)`

Return pointer to last element of `array` and remove it from `array`.

`void* gt_array_get_space(const GtArray *array)`

Return pointer to the internal space of `array` where the elements are stored.

`#define gt_array_add(array, elem)`

Add element `elem` to `array`. The size of `elem` must equal the given element size when the `array` was created and is determined automatically with the `sizeof` operator.

`void gt_array_add_elem(GtArray *array, void *elem, size_t size_of_elem)`

Add element `elem` with size `size_of_elem` to `array`. `size_of_elem` must equal the given element size when the `array` was created. Usually, this method is not used directly and the macro `gt_array_add()` is used instead.

`void gt_array_add_array(GtArray *dest, const GtArray *src)`

Add all elements of array `src` to the array `dest`. The element sizes of both arrays must be equal.

`void gt_array_rem(GtArray *array, unsigned long index)`

Remove element with number `index` from `array` in O(`gt_array_size(array)`) time. `index` has to be smaller than `gt_array_size(array)`.

`void gt_array_rem_span(GtArray *array, unsigned long frompos, unsigned long topos)`

Remove elements starting with number `frompos` up to (and including) `topos` from `array` in O(`gt_array_size(array)`) time. `frompos` has to be smaller or equal than `topos` and both have to be smaller than `gt_array_size(array)`.

`void gt_array_reverse(GtArray *array)`

Reverse the order of the elements in `array`.

`void gt_array_set_size(GtArray *array, unsigned long size)`

Set the size of `array` to `size`. `size` must be smaller or equal than `gt_array_size(array)`.

`void gt_array_reset(GtArray *array)`

Reset the `array`. That is, afterwards the array has size 0.

`size_t gt_array_elem_size(const GtArray *array)`

Return the size of the elements stored in `array`.

`unsigned long gt_array_size(const GtArray *array)`

Return the number of elements in `array`. If `array` equals `NULL`, 0 is returned.

`void gt_array_sort(GtArray *array, GtCompare compar)`

Sort `array` with the given compare function `compar`.

`void gt_array_sort_stable(GtArray *array, GtCompare compar)`

Sort `array` in a stable way with the given compare function `compar`.

`void gt_array_sort_with_data(GtArray *array, GtCompareWithData compar, void *data)`

Sort `array` with the given compare function `compar`. Passes a pointer with userdata `data` to `compar`.

`void gt_array_sort_stable_with_data(GtArray *array, GtCompareWithData compar, void *data)`

Sort `array` in a stable way with the given compare function `compar`. Passes a pointer with userdata `data` to `compar`.

`int gt_array_cmp(const GtArray *array_a, const GtArray *array_b)`

Compare the content of `array_a` with the content of `array_b`. `array_a` and `array_b` must have the same `gt_array_size()` and `gt_array_elem_size()`.

`void gt_array_delete(GtArray *array)`

Decrease the reference count for `array` or delete it, if this was the last reference.

C.3 Class GtBEDInStream

Implements the `GtNodeStream` interface. A `GtBEDInStream` allows one to parse a BED file and return it as a stream of `GtGenomeNode` objects.

Methods

`GtNodeStream* gt_bed_in_stream_new(const char *filename)`

Return a `GtBEDInStream` object which subsequently reads the BED file with the given `filename`. If `filename` equals `NULL`, the BED data is read from `stdin`.

`void gt_bed_in_stream_set_feature_type(GtBEDInStream *bed_in_stream, const char *type)`

Create BED features parsed by `bed_in_stream` with given `type` (instead of the default "BED_feature").

`void gt_bed_in_stream_set_thick_feature_type(GtBEDInStream *bed_in_stream, const char *type)`

Create thick BED features parsed by `bed_in_stream` with given `type` (instead of the default "BED_thick_feature").

`void gt_bed_in_stream_set_block_type(GtBEDInStream *bed_in_stream, const char *type)`

Create BED blocks parsed by `bed_in_stream` with given `type` (instead of the default "BED_block").

C.4 Class GtBittab

Implements arbitrary-length bit arrays and various operations on them.

Methods

`GtBittab* gt_bittab_new(unsigned long num_of_bits)`

Return a new `GtBittab` of length `num_of_bits`, initialised to 0.

`void gt_bittab_set_bit(GtBittab *bittab, unsigned long i)`

Set bit `i` in `bittab` to 1.

`void gt_bittab_unset_bit(GtBittab *bittab, unsigned long i)`

Set bit `i` in `bittab` to 0.

`void gt_bittab_complement(GtBittab *bittab_a, const GtBittab *bittab_b)`

Set `bittab_a` to be the complement of `bittab_b`.

`void gt_bittab_equal(GtBittab *bittab_a, const GtBittab *bittab_b)`

Set `bittab_a` to be equal to `bittab_b`.

`void gt_bittab_and(GtBittab *bittab_a, const GtBittab *bittab_b, const GtBittab *bittab_c)`

Set `bittab_a` to be the bitwise AND of `bittab_b` and `bittab_c`.

`void gt_bittab_or(GtBittab *bittab_a, const GtBittab *bittab_b, const GtBittab *bittab_c)`

Set `bittab_a` to be the bitwise OR of `bittab_b` and `bittab_c`.

`void gt_bittab_nand(GtBittab *bittab_a, const GtBittab *bittab_b, const GtBittab *bittab_c)`

Set `bittab_a` to be `bittab_b` NAND `bittab_c`.

`void gt_bittab_and_equal(GtBittab *bittab_a, const GtBittab *bittab_b)`

Set `bittab_a` to be the bitwise AND of `bittab_a` and `bittab_b`.

`void gt_bittab_or_equal(GtBittab *bittab_a, const GtBittab *bittab_b)`

Set `bittab_a` to be the bitwise OR of `bittab_a` and `bittab_b`.

`void gt_bittab_shift_left_equal(GtBittab *bittab)`

Shift `bittab` by one position to the left.

`void gt_bittab_shift_right_equal(GtBittab *bittab)`

Shift `bittab` by one position to the right.

`void gt_bittab_unset(GtBittab *bittab)`

Set all bits in `bittab` to 0.

`void gt_bittab_show(const GtBittab *bittab, FILE *fp)`

Output a representation of `bittab` to `fp`.

`void gt_bittab_get_all_bitnums(const GtBittab *bittab, GtArray *array)`

Fill `array` with the indices of all set bits in `bittab`.

`bool gt_bittab_bit_is_set(const GtBittab *bittab, unsigned long i)`

Return `true` if bit `i` is set in `bittab`.

`bool gt_bittab_cmp(const GtBittab *bittab_a, const GtBittab *bittab_b)`

Return `true` if `bittab_a` and `bittab_b` are identical.

`unsigned long gt_bittab_get_first_bitnum(const GtBittab *bittab)`

Return the index of the first set bit in `bittab`.

`unsigned long gt_bittab_get_last_bitnum(const GtBittab *bittab)`

Return the index of the last set bit in `bittab`.

`unsigned long gt_bittab_get_next_bitnum(const GtBittab *bittab, unsigned long i)`

Return the index of the next set bit in `bittab` with an index greater than `i`.

`unsigned long gt_bittab_count_set_bits(const GtBittab *bittab)`

Return the number of set bits in `bittab`.

`unsigned long gt_bittab_size(GtBittab *bittab)`

Return the total number of bits of `bittab`.

`void gt_bittab_delete(GtBittab *bittab)`

Delete `bittab`.

C.5 Class GtCDSStream

Implements the `GtNodeStream` interface. A `GtCDSStream` determines the coding sequence (CDS) for sequences determined by feature nodes of type *exon* and adds them as feature nodes of type *CDS*.

Methods

`GtNodeStream* gt_cds_stream_new(GtNodeStream *in_stream, GtRegionMapping *region_mapping, unsigned int minorflen, const char *source, bool start_codon, bool final_stop_codon, bool generic_star_codons)`

Create a `GtCDSStream*` which determines the coding sequence (CDS) for sequences determined by feature nodes of type *exon* it retrieves from `in_stream`, adds them as feature nodes of type *CDS* and returns all nodes. `region_mapping` is used to map the sequence IDs of the feature nodes to the regions of the actual sequences. `minorflen` is the minimum length an ORF must have in order to be added. The CDS features are created with the given `source`. If `start_codon` equals `true` an ORF must begin with a start codon, otherwise it can start at any position. If `final_stop_codon` equals `true` the final ORF must end with a stop codon. If `generic_start_codons` equals `true`, the start codons of the standard translation scheme are used as start codons (otherwise the amino acid 'M' is regarded as a start codon).

C.6 Class GtCSAStream

Implements the `GtNodeStream` interface. A `GtCSAStream` takes spliced alignments and transforms them into consensus spliced alignments.

Methods

`GtNodeStream* gt_csa_stream_new(GtNodeStream *in_stream, unsigned long join_length)`

Create a `GtCSAStream*` which takes spliced alignments from its `in_stream` (which are at most `join_length` many bases apart), transforms them into consensus spliced alignments, and returns them.

C.7 Class GtCommentNode

Implements the `GtGenomeNode` interface. Comment nodes correspond to comment lines in GFF3 files (i.e., lines which start with a single "`#`").

Methods

`GtGenomeNode* gt_comment_node_new(const char *comment)`

Return a new `GtCommentNode` object representing a `comment`. Please note that the single leading "#" which denotes comment lines in GFF3 files should not be part of `comment`.

`const char* gt_comment_node_get_comment(const GtCommentNode *comment_node)`

Return the comment stored in `comment_node`.

C.8 Class GtCstrTable

Implements a table of C strings.

Methods

`GtCstrTable* gt_cstr_table_new(void)`

Return a new `GtCstrTable` object.

`void gt_cstr_table_add(GtCstrTable *table, const char *cstr)`

Add `cstr` to `table`. `table` must not already contain `cstr`!

`const char* gt_cstr_table_get(const GtCstrTable *table, const char *cstr)`

If a C string equal to `cstr` is contained in `table`, it is returned. Otherwise `NULL` is returned.

`GtStrArray* gt_cstr_table_get_all(const GtCstrTable *table)`

Return a `GtStrArray*` which contains all `cstr`s added to `table` in alphabetical order. The caller is responsible to free it!

`void gt_cstr_table_remove(GtCstrTable *table, const char *cstr)`

Remove `cstr` from `table`.

`void gt_cstr_table_reset(GtCstrTable *table)`

Reset `table` (that is, remove all contained C strings).

`void gt_cstr_table_delete(GtCstrTable *table)`

Delete C string `table`.

C.9 Class GtDlist

A double-linked list which is sorted according to a `GtCompare` compare function (`qsort(3)`-like, only if one was supplied to the constructor).

Methods

`GtDlist* gt_dlist_new(GtCompare compar)`

Return a new `GtDlist` object sorted according to `compar` function. If `compar` equals `NULL`, no sorting is enforced.

`GtDlistelem* gt_dlist_first(const GtDlist *dlist)`

Return the first `GtDlistelem` object in `dlist`.

`GtDlistelem* gt_dlist_last(const GtDlist *dlist)`

Return the last `GtDlistelem` object in `dlist`.

`GtDlistelem* gt_dlist_find(const GtDlist *dlist, void *data)`

Return the first `GtDlistelem` object in `dlist` which contains data identical to `data`. Takes O(n) time.

`unsigned long gt_dlist_size(const GtDlist *dlist)`

Return the number of `GtDlistelem` objects in `dlist`.

`void gt_dlist_add(GtDlist *dlist, void *data)`

Add a new `GtDlistelem` object containing `data` to `dlist`. Usually O(n), but O(1) if data is added in sorted order.

`void gt_dlist_remove(GtDlist *dlist, GtDlistelem *dlistelem)`

Remove `dlistelem` from `dlist` and free it.

`int gt_dlist_example(GtError *err)`

Example for usage of the `GtDlist` class.

`void gt_dlist_delete(GtDlist *dlist)`

Delete `dlist`.

C.10 Class GtDlistelem

`GtDlistelem* gt_dlistelem_next(const GtDlistelem *dlistelem)`

Return the successor of `dlistelem`, or `NULL` if the element is the last one in the `GtDlist`.

`GtDlistelem* gt_dlistelem_previous(const GtDlistelem *dlistelem)`

Return the predecessor of `dlistelem`, or `NULL` if the element is the first one in the `GtDlist`.

`void* gt_dlistelem_get_data(const GtDlistelem *dlistelem)`

Return the data pointer attached to `dlistelem`.

C.11 Class GtError

This class is used for the handling of **user errors** in *GenomeTools*. Thereby, the actual `GtError` object is used to store the *error message* while it is signaled by the return value of the called function, if an error occured.
By convention in *GenomeTools*, the `GtError` object is always passed into a function as the last parameter and -1 (or `NULL` for constructors) is used as return value to indicate that an error occurred. Success is usually indicated by 0 as return value or via a non-`NULL` object pointer for constructors.
It is possible to use `NULL` as an `GtError` object, if one is not interested in the actual error message.
Functions which do not get an `GtError` object cannot fail due to a user error and it is not necessary to check their return code for an error condition.

Methods

`GtError* gt_error_new(void)`

Return a new `GtError` object

`#define gt_error_check(err)`

Insert an assertion to check that the error `err` is not set or is `NULL`. This macro should be used at the beginning of every routine which has an `GtError*` argument to make sure the error propagation has been coded correctly.

`void gt_error_set(GtError *err, const char *format, ...)`

Set the error message stored in `err` according to `format` (as in `printf(3)`).

`void gt_error_vset(GtError *err, const char *format, va_list ap)`

Set the error message stored in `err` according to `format` (as in `vprintf(3)`).

`void gt_error_set_nonvariadic(GtError *err, const char *msg)`

Set the error message stored in `err` to `msg`.

`bool gt_error_is_set(const GtError *err)`

Return `true` if the error `err` is set, `false` otherwise.

`void gt_error_unset(GtError *err)`

Unset the error `err`.

`const char* gt_error_get(const GtError *err)`

Return the error string stored in `err` (the error must be set).

`void gt_error_delete(GtError *err)`

Delete the error object `err`.

C.12 Class GtExtractFeatureStream

Implements the `GtNodeStream` interface. A `GtExtractFeatureStream` extracts the corresponding sequences of features.

Methods

```
GtNodeStream* gt_extract_feature_stream_new(GtNodeStream
*in_stream, GtRegionMapping *region_Mapping, const char *type,
bool join, bool translate, bool seqid, bool target, unsigned
long width, GtFile *outfp)
```

Create a `GtExtractFeatureStream*` which extracts the corresponding sequences of feature nodes (of the given `type`) it retrieves from `in_stream` and writes them in FASTA format (with the given `width`) to `outfp`. If `join` is `true`, features of the given `type` are joined together before the sequence is extracted. If `translate` is `true`, the sequences are translated into amino acid sequences before they are written to `outfp`. If `seqid` is `true` the sequence IDs of the extracted features are added to the FASTA header. If `target` is `true` the target IDs of the extracted features are added to the FASTA header. Takes ownership of `region_mapping`!

C.13 Class GtFeatureNode

Implements the `GtGenomeNode` interface. A single feature node corresponds to a GFF3 feature line (i.e., a line which does not start with `#`). Part-of relationships (which are realized in GFF3 with the `Parent` and `ID` attributes) are realized in the C *GenomeTools* API with the `gt_feature_node_add_child()` method.

Besides the "mere" feature nodes two "special" feature nodes exist: multi-features and pseudo-features.
Multi-features represent features which span multiple lines (it is indicated in GFF3 files by the fact, that each line has the same ID attribute).
To check if a feature is a multi-feature use the method `gt_feature_node_is_multi()`.
Multi-features are connected via a "representative". That is, two features are part of the same multi-feature if they have the same representative. The feature node representative can be be retrieved via the `gt_feature_node_get_multi_representative()` method.
Pseudo-features became a technical necessity to be able to pass related top-level features as a single entity through the streaming machinery. There are two cases in which a pseudo-feature has to be introduced.
First, if a multi-feature has no parent. In this case all features which comprise the multi-feature become the children of a pseudo-feature.
Second, if two or more top-level features have the same children (and are thereby connected). In this case all these top-level features become the children of a pseudo-feature.
It should be clear from the explanation above that pseudo-features make only sense as top-level features (a fact which is enforced in the code).
Pseudo-features are typically ignored during a traversal to give the illusion that they do not exist.

Methods

`GtGenomeNode* gt_feature_node_new(GtStr *seqid, const char *type, unsigned long start, unsigned long end, GtStrand strand)`

> Return an new `GtFeatureNode` object on sequence with ID `seqid` and type `type` which lies from `start` to `end` on strand `strand`. The `GtFeatureNode*` stores a new reference to `seqid`, so make sure you do not modify the original `seqid` afterwards! `start` and `end` always refer to the forward strand, therefore `start` has to be smaller or equal than `end`.

`GtGenomeNode* gt_feature_node_new_pseudo(GtStr *seqid, unsigned long start, unsigned long end, GtStrand strand)`

> Return a new pseudo-`GtFeatureNode` object on sequence with ID `seqid` which lies from `start` to `end` on strand `strand`. Pseudo-features do not have a type. The <GtFeatureNode > stores a new reference to `seqid`, so make sure you do not modify the original `seqid` afterwards. `start` and `end` always refer to the forward strand, therefore `start` has to be smaller or equal than `end`.

`GtGenomeNode* gt_feature_node_new_pseudo_template(GtFeatureNode *feature_node)`

> Return a new pseudo-`GtFeatureNode` object which uses `feature_node` as template. That is, the sequence ID, range, strand, and source are taken from `feature_node`.

`GtGenomeNode* gt_feature_node_new_standard_gene(void)`

Return the "standard gene" (mainly for testing purposes).

`void gt_feature_node_add_child(GtFeatureNode *parent, GtFeatureNode *child)`

Add `child` feature node to `parent` feature node. `parent` takes ownership of `child`.

`const char* gt_feature_node_get_source(const GtFeatureNode *feature_node)`

Return the source of `feature_node`. If no source has been set, "." is returned. Corresponds to column 2 of GFF3 feature lines.

`void gt_feature_node_set_source(GtFeatureNode *feature_node, GtStr *source)`

Set the `source` of `feature_node`. Stores a new reference to `source`. Corresponds to column 2 of GFF3 feature lines.

`bool gt_feature_node_has_source(const GtFeatureNode *feature_node)`

Return `true` if `feature_node` has a defined source (i.e., on different from "."). `false` otherwise.

`const char* gt_feature_node_get_type(const GtFeatureNode *feature_node)`

Return the type of `feature_node`. Corresponds to column 3 of GFF3 feature lines.

`void gt_feature_node_set_type(GtFeatureNode *feature_node, const char *type)`

Set the type of `feature_node` to `type`.

`bool gt_feature_node_has_type(GtFeatureNode *feature_node, const char *type)`

Return `true` if `feature_node` has given `type`, `false` otherwise.

`unsigned long gt_feature_node_number_of_children(const GtFeatureNode *feature_node)`

Return the number of children for given `feature_node`.

`unsigned long gt_feature_node_number_of_children_of_type(const GtFeatureNode *parent, const GtFeatureNode *node)`

Return the number of children of type `node` for given GtFeatureNode `parent`.

`bool gt_feature_node_score_is_defined(const GtFeatureNode *feature_node)`

Return `true` if the score of `feature_node` is defined, `false` otherwise.

`float gt_feature_node_get_score(const GtFeatureNode *feature_node)`

Return the score of `feature_node`. The score has to be defined. Corresponds to column 6 of GFF3 feature lines.

`void gt_feature_node_set_score(GtFeatureNode *feature_node, float score)`

Set the score of `feature_node` to `score`.

`void gt_feature_node_unset_score(GtFeatureNode *feature_node)`

Unset the score of `feature_node`.

`GtStrand gt_feature_node_get_strand(const GtFeatureNode *feature_node)`

Return the strand of `feature_node`. Corresponds to column 7 of GFF3 feature lines.

`void gt_feature_node_set_strand(GtFeatureNode *feature_node, GtStrand strand)`

Set the strand of `feature_node` to `strand`.

`GtPhase gt_feature_node_get_phase(const GtFeatureNode *feature_node)`

Return the phase of `feature_node`. Corresponds to column 8 of GFF3 feature lines.

`void gt_feature_node_set_phase(GtFeatureNode *feature_node, GtPhase phase)`

Set the phase of `feature_node` to `phase`.

`const char* gt_feature_node_get_attribute(const GtFeatureNode *feature_node, const char *name)`

Return the attribute of `feature_node` with the given `name`. If no such attribute has been added, `NULL` is returned. The attributes are stored in column 9 of GFF3 feature lines.

`GtStrArray* gt_feature_node_get_attribute_list(const GtFeatureNode *feature_node)`

Return a string array containing the used attribute names of `feature_node`. The caller is responsible to free the returned `GtStrArray*`.

`void gt_feature_node_add_attribute(GtFeatureNode *feature_node, const char *tag, const char *value)`

Add attribute `tag=value` to `feature_node`. `tag` and `value` must at least have length 1. `feature_node` must not contain an attribute with the given `tag` already. You should not add Parent and ID attributes, use `gt_feature_node_add_child()` to denote part-of relationships.

`void gt_feature_node_set_attribute(GtFeatureNode* feature_node, const char *tag, const char *value)`

Set attribute `tag` to new `value` in `feature_node`, if it exists already. Otherwise the attribute `tag=value` is added to `feature_node`. `tag` and `value` must at least have length 1. You should not set Parent and ID attributes, use `gt_feature_node_add_child()` to denote part-of relationships.

`void gt_feature_node_remove_attribute(GtFeatureNode* feature_node, const char *tag)`

Remove attribute `tag` from `feature_node`. `feature_node` must contain an attribute with the given `tag` already! You should not remove Parent and ID attributes.

`bool gt_feature_node_is_multi(const GtFeatureNode *feature_node)`

Return `true` if `feature_node` is a multi-feature, `false` otherwise.

`bool gt_feature_node_is_pseudo(const GtFeatureNode *feature_node)`

Return `true` if `feature_node` is a pseudo-feature, `false` otherwise.

`void gt_feature_node_make_multi_representative(GtFeatureNode *feature_node)`

Make `feature_node` the representative of a multi-feature. Thereby `feature_node` becomes a multi-feature.

`void gt_feature_node_set_multi_representative(GtFeatureNode *feature_node, GtFeatureNode *representative)`

Set the multi-feature representative of `feature_node` to `representative`. Thereby `feature_node` becomes a multi-feature.

`void gt_feature_node_unset_multi(GtFeatureNode *feature_node)`

Unset the multi-feature status of `feature_node` and remove its multi-feature representative.

`GtFeatureNode* gt_feature_node_get_multi_representative(GtFeatureNode *feature_node)`

Return the representative of the multi-feature `feature_node`.

`bool gt_feature_node_is_similar(const GtFeatureNode *feature_node_a, const GtFeatureNode *feature_node_b)`

Returns `true`, if the given `feature_node_a` has the same seqid, feature type, range, strand, and phase as `feature_node_b`. Returns `false` otherwise.

`void gt_feature_node_mark(GtFeatureNode*)`

Marks the given `feature_node`.

`void gt_feature_node_unmark(GtFeatureNode*)`

If the given `feature_node` is marked it will be unmarked.

`bool gt_feature_node_contains_marked(GtFeatureNode *feature_node)`

Returns `true` if the given `feature_node` graph contains a marked node.

`bool gt_feature_node_is_marked(const GtFeatureNode *feature_node)`

Returns `true` if the (top-level) `feature_node` is marked.

C.14 Class GtFeatureNodeIterator

`GtFeatureNodeIterator* gt_feature_node_iterator_new(const GtFeatureNode *feature_node)`

Return a new `GtFeatureNodeIterator*` which performs a depth-first traversal of `feature_node` (including `feature_node` itself). It ignores pseudo-features.

`GtFeatureNodeIterator* gt_feature_node_iterator_new_direct(const GtFeatureNode *feature_node)`

Return a new `GtFeatureNodeIterator*` which iterates over all direct children of `feature_node` (without `feature_node` itself).

`GtFeatureNode* gt_feature_node_iterator_next(GtFeatureNodeIterator *feature_node_iterator)`

Return the next `GtFeatureNode*` in `feature_node_iterator` or `NULL` if none exists.

`void gt_feature_node_iterator_delete(GtFeatureNodeIterator *feature_node_iterator)`

Delete `feature_node_iterator`.

C.15 Class GtFile

This class defines (generic) files in *GenomeTools*. A generic file is is a file which either uncompressed or compressed (with gzip or bzip2). A `NULL`-pointer as generic file implies `stdout`.

Methods

`GtFile* gt_file_new(const char *path, const char *mode, GtError *err)`

Return a new `GtFile` object for the given `path` and open the underlying file handle with given `mode`. Returns `NULL` and sets `err` accordingly, if the file `path` could not be opened. The compression mode is determined by the ending of `path` (gzip compression if it ends with '.gz', bzip2 compression if it ends with '.bz2', and uncompressed otherwise).

`void gt_file_xprintf(GtFile *file, const char *format, ...)`

`printf(3)` for generic `file`.

`void gt_file_xfputs(const char *cstr, GtFile *file)`

Write `\0`-terminated C string `cstr` to `file`. Similar to `fputs(3)`, but terminates on error.

`int gt_file_xfgetc(GtFile *file)`

Return next character from `file` or `EOF`, if end-of-file is reached.

`int gt_file_xread(GtFile *file, void *buf, size_t nbytes)`

Read up to `nbytes` from generic `file` and store result in `buf`, returns bytes read.

`void gt_file_xwrite(GtFile *file, void *buf, size_t nbytes)`

Write `nbytes` from `buf` to given generic `file`.

`void gt_file_xrewind(GtFile *file)`

Rewind the generic `file`.

`void gt_file_delete(GtFile *file)`

Close the underlying file handle and destroy the `file` object.

C.16 Class GtGFF3InStream

Implements the `GtNodeStream` interface. A `GtGFF3InStream` parses GFF3 files and returns them as a stream of `GtGenomeNode` objects.

Methods

`GtNodeStream* gt_gff3_in_stream_new_unsorted(int num_of_files, const char **filenames)`

Return a `GtGFF3InStream` object which subsequently reads the `num_of_files` many GFF3 files denoted in `filenames`. The GFF3 files do not have to be sorted. If `num_of_files` is 0 or a file name is "-", it is read from `stdin`. The memory footprint is O(file size) in the worst-case.

`GtNodeStream* gt_gff3_in_stream_new_sorted(const char *filename)`

Create a `GtGFF3InStream*` which reads the sorted GFF3 file denoted by `filename`. If filename is `NULL`, it is read from `stdin`. The memory footprint is O(1) on average.

`void gt_gff3_in_stream_check_id_attributes(GtGFF3InStream *gff3_in_stream)`

Make sure all ID attributes which are parsed by `gff3_in_stream` are correct. Increases the memory footprint to O(file size).

`void gt_gff3_in_stream_enable_tidy_mode(GtGFF3InStream *gff3_in_stream)`

Enable tidy mode for `gff3_in_stream`. That is, the GFF3 parser tries to tidy up features which would normally lead to an error.

`void gt_gff3_in_stream_show_progress_bar(GtGFF3InStream *gff3_in_stream)`

Show progress bar on `stdout` to convey the progress of parsing the GFF3 files underlying `gff3_in_stream`.

C.17 Class GtGFF3OutStream

Implements the `GtNodeStream` interface. A `GtGFF3OutStream` produces GFF3 output. It automatically inserts termination lines at the appropriate places.

Methods

`GtNodeStream* gt_gff3_out_stream_new(GtNodeStream *in_stream, GtFile *outfp)`

Create a `GtGFF3OutStream*` which uses `in_stream` as input. It shows the nodes passed through it as GFF3 on `outfp`.

`void gt_gff3_out_stream_set_fasta_width(GtGFF3OutStream *gff3_out_stream, unsigned long fasta_width)`

Set the width with which the FASTA sequences of `GtSequenceNode`s passed through `gff3_out_stream` are shown to `fasta_width`. Per default, each FASTA entry is shown on a single line.

`void gt_gff3_out_stream_retain_id_attributes(GtGFF3OutStream *gff3_out_stream)`

If this method is called upon `gff3_out_stream`, use the original ID attributes provided in the input (instead of creating new ones, which is the default). Memory consumption for `gff3_out_stream` is raised from O(1) to O(`input_size`), because bookkeeping of used IDs becomes necessary to avoid ID collisions.

C.18 Class GtGFF3Parser

A `GtGFF3Parser` can be used to parse GFF3 files and convert them into `GtGenomeNode` objects. If the GFF3 files do not contain the encouraged sequence-region meta directives, the GFF3 parser introduces the corresponding region nodes automatically. This is a low-level class and it is usually not used directly. Normally, a `GtGFF3InStream` is used to parse GFF3 files.

Methods

`GtGFF3Parser* gt_gff3_parser_new(GtTypeChecker *type_checker)`

Return a new `GtGFF3Parser` object with optional `type_checker`. If a `type_checker` was given, the `GtGFF3Parser` stores a new reference to it internally and uses the `type_checker` to check types during parsing.

`void gt_gff3_parser_check_id_attributes(GtGFF3Parser *gff3_parser)`

Enable ID attribute checking in `gff3_parser`. Thereby, the memory consumption of the `gff3_parser` becomes proportional to the input file size(s).

`void gt_gff3_parser_check_region_boundaries(GtGFF3Parser *gff3_parser)`

Enable sequence region boundary checking in `gff3_parser`. That is, encountering features outside the sequence region boundaries will result in an error.

`void gt_gff3_parser_do_not_check_region_boundaries(GtGFF3Parser *gff3_parser)`

Disable sequence region boundary checking in `gff3_parser`. That is, features outside the sequence region boundaries will be permitted.

`void gt_gff3_parser_set_offset(GtGFF3Parser *gff3_parser, long offset)`

Transform all features parsed by `gff3_parser` by the given `offset`.

`void gt_gff3_parser_set_type_checker(GtGFF3Parser *gff3_parser, GtTypeChecker *type_checker)`

Set `type_checker` used by `gff3_parser`.

`void gt_gff3_parser_enable_tidy_mode(GtGFF3Parser *gff3_parser)`

Enable the tidy mode in `gff3_parser`. In tidy mode the `gff3_parser` parser tries to tidy up features which would normally lead to a parse error.

`int gt_gff3_parser_parse_genome_nodes(GtGFF3Parser *gff3_parser, int *status_code, GtQueue *genome_nodes, GtCstrTable *used_types, GtStr *filenamestr, unsigned long long *line_number, GtFile *fpin, GtError *err)`

Use `gff3_parser` to parse genome nodes from file pointer `fpin`. `status_code` is set to 0 if at least one genome node was created (and stored in `genome_nodes`) and to `EOF` if no further genome nodes could be parsed from `fpin`. Every encountered (genome feature) type is recorded in the C string table `used_types`. The parser uses the given `filenamestr` to store the file name of `fpin` in the created genome nodes or to give the correct filename in error messages, if necessary. `line_number` is increased accordingly during parsing and has to be set to 0 before parsing a new `fpin`. If an error occurs during parsing this method returns -1 and sets `err` accordingly.

`void gt_gff3_parser_reset(GtGFF3Parser *gff3_parser)`

Reset the `gff3_parser` (necessary if the input file is switched).

`void gt_gff3_parser_delete(GtGFF3Parser *gff3_parser)`

Delete the `gff3_parser`.

C.19 Class GtGFF3Visitor

Implements the `GtNodeVisitor` interface with a visitor that produces GFF3 output. This is a low-level class and it is usually not used directly. Normally, a `GtGFF3OutStream` is used to produce GFF3 output.

Methods

`GtNodeVisitor* gt_gff3_visitor_new(GtFile *outfp)`

Create a new `GtNodeVisitor*` which writes the output it produces to the given output file pointer `outfp`. If `outfp` is `NULL`, the output is written to `stdout`.

`void gt_gff3_visitor_set_fasta_width(GtGFF3Visitor *gff3_visitor, unsigned long fasta_width)`

Set the width with which the FASTA sequences of `GtSequenceNode`s visited by `gff3_visitor` are shown to `fasta_width`. Per default, each FASTA entry is shown on a single line.

`void gt_gff3_visitor_retain_id_attributes(GtGFF3Visitor *gff3_visitor)`

Retain the original ID attributes (instead of creating new ones), if possible. Memory consumption for `gff3_visitor` is raised from O(1) to O(`input_size`), because bookkeeping of used IDs becomes necessary to avoid ID collisions.

C.20 Class GtGTFInStream

Implements the `GtNodeStream` interface. A `GtGTFInStream` parses a GTF2.2 file and returns it as a stream of `GtGenomeNode` objects.

Methods

`GtNodeStream* gt_gtf_in_stream_new(const char *filename)`

Create a `GtGTFInStream*` which subsequently reads the GTF file with the given `filename`. If `filename` equals `NULL`, the GTF data is read from `stdin`.

C.21 Class GtGTFOutStream

Implements the `GtNodeStream` interface. A `GtGTFOutStream` produces GTF2.2 output.

Methods

`GtNodeStream* gt_gtf_out_stream_new(GtNodeStream *in_stream, GtFile *outfp)`

Create a `GtNodeStream*` which uses `in_stream` as input. It shows the nodes passed through it as GTF2.2 on `outfp`.

C.22 Class GtGenomeNode

The `GtGenomeNode` interface. The different implementation of the `GtGenomeNode` interface represent different parts of genome annotations (as they are usually found in GFF3 files).

Methods

`GtGenomeNode* gt_genome_node_ref(GtGenomeNode *genome_node)`

Increase the reference count for `genome_node` and return it. `genome_node` cannot be `NULL`.

`GtStr* gt_genome_node_get_seqid(GtGenomeNode *genome_node)`

Return the sequence ID of `genome_node`. Corresponds to column 1 of GFF3 feature lines.

`GtRange gt_genome_node_get_range(GtGenomeNode *genome_node)`

Return the genomic range of of `genome_node`. Corresponds to columns 4 and 5 of GFF3 feature lines.

`unsigned long gt_genome_node_get_start(GtGenomeNode *genome_node)`

Return the start of `genome_node`. Corresponds to column 4 of GFF3 feature lines.

`unsigned long gt_genome_node_get_end(GtGenomeNode *genome_node)`

Return the end of `genome_node`. Corresponds to column 5 of GFF3 feature lines.

`unsigned long gt_genome_node_get_length(GtGenomeNode *genome_node)`

Return the length of `genome_node`. Computed from column 4 and 5 of GFF3 feature lines.

`const char* gt_genome_node_get_filename(const GtGenomeNode* genome_node)`

Return the filename the `genome_node` was read from. If the node did not originate from a file, an appropriate string is returned.

`unsigned int gt_genome_node_get_line_number(const GtGenomeNode*)`

Return the line of the source file the `genome_node` was encountered on (if the node was read from a file, otherwise 0 is returned).

`void gt_genome_node_set_range(GtGenomeNode *genome_node, const GtRange *range)`

Set the genomic range of `genome_node` to given `range`.

`void gt_genome_node_add_user_data(GtGenomeNode *genome_node, const char *key, void *data, GtFree free_func)`

Attach a pointer to `data` to the `genome_node` using a given string as `key`. `free_func` is the optional destructor for `data`.

`void* gt_genome_node_get_user_data(const GtGenomeNode *genome_node, const char *key)`

Return the pointer attached to the `genome_node` for a given `key`.

`void gt_genome_node_release_user_data(GtGenomeNode *genome_node, const char *key)`

Call the destructor function associated with the user data attached to `genome_node` under the `key` on the attached data.

`int gt_genome_node_cmp(GtGenomeNode *genome_node_a, GtGenomeNode *genome_node_b)`

Compare `genome_node_a` with `genome_node_b` and return the result (similar to `strcmp(3)`). This method is the criterion used to sort genome nodes.

`void gt_genome_nodes_sort(GtArray *nodes)`

Sort node array `nodes`

`void gt_genome_nodes_sort_stable(GtArray *nodes)`

Sort node array `nodes` in a stable way

`int gt_genome_node_accept(GtGenomeNode *genome_node, GtNodeVisitor *node_visitor, GtError *err)`

Let `genome_node` accept the `node_visitor`. In the case of an error, -1 is returned and `err` is set accordingly.

`void gt_genome_node_delete(GtGenomeNode *genome_node)`

Decrease the reference count for `genome_node` or delete it, if this was the last reference.

C.23 Class GtHashmap

A hashmap allowing to index any kind of pointer (as a value). As keys, strings or any other pointer can be used.

Methods

`GtHashmap* gt_hashmap_new(GtHashType keyhashtype, GtFree keyfree, GtFree valuefree)`

Creates a new `GtHashmap` object of type `keyhashtype`. If `keyfree` and/or `valuefree` are given, they will be used to free the hashmap members when the `GtHashmap` is deleted. `keyhashtype` defines how to hash the keys given when using the `GtHashmap`. `GT_HASH_DIRECT` uses the key pointer as a basis for the hash function. Equal pointers will refer to the same value. If `GT_HASH_STRING` is used, the keys will be evaluated as strings and keys will be considered equal if the strings are identical, regardless of their address in memory

`GtHashmap* gt_hashmap_ref(GtHashmap *hm)`

Increase the reference count of `hm`.

`void* gt_hashmap_get(GtHashmap *hashmap, const void *key)`

Return the value stored in `hashmap` for `key` or `NULL` if no such key exists.

`void gt_hashmap_add(GtHashmap *hashmap, void *key, void *value)`

Set the value stored in `hashmap` for `key` to `value`, overwriting the prior value for that key if present.

`void gt_hashmap_remove(GtHashmap *hashmap, const void *key)`

Remove the member with key `key` from `hashmap`.

`int gt_hashmap_foreach_ordered(GtHashmap *hashmap, GtHashmapVisitFunc func, void *data, GtCompare cmp, GtError *err)`

Iterate over `hashmap` in order given by compare function `cmp`. For each member, `func` is called (see interface).

`int gt_hashmap_foreach(GtHashmap *hashmap, GtHashmapVisitFunc func, void *data, GtError *err)`

Iterate over `hashmap` in arbitrary order. For each member, `func` is called (see interface).

`int gt_hashmap_foreach_in_key_order(GtHashmap *hashmap, GtHashmapVisitFunc func, void *data, GtError *err)`

Iterate over `hashmap` in either alphabetical order (if `GtHashType` was specified as `GT_HASH_STRING`) or numerical order (if `GtHashType` was specified as `GT_HASH_DIRECT`).

`void gt_hashmap_reset(GtHashmap *hashmap)`

Reset `hashmap` by unsetting values for all keys, calling the free function if necessary.

`void gt_hashmap_delete(GtHashmap *hashmap)`

Delete `hashmap`, calling the free function if necessary.

C.24 Class GtIDToMD5Stream

Implements the `GtNodeStream` interface. A `GtIDToMD5Stream` converts "regular" sequence IDs to MD5 fingerprints.

Methods

`GtNodeStream* gt_id_to_md5_stream_new(GtNodeStream *in_stream, GtRegionMapping *region_mapping, bool substitute_target_ids)`

Create a `GtIDToMD5Stream` object which converts "regular" sequence IDs from nodes it retrieves from its `in_stream` to MD5 fingerprints (with the help of the given `region_mapping`). If `substitute_target_ids` is true, the IDs of Target attributes are also converted to MD5 fingerprints. Takes ownership of `region_mapping`!

C.25 Class GtInterFeatureStream

Implements the `GtNodeStream` interface. A `GtInterFeatureStream` inserts new feature nodes between existing feature nodes of a certain type.

Methods

`GtNodeStream* gt_inter_feature_stream_new(GtNodeStream *in_stream, const char *outside_type, const char *inter_type)`

Create a `GtInterFeatureStream*` which inserts feature nodes of type `inter_type` between the feature nodes of type `outside_type` it retrieves from `in_stream` and returns them.

C.26 Class GtMD5ToIDStream

Implements the `GtNodeStream` interface. A `GtMD5ToIDStream` converts MD5 fingerprints used as sequence IDs to "regular" ones.

Methods

`GtNodeStream* gt_md5_to_id_stream_new(GtNodeStream *in_stream, GtRegionMapping *region_mapping)`

Create a `GtMD5toIDStream*` which converts MD5 sequence IDs from nodes it retrieves from its `in_stream` to "regular" ones (with the help of the given `region_mapping`). Takes ownership of `region_mapping`!

C.27 Class GtMergeFeatureStream

Implements the `GtNodeStream` interface. A `GtMergeFeatureStream` merges adjacent features of the same type.

Methods

`GtNodeStream* gt_merge_feature_stream_new(GtNodeStream *in_stream)`

Create a `GtMergeFeatureStream*` which merges adjacent features of the same type it retrieves from `in_stream` and returns them (and all other unmodified features).

C.28 Class GtMergeStream

Implements the `GtNodeStream` interface. A `GtMergeStream` allows one to merge a set of sorted streams in a sorted fashion.

Methods

`GtNodeStream* gt_merge_stream_new(const GtArray *node_streams)`

Create a `GtMergeStream*` which merges the given (sorted) `node_streams` in a sorted fashion.

C.29 Class GtNodeStream

The `GtNodeStream` interface. `GtNodeStream` objects process `GtGenomeNode` objects in a pull-based architecture and can be chained together.

Methods

`GtNodeStream* gt_node_stream_ref(GtNodeStream *node_stream)`

Increase the reference count for `node_stream` and return it.

`int gt_node_stream_next(GtNodeStream *node_stream, GtGenomeNode **genome_node, GtError *err)`

Try to get the the next `GtGenomeNode` from `node_stream` and store it in `genome_node` (transfers ownership to `genome_node`). If no error occurs, 0 is returned and `genome_node` contains either the next `GtGenomeNode` or `NULL`, if the `node_stream` is exhausted. If an error occurs, -1 is returned and `err` is set accordingly (the status of `genome_node` is undefined, but no ownership transfer occured).

`int gt_node_stream_pull(GtNodeStream *node_stream, GtError *err)`

Calls `gt_node_stream_next()` on `node_stream` repeatedly until the `node_stream` is exhausted (0 is returned) or an error occurs (-1 is returned and `err` is set). All retrieved `GtGenomeNodes` are deleted automatically with calls to `gt_genome_node_delete()`. This method is basically a convenience method which simplifies calls to `gt_node_stream_next()` in a loop where the retrieved `GtGenomeNode` objects are not processed any further.

`bool gt_node_stream_is_sorted(GtNodeStream *node_stream)`

Return `true` if `node_stream` is a sorted stream, `false` otherwise.

`void gt_node_stream_delete(GtNodeStream *node_stream)`

Decrease the reference count for `node_stream` or delete it, if this was the last reference.

`GtNodeStream* gt_node_stream_create(const GtNodeStreamClass *node_stream_class, bool ensure_sorting)`

Create a new object of the given `node_stream_class`. If `ensure_sorting` is `true`, it is enforced that all genome node objects pulled from this class are sorted. That is, for consecutive nodes `a` and `b` obtained from the given `node_stream_class` the return code of `gt_genome_node_compare(a,b)` has to be smaller or equal than 0. If this condition is not met, an assertion fails.

`void* gt_node_stream_cast(const GtNodeStreamClass *node_stream_class, GtNodeStream *node_stream)`

Cast `node_stream` to the given `node_stream_class`. That is, if `node_stream` is not from the given `node_stream_class`, an assertion will fail.

C.30 Class GtNodeStreamClass

`const GtNodeStreamClass* gt_node_stream_class_new(size_t size, GtNodeStreamFreeFunc free, GtNodeStreamNextFunc next)`

Create a new node stream class (that is, a class which implements the node stream interface). `size` denotes the size of objects of the new node stream class. The optional `free` method is called once, if an object of the new class is deleted. The mandatory `next` method has to implement the `gt_node_stream_next()` semantic for the new class.

C.31 Class GtNodeVisitor

The `GtNodeVisitor` interface, a visitor for `GtGenomeNode` objects.

Methods

`int gt_node_visitor_visit_comment_node(GtNodeVisitor *node_visitor, GtCommentNode *comment_node, GtError *err)`

Visit `comment_node` with `node_visitor`.

`int gt_node_visitor_visit_feature_node(GtNodeVisitor *node_visitor, GtFeatureNode *feature_node, GtError *err)`

Visit `feature_node` with `node_visitor`.

`int gt_node_visitor_visit_meta_node(GtNodeVisitor *node_visitor, GtMetaNode *meta_node, GtError *err)`

Visit `meta_node` with `node_visitor`.

`int gt_node_visitor_visit_region_node(GtNodeVisitor *node_visitor, GtRegionNode *region_node, GtError *err)`

Visit `region_node` with `node_visitor`.

`int gt_node_visitor_visit_sequence_node(GtNodeVisitor *node_visitor, GtSequenceNode *sequence_node, GtError *err)`

Visit `sequence_node` with `node_visitor`.

`void gt_node_visitor_delete(GtNodeVisitor *node_visitor)`

Delete `node_visitor`.

C.32 Class GtOption

`GtOption` objects represent command line options (which are used in a `GtOptionParser`). Option descriptions are automatically formatted to `GT_OPTION_PARSER_TERMINAL_WIDTH`,

but it is possible to embed newlines into the descriptions to manually affect the formatting.

Methods

`GtOption* gt_option_new_bool(const char *option_string, const char *description, bool *value, bool default_value)`

> Return a new `GtOption` with the given `option_string`, `description`, and `default_value`. The result of the option parsing is stored in `value`.

`GtOption* gt_option_new_double(const char *option_string, const char *description, double *value, double default_value)`

> Return a new `GtOption` with the given `option_string`, `description`, and `default_value`. The result of the option parsing is stored in `value`.

`GtOption* gt_option_new_double_min(const char *option_string, const char *description, double *value, double default_value, double minimum_value)`

> Return a new `GtOption` with the given `option_string`, `description`, and `default_value`. The result of the option parsing is stored in `value`. The argument to this option must at least have the `minimum_value`.

`GtOption* gt_option_new_double_min_max(const char *option_string, const char *description, double *value, double default_value, double minimum_value, double maximum_value)`

> Return a new `GtOption` with the given `option_string`, `description`, and `default_value`. The result of the option parsing is stored in `value`. The argument to this option must at least have the `minimum_value` and at most the `maximum_value`.

`GtOption* gt_option_new_probability(const char *option_string, const char *description, double *value, double default_value)`

> Return a new `GtOption` with the given `option_string`, `description`, and `default_value`. The result of the option parsing is stored in `value`. The argument to this option must at larger or equal than 0.0 and smaller or equal than 1.0.

`GtOption* gt_option_new_int(const char *option_string, const char *description, int *value, int default_value)`

> Return a new `GtOption` with the given `option_string`, `description`, and `default_value`. The result of the option parsing is stored in `value`.

`GtOption* gt_option_new_int_min(const char *option_string, const char *description, int *value, int default_value, int minimum_value)`

Return a new `GtOption` with the given `option_string`, `description`, and `default_value`. The result of the option parsing is stored in `value`. The argument to this option must at least have the `minimum_value`.

`GtOption* gt_option_new_int_max(const char *option_string, const char *description, int *value, int default_value, int maximum_value)`

Return a new `GtOption` with the given `option_string`, `description`, and `default_value`. The result of the option parsing is stored in `value`. The argument to this option must at most have the `maximum_value`.

`GtOption* gt_option_new_int_min_max(const char *option_string, const char *description, int *value, int default_value, int minimum_value, int maximum_value)`

Return a new `GtOption` with the given `option_string`, `description`, and `default_value`. The result of the option parsing is stored in `value`. The argument to this option must at least have the `minimum_value` and at most the `maximum_value`.

`GtOption* gt_option_new_uint(const char *option_string, const char *description, unsigned int *value, unsigned int default_value)`

Return a new `GtOption` with the given `option_string`, `description`, and `default_value`. The result of the option parsing is stored in `value`.

`GtOption* gt_option_new_uint_min(const char *option_string, const char *description, unsigned int *value, unsigned int default_value, unsigned int minimum_value)`

Return a new `GtOption` with the given `option_string`, `description`, and `default_value`. The result of the option parsing is stored in `value`. The argument to this option must at least have the `minimum_value`.

`GtOption* gt_option_new_uint_max(const char *option_string, const char *description, unsigned int *value, unsigned int default_value, unsigned int maximum_value)`

Return a new `GtOption` with the given `option_string`, `description`, and `default_value`. The result of the option parsing is stored in `value`. The argument to this option must at most have the `maximum_value`.

`GtOption* gt_option_new_uint_min_max(const char *option_string, const char *description, unsigned int *value, unsigned int default_value, unsigned int minimum_value, unsigned int maximum_value)`

Return a new `GtOption` with the given `option_string`, `description`, and `default_value`. The result of the option parsing is stored in `value`. The argument to this option must at least have the `minimum_value` and at most the `maximum_value`.

`GtOption* gt_option_new_long(const char *option_string, const char *description, long *value, long default_value)`

Return a new `GtOption` with the given `option_string`, `description`, and `default_value`. The result of the option parsing is stored in `value`.

`GtOption* gt_option_new_ulong(const char *option_string, const char *description, unsigned long *value, unsigned long default_value)`

Return a new `GtOption` with the given `option_string`, `description`, and `default_value`. The result of the option parsing is stored in `value`.

`GtOption* gt_option_new_ulong_min(const char *option_string, const char *description, unsigned long *value, unsigned long default_value, unsigned long minimum_value)`

Return a new `GtOption` with the given `option_string`, `description`, and `default_value`. The result of the option parsing is stored in `value`. The argument to this option must at least have the `minimum_value`.

`GtOption* gt_option_new_ulong_min_max(const char *option_string, const char *description, unsigned long *value, unsigned long default_value, unsigned long minimum_value, unsigned long maximum_value)`

Return a new `GtOption` with the given `option_string`, `description`, and `default_value`. The result of the option parsing is stored in `value`. The argument to this option must at least have the `minimum_value` and at most the `maximum_value`.

`GtOption* gt_option_new_range(const char *option_string, const char *description, GtRange *value, GtRange *default_value)`

Return a new `GtOption` with the given `option_string`, `description`, and `default_value`. The result of the option parsing is stored in `value`. If `default_value` equals `NULL`, `GT_UNDEF_LONG` will be used as the default start and end point of `value`.

`GtOption* gt_option_new_range_min_max(const char *option_string, const char *description, GtRange *value, GtRange *default_value, unsigned long minimum_value, unsigned long maximum_value)`

Return a new `GtOption` with the given `option_string`, `description`, and `default_value`. The result of the option parsing is stored in `value`. The first argument to this option (which will be used as the start) must at least have the `minimum_value` and the second argument (which will be used as the end) at most the `maximum_value`.

`GtOption* gt_option_new_string(const char *option_string, const char *description, GtStr *value, const char *default_value)`

Return a new `GtOption` with the given `option_string`, `description`, and `default_value`. The result of the option parsing is stored in `value`.

`GtOption* gt_option_new_string_array(const char *option_string, const char *description, GtStrArray *value)`

Return a new `GtOption` with the given `option_string`, `description`, and `default_value`. The result of the option parsing are stored in `value`.

`GtOption* gt_option_new_choice(const char *option_string, const char *description, GtStr *value, const char *default_value, const char **domain)`

Return a `GtOption` with the given `option_string`, `description`, and `default_value` which allows only arguments given in the `NULL`-terminated `domain` (`default_value` must be an entry of `domain` or `NULL`).

`GtOption* gt_option_new_filename(const char *option_string, const char *description, GtStr *filename)`

Return a new `GtOption` with the given `option_string`, `description`, and `default_value`. The result of the option parsing are stored in `value`.

`GtOption* gt_option_new_filename_array(const char *option_string, const char *description, GtStrArray *filename_array)`

Return a new `GtOption` with the given `option_string`, `description`, and `default_value`. The results of the option parsing are stored in `value`.

`GtOption* gt_option_new_debug(bool *value)`

Return a new debug `GtOption` object: `-debug`, "enable debugging output", default is `false`. The result of the option parsing is stored in `value`

`GtOption* gt_option_new_verbose(bool *value)`

Return a new verbose `GtOption` object: `-v`, "be verbose", default is `false`. The result of the option parsing is stored in `value`

`GtOption* gt_option_new_width(unsigned long *value)`

Return a new width `GtOption` object: `-width`, "set output width for FASTA sequence printing (0 disables formatting)", default is 0. The result of the option parsing is stored in `value`

`GtOption* gt_option_ref(GtOption *option)`

Increase the reference count for `option` and return it.

`const char* gt_option_get_name(const GtOption * option)`

Return the name of `option`

`void gt_option_is_mandatory(GtOption *option)`

Make `option` mandatory.

`void gt_option_is_mandatory_either(GtOption *option_a, const GtOption *option_b)`

Make it mandatory, that either `option_a` or `option_b` is used.

`void gt_option_is_mandatory_either_3(GtOption *option_a, const GtOption *option_b, const GtOption *option_c)`

Make it mandatory, that one of the options `option_a`, `option_b`, or `option_c` is used.

`void gt_option_is_extended_option(GtOption *option)`

Set that `option` is only shown in the output of `-help+`.

`void gt_option_is_development_option(GtOption *option)`

Set that `option` is only shown in the output of `-helpdev`.

`void gt_option_imply(GtOption *option_a, const GtOption *option_b)`

Make `option_a` imply `option_b`.

`void gt_option_imply_either_2(GtOption *option_a, const GtOption *option_b, const GtOption *option_c)`

Make `option_a` imply either `option_b` or `option_c`

`void gt_option_exclude(GtOption *option_a, GtOption *option_b)`

Set that the options `option_a` and `option_b` exclude each other.

`void gt_option_hide_default(GtOption *option)`

Hide the default value of `option` in `-help` output.

`void gt_option_argument_is_optional(GtOption *option)`

Set that the argument to `option` is optional

`bool gt_option_is_set(const GtOption *option)`

Return `true` if `option` was set, `false` otherwise.

`void gt_option_delete(GtOption*)`

Delete `option`.

`int gt_option_parse_spacespec(unsigned long *maximumspace, const char *optname, const GtStr *memlimit, GtError *err)`

Parse the argument to option -memlimit. Could be made into a special parser, but I do not know how. SK. 2011-09-19

C.33 Class GtOptionParser

`GtOptionParser` objects can be used to parse command line options.

Methods

`#define GT_OPTION_PARSER_TERMINAL_WIDTH`

The default terminal width used in the output of the `GtOptionParser`.

`GtOptionParser* gt_option_parser_new(const char *synopsis, const char *one_liner)`

Return a new `GtOptionParser` object. The `synopsis` should summarize the command line arguments and mandatory arguments in a single line. The `one_liner` should describe the program for which the `GtOptionParser` is used in a single line and must have an upper case letter at the start and a '.' at the end.

`void gt_option_parser_add_option(GtOptionParser *option_parser, GtOption *option)`

Add `option` to `option_parser`. Takes ownership of `option`.

`GtOption* gt_option_parser_get_option(GtOptionParser *option_parser, const char *option_string)`

Return the `GtOption` object if an option named `option_string` is present in `option_parser`, and `NULL` if no such option exists.

`void gt_option_parser_refer_to_manual(GtOptionParser *option_parser)`

Refer to manual at the end of `-help` output of `opion_parser`.

`void gt_option_parser_set_comment_func(GtOptionParser *option_parser, GtShowCommentFunc comment_func, void *data)`

Set `comment_func` in `option_parser` (`data` is passed along).

`void gt_option_parser_set_version_func(GtOptionParser *option_parser, GtShowVersionFunc version_func)`

Set the version function used by `option_parser` to `version_func`. This version function takes precedence to the one supplied to `gt_option_parser_parse()`.

`void gt_option_parser_set_mail_address(GtOptionParser*, const char *mail_address)`

Set the `mail_address` used in the final "Report bugs to" line of the `-help` output. It should be of the form <`bill@microsoft.com`> (email address enclosed in one pair of angle brackets).

`void gt_option_parser_register_hook(GtOptionParser *option_parser, GtOptionParserHookFunc hook_function, void *data)`

Register a `hook_function` with `option_parser`. All registered hook functions are called at the end of `gt_option_parser_parse()`. This allows one to have a module which registers a bunch of options in the option parser and automatically performs necessary postprocessing after the option parsing has been done via the hook function.

`void gt_option_parser_set_min_args(GtOptionParser *option_parser, unsigned int minimum)`

The the `minimum` number of additional command line arguments `option_parser` must parse in order to succeed.

`void gt_option_parser_set_max_args(GtOptionParser *option_parser, unsigned int maximum)`

The the `maximum` number of additional command line arguments `option_parser` must parse in order to succeed.

`void gt_option_parser_set_min_max_args(GtOptionParser *option_parser, unsigned int minimum, unsigned int maximum)`

The the `minimum` and `maximum` number of additional command line arguments `option_parser` must parse in order to succeed.

`GtOPrval gt_option_parser_parse(GtOptionParser *option_parser, int *parsed_args, int argc, const char **argv, GtShowVersionFunc version_func, GtError *err)`

Use `option_parser` to parse options given in argument vector `argv` (with `argc` many arguments). The number of parsed arguments is stored in `parsed_args`. `version_func` is used for the output of option `-version`. In case of error, `GT_OPTION_PARSER_ERROR` is returned and `err` is set accordingly.

`void gt_option_parser_delete(GtOptionParser *option_parser)`

Delete `option_parser`.

C.34 Class GtPhase

This enum type defines the possible phases. The following phases are defined: `GT_PHASE_ZERO`, `GT_PHASE_ONE`, `GT_PHASE_TWO`, and `GT_PHASE_UNDEFINED`.

Methods

`#define GT_PHASE_CHARS`

Use this string to map phase enum types to their corresponding character.

`GtPhase gt_phase_get(char phase_char)`

Map `phase_char` to the corresponding phase enum type. An assertion will fail if `phase_char` is not a valid one.

C.35 Class GtQueue

`GtQueue` objects are generic queues which can be used to process objects of any type in an First-In-First-Out (FIFO) fashion.

Methods

`GtQueue* gt_queue_new(void)`

Return a new `GtQueue` object.

`void gt_queue_add(GtQueue *queue, void *elem)`

Add `elem` to `queue` (*enqueue* in computer science terminology).

`void* gt_queue_get(GtQueue *queue)`

Remove the first element from non-empty `queue` and return it (*dequeue* in computer science terminology).

`void* gt_queue_head(GtQueue *queue)`

Return the first element in non-empty `queue` without removing it.

`void gt_queue_remove(GtQueue *queue, void *elem)`

Remove `elem` from `queue` (`elem` has to be in `queue`). Thereby `queue` is traversed in reverse order, leading to O(`gt_queue_size(queue)`) worst-case running time.

`unsigned long gt_queue_size(const GtQueue *queue)`

Return the number of elements in `queue`.

`void gt_queue_delete(GtQueue *queue)`

Delete `queue`. Elements contained in `queue` are not freed!

C.36 Class GtRange

The `GtRange` class is used to represent genomic ranges in *GenomeTools*. Thereby, the `start` must **always** be smaller or equal than the `end`.

Methods

`int gt_range_compare(const GtRange *range_a, const GtRange *range_b)`

Compare `range_a` and `range_b`. Returns 0 if `range_a` equals `range_b`, -1 if `range_a` starts before `range_b` or (for equal starts) `range_a` ends before `range_b`, and 1 else.

`int gt_range_compare_with_delta(const GtRange *range_a, const GtRange *range_b, unsigned long delta)`

Compare `range_a` and `range_b` with given `delta`. Returns 0 if `range_a` equals `range_b` modulo `delta` (i.e., the start and end points of `range_a` and `range_b` are at most `delta` bases apart), -1 if `range_a` starts before `range_b` or (for equal starts) `range_a` ends before `range_b`, and 1 else.

`bool gt_range_overlap(const GtRange *range_a, const GtRange *range_b)`

Returns `true` if `range_a` and `range_b` overlap, `false` otherwise.

`bool gt_range_overlap_delta(const GtRange *range_a, const GtRange *range_b, unsigned long delta)`

Returns `true` if `range_a` and `range_b` overlap **at least** `delta` many positions, `false` otherwise.

`bool gt_range_contains(const GtRange *range_a, const GtRange *range_b)`

Returns `true` if `range_b` is contained in `range_a`, `false` otherwise.

`bool gt_range_within(const GtRange *range, unsigned long point)`

Returns `true` if `point` lies within `range`, `false` otherwise.

`GtRange gt_range_join(const GtRange *range_a, const GtRange *range_b)`

Join `range_a` and `range_b` and return the result.

`GtRange gt_range_offset(const GtRange *range, long offset)`

Transform start and end of `range` by `offset` and return the result.

`unsigned long gt_range_length(const GtRange *range)`

Returns the length of the given `range`.

C.37 Class GtRegionMapping

A GtRegionMapping objects maps sequence-regions to the corresponding entries of sequence files.

Methods

GtRegionMapping* gt_region_mapping_new_mapping(GtStr *mapping_filename, GtError *err)

Return a new GtRegionMapping object for the mapping file with the given mapping_filename. In the case of an error, NULL is returned and err is set accordingly.

GtRegionMapping* gt_region_mapping_new_seqfiles(GtStrArray *sequence_filenames, bool matchdesc, bool usedesc)

Return a new GtRegionMapping object for the sequence files given in sequence_filenames. If matchdesc is true, the sequence descriptions from the input files are matched for the desired sequence IDs (in GFF3).
If usedesc is true, the sequence descriptions are used to map the sequence IDs (in GFF3) to actual sequence entries. If a description contains a sequence range (e.g., III:1000001..2000000), the first part is used as sequence ID ('III') and the first range position as offset ('1000001').
matchdesc and usedesc cannot be true at the same time.

GtRegionMapping* gt_region_mapping_new_rawseq(const char *rawseq, unsigned long length, unsigned long offset)

Return a new GtRegionMapping object which maps to the given sequence rawseq with the corresponding length and offset.

GtRegionMapping* gt_region_mapping_ref(GtRegionMapping *region_mapping)

Increase the reference count for region_mapping and return it.

int gt_region_mapping_get_raw_sequence(GtRegionMapping *region_mapping, const char **rawseq, unsigned long *length, unsigned long *offset, GtStr *seqid, const GtRange *range, GtError *err)

Use region_mapping to map the given sequence ID seqid and its corresponding range to an actual sequence. The sequence is returned in rawseq, its length and offset in length and offset. In the case of an error, -1 is returned and err is set accordingly.

`int gt_region_mapping_get_description(GtRegionMapping *region_mapping, GtStr *desc, GtStr *seqid, GtError *err)`

Use `region_mapping` to get the description of the MD5 sequence ID `seqid`. The description is appended to `desc`. In the case of an error, -1 is returned and `err` is set accordingly.

`const char* gt_region_mapping_get_md5_fingerprint(GtRegionMapping *region_mapping, GtStr *seqid, const GtRange *range, unsigned long *offset, GtError *err)`

Use `region_mapping` to return the MD5 fingerprint of the sequence with the sequence ID `seqid` and its corresponding `range`. The offset of the sequence is stored in `offset`. In the case of an error, `NULL` is returned and `err` is set accordingly.

`void gt_region_mapping_delete(GtRegionMapping *region_mapping)`

Delete `region_mapping`.

C.38 Class GtRegionNode

Implements the `GtGenomeNode` interface. Region nodes correspond to the `##sequence-region` lines in GFF3 files.

Methods

`GtGenomeNode* gt_region_node_new(GtStr *seqid, unsigned long start, unsigned long end)`

Create a new `GtRegionNode*` representing sequence with ID `seqid` from base position `start` to base position `end` (1-based). `start` has to be smaller or equal than `end`. The `GtRegionNode*` stores a new reference to `seqid`, so make sure you do not modify the original `seqid` afterwards!

C.39 Class GtSelectStream

Implements the `GtNodeStream` interface. A `GtSelectStream` selects certain nodes it retrieves from its node source and passes them along.

Methods

`GtNodeStream* gt_select_stream_new(GtNodeStream *in_stream, GtStr *seqid, GtStr *source, const GtRange *contain_range, const GtRange *overlap_range, GtStrand strand, GtStrand targetstrand, bool has_CDS, unsigned long max_gene_length, unsigned long max_gene_num, double min_gene_score, double max_gene_score, double min_average_splice_site_prob, unsigned long feature_num, GtStrArray *select_files, GtStr *select_logic, GtError *err)`

Create a `GtSelectStream` object which selects genome nodes it retrieves from its `in_stream` and passes them along if they meet the criteria defined by the other arguments. All comment nodes are selected. If `seqid` is defined, a genome node must have it to be selected. If `source` is defined, a genome node must have it to be selected. If `contain_range` is defined, a genome node must be contained in it to be selected. If `overlap_range` is defined, a genome node must overlap it to be selected. If `strand` is defined, a (top-level) genome node must have it to be selected. If `targetstrand` is defined, a feature with a target attribute must have exactly one of it and its strand must equal `targetstrand`. If `had_cds` is `true`, all top-level features are selected which have a child with type *CDS*. If `max_gene_length` is defined, only genes up to the this length are selected. If `max_gene_num` is defined, only so many genes are selected. If `min_gene_score` is defined, only genes with at least this score are selected. If `max_gene_score` is defined, only genes with at most this score are selected. If `min_average_splice_site_prob` is defined, feature nodes which have splice sites must have at least this average splice site score to be selected. If `feature_num` is defined, just the `feature_num`th feature node occurring in the `in_stream` is selected. If `select_files` is defined and has at least one entry, the entries are evaluated as Lua scripts containing functions taking `GtGenomeNodes` that are evaluated to boolean values to determine selection. `select_logic` can be "OR" or "AND", defining how the results from the select scripts are combined. Returns a pointer to a new `GtSelectStream` or NULL on error (`err` is set accordingly).

`void gt_select_stream_set_drophandler(GtSelectStream *sstr, GtSelectNodeFunc fp, void *data)`

Sets `fp` as a handler function to be called for every `GtGenomeNode` not selected by `sstr`. The void pointer `data` can be used for arbitrary user data.

C.40 Class GtSequenceNode

Implements the `GtGenomeNode` interface. Sequence nodes correspond to singular embedded FASTA sequences in GFF3 files.

Methods

`GtGenomeNode* gt_sequence_node_new(const char *description, GtStr *sequence)`

Create a new `GtSequenceNode*` representing a FASTA entry with the given `description` and `sequence`. Takes ownership of `sequence`.

`const char* gt_sequence_node_get_description(const GtSequenceNode *sequence_node)`

Return the description of `sequence_node`.

`const char* gt_sequence_node_get_sequence(const GtSequenceNode *sequence_node)`

Return the sequence of `sequence_node`.

`unsigned long gt_sequence_node_get_sequence_length(const GtSequenceNode *sequence_node)`

Return the sequence length of `sequence_node`.

C.41 Class GtSortStream

Implements the `GtNodeStream` interface. A `GtSortStream` sorts the `GtGenomeNode` objects it retrieves from its node source.

Methods

`GtNodeStream* gt_sort_stream_new(GtNodeStream *in_stream)`

Create a `GtSortStream*` which sorts the genome nodes it retrieves from `in_stream` and returns them unmodified, but in sorted order.

C.42 Class GtStatStream

Implements the `GtNodeStream` interface. A `GtStatStream` gathers statistics about the `GtGenomeNode` objects it retrieves from its node source and passes them along unmodified.

Methods

`GtNodeStream* gt_stat_stream_new(GtNodeStream *in_stream, bool gene_length_distribution, bool gene_score_distribution, bool exon_length_distribution, bool exon_number_distribution, bool intron_length_distribution, bool cds_length_distribution, bool used_sources)`

Create a `GtStatStream` object which gathers statistics about the `GtGenomeNode` objects it retrieves from its `in_stream` and returns them unmodified. Besides the basic statistics, statistics about the following distributions can be gathered, if the corresponding argument equals `true`: `gene_length_distribution`, `gene_score_distribution`, `exon_length_distribution`, `exon_number_distribution`, `intron_length_distribution`, `cds_length_distribution`.
If `used_sources` equals `true`, it is recorded which source tags have been encountered.

`void gt_stat_stream_show_stats(GtStatStream *stat_stream, GtFile *outfp)`

Write the statistics gathered by `stat_stream` to `outfp`.

C.43 Class GtStr

Objects of the `GtStr` class are strings which grow on demand.

Methods

`GtStr* gt_str_new(void)`

Return an empty `GtStr` object.

`GtStr* gt_str_new_cstr(const char *cstr)`

Return a new `GtStr` object whose content is set to `cstr`.

`GtStr* gt_str_clone(const GtStr *str)`

Return a clone of `str`.

`GtStr* gt_str_ref(GtStr *str)`

Increase the reference count for `str` and return it. If `str` is `NULL`, `NULL` is returned without any side effects.

`char* gt_str_get(const GtStr *str)`

Return the content of `str`. Never returns NULL, and the content is always \0-terminated

`void gt_str_set(GtStr *str, const char *cstr)`
: Set the content of `str` to `cstr`.

`void gt_str_append_str(GtStr *dest, const GtStr *src)`
: Append the string `src` to `dest`.

`void gt_str_append_cstr(GtStr *str, const char *cstr)`
: Append the \0-terminated `cstr` to `str`.

`void gt_str_append_cstr_nt(GtStr *str, const char *cstr, unsigned long length)`
: Append the (not necessarily \0-terminated) `cstr` with given `length` to `str`.

`void gt_str_append_char(GtStr *str, char c)`
: Append character `c` to `str`.

`void gt_str_append_double(GtStr *str, double d, int precision)`
: Append double `d` to `str` with given `precision`.

`void gt_str_append_ulong(GtStr *str, unsigned long ulong)`
: Append `ulong` to `str`.

`void gt_str_append_int(GtStr *str, int intval)`
: Append `intval` to `str`.

`void gt_str_append_uint(GtStr *str, unsigned int uint)`
: Append `uint` to `str`.

`void gt_str_set_length(GtStr *str, unsigned long length)`
: Set length of `str` to `length`. `length` must be smaller or equal than `gt_str_length(str)`.

`void gt_str_reset(GtStr *str)`
: Reset `str` to length 0.

`int gt_str_cmp(const GtStr *str1, const GtStr *str2)`
: Compare `str1` and `str2` and return the result (similar to `strcmp(3)`).

`unsigned long gt_str_length(const GtStr *str)`
: Return the length of `str`. If `str` is `NULL`, 0 is returned.

`void gt_str_delete(GtStr *str)`
: Decrease the reference count for `str` or delete it, if this was the last reference.

C.44 Class GtStrArray

`GtStrArray*` objects are arrays of string which grow on demand.

Methods

`GtStrArray* gt_str_array_new(void)`

Return a new `GtStrArray` object.

`GtStrArray* gt_str_array_ref(GtStrArray*)`

Increases the reference to a GtStrArray.

`void gt_str_array_add_cstr(GtStrArray *str_array, const char *cstr)`

Add `cstr` to `str_array`. Thereby, an internal copy of `cstr` is created.

`void gt_str_array_add_cstr_nt(GtStrArray *str_array, const char *cstr, unsigned long length)`

Add the non `\0`-terminated `cstr` with given `length` to `str_array`. Thereby, an internal copy of `cstr` is created.

`void gt_str_array_add(GtStrArray *str_array, const GtStr *str)`

Add `str` to `str_array`. Thereby, an internal copy of `str` is created.

`const char* gt_str_array_get(const GtStrArray *str_array, unsigned long strnum)`

Return pointer to internal string with number `strnum` of `str_array`. `strnum` must be smaller than `gt_str_array_size(str_array)`.

`void gt_str_array_set_cstr(GtStrArray *str_array, unsigned long strnum, const char *cstr)`

Set the string with number `strnum` in `str_array` to `cstr`.

`void gt_str_array_set(GtStrArray *str_array, unsigned long strnum, const GtStr *str)`

Set the string with number `strnum` in `str_array` to `str`.

`void gt_str_array_set_size(GtStrArray *str_array, unsigned long size)`

Set the size of `str_array` to `size`. `size` must be smaller or equal than `gt_str_array_size(str_array)`.

`void gt_str_array_reset(GtStrArray *str_array)`

Set the size of `str_array` to 0.

`unsigned long gt_str_array_size(const GtStrArray *str_array)`

Return the number of strings stored in `str_array`.

`void gt_str_array_delete(GtStrArray *str_array)`

Delete `str_array`.

C.45 Class GtStrand

This enum type defines the possible strands. The following strands are defined: `GT_STRAND_FORWARD`, `GT_STRAND_REVERSE`, `GT_STRAND_BOTH`, and `GT_STRAND_UNKNOWN`.

Methods

`#define GT_STRAND_CHARS`

Use this string to map strand enum types to their corresponding character.

`GtStrand gt_strand_get(char strand_char)`

Map `strand_char` to the corresponding strand enum type. Returns `GT_NUM_OF_STRAND_TYPES` if `strand_char` is not a valid one.

C.46 Class GtTagValueMap

A very simple tag/value map absolutely optimized for space (i.e., memory consumption) on the cost of time. Basically, each read/write access costs O(n) time, whereas n denotes the accumulated length of all tags and values contained in the map. Tags and values cannot have length 0.
The implementation as a char* shines through (also to save one additional memory allocation), therefore the usage is a little bit different compared to other *GenomeTools* classes. See the implementation of `gt_tag_value_map_example()` for an ussage example.

Methods

`GtTagValueMap gt_tag_value_map_new(const char *tag, const char *value)`

Return a new `GtTagValueMap` object which stores the given `tag`/`value` pair.

`void gt_tag_value_map_add(GtTagValueMap *tag_value_map, const char *tag, const char *value)`

Add `tag`/`value` pair to `tag_value_map`. `tag_value_map` must not contain the given `tag` already!

`void gt_tag_value_map_set(GtTagValueMap *tag_value_map, const char *tag, const char *value)`

Set the given `tag` in `tag_value_map` to `value`.

`const char* gt_tag_value_map_get(const GtTagValueMap tag_value_map, const char *tag)`

Return value corresponding to `tag` from `tag_value_map`. If `tag_value_map` does not contain such a value, `NULL` is returned.

`void gt_tag_value_map_remove(GtTagValueMap *tag_value_map, const char *tag)`

Removes the given `tag` from `tag_value_map`. `tag_value_map` must contain the given `tag` already!

`void gt_tag_value_map_foreach(const GtTagValueMap tag_value_map, GtTagValueMapIteratorFunc iterator_func, void *data)`

Apply `iterator_func` to each tag/value pair contained in `tag_value_map` and pass `data` along.

`int gt_tag_value_map_example(GtError *err)`

Implements an example useage of a tag/value map.

`void gt_tag_value_map_delete(GtTagValueMap tag_value_map)`

Delete `tag_value_map`.

C.47 Class GtTypeChecker

The `GtTypeChecker` interface, allows one to check the validity of (genome feature) types.

Methods

`GtTypeChecker* gt_type_checker_ref(GtTypeChecker *type_checker)`

Increase the reference count for `type_checker` and return it.

`bool gt_type_checker_is_valid(GtTypeChecker *type_checker, const char *type)`

Return `true` if `type` is a valid type for the given `type_checker`, `false` otherwise.

`void gt_type_checker_delete(GtTypeChecker *type_checker)`

Decrease the reference count for `type_checker` or delete it, if this was the last reference.

C.48 Class GtTypeCheckerOBO

Implements the `GtTypeChecker` interface with types from an OBO file.

Methods

`GtTypeChecker* gt_type_checker_obo_new(const char *obo_file_path, GtError *err)`

Create a new `GtTypeChecker*` for OBO file with given `obo_file_path`. If the OBO file cannot be parsed correctly, `NULL` is returned and `err` is set correspondingly.

C.49 Class GtUniqStream

Implements the `GtNodeStream` interface. A `GtUniqStream` filters out repeated features it retrieves from its node source.

Methods

`GtNodeStream* gt_uniq_stream_new(GtNodeStream*)`

Create a `GtUniqStream` object which filters out repeated feature node graphs it retrieves from the sorted `in_stream` and return all other nodes. Two feature node graphs are considered to be *repeated* if they have the same depth-first traversal and each corresponding feature node pair is similar according to the `gt_feature_node_is_similar()` method. For such a repeated feature node graph the one with the higher score (of the top-level feature) is kept. If only one of the feature node graphs has a defined score, this one is kept.

C.50 Class GtVisitorStream

Implements the `GtNodeStream` interface.

Methods

`GtNodeStream* gt_visitor_stream_new(GtNodeStream *in_stream, GtNodeVisitor *node_visitor)`

Create a new `GtVisitorStream*`, takes ownership of `node_visitor`. This stream applies `node_visitor` to each node which passes through it. Can be used to implement all streams with such a functionality.

C.51 Module FunctionPointer

`int (*GtCompare)(const void *a, const void *b)`

Functions of this type return less than 0 if `a` is *smaller* than `b`, 0 if `a` is *equal* to `b`, and greater 0 if `a` is *larger* than `b`. Thereby, the operators *smaller*, *equal*, and *larger* are implementation dependent. Do not count on these functions to return -1, 0, or 1!

`int (*GtCompareWithData)(const void*, const void*, void *data)`

Similar to `GtCompare`, but with an additional `data` pointer.

`void (*GtFree)(void*)`

The generic free function pointer type.

C.52 Module Init

`void gt_lib_init(void)`

Initialize this *GenomeTools* library instance. This has to be called before the library is used!

`void gt_lib_reg_atexit_func(void)`

Registers exit function which calls `gt_lib_clean()` at exit.

`int gt_lib_clean(void)`

Returns 0 if no memory map, file pointer, or memory has been leaked and a value != 0 otherwise.

C.53 Module Strcmp

`int gt_strcmp(const char *s1, const char *s2)`

Returns 0 if `s1 == s2`, otherwise the equivalent of `strcmp(s1,s2)`. Useful as a performance improvement in some cases (for example, to compare symbols).

C.54 Module Symbol

`const char* gt_symbol(const char *cstr)`

Return a symbol (a canonical representation) for `cstr`. An advantage of symbols is that they can be compared for equality by a simple pointer comparison, rather than using `strcmp()` (as it is done in `gt_strcmp()`). Furthermore, a symbol is stored only once in memory for equal `cstr`s, but keep in mind that this memory can never be freed safely during the lifetime of the calling program. Therefore, it should only be used for a small set of `cstr`s.

C.55 Module Undef

`#define GT_UNDEF_BOOL`

The undefined `bool` value.

`#define GT_UNDEF_CHAR`

The undefined `char` value.

`#define GT_UNDEF_DOUBLE`

The undefined `double` value.

`#define GT_UNDEF_FLOAT`

The undefined `float` value.

`#define GT_UNDEF_INT`

The undefined `int` value.

`#define GT_UNDEF_LONG`

The undefined `long` value.

`#define GT_UNDEF_UCHAR`

The undefined <unsigned char> value.

`#define GT_UNDEF_UINT`

The undefined <unsigned int> value.

`#define GT_UNDEF_ULONG`

The undefined <unsigned long> value.

Appendix D

Example GFF3 Sorter Program

A simple GFF3 sorter implemented using `libgenometools`:

```
#include "genometools.h"

int main(int argc, const char *argv[])
{
  GtNodeStream *gff3_in_stream, *sort_stream, *gff3_out_stream;
  GtGenomeNode *gn;
  GtError *err;
  int had_err;

  gt_lib_init();
  err = gt_error_new();

  gff3_in_stream = gt_gff3_in_stream_new_unsorted(argc-1, argv+1);
  sort_stream = gt_sort_stream_new(gff3_in_stream);
  gff3_out_stream = gt_gff3_out_stream_new(sort_stream, NULL);

  while (!(had_err = gt_node_stream_next(gff3_out_stream, &gn, err)) && gn)
    gt_genome_node_delete(gn);

  if (had_err)
    fprintf(stderr, "%s: error: %s\n", argv[0], gt_error_get(err));

  gt_node_stream_delete(gff3_out_stream);
  gt_node_stream_delete(sort_stream);
  gt_node_stream_delete(gff3_in_stream);
  gt_error_delete(err);

  if (had_err)
    return EXIT_FAILURE;
  return EXIT_SUCCESS;
}
```

In line 10 `libgenometools` is initialized and in line 11 an error object (see Section C.11) is created. In line 13 the GFF3 input stream (see Section C.16) is created which takes the command line arguments of the tool as filenames. The GFF3 input stream is then used as node source for the sort stream (see Section C.41) which is created in line 14. The sort stream in turn is used as the node source for the GFF3 output stream (see Section C.17) which is created in line 15 (the second argument is the output file pointer and `NULL` denotes that the stream writes its output to `stdout`). In line 17 the actual streaming happens: in the while-loop genome nodes are requested from the GFF3 output stream and freed directly afterwards (the output was already produced in the GFF3 output stream). The GFF3 output stream requests the genome nodes internally from its node source (the sort stream). The sort stream in turn requests its nodes from the GFF3 in stream who reads it from the input files (with the help of the GFF3 parser documented in section C.18). The sort stream buffers all nodes internally before it sorts them and starts handing them out, but this is completely transparent for the user. If an error occurs in one of the streams, the while loop will terminate with a `had_err` value of -1 and the error is shown in line 21.

Appendix E

GenomeTools Contributors

The majority of *GenomeTools* has been developed by Gordon Gremme, Sascha Steinbiß and Stefan Kurtz. The following persons contributed to the development (in alphabetical order):

- Stefan Bienert (patches)
- Joachim Bonnet (Huffman/Golomb/Elias coding, `GtBit{In,Out}Stream`)
- David Ellinghaus (parts of `ltr/` subdirectory, `ltrharvest` tool)
- Johannes Fischer (range minimum query code)
- Giorgio Gonnella (codegen, `simreads` tool, patches)
- Thomas Jahns (`BitPackArray` and `BitPackString` class, block-compressed FM-index)
- Malte Mader (*AnnotationSketch*, `sketch` tool)
- Brent Pedersen (python bindings, patches)
- Christin Schärfer (*AnnotationSketch*, `sketch` tool)
- David Schmitz-Hübsch (`mgth/` subdirectory, `mgth` tool)
- Dirk Willrodt (`genomediff` tool, patches)

An up-to-date list can be found in the `CONTRIBUTORS` file which is part of the *GenomeTools* distribution (available for download at `http://genometools.org`).

Some parts of *GenomeTools* have been published: `ltrharvest` [EKW08], `tallymer` [KNSW08], `uniquesub` [GNK+07], *AnnotationSketch* [SGS+09], `ltrdigest` [SWGK09], `mgth` [SHK10], and `readjoiner` [GK12].

Appendix F

GenomeTools License

GenomeTools has been published under the following open source license:

```
/*
  Copyright (c) 2003-2012 G. Gremme, S. Steinbiss, S. Kurtz, and CONTRIBUTORS
  Copyright (c) 2003-2012 Center for Bioinformatics, University of Hamburg

  Permission to use, copy, modify, and distribute this software for any
  purpose with or without fee is hereby granted, provided that the above
  copyright notice and this permission notice appear in all copies.

  THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL WARRANTIES
  WITH REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF
  MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR
  ANY SPECIAL, DIRECT, INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
  WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN
  ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF
  OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
*/
```

The *GenomeThreader* copyright policy is modeled after the one of the OpenBSD operating system (`http://openbsd.org/`). We chose one of the most simple and free licenses, the ICS license. It is functionally equivalent to a two-term BSD copyright with language removed that is made unnecessary by the Berne convention.

We use external sources which are covered by their respective licenses. See the `LICENSE` file in the *GenomeTools* distribution (available at `http://genometools.org`) for details.

Appendix G

GFF3 Test Runs in Detail

G.1 Retrieving the Original Files

This appendix describes the GFF3 test runs (Sections 5.5.5 and 5.5.6) in detail to make them reproducible. For the evaluation, the following GFF3 files have been retrieved on 2011/05/31:

```
ftp://ftp.arabidopsis.org/home/tair/Genes/TAIR10_genome_release/TAIR10_gff3/TAIR10_GFF3_genes.gff
ftp://ftp.fruitfly.org/pub/genomic/fasta/EST.gff.gz
Homo_sapiens_ENSEMBL.gff3 from
http://galaxy.fml.mpg.de/library_common/browse_library?show_deleted=False&cntrller=library
&use_panels=False&id=2f94e8ae9edff68a
```

An archive file containing all of the GFF3 files listed above (in uncompressed form) can be downloaded from:

```
ftp://genomethreader.org/pub/dissertation/gff3_test_runs.tar.gz
```

The file has the following MD5 hash:

```
67055d2715e2ab630f75a9f252173a17
```

The files have been sorted (and tidied) as follows):

```
gt gff3 -sort -tidy -o tair.gff3 TAIR10_GFF3_genes.gff
gt gff3 -sort -o fruitfly.gff3 EST.gff
gt gff3 -sort -o ensembl.gff3 Homo_sapiens_ENSEMBL.gff3
```

Now we have three test files `tair.gff3`, `fruitfly.gff3` and `ensembl.gff3` whose properties are described in more detail in Table 5.1. With these files a 1.5 GB large GFF3 file (for testing purposes) was created as follows (the file `all.gff3` was added multiple times in order to create a large enough output file):

```
gt merge -o all.gff3 tair.gff3 fruitfly.gff3 ensembl.gff3
gt merge -o 1.5GB.gff3 all.gff3 all.gff3 all.gff3 all.gff3 all.gff3 fruitfly.gff3
```

G.2 The Test Runs

To test the GFF3 parsing machinery of *GenomeTools* the `gff3` tool (see Section B.7) was used with option `-show no`. An overview of the used hardware and software setup is given in Appendix K. The runtime was measured with the UNIX `time` command. To measure the memory usage, the `massif` tool from the Valgrind [NS07] instrumentation framework was used (see `http://valgrind.org/`).

The following `parse.pl` script uses the *Bio::Tools:GFF* parser from the BioPerl [SBB+02] package (see `http://bioperl.org/`) to parse the GFF3 files.

```
#!/usr/bin/perl -w

use strict;
use Bio::Tools::GFF;

my $gff3file = $ARGV[0];

# create GFF3 parser
my $gffio = Bio::Tools::GFF->new(-file=>"$gff3file", -gff_version=>3);

while( my $feature = $gffio->next_feature() ) {
  # discard feature
}
```

The following `parse_and_store.pl` script is similar to the `parse.pl` script above, but stores the parsed features. This allows to measure the memory consumption of all features held in main memory.

```
#!/usr/bin/perl -w

use strict;
use Bio::Tools::GFF;

my $gff3file = $ARGV[0];

# create GFF3 parser
my $gffio = Bio::Tools::GFF->new(-file=>"$gff3file", -gff_version=>3);

my @features;

while( my $feature = $gffio->next_feature() ) {
  push @features, $feature # store feature
}
```

Appendix H

The nGASP Evaluation in Detail

H.1 Retrieving the Original Files

This appendix describes the nGASP evaluation (Section 6.1) in detail to make it reproducible. For the evaluation, the following nGASP files have been retrieved on 2009/07/04:

```
http://dev.wormbase.org/ngasp/datasets/training_regions.fa
http://dev.wormbase.org/ngasp/datasets/test_regions.fa
http://dev.wormbase.org/ngasp/datasets/protein_matches_complete_filtered.fa
http://dev.wormbase.org/ngasp/datasets/rna_matches.fa
http://dev.wormbase.org/ngasp/datasets/confirmed.gff3
http://dev.wormbase.org/ngasp/datasets/unconfirmed.gff3
ftp://ftp.wormbase.org/pub/wormbase/nGASP/wormbase_genes/test_regions_confirmed.gff3.gz
ftp://ftp.wormbase.org/pub/wormbase/nGASP/wormbase_genes/test_regions_unconfirmed.gff3.gz
```

An archive file containing all of the nGASP files listed above can be downloaded from:

```
ftp://genomethreader.org/pub/dissertation/ngasp.tar.gz
```

The file has the following MD5 hash:

```
9a668f29b14269b169b7a8905a80f75b  ngasp.tar.gz
```

H.2 Preparing the Annotation Files

The GFF3 files have been tidied up with the `gff3` tool (described in Section B.7) using the following commands[1]:

```
$ gt gff3 -tidy -addids no -o training_confirmed_tidy.gff3 confirmed.gff3
```

[1]The option `-addidsno` was used to prevent sequence-region collisions in the usage of the `id_to_md5` tool later on.

```
$ gt gff3 -tidy -addids no -o training_unconfirmed_tidy.gff3 unconfirmed.gff3

$ gt gff3 -tidy -addids no -o test_confirmed_tidy.gff3 test_regions_confirmed.gff3.gz

$ gt gff3 -tidy -addids no -o test_unconfirmed_tidy.gff3 test_regions_unconfirmed.gff3.gz
```

Afterwards, the sequence IDs contained in the files are replaced by the MD5 hashes of the actual sequences using the `id_to_md5` tool (described in Section B.11)[2]. This makes all subsequent mappings between a sequence ID and its corresponding sequence less error-prone.

```
$ gt id_to_md5 -usedesc -seqfile training_regions.fa -o training_confirmed_md5.gff3 training_confirmed_tidy.gf

$ gt id_to_md5 -usedesc -seqfile training_regions.fa -o training_unconfirmed_md5.gff3 training_unconfirmed_tid

$ gt id_to_md5 -usedesc -seqfile test_regions.fa -o test_confirmed_md5.gff3 test_confirmed_tidy.gff3

$ gt id_to_md5 -usedesc -seqfile test_regions.fa -o test_unconfirmed_md5.gff3 test_unconfirmed_tidy.gff3
```

For better processing, the GFF3 files have been sorted:

```
$ gt gff3 -sort -o training_confirmed_sorted.gff3 training_confirmed_md5.gff3

$ gt gff3 -sort -o training_unconfirmed_sorted.gff3 training_unconfirmed_md5.gff3

$ gt gff3 -sort -o test_confirmed_sorted.gff3 test_confirmed_md5.gff3

$ gt gff3 -sort -o test_unconfirmed_sorted.gff3 test_unconfirmed_md5.gff3
```

As described in Section 6.1.1 and in [CFM+08], two gene sets were used to evaluate the prediction qualities in the nGASP competition. **ref1** consists only of confirmed genes and **ref2** consists of confirmed and unconfirmed genes. The GFF3 files corresponding to these sets have been created as follows (for the training and test data, respectively):

```
$ cp training_confirmed_sorted.gff3 training_ref1.gff3

$ gt merge -o training_ref2.gff3 training_confirmed_sorted.gff3 training_unconfirmed_sorted.gff3

$ cp test_confirmed_sorted.gff3 test_ref1.gff3

$ gt merge -o test_ref2.gff3 test_confirmed_sorted.gff3 test_unconfirmed_sorted.gff3
```

The `merge` tool used above is described in Section B.14. At this point, the annotation files are ready for later processing. The stepwise process outlined above to create them could have been considerably shortened by using pipes to connect the tools directly instead of using intermediate files. It was done in this way for explanatory purposes. The combined calls are as follows:

[2]The reverse operation can be performed with the `md5_to_id` tool described in Section B.13

```
$ gt gff3 -tidy -addids no confirmed.gff3 |
  gt id_to_md5 -usedesc -seqfile training_regions.fa |
  gt gff3 -sort -o training_ref1.gff3

$ gt gff3 -tidy -addids no confirmed.gff3 unconfirmed.gff3 |
  gt id_to_md5 -usedesc -seqfile training_regions.fa |
  gt gff3 -sort -o training_ref2.gff3

$ gt gff3 -tidy -addids no test_regions_confirmed.gff3.gz |
  gt id_to_md5 -usedesc -seqfile test_regions.fa |
  gt gff3 -sort -o test_ref1.gff3

$ gt gff3 -tidy -addids no test_regions_confirmed.gff3.gz test_regions_unconfirmed.gff3.gz |
  gt id_to_md5 -usedesc -seqfile test_regions.fa |
  gt gff3 -sort -o test_ref2.gff3
```

H.3 Preparing the Sequence Files

The protein sequence file needs some preparation, because the protein sequence do not contain stop amino acids, which are improving the predictions made by *GenomeThreader*. They have been added with the `seqtransform` tool described in Section B.18:

```
$ gt seqtransform -addstopaminos -o protein_matches.fa protein_matches_complete_filtered.fa
```

H.4 Creating a Custom Nematode BSSM

With the given training data we can train a custom nematode BSSM to improve the prediction.

The `splicesiteinfo` tool (described in Section B.20) shows that GT donor sites are predominant in the training set and there are not enough GC donor sites to train a custom BSSM model:

```
$ gt splicesiteinfo -seqfile training_regions.fa training_ref2.gff3
gc:   0.48% (n=46)
gt:  98.33% (n=9420)
```

To train the model the `gthbssmtrain` tool was used to extract BSSM training data from the training set **ref2**:

```
$ gthbssmtrain -filtertype CDS -seed 2040532791 -seqfile training_regions.fa training_ref2.gff3
gt-ag:  98.13% (n=9401)
gc-ag:   0.46% (n=44)
```

Afterwards the actual BSSM file `ngasp.bssm` has been build from the extracted training data, containing a GT donor and an AG acceptor model:

```
$ gthbssmbuild -bssmfile ngasp.bssm -datapath training_data -gtdonor -agacceptor
```

The separation of this process into the two tools `gthbssmtrain` tool and `gthbssmbuild` tool allows to create BSSM files with different splice sites models (with the `gthbssmbuild` tool) from the same training data (without the need to rerun the `gthbssmtrain` tool).

H.5 Predicting the Genes

Before we call *GenomeThreader* we need to determine the intron length ranges. The options were chosen as follows.

With the help of the option `-intronlengthdistri` of the `stat` tool (see Section B.21) we looked at the distribution of intron length in training file **ref2**. The shortest intron length in this file was 15, the longest 103002 and the second longest 20948 (all three lengths occurring exactly once). Therefore we use option `-dpminintronlen` to 10 and `-gcmaxgapwidth` to 22000 (treating the longest intron as an outlier and adding some tolerance).

Because chain enrichment (see Section 4.5.4) does not improve the gene prediction results significantly (see Section 6.1.6), the option `-enrichchains` was not used by default.

To include all exons in the similarity score computation we set option `-scoreminexonlen` to 1. Furthermore, option `-maskpolyatails` was used to mask poly(A)-tails in the cDNA file.

To exclude very bad alignments from the consensus spliced alignment phase (see Section 4.7 option `-minalignmentscore` was set to 0.85. This options sets a minimum similarity score a spliced alignment must have to be added to the set of spliced alignments and be included in the consensus spliced alignment phase.

Since we predict complete genes here, we require that an ORF must begin with a start codon by setting option `-startcodon` to `yes`.

Because we are only interested in the predicted coding sequences of the consensus spliced alignment phase in GFF3 format, the options `-skipalignmentout` and `-gff3out` have been used. Furthermore, to be able to compare the results with the annotation files written above, the option `-md5ids` was used to show MD5 fingerprints as sequence IDs in the GFF3 output.

Finally, the complete call looked like this:

```
gth
-genomic test_regions.fa
-cdna rna_matches.fa
-protein protein_matches.fa
-bssm ngasp
-dpminintronlen 10
-gcmaxgapwidth 22000
-scoreminexonlen 1
-maskpolyatails
-minalignmentscore 0.85
-startcodon yes
-skipalignmentout
-gff3out
-md5ids
-o gth_bssm.gff3
```

Table H.1 gives an overview of the options used on the nGASP dataset and their descriptions.

Option	Description
`-genomic test_regions.fa`	genomic input file
`-cdna rna_matches.fa`	cDNA/EST input file
`-protein protein_matches.fa`	protein input file
`-bssm ngasp`	BSSM parameter file
`-dpminintronlen 10`	the minimum intron length for the dynamic programming
`-gcmaxgapwidth 22000`	the maximum gap width for global chains
`-scoreminexonlen 1`	the minimum exon length (for score computation)
`-maskpolyatails`	mask poly(A)-tails in the cDNA/EST files
`-minalignmentscore 0.85`	minimum similarity score a spliced alignment must have
`-startcodon yes`	require than an ORF must begin with a start codon
`-skipalignmentout`	skip output of spliced alignments
`-gff3out`	show output in GFF3 format
`-md5ids`	show MD5 fingerprints as sequence IDs
`-o gth_bssm.gff3`	redirect output to specified file

Table H.1: *GenomeThreader* nGASP options.

H.6 Alternative Approach Using Intermediate Files

An alternative approach to compute the gene predictions of the previous Section is to use intermediate files. First, we call *GenomeThreader* in a similar fashion, but write the results in the the intermediate XML format using the options `-intermediate` and `-xmlout`. This intermediate format stores the computed spliced alignments in a lossless fashion and allows to derive all other output formats from it. To save disk space we compress the output file on the fly using the option `-gzip`:

```
gth
-genomic test_regions.fa
-cdna rna_matches.fa
-protein protein_matches.fa
-bssm ngasp
-dpminintronlen 10
-gcmaxgapwidth 22000
-scoreminexonlen 1
-maskpolyatails
-intermediate
-xmlout
-gzip
-o ngasp.inter.gz
```

From output file a sorted intermediate GFF3 file was derived (using `gthconsensus`) with the following command:

```
gthconsensus -intermediate -gff3out -md5ids -o ngasp.inter.gff3 ngasp.inter.gz
```

With the following three tools we can reproduce the results from the previous Section:

1. The `select` tool (described in Section B.16) allows us with option `-mingenescore` to select only the spliced alignments above a certain score, similar to `-minalignmentscore` passed to `gth`.

2. The `csa` tool (described in Section B.4) computes the consensus spliced alignments in exactly the same fashion as it is done in *GenomeThreader*, but based on GFF3 input.

3. The `cds` tool (described in Section B.3) adds the coding sequences (CDS) to the consensus spliced alignments.

The complete command looks like this:

```
gt select -mingenescore 0.85 ngasp.inter.gff3 |
gt csa |
gt cds -seqfile test_regions.fa -startcodon yes -o gth_bssm.gff3
```

Of course, the direct approach described in the previous Section is the simpler one, but this approach allows to test different parameter settings (for example, different thresholds passed to the `select` tool or different minimum ORF lengths passed to the `cds` tool) without the need to recompute the spliced alignments with `gth` or parsing the versatile intermediate XML format with `gthconsensus` multiple times.

Using the efficient GFF3-based tools from *GenomeTools* allows to try different parameter setting in near "real-time" which makes it much more convenient – the user is not interrupted by long waiting times and the multitasking that usually implies.

H.7 Evaluating the Predictions

The final prediction stored in the file `gth_bssm.gff3` was evaluated by the `eval` tool described in Section B.5.

As it was done in the nGASP paper [CFM+08], the `ref1` test set was used to assess the sensitivity of the gene predictions and the `ref2` to assess the specificity. From the output of the `eval` tool, the CDS level (collapsed) was used to pull out the corresponding values, because this corresponds to the metric used in the paper. The actual calls looked like this:

```
gt eval test_ref1.gff3 gth_bssm.gff3
gt eval test_ref2.gff3 gth_bssm.gff3
```

H.8 Evaluating the Competing Prediction Programs

The sensitivity and specificity values of the competing gene prediction programs have also been computed with the `eval` tool from their GFF3 files submitted to the nGASP competition.

We downloaded all annotations in category 1-3. Category 3 is the relevant category to compare *GenomeThreader* against ("gene-finders that take advantage of alignments of expressed sequences such as proteins, ESTs, and assembled mRNAs" [CFM+08]). From

```
ftp://ftp.wormbase.org/pub/wormbase/nGASP/submissions_phaseI/
```

the following files have been retrieved on 2010/12/05:

```
GENE_validated.gff3.1
AUGUSTUS_validated.gff3.2
CRAIG_validated.gff3.1
EUGENE_validated.gff3
EXONHUNTER_validated.gff3.2
FGENESH++_validated.gff3.1
FGENESH_validated.gff3.1
GENEID_validated.gff3
GENEMARKHMM_validated.gff3.3
GLIMMERHMM_validated.gff3.2
GRAMENE_validated.gff3.1
MAKER_validated.gff3.1
MGENE_validated.gff3.1
NSCAN_validated.gff3.3
SGP2_validated.gff3
SNAP_validated.gff3.2
```

An archive file containing all of the annotation files listed above and a preparation script `prepare.sh` can be downloaded from:

```
ftp://genomethreader.org/pub/dissertation/ngasp_competitors.tar.gz
```

The file has the following MD5 hash:

```
27df5958d9d2abf1441bb3a3020a2885  ngasp_competitors.tar.gz
```

For the evaluation the annotation files have been prepared with the `prepare.sh` preparation script:

```
$ prepare.sh
```

The preprocessed annotation file have then been used to compute the results of the competing programs with the `eval` tool, similar to the approach used in Section H.7. For example like this (for *agene*):

```
$ gt eval test_ref1.gff3 agene.gff3
$ gt eval test_ref2.gff3 agene.gff3
```

The computed results differ slightly (they are a little bit worse) than the results published in the paper. To find the reason for the difference between the sensitivity and specificity values computed by the `eval` tool and the script used in [CFM+08] we tried multiple times to receive the evaluation script from the authors of the publication. Although it was promised, we didn't receive the script.

Appendix I

Mapping 454 Sequences in Detail

I.1 Used Files

This appendix describes the mapping of 454 sequences (Section 6.2) in detail to make it reproducible. For the evaluation, the following files from the nGASP evaluation (see Section 6.1 and Appendix H) have been reused:

```
test_ref1.gff3
test_ref2.gff3
test_regions.fa
```

An archive file containing all files which are relevant for the 454 mapping can be downloaded from:

```
ftp://genomethreader.org/pub/dissertation/454.tar.gz
```

The file has the following MD5 hash:

```
298e4f9bac7791103402c72acf036295  454.tar.gz
```

I.2 Extract mRNA Sequences

The mRNA sequences were extracted from gene reference set files `test_ref1.gff3` and `test_ref2.gff3` and the sequence file `test_regions.fa` with the help of the `extractfeat` tool (see Section B.6) as follows:

```
$ gt extractfeat -type CDS -join -seqfile test_regions.fa -o test_ref1.fa test_ref1.gff3
$ gt extractfeat -type CDS -join -seqfile test_regions.fa -o test_ref2.fa test_ref2.gff3
```

The resulting mRNA sequences files `test_ref1.fa` and `test_ref2.fa` contain 894 and 2549 sequences, respectively.

I.3 Simulate 454 Reads

The 454 reads were simulated with *Flowsim* [BML+10] with different number of reads (x-fold the number of sequences in the mRNA file, for $x \in \{5, 10, 25, 50, 100, 200\}$).

For example, for a 10-fold number of reads the following commands were used:

```
clonesim -c 8940 test_ref1.fa | kitsim | flowsim -o test_ref1x10.sff
clonesim -c 25490 test_ref2.fa | kitsim | flowsim -o test_ref2x10.sff
```

The read simulations result produce SFF files, which are usually used to store 454 sequence reads. To extract sequences in FASTA format from the SFF files, the `sff_extract` script was used. See Appendix K for a detailed overview of the hardware and software setup.

I.4 Align 454 Reads

The simulated 454 read files were aligned with *GenomeThreader* using the similar parameter setting as documented in Table H.1. For example, to align the 5-fold reads from the **ref1** gene set against the test regions contained the following command was used:

```
gth
-genomic test_regions.fa
-cdna test_ref1x5.fasta
-bssm ngasp
-dpminintronlen 10
-gcmaxgapwidth 22000
-scoreminexonlen 1
-maskpolyatails
-minalignmentscore 0.85
-startcodon yes
-skipalignmentout
-gff3out
-md5ids
-o gth_ref1x5.gff3
```

The evaluation of the results was done as similar to the evaluation described in Section H.7. All test scripts and the result files are also contained in the `454.tar.gz` file mentioned above.

Appendix J

The ENCODE Evaluation in Detail

J.1 Retrieving the Original Files

This appendix describes the ENCODE evaluation (Section 6.3) in detail to make it reproducible. For the evaluation, the following ENCODE files have been retrieved on 2012/04/05:

```
http://hgdownload.cse.ucsc.edu/goldenPath/hg18/encode/regions/hg18.fa.gz
http://hgdownload.cse.ucsc.edu/goldenPath/hg18/encode/database/encodeGencodeGeneKnownMar07.sql
http://hgdownload.cse.ucsc.edu/goldenPath/hg18/encode/database/encodeGencodeGeneKnownMar07.txt.gz
```

An archive file containing all of the ENCODE files listed above (in uncompressed form) can be downloaded from:

```
ftp://genomethreader.org/pub/dissertation/encode.tar.gz
```

The file has the following MD5 hash:

```
367c9dc92232b24b8da27d389c892f66  encode.tar.gz
```

J.2 Preparing the Annotation Files

With the script `encodesql2gff3` contained in the *GenomeTools* distribution, we convert the ENCODE SQL dump to GFF3:

```
encodesql2gff3 encodeGencodeGeneKnownMar07.txt | gt gff3 -tidy -sort -o gencode_chr.gff3
```

We consider only protein-coding genes:

```
gt select -hascds -o gencode_cds.gff3 gencode_chr.gff3
```

The positions in resulting file `gencode_cds.gff3` refers to chromosomes and not to the ENCODE regions contained in the downloaded sequence file `hg18.fa`. Therefore, we change the chromosome positions to region positions a script named `create_gencode_region_file`. The script uses the `chseqids` tool and `select` tool which are described in Appendices B.2 and B.16, respectively. This results in a GFF3 file `gencode_regions.gff3`. Finally, we map the IDs contained in this files to MD5s for better usability:

```
gt id_to_md5 -seqfile hg18.fa -usedesc gencode_regions.gff3 | gt gff3 -sort -o gencode.gff3
```

J.3 Extract mRNA Sequences

The mRNA sequences were extracted from gene reference set file `gencode.gff3` and the sequence file `hg18.fa` with the help of the `extractfeat` tool (see Section B.6) as follows:

```
gt extractfeat -seqfile hg18.fa -type exon -join -o mRNAs_rate_0.fas gencode.gff3
```

To mutate the sequences with different mutation rates, the `seqmutate` tool was used like in the following example:

```
gt extractfeat -seqfile hg18.fa -type exon -join gencode.gff3 | gt seqmutate -rate 5 -o mRNAs_rate_5.fas -
```

The mutation procedure is described in Section 3.7.1 and the `seqmutate` manual is given in Appendix B.17.

J.4 Aligning with *GenomeThreader*

To align the simulated mRNAs with *GenomeThreader*, different parameter combinations were tested. Table J.1 options shows the used options and their used parameters. To make the results comparable with *GMAP*, only the spliced alignments have been used to predict genes (that is, *GenomeThreader* was used without the consensus spliced alignment phase). The resulting GFF3 files were processed with the `select` tool before the evaluation (see Appendix B.16). Thereby, the options `-targetbest` and `-mingenescore` were used. With option `-targetbest` only the best alignment for every simulated mRNA is used and with `-mingenescore` only alignments above a certain minimum alignment score are kept. The minimum alignment score was set according to the mutation rate of the simulated mRNAs as documented in Table J.2.

Option	Used Parameter	Comment
`-genomic`	`hg18.fa`	
`-cdna`	`mRNAs_rate_x.fa`	for $x \in \{0, 1, 3, 5, 10, 15\}$
`-species`	`human`	
`-scoreminexonlen`	`1`	
`-maskpolyatails`	`true`	default
`-intermediate`	`true`	default
`-gff3out`	`true`	default
`-md5ids`	`true`	default
`-introncutout`	`true`	default
`-autointroncutout`	`1024`	
`-enrichchains`	`true`, `false`	
`-seedlength`	`12`, `15`, `18`	combined with `-minmatchlen`. That is, only $\{12, 12\}$, $\{15, 15\}$, and $\{18, 20\}$ were used
`-minmatchlen`	`12`, `15`, `20`	
`-o`	`gth_rate_x.gff3`	

Table J.1: *GenomeThreader* ENCODE options.

Mutation Rate	Min. Alignment Score
< 3	0.95
≥ 3 and ≤ 5	0.90
> 5 and ≤ 10	0.85

Table J.2: Minimum alignment scores for different mutation rates.

J.5 Aligning with *GMAP*

To align the simulated mRNAs with *GMAP* we first had to build an index for the genomic sequence file:

```
$ gmap_build -d hg18 hg18.fa
```

Afterwards, *GMAP* could be used to map the mRNAs against that index. We set the output format to GFF3 and run *GMAP* once with option `-n 1` and once without on the different mRNA files. Option `-n 1` lets *GMAP* keep only the best alignment path for every mRNA.

The complete call looks like this:

```
$ gmap -d hg18 -f gff3_gene -n 1 mRNAs_rate_0.fas
```

The `id_to_md5` tool (see Appendix B.11) with option `-matchdesc` was used to change the GFF3 sequence IDs to MD5 fingerprints.

J.6 Evaluating the Prediction Results

The evaluation of the results was done with the `eval` tool (documented in Appendix B.5) like this:

```
$ gt eval gencode.gff3 gmap_rate_0.gff3
```

Appendix K

Hardware and Software Setup

The test runs have been performed on an Intel i7-920 Quad-Core processor with 8 GB memory running Ubuntu Linux 10.04.2 LTS. Table K.1 shows the versions of all software packages used in this thesis.

Program	Version	Notes
GenomeThreader	1.5.2	64-bit version, `http://genomethreader.org`
GenomeTools	1.4.2	64-bit version, `http://genometools.org`
GMAP	2012-06-02	`http://research-pub.gene.com/gmap/`
Flowsim	0.3	from Biohaskell (`http://biohaskell.org`)
`sff_extract`	0.2.13	`http://bioinf.comav.upv.es/sff_extract/`

Table K.1: Used software versions for all packages in this thesis.

Bibliography

[AGA+08] P. Abad, J. Gouzy, J.-M. Aury, P. Castagnone-Sereno, E.G.J. Danchin, E. Deleury, L. Perfus-Barbeoch, V. Anthouard, F. Artiguenave, V.C. Blok, M.-C. Caillaud, P.M. Coutinho, C. Dasilva, F. De Luca, F. Deau, M. Esquibet, T. Flutre, J.V. Goldstone, N. Hamamouch, T. Hewezi, O. Jaillon, C. Jubin, P. Leonetti, M. Magliano, T.R. Maier, G.V. Markov, P. McVeigh, G. Pesole, J. Poulain, M. Robinson-Rechavi, E. Sallet, B. Segurens, D. Steinbach, T. Tytgat, E. Ugarte, C. van Ghelder, P. Veronico, T.J. Baum, M. Blaxter, T. Bleve-Zacheo, E.L. Davis, J.J. Ewbank, B. Favery, E. Grenier, B. Henrissat, J.T. Jones, V. Laudet, A.G. Maule, H. Quesneville, M.-N. Rosso, T. Schiex, G. Smant, J. Weissenbach, and P. Wincker. Genome Sequence of the Metazoan Plant-Parasitic Nematode *Meloidogyne incognita*. *Nature Biotechnology*, **89**(26):909–915, 2008.

[AGM+90] S.F. Altschul, W. Gish, W. Miller, E.W. Myers, and D.J. Lipman. Basic Local Alignment Search Tool. *J. Mol. Biol.*, **215**:403–410, 1990.

[AJL+02] B. Alberts, A. Johnson, J. Lewis, M. Raff, K. Roberts, and P. Walter. *Molecular Biology of the Cell*. Garland Science, 2002.

[AKB11] M. Alawi, S. Kurtz, and M. Beckstette. CASSys: An Integrated Software System for the Interactive Analysis of ChIP-Seq Data. *J. Integr. Bioinform.*, **8**(2):155, 2011.

[AKO04] M.I. Abouelhoda, S. Kurtz, and E. Ohlebusch. Replacing Suffix Trees with Enhanced Suffix Arrays. *Journal of Discrete Algorithms*, **2**:53–86, 2004.

[AMM44] O.T. Avery, C.M. MacLeod, and M. McCarty. Studies on the Chemical Nature of the Substance Inducing Transformation of Pneumococcal Types. *J. Exp. Med.*, **79**(2):137–158, 1944.

[AMS+97] S.F. Altschul, T.L. Madden, A.A. Schäffer, J. Zhang, Z. Zhang, W. Miller, and D.J. Lipman. Gapped BLAST and PSI-BLAST: A New Generation of Protein Database Search Programs. *Nucleic Acids Res.*, **25**(17):3389–3402, 1997.

[AS05] J.E. Allen and S.L. Salzberg. JIGSAW: Integration of Multiple Sources of Evidence for Gene Prediction. *Bioinformatics*, 21(**18**):3596–3603, 2005.

[ATE+06] P. Akiva, A. Toporik, S. Edelheit, Y. Peretz, A. Diber, R. Shemesh, A. Novik, and R. Sorek. Transcription-Mediated Gene Fusion in the Human Genome. *Genome Res.*, **16**:30–36, 2006.

[BAG+10] A. Bazzini, R. Asís, V. González, S. Bassi, M. Conte, M. Soria, A. Fernie, S. Asurmendi, and F. Carrari. miSolRNA: A Tomato Micro RNA Relational Database. *BMC Plant Biology*, **10**(1):240, 2010.

[Bat] W. Bateson. Letter from William Bateson to Alan Sedgwick in 1905. The John Innes Centre, `http://www.jic.ac.uk/corporate/about/bateson.htm`. Retrived on 2008-05-01.

[BBG+06] R. Bruggmann, A.K. Bharti, H. Gundlach, J. Lai, S. Young, A.C. Pontaroli, F. Wei, G. Haberer, G. Fuks, C. Du, C. Raymond, M.C. Estep, R. Liu, J.L. Bennetzen, A.P. Chan, P.D. Rabinowicz, J. Quackenbush, W.B. Barbazuk, R.A. Wing, B. Birren, C. Nusbaum, S. Rounsley, K.F.X. Mayer, and J. Messing. Uneven Chromosome Contraction and Expansion in the Maize Genome. *Genome Res.*, **16**(10):1241–1251, 2006.

[BBLV05] B. Brejová, D.G. Brown, M. Li, and T. Vinař. ExonHunter: A Comprehensive Approach to Gene Finding. *Bioinformatics*, **21**(suppl 1):i57–i65, 2005.

[BCD04] E. Birney, M. Clamp, and R. Durbin. GeneWise and Genomewise. *Genome Res.*, **14**(5):988–995, 2004.

[BD97] E. Birney and R. Durbin. Dynamite: A Flexible Code Generation Language for Dynamic Programming Methods Used in Sequence Comparison. In *Proc. of ISMB 97*, pages 56–64, 1997.

[Bel57] R. Bellman. *Dynamic Programming*. Princeton University Press, 1957.

[BG96] M. Burset and R. Guigó. Evaluation of Gene Structure Prediction Programs. *Genomics*, **34**(3):353–367, 1996.

[BG04] M.R. Brent and R. Guigo. Recent Advances in Gene Structure Prediction. *Curr. Opin. Struct. Biol.*, **14**(3):264–272, 2004.

[BH00] V. Bafna and D.H. Huson. The Conserved Exon Method for Gene Finding. In *Proc. 8th Int. Conference on Intelligent Systems for Molecular Biology*, pages 3–12. AAAI Press, 2000.

[BJV+07] A. Ballvora, A. Jocker, P. Viehover, H. Ishihara, J. Paal, K. Meksem, R. Bruggmann, H. Schoof, B. Weisshaar, and C. Gebhardt. Comparative Sequence Analysis of Solanum and Arabidopsis in a Hot Spot for Pathogen Resistance on Potato Chromosome V Reveals a Patchwork of Conserved and Rapidly Evolving Genome Segments. *BMC Genomics*, **8**(1):112, 2007.

[BK97] C. Burge and S. Karlin. Prediction of Complete Gene Structures in Human Genomic DNA. *J. Mol. Biol.*, **268**:78–94, 1997.

[BK98] C.B. Burge and S. Karlin. Finding the Genes in Genomic DNA. *Curr. Opin. Struct. Biol.*, **8**(3):346–354, 1998.

[BLT93] M.S. Boguski, T.M.J. Lowe, and C.M. Tolstoshev. dbEST – Database for "Expressed Sequence Tags". *Nature Genetics*, **4**:332–333, 1993.

[Blu05] T. Blumenthal. Trans-Splicing and Operons. In The *C. elegans* Research Community, editor, *WormBook*. 2005. doi/10.1895/wormbook.1.5.1, http://www.wormbook.org.

[BML+10] S. Balzer, K. Malde, A. Lanzén, A. Sharma, and I. Jonassen. Characteristics of 454 Pyrosequencing Data—Enabling Realistic Simulation with Flowsim. *Bioinformatics*, **26**(18):i420–i425, 2010.

[BMS77] S.M. Berget, C. Moore, and P.A. Sharp. Spliced Segments at the 5' Terminus of Adenovirus 2 Late mRNA. *Proc. Nat. Acad. Sci. U.S.A.*, **74**:3171–3175, 1977.

[BPM+00] S. Batzoglu, L. Pachter, J.P. Mesirov, B. Berger, and E.S. Lander. Human and Mouse Gene Structure: Comparative Analysis and Application to Exon Prediction. *Genome Res.*, **10**:950–958, 2000.

[Bre02] M.R. Brent. Predicting Full-Length Transcripts. *Trends Biotechnol.*, **20**(7):273–275, 2002. News.

[Bre05] M.R. Brent. Genome Annotation Past, Present, and Future: How to Define an ORF at Each Locus. *Genome Res.*, **15**(12):1777–1786, 2005.

[Bre08] M.R. Brent. Steady Progress and Recent Breakthroughs in the Accuracy of Automated Genome Annotation. *Nature Reviews Genetics*, **9**:62–73, 2008.

[BS97] J.L. Bentley and R. Sedgewick. Fast algorithms for sorting and searching strings. In *SODA '97: Proceedings of the eighth annual ACM-SIAM symposium on Discrete algorithms*, pages 360–369, Philadelphia, PA, USA, 1997. Society for Industrial and Applied Mathematics.

[BSR+04] P. Bertone, V. Stolc, T.E. Royce, J.S. Rozowsky, A.E. Urban, X. Zhu, J.L. Rinn, W. Tongprasit, M. Samanta, S. Weissman, M. Gerstein, and M. Snyder. Global Identification of Human Transcribed Sequences with Genome Tiling Arrays. *Science*, **306**:2242–2246, 2004.

[BT41] G.W. Beadle and E.L. Tatum. Genetic Control of Biochemical Reactions in *Neurospora*. *Proc. Nat. Acad. Sci. U.S.A.*, **27**(11):499–506, 1941.

[BXZ04] V. Brendel, L. Xing, and W. Zhu. Gene Structure Prediction from Consensus Spliced Alignment of Multiple ESTs Matching the Same Genomic Locus. *Bioinformatics*, **20**(7):1157–1169, 2004.

[CBM08] M. Calviõ, R. Bruggmann, and J. Messing. Screen of Genes Linked to High-Sugar Content in Stems by Comparative Genomics. *Rice*, **1**:166–176, 2008.

[CBN+04] S. Cawley, S. Bekiranov, H.H. Ng, P. Kapranov, E.A. Sekinger, D. Kampa, A. Piccolboni, V. Sementchenko, J. Cheng, A.J. Williams, R. Wheeler, B. Wong, J. Drenkow, M. Yamanaka, S. Patel, S. Brubaker, H. Tammana, G. Helt, K. Struhl, and T.R. Gingeras. Unbiased Mapping of Transcription Factor Binding Sites along Human Chromosomes 21 and 22 Points to Widespread Regulation of Noncoding RNAs. *Cell*, **116**:499–509, 2004.

[CD07] A. Coghlan and R. Durbin. Genomix: A Method for Combining Gene-Finders' Predictions, Which Uses Evolutionary Conservation of Sequence and Intron-Exon Structure. *Bioinformatics*, **23**(12):1468–1475, 2007.

[CDT+08] M.L. Chiusano, N. D'Agostino, A. Traini, C. Licciardello, E. Raimondo, M. Aversano, L. Frusciante, and L. Monti. ISOL@: an Italian SOLAnaceae Genomics Resource. *BMC Bioinformatics*, **9**(Suppl 2):S7, 2008.

[CFM+08] A. Coghlan, T. Fiedler, S. McKay, P. Flicek, T. Harris, D. Blasiar, the nGASP Consortium, and L. Stein. nGASP - The Nematode Genome Annotation Assessment Project. *BMC Bioinformatics*, **9**(1):549, 2008.

[CGBR77] L.T. Chow, R.E. Gelinas, T.R. Broker, and R.J. Roberts. An Amazing Sequence Arrangement at the 5' Ends of Adenovirus 2 Messenger RNA. *Cell*, **12**:1–8, 1977.

[Cha51] E. Chargaff. Some Recent Studies on the Composition and Structure of Nucleic Acids. *J. Cell Physiol. Suppl.*, **38**:41–59, 1951.

[CKD+05] J. Cheng, P. Kapranov, J. Drenkow, S. Dike, S. Brubaker, S. Patel, J. Long, D. Stern, H. Tammana, G. Helt, V. Sementchenko, A. Piccolboni, S. Bekiranov, D.K. Bailey, M. Ganesh, S. Ghosh, I. Bell, D.S. Gerhard, and T.R. Gingeras. Transcriptional Maps of 10 Human Chromosomes at 5-Nucleotide Resolution. *Science*, **308**:1149–1154, 2005.

[CKR+08] B.L. Cantarel, I. Korf, S.M.C. Robb, G. Parra, E. Ross, B. Moore, C. Holt, A. Sánchez Alvarado, and M. Yandell. MAKER: An Easy-to-Use Annotation Pipeline Designed for Emerging Model Organism Genomes. *Genome Res.*, **18**(1):188–196, 2008.

[CLR90] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 1990.

[Coc10] Cock, J.M. and Sterck, L. and Rouze, P. and Scornet, D. and Allen, A.E. and Amoutzias, G. and Anthouard, V. and Artiguenave, F. and Aury, J.-M. and Badger, J.H. and Beszteri, B. and Billiau, K. and Bonnet, E. and Bothwell, J.H. and Bowler, C. and Boyen, C. and Brownlee, C. and Carrano, C.J. and Charrier, B. and Cho, G.Y. and Coelho, S.M. and Collen, J. and Corre, E. and Da Silva, C. and Delage, L. and Delaroque, N. and Dittami, S.M. and Doulbeau, S. and Elias, M. and Farnham, G. and Gachon, C.M.M. and Gschloessl, B. and Heesch, S. and Jabbari, K. and Jubin, C. and Kawai, H. and Kimura, K. and Kloareg, B. and Kupper, F.C. and Lang, D. and Le Bail, A. and Leblanc, C. and Lerouge, P. and Lohr, M. and Lopez, P.J. and Martens, C. and Maumus, F. and Michel, G. and Miranda-Saavedra, D. and Morales, J. and Moreau, H. and Motomura, T. and Nagasato, C. and Napoli, C.A. and Nelson, D.R. and Nyvall-Collen, P. and Peters, A.F. and Pommier, C. and Potin, P. and Poulain, J. and Quesneville, H. and Read, B. and Rensing, S.A. and Ritter, A. and Rousvoal, S. and Samanta, M. and Samson, G. and Schroeder, D.C. and Segurens, B. and Strittmatter, M. and Tonon, T. and Tregear, J.W. and Valentin, K. and von Dassow, P. and Yamagishi, T. and Van de Peer, Y. and Wincker, P. The *Ectocarpus* Genome and the Independent Evolution of Multicellularity in Brown Algae. *Nature*, **465**:617–621, 2010.

[Con04] The ENCODE Project Consortium. The ENCODE (ENCyclopedia Of DNA Elements) Project. *Science*, **306**:636–640, 2004.

[Con07] The ENCODE Project Consortium. Identification and Analysis of Functional Elements in 1% of the Human Genome by the ENCODE Pilot Project. *Nature*, **447**:799–816, 2007.

[Con12] The UniProt Consortium. Reorganizing the Protein Space at the Universal Protein Resource (UniProt). *Nucleic Acids Res.*, **40**(D1):D71–D75, 2012.

[Cri51] F.H.C. Crick. On Protein Synthesis. *Symp. Soc. Exp. Biol.*, **XII**:138–163, 1951.

[CRVdVF77] R. Contreras, R. Rogiers, A. Van de Voorde, and W. Fiers. Overlapping of the VP2-VP3 Gene and the VP1 Gene in the SV40 Genome. *Cell*, **12**:529–538, 1977.

[Cyg] Cygwin Project. `http://cygwin.com/`. Retrieved on 2011-06-12.

[DFM+08] J. Duvick, A. Fu, U. Muppirala, M. Sabharwal, M.D. Wilkerson, C.J. Lawrence, C. Lushbough, and V. Brendel. PlantGDB: A Resource for Comparative Plant Genomics. *Nucleic Acids Res.*, **36**(suppl 1):D959–D965, 2008.

[DTFC09] N. D'Agostino, A. Traini, L. Frusciante, and M. Chiusano. SolEST Database: A "One-Stop Shop" Approach to the Study of *Solanaceae* Transcriptomes. *BMC Plant Biology*, **9**(1):142, 2009.

[DWB07] Q. Dong, M. Wilkerson, and V. Brendel. *Tracembler* - Software for *in-silico* Chromosome Walking in Unassembled Genomes. *BMC Bioinformatics*, **8**(1):151, 2007.

[Edd11] S.R. Eddy. Accelerated Profile HMM Searches. *PLoS Comput. Biol.*, **7**(10):e1002195, October 2011.

[EKW08] D. Ellinghaus, S. Kurtz, and U. Willhoeft. LTRharvest, an efficient and flexible software for *de novo* detection of LTR retrotransposons. *BMC Bioinformatics*, **9**:18, 2008.

[ELM+05] K. Eilbeck, S. Lewis, C. Mungall, M. Yandell, L. Stein, R. Durbin, and M. Ashburner. The Sequence Ontology: A Tool for the Unification of Genome Annotations. *Genome Biology*, **6**(5):R44, 2005.

[EMR+07] C. Elsik, A. Mackey, J. Reese, N. Milshina, D. Roos, and G. Weinstock. Creating a Honey Bee Consensus Gene Set. *Genome Biology*, **8**(1):R13, 2007.

[ERB+04] G. Euskirchen, T.E. Royce, P. Bertone, R. Martone, J.L. Rinn, F.K. Nelson, F. Sayward, N.M. Luscombe, P. Miller, M. Gerstein, S. Weissman, and M. Snyder. CREB Binds to Multiple Loci on Human Chromosome 22. *Mol. Cell. Biol.*, **24**:3804–3814, 2004.

[FAB+11] P. Flicek, M.R. Amode, D. Barrell, K. Beal, S. Brent, Y. Chen, P. Clapham, G. Coates, S. Fairley, S. Fitzgerald, L. Gordon, M. Hendrix, T. Hourlier, N. Johnson, A. Kähäri, D. Keefe, S. Keenan, R. Kinsella, F. Kokocinski, E. Kulesha, P. Larsson, I. Longden, W. McLaren, B. Overduin, B. Pritchard, H.S. Riat, D. Rios, G.R.S. Ritchie, M. Ruffier, M. Schuster, D. Sobral, G. Spudich, Y.A. Tang, S. Trevanion, J. Vandrovcova, A.J. Vilella, S. White, S.P. Wilder, A. Zadissa, J. Zamora, B.L. Aken, E. Birney, F. Cunningham, I. Dunham, R. Durbin, X.M. Fernández-Suarez, J. Herrero, T.J.P. Hubbard, A. Parker, G. Proctor, J. Vogel, and S.M.J. Searle. Ensembl 2011. *Nucleic Acids Res.*, **39**(suppl 1):D800–D806, 2011.

[FAW+95] R.D. Fleischmann, M.D. Adams, O. White, R.A. Clayton, E.F. Kirkness, A.R. Kerlavage, C.J. Bult, J.F. Tomb, B.A. Dougherty, J.M. Merrick, and et al. Whole-Genome Random Sequencing and Assembly of *Haemophilus influenzae* Rd. *Science*, **269**:496–512, 1995.

[FCD+76] W. Fiers, R. Contreras, F. Duerinck, G. Haegeman, D. Iserentant, J. Merregaert, W. Min Jou, F. Molemans, A. Raeymaekers, A. Van den Berghe, G. Volckaert, and M. Ysebaert. Complete Nucleotide Sequence of Bacteriophage MS2 RNA: Primary and Secondary Structure of the Replicase Gene. *Nature*, **260**:500–507, 1976.

[FCDW+71] W. Fiers, R. Contreras, R. De Wachter, G. Haegeman, J. Merregaert, W.M. Jou, and A. Vandenberghe. Recent Progress in the Sequence Determination of Bacteriophage MS2 RNA. *Biochimie*, **53**(4):495–506, 1971.

[FHZ+98] L. Florea, G. Hartzell, Z. Zhang, G.M. Rubin, and W. Miller. A Computer Program for Aligning a cDNA Sequence with a Genomic DNA Sequence. *Genome Res.*, **8**(9):967–974, 1998.

[Fil10] I. Filippis. Personal Communication, October 2010.

[FS05] S. Foissac and T. Schiex. Integrating Alternative Splicing Detection into Gene Prediction. *BMC Bioinformatics*, **6**(1):25, 2005.

[FTD+12] M.D. Filippo, A. Traini, N. D'Agostino, L. Frusciante, and M.L. Chiusano. Euchromatic and Heterochromatic Compositional Properties Emerging from the Analysis of *Solanum lycopersicum* BAC Sequences. *Gene*, **499**(1):176–181, 2012.

[FWF+06] M.C. Frith, L.G. Wilming, A. Forrest, H. Kawaji, S.L. Tan, C. Wahlestedt, V.B. Bajic, C. Kai, J. Kawai, P. Carninci, Y. Hayashizaki, T.L. Bailey, and L. Huminiecki. Pseudo-Messenger RNA: Phantoms of the Transcriptome. *PLoS Genet.*, **2**:e23, 2006.

[GAA+00] R. Guigó, P. Agarwal, J.F. Abril, M. Burset, and J.W. Fickett. An Assessment of Gene Prediction Accuracy in Large DNA Sequences. *Genome Res.*, **10**(10):1631–1642, 2000.

[GB06] S.S. Gross and M.R. Brent. Using Multiple Alignments to Improve Gene Prediction. *J. Comput. Biol.*, **13**(2):379–393, 2006.

[GBGS+11] L. Grenville-Briggs, C.M.M. Gachon, M. Strittmatter, L. Sterck, F.C. Küpper, and P. van West. A Molecular Insight into Algal-Oomycete Warfare: cDNA Analysis of *Ectocarpus siliculosus* Infected with the Basal Oomycete *Eurychasma dicksonii*. *PLoS ONE*, **6**(9):e24500, September 2011.

[GBR+07] M.B. Gerstein, C. Bruce, J.S. Rozowsky, D. Zheng, J. Du, J.O. Korbel, O. Emanuelsson, Z.D. Zhang, S. Weissman, and M. Snyder. What is a Gene, Post-ENCODE? History and Updated Definition. *Genome Res.*, **17**(6):669–681, 2007.

[GBSK05] G. Gremme, V. Brendel, M.E. Sparks, and S. Kurtz. Engineering a Software Tool for Gene Structure Prediction in Higher Organisms. *Information and Software Technology*, **47**(15):965–978, 2005.

[GFA+06] R. Guigo, P. Flicek, J. Abril, A. Reymond, J. Lagarde, F. Denoeud, S. Antonarakis, M. Ashburner, V. Bajic, E. Birney, R. Castelo, E. Eyras, C. Ucla, T. Gingeras,

J. Harrow, T. Hubbard, S. Lewis, and M. Reese. EGASP: the human ENCODE Genome Annotation Assessment Project. *Genome Biology*, **7**(Suppl 1):S2, 2006.

[GFF] Generic Feature Format Version 2. `http://gmod.org/wiki/GFF2`. Retrieved on 2011-04-07.

[GHJV94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.

[GK12] G. Gonnella and S. Kurtz. Readjoiner: A Fast and Memory Efficient String Graph-Based Sequence Assembler. *BMC Bioinformatics*, **13**(1):82, 2012.

[GMP96] M.S. Gelfand, A.A. Mironov, and P.A. Pevzner. Gene Recognition via Spliced Alignment. *Proc. Nat. Acad. Sci. U.S.A.*, **93**:9061–9066, 1996.

[GNK+07] S. Gräf, F.G.G. Nielsen, S. Kurtz, M.A. Huynen, E. Birney, H. Stunnenberg, and P. Flicek. Optimized Design and Assessment of Whole Genome Tiling Arrays. *Bioinformatics*, 23 ISMB/ECCB 2007:i195–i204, 2007.

[GR77] R.E. Gelinas and R.J. Roberts. One Predominant 5'-Undecanucleotide in Adenovirus 2 Late Messenger RNAs. *Cell*, **11**:533–544, 1977.

[GR05] R. Guigó and M.G. Reese. EGASP: Collaboration Through Competition to Find Human Genes. *Nat. Meth.*, **2**(8):575–577, 2005.

[Gri28] F. Griffith. The Significance of Pneumococcal Types. *J. Hyg.*, **27**:113–159, 1928.

[GS06] P.E. Griffiths and K. Stotz. Genes in the Postgenomic Era. *Theor. Med. Bioeth.*, **27**:499–521, 2006.

[GSK13] G. Gremme, S. Steinbiss, and S. Kurtz. *GenomeTools*: A Comprehensive Software Library for Efficient Processing of Structured Genome Annotations. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, **99**(PrePrints):1, 2013.

[GTF] GTF2.2: A Gene Annotation Format. `http://mblab.wustl.edu/GTF22.html`. Retrieved on 2011-04-07.

[GW03] R. Guigó and T. Wiehe. Gene Prediction Accuray in Large DNA Sequences. In M.Y. Galperin and E.V. Koonin, editors, *Frontiers in Computational Genomics*, chapter 1. Caister Academic Press, Wymondham, UK, 2003.

[HC52] A.D. Hershey and M. Chase. Independent Functions of Viral Protein and Nucleic Acid in Growth of Bacteriophage. *J. Gen. Physiol.*, **36**(1):39–56, 1952.

[HDF+06] J. Harrow, F. Denoeud, A. Frankish, A. Reymond, C.-K. Chen, J. Chrast, J. Lagarde, J. Gilbert, R. Storey, D. Swarbreck, C. Rossier, C. Ucla, T. Hubbard, S. Antonarakis, and R. Guigo. GENCODE: Producing a Reference Annotation for ENCODE. *Genome Biology*, **7**(Suppl 1):S4, 2006.

[HDM+03] 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, and O. White. Improving the *Arabidopsis* Genome Annotation Using Maximal Transcript Alignment Assemblies. *Nucleic Acids Res.*, **31**(19):5654–5666, 2003.

[HH92] S. Henikoff and J.G. Henikoff. Amino Acid Substitution Matrices from Protein Blocks. *Proceedings of the National Academy of Sciences*, **89**(22):10915–10919, 1992.

[HKFF86] S. Henikoff, M.A. Keene, K. Fechtel, and J.W. Fristrom. Gene Within a Gene: Nested *Drosophila* Genes Encode Unrelated Proteins on Opposite DNA Strands. *Cell*, **44**:32–42, 1986.

[HM76] P. Henderson and J.H. Morris, Jr. A Lazy Evaluator. In *Proceedings of the 3rd ACM SIGACT-SIGPLAN symposium on Principles on programming languages*, POPL '76, pages 95–103, New York, NY, USA, 1976. ACM.

[HPB+11] T.T. Hu, P. Pattyn, E.G. Bakker, J. Cao, J.-F. Cheng, R.M. Clark, N. Fahlgren, J.A. Fawcett, J. Grimwood, H. Gundlach, G. Haberer, J.D. Hollister, S. Ossowski, R.P. Ottilar, A.A. Salamov, K. Schneeberger, M. Spannagl, X. Wang, L. Yang, M.E. Nasrallah, J. Bergelson, J.C. Carrington, B.S. Gaut, J. Schmutz, K.F.X. Mayer, Y. Van de Peer, I.V. Grigoriev, M. Nordborg, D. Weigel, and Y.-L. Guo. The *Arabidopsis lyrata* Genome Sequence and the Basis of Rapid Genome Size Change. *Nat. Genet.*, **43**:476–481, 2011.

[HVT+02] B.J. Haas, N. Volfovsky, C.D. Town, M. Troukhan, N. Alexandrov, K.A. Feldmann, R.B. Flavell, O. White, and S.L. Salzberg. Full-length messenger RNA sequences greatly improve genome annotation. *Genome Biol.*, **3**(6):RESEARCH0029, 2002.

[HZZ+05] P.M. Harrison, D. Zheng, Z. Zhang, N. Carriero, and M. Gerstein. Transcribed Processed Pseudogenes in the Human Genome: An Intermediate Form of Expressed Retrosequence Lacking Protein-Coding Ability. *Nucleic Acids Res.*, **33**:2374–2383, 2005.

[Int01] International Human Genome Sequencing Consortium. Initial Sequencing and Analysis of the Human Genome. *Nature*, **409**:860–921, 2001.

[Joh09] W. Johannsen. *Elemente der Exakten Erblichkeitslehre*. Gustav Fischer, Jena, 1909.

[KAA+05] 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, and R. Apweiler. The EMBL Nucleotide Sequence Database. *Nucleic Acids Res.*, **33 Database Issue**:29–33, 2005.

[KBZ+05] T.H. Kim, L.O. Barrera, M. Zheng, C. Qu, M.A. Singer, T.A. Richmond, Y. Wu, R.D. Green, and B. Ren. A High-Resolution Map of Active Promoters in the Human Genome. *Nature*, **436**:876–880, 2005.

[Ken02] W.J. Kent. BLAT—The BLAST-Like Alignment Tool. *Genome Res.*, **12**(4):656–664, 2002.

[KFDB01] I. Korf, P. Flicek, D. Duan, and M. R. Brent. Integrating Genomic Homology into Gene Structure Prediction. *Bioinformatics*, **17**:S140–148, 2001.

[KHRE96] D. Kulp, D. Haussler, M.G. Reese, and F.H. Eeckman. A Generalized Hidden Markov Model for the Recognition of Human Genes in DNA. In D.J. States and P. Agarwal, editors, *Proc. Conf. on Intelligent Systems for Molecular Biology*, pages 134–142. AAAI/MIT Press, 1996.

[KHV+98] J. Kleffe, K. Hermann, W. Vahrson, B. Wittig, and V. Brendel. GeneGenerator–a Flexible Algorithm for Gene Prediction and its Application to Maize Sequences. *Bioinformatics*, **14**(3):232–243, 1998.

[Kni97] R. Knippers. *Molekulare Genetik*. Thieme, 1997.

[KNSW08] S. Kurtz, A. Narechania, J.C. Stein, and D. Ware. A New Method to Compute K-mer Frequencies and its Application to Annotate Large Repetitive Plant Genomes. *BMC Genomics*, **9**:517, 2008.

[Kro97] A. Krogh. Two Methods for Improving Performance of an HMM and Their Application for Gene Finding. In *Proc. of ISMB 97*, pages 179–186, 1997.

[KSF+02] W.J. Kent, C.W. Sugnet, T.S. Furey, K.M. Roskin, T.H. Pringle, A.M. Zahler, and D. Haussler. The Human Genome Browser at UCSC. *Genome Res.*, **12**(6):996–1006, 2002.

[Kur03] S. Kurtz. *Foundations of Sequence Analysis*. Unpublished Lecture Notes, May 2003.

[Kur12] S. Kurtz. *Genominformatik*. Unpublished Lecture Notes, July 2012.

[LB98] A.V. Lukashin and M. Borodovsky. GeneMark.hmm: New Solutions for Gene Finding. *Nucleic Acids Res.*, **26**(4):1107–1115, 1998.

[LBNS+09] C. Lelandais-Brière, L. Naya, E. Sallet, F. Calenge, F. Frugier, C. Hartmann, J. Gouzy, and M. Crespi. Genome-Wide *Medicago truncatula* Small RNA Analysis Revealed Novel MicroRNAs and Isoforms Differentially Regulated in Roots and Nodules. *The Plant Cell Online*, **21**(9):2780–2796, 2009.

[LCC+12] H.Y.K. Lam, M.J. Clark, R. Chen, R. Chen, G. Natsoulis, M. O'Huallachain, F.E. Dewey, L. Habegger, E.A. Ashley, M.B. Gerstein, A.J. Butte, H.P. Ji, and M. Snyder. Performance Comparison of Whole-Genome Sequencing Platforms. *Nature Biotechnology*, **30**:78–82, 2012.

[Lew04] B. Lewin. *Genes VIII*. Prentice Hall, 2004.

[LMD+12] N.J. Loman, R.V. Misra, T.J. Dallman, C. Constantinidou, S.E. Gharbia, J. Wain, and M.J. Pallen. Performance Comparison of Benchtop High-Throughput Sequencing Platforms. *Nature Biotechnology*, **30**:434–439, 2012.

[LMRP08] Q. Liu, A.J. Mackey, D.S. Roos, and F.C.N. Pereira. Evigan: A Hidden Variable Model for Integrating Gene Evidence for Eukaryotic Gene Prediction. *Bioinformatics*, **24**(5):597–605, 2008.

[LSH+02] S.E. Lewis, S.M.J. Searle, N. Harris, M. Gibson, V. Iyer, J. Richter, C. Wiel, L. Bayraktaroglu, E. Birney, M.A. Crosby, J.S. Kaminker, B.B. Matthews, S.E. Prochnik, C.D. Smith, J.L. Tupy, G.M. Rubin, S. Misra, C.J. Mungall, and M.E. Clamp. Apollo: A Sequence Annotation Editor. *Genome Biology*, **3**(12):research0082.1–0082.14, 2002. This article is part of a series of refereed research articles from Berkeley Drosophila Genome Project, FlyBase and colleagues, describing Release 3 of the Drosophila genome, which are freely available at http://genomebiology.com/drosophila/.

[LSH+10] J.-Y. Lin, R.M. Stupar, C. Hans, D.L. Hyten, and S.A. Jackson. Structural and Functional Divergence of a 1-Mb Duplicated Region in the Soybean (*Glycine max*) Genome and Comparison to an Orthologous Region from *Phaseolus vulgaris*. *The Plant Cell Online*, **22**(8):2545–2561, 2010.

[Mar08a] E.R. Mardis. Next-Generation DNA Sequencing Methods. *Annu. Rev. Genomics Hum. Genet.*, **9**:387–402, 2008.

[Mar08b] E.R. Mardis. The Impact of Next-Generation Sequencing Technology on Genetics. *Trends in Genetics*, **24**:133–141, 2008.

[McC29] B. McClintock. A Cytological and Genetical Study of Triploid Maize. *Genetics*, **14**:180–222, 1929.

[McC48] B. McClintock. Mutable Loci in Maize. *Carnegie Inst. of Wash. Year Book*, **47**:155–169, 1948.

[MCC+08] J. Mondego, M. Carazzolle, G. Costa, E. Formighieri, L. Parizzi, J. Rincones, C. Cotomacci, D. Carraro, A. Cunha, H. Carrer, R. Vidal, R. Estrela, O. García, D. Thomazella, B. de Oliveira, A. Pires, M. Rio, M. Araújo, M. de Moraes, L. Castro, K. Gramacho, M. Goncalves, J. Neto, A. Neto, L. Barbosa, M. Guiltinan, B. Bailey, L. Meinhardt, J. Cascardo, and G. Pereira. A Genome Survey of *Moniliophthora perniciosa* Gives New Insights into Witches' Broom Disease of Cacao. *BMC Genomics*, **9**(1):548, 2008.

[MCM+02] S Misra, M. Crosby, C. Mungall, B. Matthews, K. Campbell, P. Hradecky, Y. Huang, J. Kaminker, G. Millburn, S. Prochnik, C. Smith, J. Tupy, E. Whitfield, L. Bayraktaroglu, B. Berman, B. Bettencourt, S. Celniker, A. de Grey, R. Drysdale, N. Harris, J. Richter, S. Russo, A. Schroeder, S. Shu, M. Stapleton, C. Yamada, M. Ashburner, W. Gelbart, G. Rubin, and S. Lewis. Annotation of the *Drosophila melanogaster* Euchromatic Genome: A Systematic Review. *Genome Biology*, **3**(12):research0083.1–0083.22, 2002.

[MEA+05] M. Margulies, M. Egholm, W.E. Altman, S. Attiya, J.S. Bader, L.A. Bemben, J. Berka, M.S. Braverman, Y.-J. Chen, Z. Chen, S.B. Dewell, L. Du, J.M. Fierro, X.V. Gomes, B.C. Godwin, W. He, S. Helgesen, C.H. Ho, G.P. Irzyk, S.C. Jando, M.L.I. Alenquer, T.P. Jarvie, K.B. Jirage, J.-B. Kim, J.R. Knight, J.R. Lanza, J.H. Leamon, S.M. Lefkowitz, M. Lei, J. Li, K.L. Lohman, H. Lu, V.B. Makhijani, K.E. McDade, M.P. McKenna, E.W. Myers, E. Nickerson, J.R. Nobile, R. Plant, B.P. Puc, M.T. Ronan, G.T. Roth, G.J. Sarkis, J.F. Simons, J.W. Simpson, M. Srinivasan, K.R. Tartaro, A. Tomasz, K.A. Vogt, G.A. Volkmer, S.H. Wang, Y. Wang, M.P. Weiner, P. Yu, R.F. Begley, and J.M. Rothberg. Genome Sequencing in Microfabricated High-Density Picolitre Reactors. *Nature*, **437**:376–380, 2005.

[MEC07] C.J. Mungall, D.B. Emmert, and The FlyBase Consortium. A Chado Case Study: An Ontology-Based Modular Schema for Representing Genome-Associated Biological Information. *Bioinformatics*, **23**(13):i337–i346, 2007.

[Men66] G. Mendel. *Versuche über Pflanzenhybriden*, pages 3–47. Number 4 in Abhandlungen. Verhandlungen des Naturforschenden Vereines in Brünn, 1866.

[Met10] M.L. Metzker. Sequencing Technologies — The Next Generation. *Nat. Rev. Genet.*, **11**:31–46, 2010.

[MJ10] P. Montalent and J. Joets. EuGène-Maize: A Web Site for Maize Gene Prediction. *Bioinformatics*, **26**(9):1254–1255, 2010.

[MLH+09] A. Martin, D. Lang, J. Heckmann, A.D. Zimmer, M. Vervliet-Scheebaum, and R. Reski. A Uniquely High Number of ftsZ Genes in the Moss *Physcomitrella patens*. *Plant Biology*, **11**(5):744–750, 2009.

[MSMB15] T.H. Morgan, A.H. Sturtevant, H.J. Muller, and C.B. Bridges. *The Mechanism of Mendelian Heredity*. Henry Holt & Company, New York, 1915.

[MSSK11] M. Mader, R. Simon, S. Steinbiss, and S. Kurtz. FISH Oracle: A Web Server for Flexible Visualization of DNA Copy Number Data in a Genomic Context. *Journal of Clinical Bioinformatics*, **1**:20, 2011.

[MSSR02] C. Mathe, M.-F. Sagot, T. Schiex, and P. Rouze. Current Methods of Gene Prediction, their Strengths and Weaknesses. *Nucleic Acids Res.*, **30**(19):4103–4117, 2002.

[Mul27] H.J. Muller. Artificial Transmutation of the Gene. *Science*, **66**:84–87, 1927.

[NC01] D.L. Nelson and M.M. Cox. *Lehninger Biochemie*. Springer, 2001.

[NLB+65] M. Nirenberg, P. Leder, M. Bernfield, R. Brimacombe, J. Trupin, F. Rottman, and C. O'Neal. RNA Codewords and Protein Synthesis, VII. On the General Nature of the RNA Code. *Proc. Nat. Acad. Sci. U.S.A.*, **53**(5):1161–1168, 1965.

[NS07] N. Nethercote and J. Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 89–100, New York, NY, USA, 2007. ACM.

[PAA+03] G. Parra, P. Agarwal, J.F. Abril, T. Wiehe, J.W. Fickett, and R. Guigó. Comparative Gene Prediction in Human and Mouse. *Genome Res.*, **13**(1):108–117, 2003.

[PAC01] L. Pachter, M. Alexandersson, and S. Cawley. Applications of Generalized Pair Hidden Markov Models to Alignment and Gene Finding Problems. In *Proc. of the Fifth International Conference on Computational Molecular Biology, (RECOMB 01)*, pages 241–248, 2001.

[PBG00] G. Parra, E. Blanco, and R. Guigó. `GeneID` in *Drosophila*. *Genome Res.*, **10**(4):511–515, 2000.

[Pea00] W.R. Pearson. Flexible sequence similarity searching with the FASTA3 program package. *Methods Mol. Biol.*, **132**:185–219, 2000.

[Pea06] H. Pearson. Genetics: What is a Gene? *Nature*, **441**:398–401, 2006.

[Pev00] P.A. Pevzner. *Computational Molecular Biology: An Algorithmic Approach*. The MIT Press, Cambridge, MA, 2000.

[PFJ+11] H. Pausch, K. Flisikowski, S. Jung, R. Emmerling, C. Edel, K.-U. Götz, and R. Fries. Genome-Wide Association Study Identifies Two Major Loci Affecting Calving Ease and Growth-Related Traits in Cattle. *Genetics*, **187**(1):289–297, January 2011.

[PL88] W.R. Pearson and D.J. Lipman. Improved tools for biological sequence comparison. *Proc. Nat. Acad. Sci. U.S.A.*, **85**:2444–2448, 1988.

[PLJ+12] I. Pagani, K. Liolios, J. Jansson, I-M.A. Chen, T. Smirnova, B. Nosrat, V.M. Markowitz, and N.C. Kyrpides. The Genomes OnLine Database (GOLD) v.4: Status of Genomic and Metagenomic Projects and their Associated Metadata. *Nucleic Acids Res.*, **40**(D1):D571–D579, 2012.

[PRD+99] N. Pavy, S. Rombauts, P. Dehais, C. Mathe, D. V. V. Ramana, P. Leroy, and P. Rouze. Evaluation of gene prediction software using a genomic data set: application to Arabidopsis thaliana sequences. *Bioinformatics*, **15**(11):887–899, 1999.

[PRD+06] G. Parra, A. Reymond, N. Dabbouseh, E.T. Dermitzakis, R. Castelo, T.M. Thomson, S.E. Antonarakis, and R. Guigó. Tandem Chimerism as a Means to Increase Protein Complexity in the Human Genome. *Genome Res.*, **16**:37–44, 2006.

[RBB+03] S.Y. Rhee, W. Beavis, T.Z. Berardini, G. Chen, D. Dixon, A. Doyle, M. Garcia-Hernandez, E. Huala, G. Lander, M. Montoya, N. Miller, L.A. Mueller, S. Mundodi, L. Reiser, J. Tacklind, D.C. Weems, Y. Wu, I. Xu, D. Yoo, J. Yoon, and P. Zhang. The *Arabidopsis* Information Resource (TAIR): A Model Organism Database Providing a Centralized, Curated Gateway to *Arabidopsis* Biology, Research Materials and Community. *Nucleic Acids Res.*, **31**(1):224–228, 2003.

[RHH+00] M.G. Reese, G. Hartzell, N.L. Harris, U. Ohler, J.F. Abril, and Lewis S.E. Genome Annotation Assessment in Drosophila Melanogaster. *Genome Res.*, **10**(4):483–501, 2000.

[RMO01] S. Rogic, A.K. Mackworth, and F.B.F. Ouellette. Evaluation of Gene-Finding Programs on Mammalian Sequences. *Genome Res.*, **11**:817–832, 2001.

[RTL+10] S. Richardt, G. Timmerhaus, D. Lang, E. Qudeimat, L. Corrêa, R. Reski, S. Rensing, and W. Frank. Microarray Analysis of the Moss *Physcomitrella patens* Reveals Evolutionarily Conserved Transcriptional Regulation of Salt Stress and Abscisic Acid Signalling. *Plant Molecular Biology*, **72**:27–45, 2010.

[SAJ+10] A. Schallau, F. Arzenton, A.J. Johnston, U. Hähnel, D. Koszegi, F.R. Blattner, L. Altschmied, G. Haberer, G. Barcaccia, and H. Bäumlein. Identification and Genetic Analysis of the *APOSPORY* locus in *Hypericum perforatum* L. *The Plant Journal*, **62**(5):773–784, 2010.

[Sal97] S.L. Salzberg. A Method for Identifying Splice Sites and Translational Start Sites in Eukaryotic mRNA. *Comput. Appl. Biosci.*, **13**(4):365–376, 1997.

[SAS+09] T.Z. Sen, C.M. Andorf, M.L. Schaeffer, L.C. Harper, M.E. Sparks, J. Duvick, V.P. Brendel, E. Cannon, D.A. Campbell, and C.J. Lawrence. MaizeGDB Becomes Sequence-Centric. *Database*, **2009**, 2009.

[SB05] M.E. Sparks and V. Brendel. Incorporation of Splice Site Probability Models for Non-Canonical Introns Improves Gene Structure Prediction in Plants. *Bioinformatics*, **21**(Suppl 3):iii20–iii30, 2005.

[SB08] M. Sparks and V. Brendel. MetWAMer: Eukaryotic Translation Initiation Site Prediction. *BMC Bioinformatics*, **9**(1):381, 2008.

[SBB+02] J.E. Stajich, D. Block, K. Boulez, S.E. Brenner, S.A. Chervitz, C. Dagdigian, G. Fuellen, J.G.R. Gilbert, I. Korf, H. Lapp, H. Lehväslaiho, C. Matsalla, C.J. Mungall, B.I. Osborne, M.R. Pocock, P. Schattner, M. Senger, L.D. Stein, E. Stupka, M.D. Wilkinson, and E. Birney. The Bioperl Toolkit: Perl Modules for the Life Sciences. *Genome Res.*, **12**(10):1611–1618, 2002.

[SDBH08] M. Stanke, M. Diekhans, R. Baertsch, and D. Haussler. Using Native and Syntenically Mapped cDNA Alignments to Improve *de novo* Gene Finding. *Bioinformatics*, **24**(5):637–644, 2008.

[SDFH98] S. Salzberg, A. Delcher, K. Fasman, and J. Henderson. A Decision Tree System for Finding Genes in DNA. *J. Comp. Biol.*, **5**(4):667–680, 1998.

[SGS+09] S. Steinbiss, G. Gremme, C. Schärfer, M. Mader, and S. Kurtz. *AnnotationSketch*: A Genome Annotation Drawing Library. *Bioinformatics*, **25**(4):533–534, 2009.

[SHK10] D.J. Schmitz-Hübsch and S. Kurtz. MetaGenomeThreader: A Software Tool for Predicting Genes in DNA-Sequences of Metagenome Projects. In Streit, W. and Daniel, R., editor, *Metagenomics. Methods and Protocols*, volume **668** of *Methods in Molecular Biology*, pages 325–338. Springer, Berlin, 2010.

[SJ08] J. Shendure and H. Ji. Next-Generation DNA Sequencing. *Nature Biotechnology*, **26**:1135–1145, 2008.

[SK03] T. Shibuya and I. Kurochkin. Match chaining algorithms for cdna mapping. In *WABI*, pages 462–475, 2003.

[SK12] S. Steinbiss and S. Kurtz. A New Efficient Data Structure for Storage and Retrieval of Multiple Biosequences. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, **9**:345–357, 2012.

[SLTF05] C.G. Spilianakis, M.D. Lalioti, G.R. Town, T. Lee, and R.A. Flavell. Interchromosomal Associations Between Alternatively Expressed Loci. *Nature*, **435**:637–645, 2005.

[SMS+02] L.D. Stein, C. Mungall, S. Shu, M. Caudy, M. Mangone, A. Day, E. Nickerson, J.E. Stajich, T.W. Harris, A. Arva, and Lewis S. The Generic Genome Browser: a Building Block for a Model Organism System Database. *Genome Res.*, **12**(10):1599–1610, 2002.

[SNC77] F. Sanger, S. Nicklen, and A.R. Coulson. DNA Sequencing with Chain-Terminating Inhibitors. *Proc. Nat. Acad. Sci. U.S.A.*, **74**(12):5463–5467, 1977.

[SOJ+65] D. Söll, E. Ohtsuka, D.S. Jones, R. Lohrmann, H. Hayatsu, S. Nishimura, and H.G. Khorana. Studies on Polynucleotides, XLIX. Stimulation of the Binding of Aminoacyl-sRNA's to Ribosomes by Ribotrinucleotides and a Survey of Codon Assignments for 20 Amino Acids. *Proc. Nat. Acad. Sci. U.S.A.*, **54**(5):1378–1385, 1965.

[SS00] A.A. Salamov and V.V. Solovyev. Ab initio Gene Finding in *Drosophila* Genomic DNA. *Genome Res.*, **10**(4):516–522, 2000.

[Ste11] L. Stein. Generic Feature Format Version 3. `http://www.sequenceontology.org/gff3.shtml`, 2011. Retrieved on 2011-04-07.

[STG+09] B. Steuernagel, S. Taudien, H. Gundlach, M. Seidel, R. Ariyadasa, D. Schulte, A. Petzold, M. Felder, A. Graner, U. Scholz, K.F.X. Mayer, M. Platzer, and N. Stein. *De novo* 454 Sequencing of Barcoded BAC Pools for Comprehensive Gene Survey and Genome Analysis in the Complex Genome of Barley. *BMC Genomics*, **10**(1):547, 2009.

[Sto00] G.D. Stormo. Gene-Finding Approaches for Eukaryotes. *Genome Res.*, **10**(4):394–397, 2000.

[Stu13] A.H. Sturtevant. The Linear Arrangement of Six Sex-Linked Factors in *Drosophila*, as Shown by Their Mode of Association. *J. Exp. Zool.*, **14**:43–59, 1913.

[SW03] M. Stanke and S. Waack. Gene prediction with a hidden Markov model and a new intron submodel. *Bioinformatics*, **19**(90002):215ii–225, 2003.

[SWF+09] P.S. Schnable, D. Ware, R.S. Fulton, J.C. Stein, F. Wei, S. Pasternak, C. Liang, J. Zhang, L. Fulton, T.A. Graves, P. Minx, A.D. Reily, L. Courtney, S.S. Kruchowski, C. Tomlinson, C. Strong, K. Delehaunty, C. Fronick, B. Courtney, S.M. Rock, E. Belter, F. Du, K. Kim, R.M. Abbott, M. Cotton, A. Levy, P. Marchetto, K. Ochoa, S.M. Jackson, B. Gillam, W. Chen, L. Yan, J. Higginbotham, M. Cardenas, J. Waligorski, E. Applebaum, L. Phelps, J. Falcone, K. Kanchi, T. Thane, A. Scimone, N. Thane, J. Henke, T. Wang, J. Ruppert, N. Shah, K. Rotter, J. Hodges, E. Ingenthron, M. Cordes, S. Kohlberg, J. Sgro, B. Delgado, K. Mead, A. Chinwalla, S. Leonard, K. Crouse, K. Collura, D. Kudrna, J. Currie, R. He, A. Angelova, S. Rajasekar, T. Mueller, R. Lomeli, G. Scara, A. Ko, K. Delaney, M. Wissotski, G. Lopez, D. Campos, M. Braidotti, E. Ashley, W. Golser, H. Kim, S. Lee, J. Lin, Z. Dujmic, W. Kim, J. Talag, A. Zuccolo, C. Fan, A. Sebastian, M. Kramer, L. Spiegel, L. Nascimento, T. Zutavern, B. Miller, C. Ambroise, S. Muller, W. Spooner, A. Narechania, L. Ren, S. Wei, S. Kumari, B. Faga, M.J.

Levy, L. McMahan, P. Van Buren, M.W. Vaughn, K. Ying, C.-T. Yeh, S.J. Emrich, Y. Jia, A. Kalyanaraman, A.-P. Hsia, W.B. Barbazuk, R.S. Baucom, T.P. Brutnell, N.C. Carpita, C. Chaparro, J.-M. Chia, J.-M. Deragon, J.C. Estill, Y. Fu, J.A. Jeddeloh, Y. Han, H. Lee, P. Li, D.R. Lisch, S. Liu, Z. Liu, D.H. Nagel, M.C. McCann, P. SanMiguel, A.M. Myers, D. Nettleton, J. Nguyen, B.W. Penning, L. Ponnala, K.L. Schneider, D.C. Schwartz, A. Sharma, C. Soderlund, N.M. Springer, Q. Sun, H. Wang, M. Waterman, R. Westerman, T.K. Wolfgruber, L. Yang, Y. Yu, L. Zhang, S. Zhou, Q. Zhu, J.L. Bennetzen, R.K. Dawe, J. Jiang, N. Jiang, G.G. Presting, S.R. Wessler, S. Aluru, R.A. Martienssen, S.W. Clifton, W.R. McCombie, R.A. Wing, and R.K. Wilson. The B73 Maize Genome: Complexity, Diversity, and Dynamics. *Science*, **326**(5956):1112–1115, 2009.

[SWGK09] S. Steinbiss, U. Willhoeft, G Gremme, and S. Kurtz. Fine-Grained Annotation and Classification of *de novo* Predicted LTR Retrotransposons. *Nucleic Acids Res.*, **37**(21):7002–7013, 2009.

[SZB+10] R. Sinha, A. Zimmer, K. Bolte, D. Lang, R. Reski, M. Platzer, S. Rensing, and R. Backofen. Identification and Characterization of NAGNAG Alternative Splicing in the Moss *Physcomitrella patens*. *BMC Plant Biology*, **10**(1):76, 2010.

[SZZ+09] G. Schweikert, A. Zien, G. Zeller, J. Behr, C. Dieterich, C.S. Ong, P. Philips, F. De Bona, L. Hartmann, A. Bohlen, N. Krüger, S. Sonnenburg, and G. Rätsch. mGene: Accurate SVM-Based Gene Finding With an Application to Nematode Genomes. *Genome Res.*, **19**(11):2133–2143, 2009.

[TRG+03] L. Taher, O. Rinner, S. Garg, A. Sczyrba, M. Brudno, S. Batzoglou, and B. Morgenstern. AGenDa: Homology-Based Gene Prediction. *Bioinformatics*, **19**(12):1575–1577, 2003.

[UB00] J. Usuka and V. Brendel. Gene Structure Prediction by Spliced Alignment of Genomic DNA with Protein Sequences: Increased Accuracy by Differential Splice Site Scoring. *J. Mol. Biol.*, **297**:1075–1085, 2000.

[UZB00] J. Usuka, W. Zhu, and V. Brendel. Optimal Spliced Alignment of Homologous cDNA to a Genomic DNA Template. *Bioinformatics*, **16**(3):203–211, 2000.

[VAM+01] J.C. Venter, M.D. Adams, E.W. Myers, P.W. Li, R.J. Mural, G.G. Sutton, H.O. Smith, M. Yandell, C.A. Evans, R.A. Holt, and et al. The Sequence of the Human Genome. *Science*, **291**:1304–1351, 2001.

[Wad71] C.P. Wadsworth. *Semantics and Pragmatics of the Lambda Calculus*. PhD thesis, Oxford University, 1971.

[WB06] WormBase web site, `http://wormbase.org`, release WS160, date July 31, 2006.

[WBB+05] 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, and E. Yaschenko. Database resources of the National Center for Biotechnology Information. *Nucleic Acids Res.*, **33 Database Issue**:39–45, 2005.

[WBL+02] H.M. Wain, E.A. Bruford, R.C. Lovering, M.J. Lush, M.W. Wright, and S. Povey. Guidelines for Human Gene Nomenclature. *Genomics*, **79**(4):464–470, 2002.

[WC53] J.D. Watson and F.H.C. Crick. A Structure for Deoxyribose Nucleic Acid. *Nature*, **171**:737–738, 1953.

[WCO01] S.J. Wheelan, D.M. Church, and J.M. Ostell. Spidey: A Tool for mRNA-to-Genomic Alignments. *Genome Res.*, **11**(11):1952–1957, 2001.

[WFV+09] R. Wang, S. Farrona, C. Vincent, A. Joecker, H. Schoof, F. Turck, C. Alonso-Blanco, G. Coupland, and M.C. Albani. PEP1 Regulates Perennial Flowering in *Arabis alpina*. *Nature*, **459**:423–427, 2009.

[WGS09] Z. Wang, M. Gerstein, and M. Snyder. RNA-Seq: A Revolutionary Tool for Transcriptomics. *Nat. Rev. Genet.*, **10**:57–63, 2009.

[WL01] T.G. Wolfsberg and D. Landsman. Expressed Sequence Tags (ESTs). In A.D Baxevanis and B.F.F. Ouellette, editors, *Bioinformatics: A Practical Guide to the Analysis of Genes and Proteins*, chapter 12. John Wiley & Sons, 2001.

[WP03] C.D. Worth and K. Packard. Cairo: Cross-device Rendering for Vector Graphics. *Proceedings of the 2003 Linux Symposium*, 2003.

[WW05] T.D. Wu and C.K. Watanabe. GMAP: A Genomic Mapping and Alignment Program for mRNA and EST Sequences. *Bioinformatics*, **21**(9):1859–1875, 2005.

[YSY+03] Y. Yano, R. Saito, N. Yoshida, A. Yoshiki, A. Wynshaw-Boris, M. Tomita, and S. Hirotsune. A New Role for Expressed Pseudogenes as ncRNA: Regulation of mRNA Stability of its Homologous Coding Gene. *J. Mol. Med.*, **82**:414–422, 2003.

[ZB03] W. Zhu and V. Brendel. Identification, Characterization and Molecular Phylogeny of U12-dependent Introns in the *Arabidopsis thaliana* Genome. *Nucleic Acids Res.*, **31**(15):4561–4572, 2003.

[ZFB+07] D. Zheng, A. Frankish, R. Baertsch, P. Kapranov, A. Reymond, S.W. Choo, Y. Lu, F. Denoeud, S.E. Antonarakis, M. Snyder, Y. Ruan, C.-L. Wei, T.R. Gingeras, R. Guigó, J. Harrow, and M.B. Gerstein. Pseudogenes in the ENCODE Regions:

Consensus Annotation, Analysis of Transcription, and Rvolution. *Genome Res.*, **17**:839–851, 2007.

[Zha97] M.Q. Zhang. Identification of Protein Coding Regions in the Human Genome by Quadratic Discriminant Analysis. *Proc. Nat. Acad. Sci. U.S.A.*, **94**:565–568, 1997.

[Zha02] M.Q. Zhang. Computational Prediction of Eukaryotic Protein-Coding Genes. *Nature Reviews Genetics*, **3**(9):698–709, 2002.

[ZPF+07] Z.D. Zhang, A. Paccanaro, Y. Fu, S. Weissman, Z. Weng, J. Chang, M. Snyder, and M.B. Gerstein. Statistical Analysis of the Genomic Distribution and Correlation of Regulatory Elements in the ENCODE Regions. *Genome Res.*, **17**:787–797, 2007.

[ZSWM00] Z. Zhang, S. Schwartz, L. Wagner, and W. Miller. A Greedy Algorithm for Aligning DNA Sequences. *J. Comp. Biol.*, **7**(1/2):203–214, 2000.

[ZZH+05] D. Zheng, Z. Zhang, P.M. Harrison, J. Karro, N. Carriero, and M. Gerstein. Integrated Pseudogene Annotation for Human Chromosome 22: Evidence for Transcription. *J. Mol. Biol.*, **349**:27–45, 2005.

Index