Efficient Probabilistic Logic Programming for Biological Sequence Analysis

Christian Theil Have

JANUARY 2013 ROSKILDE UNIVERSITY COMPUTER SCIENCE RESEARCH REPORT #137

Computer Science
Department of Communication,
Business and Information Technologies
Roskilde University
P. O. Box 260
DK–4000 Roskilde
Denmark

Telephone:	+45 4674 3839
Telefax:	+45 4674 3072
Internet:	http://www.ruc.dk/dat_en/
E-mail:	datalogi@ruc.dk

ISSN 0109–9779

Efficient Probabilistic Logic Programming for Biological Sequence Analysis

by

Christian Theil Have

A dissertation presented to the faculties
of Roskilde University in partial fulfillment of
the requirement for the PhD degree

Supervisor: Henning Christiansen

Department of Communication, Business and Information Technologies
Roskilde University, Denmark

September 2012

Contents

Abstract

This thesis is concerned with the application of probabilistic logic programming to biological sequence analysis. Probabilistic logic programming is a declarative programming paradigm which enables convenient and concise expression of a wide range of statistical models including common models from biological sequence analysis. A main advantage of this approach is that logic and control is separated – this enables that general inference algorithms can be reused for any model which is expressed using probabilistic logic programming.

The dissertation demonstrates the usefulness of probabilistic logic programming for biological sequence analysis through applications in bacterial gene finding and through programming abstractions that enable convenient expression of constraints from the domain of biological sequence analysis. To deal with central limitations with regard to the efficiency of probabilistic logic programming, a number of optimizations of the approach are introduced. With these optimizations, probabilistic logic programming is now efficient enough to deal with a wide range of problems from biological sequence analysis.

Resumé

Denne afhandling omhandler anvendelsen af probabilistisk logikprogrammering til biologisk sekvensanalyse. Probabilistisk logikprogrammering er et deklarativt programmeringsparadigme der på en nem og kortfattet måde gør det muligt at udtrykke en bred vifte af statistiske modeller. Dette inkluderer en række af de modeller som normalt bruges til biologisk sekvensanalyse.

En fordel ved denne tilgang er at der er en opdeling mellem logik (model) og kontrol (inferens). Dette muliggør at generelle inferensprocedurer kan genbruges for alle typer af modeller der udtrykkes ved hjælp af probabilistisk logikprogrammering. Dette gør det markant lettere at udvikle nye typer af modeller.

I afhandlingen demonstreres nytten af probabilistisk logikprogrammering til biologisk sekvensanalyse i kraft af anvendelser indenfor bakteriel genfinding og gennem programmeringsabstraktioner der gør det nemmere at udtrykke modelbegrænsninger i form af viden og antagelser fra biologisk sekvensanalyse. For at håndtere problemer i forbindelse med beregningseffektiviteten af probabilistisk logikprogrammering præsenterer afhandlingen en række teknikker og optimeringer.

Med disse optimeringer er probabilistisk logikprogrammering nu effektivt nok til at håndtere en lang række problemer indenfor biologisk sekvensanalyse.

Acknowledgements

It is an privilege to be able spend three years (and a bit more) to work on a really exciting subject. I am deeply grateful for this opportunity. I would like to thank my supervisor, Henning Christiansen for giving me this opportunity and for having confidence in that I would use it well.

This work is the product of many exciting collaborations with several interesting people. It has been remarkable to be part of the LoSt project and I would like to use the opportunity to thank the people I have collaborated with and who have inspired me. Thanks for Ole Skovgaard and Ana Capatana for exciting discussions and for sharing with me biological insights and assistance. I have coauthored papers with Henning Christian, Ole Torp Lassen, Matthieu Petit, Søren Mørk, Sine Zambach and Neng-Fa Zhou. Thanks to all of you for such fruitful and inspiring collaborations.

I would also like to thank all my colleagues at Roskilde University. It has been wonderful to be here and I have enjoyed the friendly atmosphere and inspirational environment. A special thanks to Ole, Matthieu, Mai, Jens Ulrik and Sine who have made RUC a such a fun place to be.

Also thank you to Henning, Mai, Søren, Sine and Ole for proofreading parts of this thesis.

I also would like to thank my family who have encouraged me and been very supportive. In particular, I would like to thank my lovely wife, Ida, and our wonderful child Johanna for putting up with me in the process and for being supportive.

Finally, I would like to thank the assessment committee for taking the time to read and comment on this dissertation. I am grateful for their time and their valuable feedback.

Chapter 1

Introduction

1.1 Problem statement

This thesis is in the intersection of probabilistic logic programming, biological sequence analysis and constraints. The thesis does not represent a complete synthesis of these subjects. Rather, it is a collection of applications and abstractions for biological sequence analysis built using probabilistic logic programming and also some new optimization techniques to deal with complexity and efficiency of these. Yet, these diverse efforts all contribute to shed light on the following three related research questions:

- *To what extent is it possible to use probabilistic logic programming for biological sequence analysis?*
- *How can constraints relevant to the domain of biological sequence analysis be combined with probabilistic logic programming?*
- *What are the limitations with regard to efficiency and how can these be dealt with?*

To motivate these research questions and the approach taken to answer them, the context of the project and my original research agenda will be discussed.

This project is part of the larger research project, *Logic-Statistic Modeling and Analysis of Biological Sequence Data* (abbreviated *LoSt*)[1]. With offset in the LoSt project and in conception phase of this project a research agenda formed [85]. The research agenda has served to guide the direction of the research. In the course of the the project, however, insights and collaboration opportunities have arisen, which leads the project to digress slightly from the original agenda. The project has been adapted to these circumstances, resulting in a more dispersed picture, which is nevertheless still highly relevant to the above high level research questions. The original agenda is formulated as follows,

"This project aims to investigate biologically inspired, logic-statistical models with constraints. The complexity and expressiveness of different kinds of

[1]website: http://lost.ruc.dk

models will be examined and algorithms to efficiently cope with inference in and training of such models will be explored. The models will be evaluated with regard to their applicability to biological sequence analysis.“

This agenda touches upon two very distinct problems. On one hand, it sets out to explore sophisticated models with constraints which can be expressed using probabilistic logic programming. On the other hand it sets out to build useful models for biological sequence analysis. This somehow indicates that such models are useful for biological sequence analysis, but does not state why this may be the case.

Biological sequence analysis is subfield of bioinformatics which aims to analyze biological sequences, e.g., DNA, RNA and proteins, to understand their features, functions and evolutionary relationships. Biological sequence analysis is a computational process which can take place when the sequences are represented as data in a digital format. The conversion from biological molecules to digital data is made possible by modern biotechnology. Models are central to biological sequence analysis. A biological sequence analysis model accounts for relationships between features of sequence data and is the basis for inferring biological information from the sequences. An inference algorithm utilizes the model to extrapolate such information.

Probabilistic logic programming is a general framework which provides the ability to express arbitrarily complex models, including a variety of models commonly used for biological sequence analysis. It supports the expression of complex models in a concise manner which resemble (or even simplify) their mathematical definition. It is certainly more convenient to create a probabilistic sequence model using probabilistic logic programming, than it is using modern imperative languages, e.g., C/C++ or Java. Such languages require the design of suitable data structures and algorithms for specific models. Another issue is that it is prohibitively difficult to reason about (and transform) programs in these languages. With probabilistic logic programs, logic and control are more separated [114]. Due to this property, the need for adapting inference algorithms for different kinds of models is eliminated. In the spirit of logic programming, the separation of logic and control enables that algorithms can be written in a general fashion, such that they may be used for any model expressible within the framework. For many types of models these general algorithms — at least, theoretically — can have identical time complexity to the best known specialized algorithms.

Logic programming is particular well-suited for modeling constraints and the integration of constraints from the domain of biological sequence analysis may lead to better and more realistic models. Constraints can be formulated in a variety of ways and may be either inherent in models or explicit to models. To be useful, they must somehow bridge the gap between the model and the problem domain. They can be seen as an abstraction — a high level language — which supports the translation of domain knowledge and assumptions to practical aspects of the model. More concretely, constraints should be understood as conditions which delimits a problem by restricting possible solutions and hence reducing the search space. This may, however, make the search space more difficult to explore if the constraints violate preconditions for efficient inference, e.g., common problem substructure in the case of inference based on dynamic programming. Constraint Logic Programming represents a more declarative paradigm, which express constraints over a logic program and where the control is in principle left implicit. The control can be delegated to specialized constraint solving algorithms, which can take advantage of the structure and interdepencies of the constraints.

The agenda also contains an algorithmic aspect – to explore and develop efficient means of inference. This an early indication of a looming problem; Even though probabilistic logic programming shows great promise in its ability to represent advanced models, it is apparent that models expressed using existing tools for probabilistic logic programming may not be efficient enough to deal with the complexity and overwhelming magnitude of data present in real biological sequence analysis problems. Furthermore, the interaction of constraints with probabilistic logic programming — for which efficient inference relies on certain assumptions about the problem structure – is not always trivial to address.

The usefulness of a tool for biological sequence analysis depends not only on how accurately its underlying model reflects relevant details in biological sequences, but also in its practical use for deriving new knowledge from such sequences. Efficient inference is a precondition for practical use.

1.1.1 How this thesis address the research goals

The contributions of this thesis is divided into three categories — Applications, Abstractions and Optimizations — each of which is concerned with a different aspect the overall research goal. They each correspond to one of the three research questions, respectively.

Applications demonstrate the use of probabilistic logic programming to deal with real and relevant biological sequence analysis problems, *i.e.*, to contribute new knowledge the field of biology or bioinformatics. Abstractions provide a language for incorporating constraints from the domain of biological sequence analysis. Optimizations deal with limitations of probabilistic logic programming that may hinder its use in biological sequence analysis.

The papers in the thesis generally fall into to at least one of the three categories. Occasionally they address more than one of the research questions, since these turn out to be intricately related.

As mentioned, this thesis does not aim to be a complete synthesis of probabilistic logic programming, constraints and biological sequence analysis. In the end, the measure of success will be if we can demonstrate that probabilistic logic programming is capable of expressing the sort of models and constraints that are relevant to biological sequence analysis and if inference for these models is efficient enough to apply them to real biological problems.

Applications One of the main goals of the LoSt project is to evaluate whether probabilistic logic programming is suitable for modeling problems of biological sequence analysis. This is intricately related to my research goal. To evaluate this goal we must identify problems within biological sequence analysis where probabilistic logic programming is useful. It is not feasible to provide an entire catalogue of such problems, but at least we should provide some successful cases.

Prokaryotic gene prediction has been in focus in the LoSt project and is also the biological sequence analysis task which is given most attention within this thesis. The motivation for prokaryotic gene prediction is partly due to cooperation opportunities within the project. Prokaryotic gene finding is perceived as an almost "solved task" within the bioinformatics community. Reasonable golden standard data is available and this makes it possible to define a precise success criteria based on prediction accuracy. For the same reason, gene prediction is also challenging – the low hanging fruits have already been reaped and new ideas are necessary to improve state of the art.

Within the LoSt project, it has already been demonstrated that probabilistic logic programs can be used to model the core approaches used by contemporary gene finders and provides a suitable environment for comparing these [137]. Existing approaches to gene prediction, which have influenced my work are summarized in section 2.3. Rather than duplicating existing efforts within prokaryotic gene finding, this thesis seeks to develop new ideas and constraints which can be modeled to improve upon the state of the art. In that light, the developed tools should be seen as experimental and when seen from the perspective of gene prediction accuracy, some of the explored methods are more successful than others.

Abstractions Abstractions increase the approachability of problems by lifting the way we think about and express such problems to a higher level. An abstraction contributes to problem clarification by focussing on relevant issues and by hiding irrelevant details.

A key advantage of using logic programming is its declaritivity (ignoring the control aspect), high expressivity and its firm foundation in mathematical logic. These properties make it possible to form well-founded abstractions which can be used to model certain kinds of problems in a more elegant and convenient manner. Probabilistic logic programming is itself such an abstraction which allows for convenient expression of probabilistic models and similarly constraint logic programming is an abstraction which introduces the explicit notion of constraints in logic programs.

In this thesis, the principle of abstraction is applied to bring probabilistic logic programming closer the problems we encounter within biological sequence analysis. Often, the abstraction takes the form of a declarative language, which can be used to express, e.g., constraints from the domain of biological sequence analysis. Several such languages are presented in this thesis including probabilistic extended regular expressions, a constraint language for Hidden Markov Models and a pipeline language which facilitates composition of probabilistic models through a principle we call Bayesian Annotation Networks [43].

Optimizations Probabilistic logic programming is a field that is still maturing and efficiency and run-time is inferior compared to imperative programming languages. There is a significant overhead associated with the execution of probabilistic logic programs and in effect they are significantly slower and more memory intensive than specialized programs written in low-level languages, e.g., C, C++ and Java. However, the gap is closing in, and probabilistic logic programs are now efficient enough for an interesting range of problems.

One of the main algorithmic devices used for inference in probabilistic models is dynamic programming. Dynamic programming is essentially the use of a polynomially compressed representation of an exponentially large search space, allowing for polynomial time inferences in this search space. Prolog/PRISM supports dynamic programming through a language feature called tabling, which is central to the efficiency of inferences in PRISM. Tabling is a recurrent theme in this thesis.

Key contributions of this thesis are optimizations for tabling of structured data, *e.g.*, sequences, and optimizations for tabling programs with constraints.

1.2 The LoSt project

My project has its offset in the larger research project, the LoSt project. The LoSt project aims to apply logic-statistical modeling to perform analysis of biological sequence data.

The LoSt project was already an active project when I joined half a year after finishing my master thesis. My master thesis, supervised by Professor Henning Christiansen, is on a different but related subject of a probabilistic logic grammar formalism [88]. At the point where I joined the project, I was largely ignorant of molecular biology, but had gained some useful knowledge on probabilistic logic programming. Retrospectively, though, I realize that I was a novice even on this subject.

Had it not been for the resourceful, helpful and collaborative people in the project, I do not think that I would have gained much ground. The social dynamics of the group have also been remarkable – it has been fun and rewarding all the way because of all the great people I have had a chance to collaborate with and have gotten to know.

1.2.1 Peers in the LoSt project

My supervisor, Professor Henning Christiansen, has been my mentor and also my key collaborative partner in the project. Henning Christiansen has been the visionary of the LoSt project and his vision for the LoSt project has had a major influence on the my Ph.D. project and this thesis. He is an inspiring person with a wealth of knowledge and ideas. Furthermore, he has an encouraging and down-to-earth style of leadership and he is generally a very nice guy. It has been an immensely gratifying experience being his student and collaborator.

I have also collaborated with the postdocs within the LoSt project, Ana Capatana, Matthieu Petit and Sine Zambach.

Ana worked on the LoSt project in the beginning of my stipend. I had just begun to learn basic molecular biology and found it difficult to access relevant literature on bioinformatics. With large generosity, she pedagogically contributed biological insights and helped me and others in the group to decipher the more biological parts of various papers.

I have had the pleasure of close cooperation with Matthieu Petit on several papers. In particular, the work on constrained hidden markov models (see chapter 5), which he had already started when I entered the LoSt project, e.g. [154]. He engaged me in this line of research and we had a very synergetic cooperation. Later, he was also quite engaged and cooperative when we started work on the LoSt framework – an effort to integrate the various sub-projects of the LoSt projects. This effort sparked the development of a pipeline for biological sequence analysis. It has been a rewarding to work with Matthieu and I appreciate him as friend.

I had the pleasure of working with Sine Zambach even before she joined the LoSt project as a postdoc. We co-supervised a student project, which resulted in a joint paper for the ESSLLI student session [84][2]. When she was later hired as postdoc in the LoSt project, we had a engaged and energetic cooperation to create a gene finder for pyrrolysine incorporating genes (see chapter 14). It is very motivating and inspiring to work with Sine Zambach and I enjoy our cooperation. In addition, Sine Zambach is a good friend and colleague who is fun and inspiring to be around.

Professor Ole Skovgaard is the senior biologist in the LoSt project and he has provided me with many biological insights and have encouraged a focus on the biological relevance of the work within project. Through discussions, he has patiently attended computer science dominated group meetings and tutored us in microbial biology and existing bioinformatics tools and approaches.

As it turns out, the LoSt project — including my project — has devoted more effort to applying probabilistic logic programming to biological sequence analysis, than it has to discovering new genes or biological insights. I am certain that Ole Skovgaard would have preferred a stronger focus on the latter, since this is in the core of his research interests. I think, however, that we managed to cross the chasm between computer science (probabilistic logic programming) and biology and I find this to be a significant accomplishment of the LoSt project. The mentoring efforts of Ole Skovgaard were fundamental in building the bridge from the biology side.

Two other PhD students — Ole Torp Lassen and Søren Mørk — have been involved in the LoSt project and have had distinct projects from mine. They both have had a collaborative spirit and my project is entangled with theirs through coauthored papers. The following paragraphs briefly summarize their projects and show how they resemble and differ from my project.

Compositionality in probabilistic logic modeling for biological sequence analysis The dissertation of Ole Torp Lassen is primarily concerned with compositionality of probabilistic logic programs for sequence analysis. The driving motivation is the optimization of probabilistic logic programs for sequence analysis and a reduction in the computational complexity of inference. Realistic probabilistic models for biological sequence analysis may be of a too complex nature to allow feasible inference. In his thesis, he presents approaches to deal with this problem based on compositionality and decomposition.

The thesis provides a gentle, yet concise introduction to complex (biological) sequence analysis, which from the ground up covers aspects from formal language theory, probability theory, logic programming, basic biology and which culminates in a engaging presentation of his own work on compositionality which draws upon all these aspects.

A theme which is prevalent in his thesis and is recurring in this thesis is Bayesian Annotation Networks (BANs). Bayesian Annotation Networks is an compositional approach which resembles Bayesian Networks, but which represent networks of probabilistic annotation models (in the form of probabilistic logic programs) rather than single random variables. These models may represent different aspects of sequence analysis which may be combined through other models. The network details how the models fit together by representing

[2]The paper is not included in this thesis, since it is on a different subject.

the interdependencies. Inference with BANs is accomplished iteratively and bottom-up, where each model produces a "best annotation", which is then used as input to depending models. The thesis of Ole Torp Lassen provides some additional details on BANs, which are not part of this thesis. For instance, he provides an information theoretical basis for BANs.

The other perspective in his thesis is to make inference feasible using approximation by decomposition. In this approach, a complex probabilistic sequence model — for which predictive inference is not feasible — is decomposed into complex and less complex parts. Predictive inference may be feasible for individual parts and predictions these locally optimal predictions may be combined by a coordinating model. The combined predictions constitute an approximation of the predictions of the original model. In experiments, he demonstrates how such approximations may be evaluated by sampling in the lack of authoritative test data.

Besides the work presented in his dissertation, Ole Torp Lassen has contributed to several of the articles constituting this thesis. In addition to articles, we have shared office, travel experiences, ideas and frustrations. In has been a pleasure to work Ole Torp Lassen and I am grateful for his friendship and the many rewarding experiences we have had together.

Probabilistic Logic Hidden Markov Models for Biological Sequence Analysis Søren Mørk has been working with various interesting probabilistic HMM-like models for biological sequence analysis – in particular with applications to gene finding and RNA modeling.

In his paper "Evaluating Bacterial Gene-Finding HMM Structures as Probabilistic Logic Programs"[137] he demonstrates the applicability of the probabilistic logic language PRISM to model and compare a range of known HMM-based bacterial gene finder architectures. The standard gene finder architectures are furthermore extended and improved with length modeling. The models are systematically compared using the built-in model selection mechanisms of PRISM and also using cross-validation.

In the paper draft "Development and Evaluation of RNA Models as Probabilistic Logic Programs", he provides a catalogue of RNA models implemented in PRISM. This catalogue includes models for a variety of common RNA bioinformatics tasks such as RNA folding, simultaneous RNA folding and alignment, mRNA models for simultaneous RNA folding and codon sequence prediction and RNA-RNA interaction models. Modeling approaches based on both Stochastic Context-Free Grammars and Probabilistic Push-Down Automata are explored and compared. Extensions which are usually associated only with Hidden Markov Models, are shown to be straight-forward to integrate the latter type of models. I am a coauthor of this paper, but my contribution is rather modest. It has, however, been inspiring and enlightening for me to have this cooperation. Since my contribution is very modest, the paper is not included in this thesis.

We have cooperated closely on an other paper on a probabilistic model which use the sequence of gene reading frames to improve the specificity of existing gene finders with a minimal loss of sensitivity. In this paper we introduce a type of model which we call a "delete-HMM". This model is used to filter the predictions of existing gene finders. The paper is included as chapter 13 of this thesis.

It has been very rewarding to work with Søren Mørk and he has inspired me on many occasions with his views, ideas and optimism. He has a delightful ability to think outside the box, which I value very much and I consider him a good friend.

1.2.1.1 External partners

Anders Krogh at the bioinformatics department of Copenhagen University has had the role of external advisor for the LoSt project. He has on several occasions provided insightful bioinformatic guidance of which I am very appreciative.

Manfred Jaeger, James Cussens and Nicos Angoupolus have also been external project partners and have participated in LoSt workshops and provided very valuable inputs and ideas.

PRISM developers The lost project has had a very close cooperation with the developers of the PRISM language and system, which is absolutely central to my PhD project and to the LoSt project more generally. The main PRISM developers, Professor Taisuke Sato and Professor Yoshitaka Kameya have been very helpful and participated in workshops in our project with great enthusiasm. They have personally helped me in analyzing and resolving performance issues of my PRISM models. Furthermore they have been very forthcoming in the discussion of ideas and feedback on our work. I am very grateful for this assistance.

I had planned a longer research visit to Professor Satos laboratory in Tokyo. The visit was unfortunately cancelled due to the earth quake in the spring of 2011. I am, however, grateful for being offered the opportunity and I am sure it would have been a great personal and learning experience for me.

I have also had a very nice cooperation with Professor Neng-Fa Zhou, who is also coauthor of PRISM and the author of the B-Prolog system which underlies PRISM. He has encouraged my work on tabling and helped me through many fruitful discussions, from which I learned a lot. Ultimately, we cooperated on a paper on an improvement of tabling in B-Prolog (chapter 12) which I am very proud to be the coauthor of.

CLC-Bio I have had the pleasure of visiting LoSt project partner company CLC-Bio, with the purpose of integrating our pipeline with their development workbench. After a three day session, we had rudimentary integration which allowed running PRISM based models from our framework from within their software and visualizing the results inside their genomic workbench software. Subsequently, this integration has been revised slightly to accommodate changes in our framework. It has been quite useful to have this integration and has helped me to debug and test models in ways would not have been possible otherwise.

1.3 Dissertation overview

This thesis is organized as follows: Chapter 2 provides some background on logic programming (section 2.1) (including probabilistic logic programming and constraint logic programming), some background on molecular biology (section 2.2) and a short review of approaches to bacterial gene finding (section 2.3). Chapter 3 outlines my contribution and explains how the papers in this thesis fit together. Chapters 4 through 14 each contain a distinct paper. Finally, chapter 15 concludes and gives directions for future research.

Chapter 2

Background

2.1 Logic Programming

Logic programming is a way of using logical inference as a means for computer programming. Logic programming has roots which can be traced to before the modern computer, but the seminal paper "programming in predicate logic" [114] by Kowalski gave a procedural interpretation (of horn clauses) which inspired the invention of the Prolog programming language by Colmerauer and Roussel [48].

A logic program, as defined by Kowalski, consists of a finite set of rules defining implications (horn clauses), *i.e.*,

$$B \text{ if } A_1 \text{ and } \ldots \text{ and } A_n$$

$A_1, \ldots, A_n$ are positive or negative (negated) literals – atomic logic formulas which can be either true or false. B is a positive literal. B is called the consequent and $A_1 \ldots A_n$ are called antecedents. There may be multiple rules with the same consequent. Problems are stated as theorems (goals) to be proved through interpretations of implications, e.g., to prove the consequent B, we recursively need to prove the antecedents A_1 through A_n by using rules where these appear as consequents, or by the absence of such rules in the case of negated literals. This recursive procedure is called backward chaining. Alternatively, through the reverse procedure — called forward chaining — implications can be derived from a given set of antecedents.

Probabilistic logic programming is a form of logic programming which deals with uncertainty. A (non-probabilistic) logic program induces a set of possible worlds, i.e., the set of derivable consequents and their alternative proofs. Probabilistic logic programming extends logic programming by assigning probabilities to each of these possible worlds and extends logical inference into probabilistic inference, as to, e.g., derive the probability of a goal or infer the most likely derivation of a goal.

2.1.1 Logic Programming with Prolog

Prolog [45] is the perhaps the most used language and formalism for logic programming. Prolog employs a closed world assumption, which means that the absence of a fact implies its falsity (negation-as-failure).

There several are other logic programming formalisms, which have different underlying assumptions and different semantics. Prominent examples include Answer Set Programming [125] and Constraint Logic Programming [101].

2.1.1.1 Syntax and terminology of Prolog programs

In this section the basic syntactic elements of Prolog are introduced.

Clauses A Prolog program consist of a finite sequence of clauses where a clause is a formula (implication) on the form,

$$B \leftarrow A_1, \ldots, A_n \quad , \qquad n \geq 0$$

where $A_1, \ldots, A_n$ and B are literals. B is called the head (and must be a positive literal) and $A_1, \ldots, A_n$ is called the body. In actual Prolog programs, the ascii-friendly syntax `:-` is used in place of $\leftarrow$ and clauses are followed by a dot which syntactically denotes the end of the clause.

Literals A literal is an atom (positive literal) or the negation of an atom (negative literal).

Facts A clause with exactly one positive literal in its head and no body is called a unit clause or more often a fact. A fact may be declared as a clause on the form, $F \leftarrow true$, or just as a clause with no body F, where the implication $true$ is implicit.

Atoms An atom is a predicate of arity n, i.e., p or $p(t_1, \ldots, t_n)$, where $t_1, \ldots, t_n$ are terms.

Terms A *term* is either a variable, a constant or the application of a functor of arity $n \geq 0$, i.e., f or $f(t_1, \ldots, t_n)$, where $t_1, \ldots, t_n$ are terms.

Lists Prolog represents lists as nested terms. A list is a term, `'.'`(t, l), where `'.'` represents the cons function, t is a term and l is a list. As syntactic sugar, Prolog allows defining lists of terms, $t_1, \ldots, t_n$, using a bracket notation `[`$t_1, \ldots, t_n$`]`, $n \geq 0$. As a further notational convenience, list composition is possible using the notation `[`$t_1, \ldots, t_n|l$`]`, $n \geq 1$, where $t_1, \ldots, t_n$ are terms and l is a list.

Constants A constant is a numeral, a string starting with a lower case letter or any single-quoted string.

Variables A variable is a named placeholder for a term and is syntactically represented by a string token — the name of the variable — starting with an upper case letter or an underscore. Multiple occurrences of the same variable, i.e., with the same name, may occur in a clause.

The underscore identifier by itself or as prefix of a variable denotes a special kind of variable called an anonymous variable. Anonymous variables are used when the programmer does not care about the value of the variable and they serve primarily a notational convenience. Multiple occurrences of single underscore anonymous variables in a clause refer to distinct variables, whereas anonymous variable occurrences with the same name (having underscore as prefix) refer to the same variable.

Ground terms A term is said to be *ground* if it contains no variables or if — at some point in the execution of the program — all variables in the term are bound to ground values.

2.1.1.2 Declarative semantics of Prolog

The clauses represent universally qualified implications. For all variables $X_1, \dots, X_m$ in a clause $B \leftarrow A_1, \dots, A_n$, B is true if all of A_1, A_2 up to A_n are true, *i.e.*,

$$\text{for all } X_1, \dots, X_m \text{ occuring in the clause } B \leftarrow A_1, \dots, A_n \;\; | \;\; \wedge_{i=1}^{n} A_i \Rightarrow B.$$

The Herbrand universe $H_\infty(P)$ of a program P is the set of all ground terms in the program, which may be defined inductively:

- Let c be a constant in P, then $c \in H_\infty(P)$,
- Let $f(t_1, \dots, t_n)$ be a functor in P, then $f(t_1, \dots, t_n) \in H_\infty(P)$ if $t_1, \dots, t_n \in H_\infty(P)$

The Herbrand universe may be infinite.

The Herbrand base $H_{base}(P)$ of a program P is the set of all ground *atoms* which can be formed by replacing variables in clauses of P with elements of $H_\infty(P)$. $H_{base}(P)$ is infinite if $H_\infty(P)$ is infinite.

Let $H_{interpretations}(P)$ be the set of all subsets of $H_{base}(P)$. A Herbrand interpretation $I \in H_{interpretations}(P)$ of a program P is a subset of its Herbrand base, i.e., $I \subseteq H_{base}(P)$. An interpretation assigns truth values to all elements of $H_{base}(P)$, *i.e.* members of the interpretation are *true* and all other elements are *false*.

A Herbrand model H_{model} is a Herbrand interpretation, inductively defined as follows:

- Let A be a ground fact in P, then $A \in H_{model}$,
- Let $B \leftarrow A_1, \dots, A_n$ be a clause in P, let B' be ground instance of B and let $A'_1, \dots, A'_n \in H_{model}$.

  $B' \in H_{model}$ if for all A_i we can replace all variables $X_1, \dots, X_m$ in a clause $B \leftarrow A_1, \dots, A_n$, with a term in A'_i such that,

  - for all of A_i, $1 \leq i \leq n \;: A_i = A'_i$
  - B' is a ground instance of B where all such variables occurrences have been replaced accordingly.

The intersection of two Herbrand models is also a Herbrand model. The declarative meaning of a program P is the minimal Herbrand model of P, which is the intersection of all Herbrand models of P. A program always has a unique, minimal Herbrand model.

2.1.1.3 Operational semantics of Prolog

Prolog may also be seen as an abstract computational machine with goal-driven computation. The general principle of inference is called SLD-resolution (Selective Linear resolution with Definite clauses) [115].

Goal-driven computation (resolution) implies that the user states a goal — a conjunction of atoms to be proven — and backward chaining is applied in order to prove the goal. The operational semantics can be characterized in terms of a rewrite system [193].

A state of the computation is the pair $\langle G; \theta \rangle$, where G is a goal and θ is a substitution. A substitution is a finite set of mappings of variables to terms, i.e., $\theta = \{X_1 \mapsto t_1, \dots, X_n \mapsto t_n\}$.

The state is updated as a result of each rewrite step. Before the first rewrite step, θ is the empty substitution and G is the goal stated by the user. Resolution proceeds non-deterministically (with regard to selection of clauses) using two transition rules – *reduce* and *fail*,

$$\langle A_1, \dots, A_i, \dots, A_n; \theta \rangle \overset{reduce}{\rightsquigarrow} \langle A_1, \dots, B_1, \dots, B_k, \dots, A_n; \theta \circ \theta' \rangle \tag{2.1}$$

$$\langle A_1, \dots, A_i, \dots, A_n; \theta \rangle \overset{fail}{\rightsquigarrow} \langle fail; \theta \rangle \tag{2.2}$$

Here, $\theta \circ \theta'$ denotes the application of the substitution θ' after the application of the substitution θ. The *reduce* transition applies if a substitution θ' can be found through *unification*: The term A_i is unified with the head of a clause in the program, $A' \leftarrow B_1, \dots, B_k$, i.e., a substitution which is a most general unifier $\theta' = mgu(\{A_i, A'\})$ is found such that after applying it to A_i and A', written as $A_i\theta'$ and $A'\theta'$, $A_i\theta' = A'\theta'$. For the substitution θ' to be a most general unifier it must be the case that for any other unifier ω, there is a unifier λ such that $\theta' = \omega \circ \lambda$.

If no substitution can be found then unification fails and the *fail* transition applies.

As a strategy to deal with the non-determinism of SLD-resolution, Prolog uses depth-first search. In Prolog the computation always proceeds by considering each clause from top to bottom when trying to unify a goal. Once a satisfying clause is found, it proceeds recursively by attempting to unify each term in the body from left to right. In the case of failure of unification, the computation backtracks to the previous state and attempts the next clause.

To use of depth-first search means that resolution is incomplete and may not terminate. The principle is complete for certain types of otherwise non-terminating programs, when Prolog is extended with tabling (see section 2.1.1.5).

2.1.1.4 Proofs, proof trees and explanation graphs

A sequence of states such that the final state consists of an (irreducible) conjunction of facts is a *derivation* of the initial goal G. If no such sequence exists, *i.e.*, if every state sequence ends in a *fail* state, the goal G is *refuted* or *failed.* In the context of logic programs, the terms "proof", "derivation" and "explanation" are synonymous, but the use may differ depending on the context, i.e., the word "derivation" has operational connotations whereas the word "proof" has a more declarative flavor and word "explanation" is more often used with abductive reasoning.

A *proof tree* is a graph $T_G = \langle V, E \rangle$ where the vertices V correspond to goals, and the edges E are induced from the sequence of states in a proof in the following way,

- The root of the tree is the initial goal G.
- For each consecutive pair of states in the proof, the tree contains a directed edges from the rewritten consequent goal A_i to each of the antecedent goals $B_1, \ldots, B_k$.

There may be more than one distinct sequence of states which proves a goal G, and we denote the set of all such proofs P_G. We can compactly represent P_G as an *explanation graph* which we denote E_G. The explanation graph is a directed acyclic hypergraph[1]. More precisely, the explanation graph is a pair $E_G = \langle V, E \rangle$, where V is a set of vertices and the edges E are pairs $\langle v_0, \langle v_1, \ldots, v_m \rangle \rangle$ such that $v_0, \ldots, v_m \in V$. The explanation graph contains an edge $(p, \langle q_1, \ldots, q_n \rangle)$ from a state p to one or more other states, $q_1, \ldots, q_n$, if there is a proof tree T_G for G with edges $(p, q_1), \ldots, (p, q_n)$. Note that this representation share states which different proofs may have in common. The proofs may be recovered by traversal of the explanation graph; every distinct path from the root node to a leaf node corresponds to a proof.

2.1.1.5 Tabling and the explanation graph

Tabling is a mechanism whereby goals and their answers are memorized once computed. If a memorized goal is encountered in a different computation, the memorized answer is used rather than recomputing the goal. This is useful, e.g., when the same goal is present in multiple proofs or multiple times in a single proof.

Tabling can be seen as implicitly building an explanation graph without explicitly generating all proofs. Tabling stores one copy of each goal – isomorphic to a distinct goal corresponding to one node in an explanation graph. A table record of a goal has pointers to each of its answers – isomorphic to the edges of the explanation graph.

With tabling the number of computational steps needed to build the explanation graph is proportional with the size of the explanation graph, which may be fewer than required for enumerating all proofs if the proofs share common substructures.

2.1.2 Probabilistic Logic Programming with PRISM

PRISM — which abbreviates **PR**ogramming **I**n **S**tatistical **M**odels — is a probabilistic extension of Prolog with probabilistic declarative semantics developed at the Tokyo Institute of Technology by Taisuke Sato and Yoshitaka Kameya. It is based the highly efficient B-Prolog system [226] by Neng-Fa Zhou. The authors of PRISM are collaboration partners of the LoSt project. The PRISM

[1]A hypergraph is a generalization of a graph where an edge can connect any number of vertices, *i.e.* edges are non-empty subsets of vertices.

system is one of the most mature systems and efficient tools for probabilistic programming and is used extensively throughout this thesis.

This section gives a brief overview of the main facilities of PRISM and outlines how its syntax and semantics tie together to provide efficient probabilistic inferences. For more detailed descriptions, the reader is referred to the PRISM manual [207] and the comprehensive papers about PRISM [175, 176, 174].

We use the terms PRISM model and PRISM program interchangeably – A PRISM program is in effect a probabilistic model.

2.1.2.1 The distribution semantics

The underlying semantics of PRISM is called the distribution semantics [173] and defines a probability distribution over Herbrand interpretations of a program.

The sample space Ω_P — or *possible worlds* — of a program P is the set of all Herbrand interpretations of P. Let $A_1, B_2, \dots$ be the atoms in the Herbrand base of P. In a given Herbrand interpretation we may have $A_i = true$ or $A_i = false$, and the set of possible Herbrand interpretations form a cartesian product, $\Omega_P = \prod_{i=1}^{\infty}\{true, false\}_i$.

Let $\Omega_P^{(n)}$ be a restriction of the Herbrand base to the first n atoms, let $I_P^{(n)}$ be the set of Herbrand interpretations of $\Omega_P^{(n)}$ and let $P_P^{(n)}$ be a probability measure for $\Omega_P^{(n)}$, i.e., assigning a probability $p_{i^{(n)}}$ to each element $i^{(n)} \in I_P^{(n)}$.

A (possibly infinite) probability measure P_P on Ω_P is obtained by merging the family of finite probability measures $P_P^{(n)}$, given that the following three conditions holds for $P_P^{(n)}$:

$$\text{for any } p_{i^{(n)}} \in P_P^{(n)}, \quad 0 \le p_{i^{(n)}} \le 1 \tag{2.3}$$

$$\sum_{p_{i^{(n)}} \in P_P^{(n)}} p_{i^{(n)}} = 1 \tag{2.4}$$

$$\text{for any } i^{(n+1)} \text{ and } i^{(n)} \text{ s.t. } i^{(n)} \subseteq i^{(n+1)}, \sum p_{i^{(n+1)}} = p_{i^{(n)}} \tag{2.5}$$

Equation 2.3 asserts that the probability measure $P_P^{(n)}$ assigns a valid probability to each interpretation. Equation 2.4, also called the uniqueness condition asserts that sum of probabilities for all possible interpretations sum to unity, i.e., interpretations are exclusive probabilistic events. Finally, equation 2.5 asserts that when extending with another atom of the Herbrand base, events (interpretation of n atoms) are divided into specialized events (interpretations of $n+1$ atoms) which together constitute the event corresponding to the interpretation of the n atoms. The sum of probabilities of these specialized events are asserted to be equal to the probability of the event they constitute.

2.1.2.2 The PRISM language

The PRISM language implements the distribution semantics through an extension of Prolog with random variables embodied as a special type of goals called multi-valued switches (`msws`).

Since PRISM is a compatible extension of Prolog, the syntax PRISM is identical to that of Prolog.

Multi-valued switches Random variables, embodied by multi-valued switches, consist of a switch `values` declaration and may be used in the program by calling the `msw/2` goal.

A multi-valued switch goal (`msw`) takes the form

msw(t,v)

where t is a ground term. This causes v to be chosen randomly from a list of outcomes according to a predefined switch `values` declaration.

A switch `values` declaration takes the form

values(t_{id},[$v_1, \ldots, v_n$])

where t_{id} is a Prolog term and may contain (anonymous) variables and [$v_1, \ldots, v_n$] is a list of possible outcomes represented as ground atoms.

In a call to a multi-valued switch msw(t,X) for some ground t, the corresponding `values` declaration is found using unification of t and t_{id}. It is assumed that any term t is unifiable only with one values declaration. The outcome `X` is selected according to a probability distribution which can either be explicitly defined (using the `set_sw` goal) or learned from observed goals. For a multi-valued switch call msw(t,X), where t unifies with t_{id}, probabilities $p_1, \ldots, p_n$ are associated with each element of [$v_1, \ldots, v_n$] such that $\sum_{i=1}^{n} p_i = 1$. The outcome probabilities $p_1, \ldots, p_n$ are referred to as a parameter.

A `values` declaration with variables in the identifier may serve as to define the outcome space of a family of multi-valued switches with the same outcome space, but which may individually have different probabilities of these outcomes. This is for instance the case in the `coin(_) values` declaration in figure 2.1.2.2.

Multi-valued switch calls are stationary in the sense that the probability of the outcomes of an `msw` does not change over time (during execution of the program), *i.e.*, in any explanation ...,msw(i,X_j),...,msw(i,X_{j+1}),..., $P(X_j = v) = P(X_{j+1} = v)$ for some fixed switch i, for any (time) j and for any outcome v.

PRISM programs and the distribution semantics The distribution semantics is defined for PRISM programs where all probabilistic ground atoms[2] in the Herbrand base of a program are probabilistically exclusive, independent and unique. To guarantee this, all non-determinism must be governed by `msw`s. Essentially, for any disjunction in the program, the disjuncts must correspond to mutually exclusive subsets of the outcomes of an `msw` (or set of such).

Given independence and exclusiveness, the probability of any atom may be computed as a sum-product; The probability of a conjunction $A \wedge B$ is the

[2]Without loss of generality it can be assumed that all ground facts are represented as multi-valued switches in accordance with the distribution semantics. In practice, though, it is common to have both `msw`s and usual prolog facts (which can be seen as probabilistic facts with probability 1).

```
values(select_coin, [fair,loaded]).
values(coin(_), [heads,tails]).

flip_coin(Result) :-
    msw(select_coin, Coin),
    msw(coin(Coin), Result).

:- set_sw(coin(fair),[0.5,0.5]).
:- set_sw(coin(loaded),[0.9,0.1]).
:- set_sw(select_coin, [0.5,0.5]).
```

Figure 2.1: A simple PRISM program that simulates randomly selecting a coin (either fair or loaded) and flipping it to obtain either heads or tails. The first two lines declare the multi-valued switches, `select_coin` and `coin(_)`. The predicate `flip_coin` first invokes the switch `select_coin` to randomly select an outcome (and unify the variable `Coin`), which is either the `fair` or `loaded` as defined by the `values(select_coin,...)` declaration. Then, the `msw(coin(Coin),Result)` msw is called and value of `Coin` indicates which probability distribution to select unify `Result` – the outcomes are `heads` or `tails` regardless of the value of `Coin`. The probability distributions are defined manually in the last three lines, e.g., if `Coin=loaded`, then the probability of `heads` is 0.9.

product[3] $P(A) \times P(B)$ and the probability of disjunction $A \vee B$ is the sum $P(A) + P(B)$. This naturally generalizes to the goals and their explanations.

The *independence condition* asserts that the probability of an explanation $Expl_G$ of a goal G is the product of probabilities of `msw` outcomes in the explanation. For an explanation $Expl_G = \texttt{msw(}i_1\texttt{,}X_1\texttt{)},\ldots,\texttt{msw(}i_m\texttt{,}X_m\texttt{)}$ this is defined as follows,

$$P(Expl_G) = \prod_{j=1}^{m} P(\texttt{msw(}i_j, X_j\texttt{)})$$

The *exclusiveness condition* asserts that the probability of a goal G is sum of the probabilities of its explanations,

$$P(G) = \sum_{Expl_G} P(Expl_G)$$

The *uniqueness condition* — corresponding to equation 2.4 — asserts that the probabilities of all observable atoms sum to unity.

Conditional probabilities Conditional dependencies between `msws` can be modeled by capturing the outcome of a multi-valued switch in a variable and

[3] Note that the probability of a conjunction of identical `msw` goals is still the product, i.e., $P(msw(i,v) \wedge msw(i,v)) = P(msw(i,v)) \times P(msw(i,v))$. Intuitively, this differs slightly from from the usual logic interpretation, where conjunction of identical goals are truth-wise identical to the single occurrence of the goal. This is necessary to be able to reuse the same values declaration.

using this variable to identify a subsequent multi-valued switch, *i.e.*, in a conjunction `msw(a,X), msw(X, Y)`, the outcome of the probabilistic choice of the first `msw` is used as identifier for the second `msw`. The example in figure 2.1.2.2 illustrates this principle.

2.1.2.3 Inference with PRISM programs

A PRISM program is effectively a probabilistic models and inferences which are usually associated with probabilistic models can be performed on PRISM programs.

Sampling The direct mode of execution of PRISM programs — called sampling — closely resembles Prologs SLD resolution procedure, with the only exception being that the outcome of `msw` facts are determined probabilistically. Sampling a goal G is either done by explicitly calling the goal or calling `sample(`G`)`. Executing this query will probabilistically derive an explanation of G.

Probability calculation Besides sampling, a common inference is to calculate the probability of a goal G. Calling `probf(`G`,`P`)` will unify P to the probability of the goal G calculated as specified in the previous section. PRISM also supports computation of hindsight probabilities (`hindsight`) which are probabilities of subgoals given a top-level goal. Newer versions of PRISM may soon include a procedure for calculating prefix probabilities as well [189].

Viterbi decoding The Viterbi algorithm provided is a generalization of the classical HMM Viterbi decoding algorithm [218] extended for arbitrary PRISM programs. The generalized Viterbi algorithm [188] of PRISM finds the most probable explanation of a goal G, *i.e.*,

$$Viterbi(G) = \operatorname*{argmax}_{e \in Expl_G} P(Expl_G)$$

In practice, this procedure is invoked using the built-in `viterbif(`G`,`E`,`P`)`, where E is unified to the most probable explanation of G and P is the probability of E. PRISM provides various extensions of this, which can for instance return the n most probable explanations.

Parameter learning Finally, the parameters of a PRISM program may be learned from examples using the built-in predicate `learn([`$G_1, \ldots, G_n$`])`. PRISM includes several procedures for parameter learning. Using default settings `learn` performs *maximum-likelihood estimation*, where the model parameters are optimized as to maximize the probability that random sampling with the program would result in $G_1, \ldots, G_n$. This is achieved using an Expectation Maximization (EM) algorithm (gEM) [107, 105] which generalizes the well-known EM algorithm for PRISM programs. The PRISM system includes other learning methods which may be preferable in a given situation. The fgEM algorithm is used for programs with failures [182, 187, 183]. The VB-EM algorithm uses a Variational Bayes method which puts dirichlet priors (essentially pseudo-counts) on parameters [186] and is useful in particular when the training data is sparse. More recently the PRISM system includes Hard-EM (Viterbi learning) which eliminates the exclusiveness condition [188] and provides for better prediction accuracy under certain conditions. Furthermore, the most recent version of PRISM includes also Markov Chain Monte Carlo (MCMC) [177]. The choice of algorithm to be used when calling `learn` is determined by setting execution flags (`set_prism_flag`)[4].

Parameter learning is fully supervised when $G_1, \ldots, G_n$ are ground, but learning become unsupervised or semi-supervised if all or some of $G_1, \ldots, G_n$ contain (possibly anonymous) variables.

The explanation graph The central inference mechanisms of PRISM — except, maybe sampling — rely on explanation graphs, which are efficiently implemented by tabling [228]. The number of different derivations in a PRISM program may be exponential, but the explanation graph represents these in polynomial space. The computed probability – maximal in the case of `viterbi` and total in the case `prob` – of a goal is stored in the explanation graph and reused for derivations which include the same goal. The use of the explanation graph allows PRISMs generic inference mechanisms to mimic specialized and efficient dynamic programming algorithms complexity-wise. For a variety of commonly used models, e.g., Hidden Markov Models and Stochastic Context-Free Grammars, PRISM theoretically achieves the same computational complexity as the best known specialized algorithms.

2.1.2.4 Introducing constraints

Constraints may be seen as a restriction of the set of possible worlds of a program.

A hard constraint reduces the set of possible worlds of a program. A program P' is a constrained version of a program P if the possible worlds of P' is a subset of the possible worlds of P, i.e., $\Omega_{P'} \subseteq \Omega_P$. Hard constraints may be introduced in logic programs through the coercion of logic variables. Such a coercion can be introduced, e.g., by asserting equality, `X=Y`, or inequality, `X\=Y`, between variables.

PRISM programs may in the same way as Prolog programs introduce constraints through coercion of logic variables. In PRISM, however, the terms bound by variables may be stochastically determined and unification failures arise from incompatible probabilistic choices. For instance, in a conjunction of msw calls `msw(`i_1`,`X`)`,`msw(`i_2`,`X`)`, the first call will unify X to the probabilistically determined outcome of i_1, but the second call i_2 may try to unify X to a different outcome.

This is useful for variety of purposes – for instance, to express Constrained Hidden Markov Models or probabilistic grammars with more than context-free expressivity [187, 88].

It also means that PRISM programs may fail – the sampling of a goal may result in failure of unification. This violates the uniqueness condition and compromises the distribution semantics since the probabilities may not sum to one – a portion of the probability mass is assigned to the failed derivations.

[4]Refer to the PRISM manual for details.

This can be alleviated by making failures explicit and mapping them instead to a form of successful derivations. PRISM incorporates the first order compiler, which given a PRISM program $Prog$, generates a failure program $\neg Prog$ that traces all failed derivations, such that $Prog \cup \neg Prog$ satisfies the uniqueness assumption.

Failures also present challenges for parameter learning. For a stochastic process, we may not be able to observe failures. We can only observe what corresponds to the successful derivations. It therefore not possible to directly estimate parameters which maximize the likelihood of *all* the (observed and unobserved) data. Usually, PRISMs EM learning procedures rely on the uniqueness assumption, which implies the falsity of all unobserved atoms. In the case of programs with failures, PRISM maximizes instead likelihood of observed data and normalizes the resulting distribution by estimating the probability of failure using the failure program. This is handled by the fgEM algorithm [182, 187, 183], which is an adaptation of Failure-Adjusted Maximization [53].

As an alternative to programs which introduce failures, constraints may instead be introduced as soft constraints. A soft constraint does not reduce the set of possible worlds, but it expresses a preference for or a bias towards a subset of possible worlds. In PRISM, conditional probabilities is the natural way to express soft constraints.

Soft constraints expressed through conditional probabilities can be data driven, in the sense that the degree of preference corresponds to observed empirical frequencies of correlations in the training data. In the total absence of (observed) correlation, conditional probabilities learned from data by maximum likelihood in effect become hard constraints since associated events are assumed to have zero probability[5]. Expressing constraints through conditional probabilities have the disadvantage of leading to a larger set of model parameters – hence requiring more training data for reliable learning and leading to degradation of inference performance.

2.1.3 Constraint logic programming

Constraint logic programming (CLP) combines logic programming and constraint satisfaction. In addition to the usual literals, constraint logic programs may include constraints which restrict the possible values for logic variables.

These constraints may for instance be equality constraints (tree constraints), *e.g.*, X=Y which forces the unification of the two variables. It may also be other types of constraints which specify the relation between variables with certain types of values such as integers or floats, e.g., `X > Y` or `X = Y + 1`. While ordinary logic programs may also include such constraints, the constraints in a constraint logic program are treated differently.

Clauses in a constraint logic program takes the form,

$$A \leftarrow B_1, \ldots, B_n, C_1, \ldots, C_m.$$

where $A, B_1, \ldots, B_n$ are literals and $C_1, \ldots, C_m$ are constraints[6]. A solution to a constraint logic program is a proof of a goal G in which all constraints occurring in clauses selected in the proof of G are satisfied.

[5]Assuming that we use plain EM, *i.e.*, no priors or pseudo-counts.

[6]The ordering $B_1, \ldots, B_n, C_1, \ldots, C_m$ is artificial and used only for notional convenience. In practice literals and constraints can occur in mixed order.

Constraint logic programs maintain a constraint store which holds all satisfied constraints. This constraint store may be rewritten or simplified during execution and constraints may be replaced by other constraints (simplification) or new derived constraints may be added to the constraint store (propagation).

2.1.3.1 Semantics of Constraint Logic Programming

The cruix of constraint logic programming is the constraint store. We formulate the semantics of constraint logic programs in a similar fashion to the formulation of semantics of Prolog programs in section 2.1.1.3. We represent the state of computation as a triple $\langle G; \theta; \gamma \rangle$, where G is a goal, θ is a substitution and γ is a constraint store. The constraint store may be represented as a set (or multiset) of constraints. Computation may then — as with Prolog programs — be characterized as a non-deterministic rewrite system,

$$\langle A_1, \ldots, A_i, \ldots, A_n; \theta; \gamma \rangle \overset{reduce}{\rightsquigarrow} \langle A_1, \ldots, B_1, \ldots, B_k, \ldots, A_n; \theta \circ \theta'; \gamma \cup \{C_1, \ldots, C_n\} \rangle \tag{2.6}$$

$$\langle A_1, \ldots, A_n; \theta; \gamma \rangle \overset{simplify}{\rightsquigarrow} \langle A_1, \ldots, A_n; \theta \circ \theta'; \gamma' \rangle \tag{2.7}$$

$$\langle A_1, \ldots, A_i, \ldots, A_n; \theta; \gamma \rangle \overset{fail}{\rightsquigarrow} \langle fail; \theta; \gamma \rangle \tag{2.8}$$

The reduce transition resembles the rule from with the same name from the Prolog semantics. Constraints, however, may be added to the constraint store as an effect of this reduce transition. The reduce transition only applies if the constraint store $\gamma \cup \{C_1, \ldots, C_n\}$ is consistent (satisfiable).

The *simplify* applies whenever $\gamma \neq \gamma'$ and $\gamma \models \gamma'$ according to the semantics of the constraint solver. The *simplify* transition updates the constraint store and applies rules of the constraint solver to simplify the constraint store. Note that the constraint solver may update the substitution.

As with Prolog programs, the *fail* transition applies if no other rule applies.

2.1.3.2 Constraint Handling Rules

Constraint Handling Rules (CHR) [71] is a rule based language which rewrites a constraint store represented as a multiset. It was originally developed as a language to write constraint solvers, but has found many applications beyond this domain [198]. The language is used within a host language — typically Prolog — and inherits notions from this language. In this thesis, only Prolog is considered as host language. With Prolog as host language, a constraint is represented as a Prolog term and CHR inherits logic variables and unification.

CHR is a declarative and rule-based language like Prolog. However, unlike Prolog, CHR is a committed-choice language, which means that once a particular rule has been applied, no backtracking occurs.

The language include three types of rules: Propagation, simplification and simpagation.

A simplification rule takes the form,

$$C_1, \ldots, C_i \texttt{ <=> } \text{Guard} \quad | \quad C_{i+1}, \ldots, C_n$$

```
leq(X,X) <=> true.
leq(X,Y), leq(Y,X) <=> X=Y.
leq(X,Y) \ leq(X,Y) <=> true.
leq(X,Y), leq(Y,Z) ==> leq(X,Z).
```

Figure 2.2: A CHR program (adapted from [71]) implementing a constraint solver for less-than-or-equal `leq` constraints. The first rule (reflexivity) is a simplification rule which remove `leq(`X`,`X`)` constraints from the constraint store. The second rule removes the two anti-symmetric constraints in the head and unifies X and Y (using the built-in constraint `=`). The third rule (a simpagation rule) removes duplicate constraints (such may occur due to the multi-set semantics of CHR). The fourth rule is a propagation rule which models the transitivity of `leq` constraints by adding the (redundant) `leq(`X`,`Z`)` constraint to the constraint store.

A simplification rule replaces the constraints in the head, $C_1, \ldots, C_i$, with the constraints in the body, $C_{i+1}, \ldots, C_n$, if the *Guard* is true. The guard is a conjunction of so-called built-in constraints, which take the form of a Prolog goal when the host language is Prolog. It may be omitted in all types of rules. The `<=>` syntax implies the relation to logical equivalence.

A propagation rule takes the form,

$$C_1, \ldots, C_i \texttt{ ==> } \text{Guard} \texttt{ | } C_{i+1}, \ldots, C_n$$

Propagation rules add the constraints in the body to the constraint store when matching the constraints in the head and if the *Guard* is true. For a particular instance of constraints, $C_1, \ldots, C_i$, the rule is only applied once. The `==>` syntax implies the relation to logical implication.

The third type of rule, simpagation, takes the form,

$$C_1, \ldots, C_i \backslash C_{i+1}, \ldots, C_m \texttt{ <=> } \text{Guard} \texttt{ | } C_{m+1}, \ldots, C_n$$

The simpagation rule is a hybrid between the simplification and the propagation rules. The constraints $C_1, \ldots, C_i$ are kept in the constraint, the constraints $C_{i+1}, \ldots, C_m$ are removed and the constraints $C_{m+1}, \ldots, C_n$ are added to the store.

Several semantics have been introduced for CHR. Informally, the abstract semantics assumes confluence[7], i.e., it assumes that the order of rule applications are not important. The refined semantics (which is used by most implementations) select the next rule to apply by giving precedence to the top-most applicable rule in the CHR program.

A small CHR program is shown in figure 2.1.3.2.

2.2 Biology and biological sequence analysis

In this section I introduce basic biological concepts which are central to the kind of biological sequence analysis presented in this thesis. The focus is on relevant concepts with regard to the biology of prokaryotic organisms. The

[7]This concept is defined more formally in chapter 7

introduction is brief and the reader is referred elsewhere for details, e.g. [4]. It does, however, contain some details which would not ordinarily be part of an introductory text on molecular biology. These details are included in anticipation of some of the more biologically oriented papers in this thesis.

2.2.1 Basic genetics

The cells of living organisms contain a blueprint of the organism in the form of DNA, short for Deoxyribo-Nucleic Acid. DNA is a molecule composed of short subunits called nucleotides. Nucleotides consist of a five carbon sugar compound, a phosphate group and a base. In DNA the base is either cytosine (C), guanine (G), adenine (A) or thymine (T). The letters, indicated for each of the bases form the four letter alphabet $\{A, G, C, T\}$.

In eukaryotes,*e.g.*, humans, plants and amoeba, the genetic material is stored in a protective core within the cell called the nucleus. In these multicellular organisms, the same genetic material is stored in each cell. The DNA is coiled up in chromosomes, of which eukaryotes usually have several.

In prokaryotes (Bacteria and Archaea), which are usually single-celled, the cells have no nucleus and the genetic material floats freely in the cytoplasm of the cell. Prokaryotes usually carry only a single chromosome.

The total genetic material of an organism — including all chromosomes and extra-chromosomal genetic material (*e.g.*, plasmids) — is called the genome. The size of the genome varies greatly between organisms. The (haploid - resting phase) human genome consists of roughly three billion basepairs, whereas the genomes of prokaryotes are usually less than ten million basepairs.

The genome consists of DNA molecules in a double helix structure. The two DNA molecules (strands) are bound together through hydrogen bonds between complementary bases: cytosine-guanine pairs and adenine-thymine pairs.

The nucleotides of each strand are joined together covalently via phosphate links between the sugars, resulting in a sugar-phosphate backbone structure. Each nucleotide is joined with neighboring nucleotides. One is connected through a bond to the $3'$ carbon atom of the sugar and the other is connected through a bond to the $5'$ carbon atom of the sugar, giving each strand a $5'$ and $3'$ directionality.

The genetic material of the genome is organized in genes.

Originally — before the age of modern molecular biology as initiated by the discovery of DNA in 1953 [220] — the term gene referred to the discrete units of heredity associated with differences and similarities between parents and their progeny. The term is now usually used to denote stretches of DNA which are transcribed to mRNA and encode proteins – the main macromolecules that perform the many functions of a cell. (See below for some more detail).

The term upstream with regard to a gene refers to the region before the $5'$ end of the gene downstream refers to region after its $3'$ end - of its "sense" strand (the one that is identical to the transcribed mRNA sequence).

2.2.2 The central dogma

The phenotype of an organism is its observable function and appearance. The genotype of an organism is the encoding of its phenotype in the genome. The central dogma of molecular biology refers to the template directed sequential synthesis processes of DNA, RNA and protein that connect the genotype with the phenotype [52].

In the *replication* process DNA is copied during cell-division. In the *transcription* process, DNA is transcribed into RNA. Some RNA sequences (messenger RNA, mRNAs) can act as templates for the accretion of amino acids that compose a protein (the *translation* process).

These "information transfer" processes between DNA, RNA and proteins are the most central ones, although other transitions are known to occur, for instance RNA to RNA (RNA replication) and RNA to DNA (Reverse Transcription).

2.2.2.1 Transcription and translation

Transcription from DNA into RNA is performed by the transcriptional apparatus, which at its core has a DNA dependent RNA polymerase enzyme. The polymerase attach to a promoter sequence in one of the DNA strands — the template strand — and separates the two strands in a short approximately 10 bases window and proceeds in the $3' \rightarrow 5'$ direction of the template strand. As it proceeds, it produces a growing strand of RNA which is complementary to the template strand through the addition of ribonucleotides at the $3'$ end of the nascent transcript.

Somewhere downstream of the promoter sequence, the polymerase encounters a termination sequence at which point it detaches from the template strand. The transcribed mRNA is released.

In eukaryotes the mRNA then migrates from the nucleus to the cytoplasm where it is translated by a massive ribonucleic-protein complex called a ribosome.

In prokaryotes — which have only the cytoplasm — the translation of the mRNA may begin while it is still being transcribed and several ribosomes may translate the same mRNA simultaneously.

The ribosome attaches to the mRNA at a ribosomal binding site — the Shine-Delgarno sequence — and proceeds downstream *i.e.* in the $5' \rightarrow 3'$ direction. Downstream the RBS, the ribosome begins translation when it encounters a specific sequence triplet called the initiation codon which is translated into the amino acid methionine. It then proceeds translating each triplet — a codon — into an amino acid. Table 2.1 show the relation between specific codons and the corresponding amino acids. The amino acids are carried to the ribosome by transfer RNAs (tRNA) which are specific to the type of amino acid they carry. The tRNAs have an anti-codon which is complementary to a triplet in the mRNA and by mediation of the ribosome they attach to such triplets. The ribosome detaches the amino acid from the tRNA and adds it the to the growing polypeptide chain. For specific codons known as stop codons, there are no tRNAs with complementary anti-codons. Instead, proteins known as release factors attach to these. When the ribosome encounters a release factor it releases the protein and detaches from the mRNA. An mRNA may encode a single protein or multiple proteins (poly-cistronic transcripts). The region upstream an initiation codon is called the $5'$ untranslated region (UTR) and the region downstream a stop codon is called the $3'$ UTR.

2.2.2.2 Replication

Prokaryotic genomes usually only have a single large circular chromosome. Replication is performed by a multi-component molecular machinery called the replisome, that initiates replication at a specific location in the genome (the origin) and proceeds along the genome in both directions (replichores) towards the region were replication terminates (terminus).

Replication is initiated by elongation of short RNA sequences called primers, that forms the beginning of the synthesis of a new anti-parrallel strand. An enzyme — DNA polymerase III — attach to the primer and elongates the new strand at its $3'$ end.

The $3' \rightarrow 5'$ template strand of each replichore is called the leading strand and opposite the $5' \rightarrow 3'$ template strand is called the lagging strand (replication proceeds slower with this strand since the polymerase can only polymerize from the $3'$ end resulting in DNA synthesis going in the opposite direction of the replisome).

On the leading strand, DNA polymerase III continuously elongates the synthesized strand in the $5' \rightarrow 3'$ direction and towards the terminus. Only one RNA primer is needed.

In the lagging strand DNA polymerase III also work in the $5' \rightarrow 3'$ direction, which is toward the origin. RNA primers attach every 1000-2000 bases and the DNA polymerase III synthesize the fragments in between. These are called okazaki fragments.

The RNA primers are subsequently removed by an exonuclease (polymerase I) and the synthesized fragments are joined together by an enzyme called DNA ligase.

It is important to note that replication and translation are not isolated processes, but occur simultaneously in prokaryotes. Halfway through the replication process the chromosome will have the two copies of the DNA near the origin, but only one copy near the terminus. Hence, genes located near the origin are more frequently expressed than genes near the terminus. Collisions between the enzymes involved in these two processes can affect the efficiency of either process. For this reason, genes are preferentially located on the leading strand.

2.2.3 Genes

In the previous section I loosely introduced a gene as a stretch of DNA which is translated to a functional protein.

As a simplified working definition, a (prokaryotic) gene is a consecutive stretch of DNA which,

- is translated to a complete protein and
- has a length which is a multiple of three (codons) and
- in which the $5'$ end starts with a codon translated as methionine and
- which contains exactly one stop codon – the last codon near the $3'$ end.

This definition is an oversimplification which besides excluding non-protein encoding genes, i.e., functional RNA genes, also excludes several classes of

First base in codon	Second base in codon				Third base in codon
	T	C	A	G	
T	TTT Phe F	TCT Ser S	TAT Tyr Y	TGT Cys Y	T
	TTC Phe F	TCC Ser S	TAC Tyr Y	TGC Cys Y	C
	TTA Leu L	TCA Ser S	TAA Stop *	TGA Stop *(U)	A
	TTG Leu L	TCG Ser S	TAG Stop *(O)	TGG Trp W	G
C	CTT Leu L	CCT Pro P	CAT His H	CGT Arg R	T
	CTC Leu L	CCC Pro P	CAC His H	CGC Arg R	C
	CTA Leu L	CCA Pro P	CAA Gln Q	CGA Arg R	A
	CTG Leu L	CCG Pro P	CAG Gln Q	CGG Arg R	G
A	ATT Ile I	ACT Thr T	AAT Asn N	AGT Ser S	T
	ATC Ile I	ACC Thr T	AAC Asn N	AGC Ser S	C
	ATA Ile I	ACA Thr T	AAA Lys K	AGA Arg R	A
	ATG Met M	ACG Thr T	AAG Lys K	AGG Arg R	G
G	GTT Val V	GCT Ala A	GAT Asp D	GGT Gly G	T
	GTC Val V	GCC Ala A	GAC Asp D	GGC Gly G	C
	GTA Val V	GCA Ala A	GAA Glu E	GGA Gly G	A
	GTG Val V	GCG Ala A	GAG Glu E	GGG Gly G	G

Table 2.1: The standard genetic code. The table shows how triplets of nucleic acid bases correspond to different amino acids. Besides the codon ATG with always codes for methionine, alternatively TTG, CTG, ATT, ATC, ATA and GTG can serve as initiation codons, in which case they are translated as methionine rather than the amino acid indicated.

rare protein encoding genes. In the case of two rare amino acids the standard genetic code is expanded and codons which are normally stop codons may encode amino acids; Selenocysteine (U) is a rare amino acid which is encoded by the stop codon TGA and similarly the amino acid pyrrolysine (O) is encoded by the stop codon TAG. Other factors such as mRNA structure contributes to translation of these stop codons as amino acids. Another exception is genes with programmed frame shifts, where the translation may skip a few bases.

On each strand, genes can occur in three different reading frames. Each reading frame begins with a different nucleotide of a triplet.

Genes in prokaryotes may overlap in the sense that the same stretch of DNA may be part of more than one gene. Overlapping genes have been observed in all transcriptional cases, *i.e.*, co-directional (→→), convergent (→←) and divergent (←→) [147] (see table 2.2). The lengths of overlaps are usually quite small – less than three bases, but longer overlaps have been observed as well, and existing prokaryotic genome annotations include several much longer overlaps, i.e., more than 100 bases. Many of the excessively long overlaps, however, are thought to be misannotations [151].

2.2.4 Evolution

Organisms evolve through modifications of the genetic material which are beneficial (or at least not overly detrimental), *i.e.*, contributing to the survival or proliferation of the organism. Genome mutations can occur due to errors in the replication process or through exposure to radiation or chemicals. Such mutations can be a point mutations where a base changes into an other base, but can also be insertions or deletions of (even multiple) bases.

Unlike eukaryotes which primarily combine genetic material with close relatives (through sexual reproduction), the acquisition of genetic material from distant organisms — called lateral or horizontal transfers — play a major role

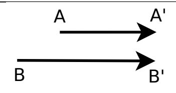
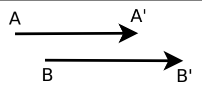
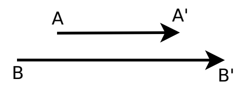
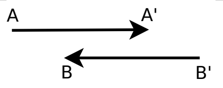
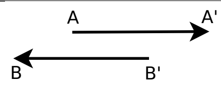
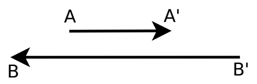
Same strand, same reading frame overlapping gene pairs	
Co-directional orientation ($\rightarrow\rightarrow$) $A > B; A' = B'$	
Same strand, different reading frame overlapping gene pairs	
Co-directional orientation ($\rightarrow\rightarrow$) $A < B; A' < B'$	
Co-directional orientation (embedded) ($\rightarrow\rightarrow$) $A > B; A' < B'$	
Sense-antisense overlapping gene pairs	
convergent orientation ($\rightarrow\leftarrow$) $A < B'; A' < B'; B < A'$	
divergent orientation ($\leftarrow\rightarrow$) $B < A; B' < A'; A < B'$	
divergent orientation (embedded) ($\leftarrow\rightarrow$) $B < A; A' < B'$	

Table 2.2: Different transcriptional cases for overlapping gene pairs. Note that a gene may be part of more than one overlapping pair. A, A', B and B' indicate gene ends – A and B correspond to the first nucleotide of the start codon of the genes and similarly A' and B' correspond to last nucleotide of the stop codon. The arrows indicate the direction of transcription of a gene, i.e., the strand that the genes are located on.

in prokaryotic evolution [148]. Lateral gene transfer can happen through a variety of different mechanisms, *e.g.*, via plasmids or bacteriophages.

Prokaryotes may carry plasmids which are covalently closed circular double stranded DNA molecules – a sort of mobile mini-chromosomes which may contain thousands of base pairs, but are typically much smaller than a prokaryotic chromosome. Plasmids do not encode genes necessary for translation and replication, but rely the on genes in the chromosome for these processes. Prokaryotes can exchange genetic material through plasmids by a mechanism called conjugation, where the plasmids pass through a pore established between two prokaryotic cells. Plasmids may recombine with genetic material of the chromosome, thus enabling transfer of chromosome encoded genes between organisms.

Bacteriophages are viruses which attack bacteria and injects their genetic material into the host chromosome. Their viral genome encode genes necessary for its own replication outside the host genome[8], but rely on the translation mechanisms of the host for expression of its genes. When such viruses replicate

[8]This process is distinct from the replication process of the host.

they may copy adjacent host genomic material into the viral genome which is then injected when the virus attacks another host organism. This form of lateral transfer is called transduction.

Genetic material can also be copied within a genome. For instance, it may be beneficial for an organism to have multiple copies of a gene for which a high level of expression is advantageous. The copying of genetic material within a genome occur through the processes of lateral transfer and a range of other mechanisms. An integron is a dynamic mechanism whereby cassettes of genes are inserted in tandem at a special site mediated by an enzyme called an integrase. Transposons are clusters of genes which are transferred in bulk and reinserted mediated by an enzyme called a transposase.

Genes which exist in multiple copies within a genome of a single specie are called paralogues. Genes which exist across species are called orthologues. Homologues is a broader term which refer to both paralogues and orthologues.

2.2.4.1 Size of prokaryotic genomes

Prokaryotic genomes are small compared to the genomes of eukaryotes, ranging from a few hundred thousand and up to ten million basepairs. The vast majority of the genome encode genes (typically around 85%) and there are little non-functional DNA. The number of functional genes in prokaryotes correlates with the size of their genomes [223].

Prokaryotes tend to shed the genes that they do not need. Having a small genome can be advantageous for proliferation, since the replication process is faster. Genes which do not confer an evolutionary advantage may mutate to become dysfunctional, i.e., they are not translated or are quickly degraded after translation. Such dysfunctional genes which represent an intermediate stage before deletion are called pseudogenes. Comparisons between pseudogenes and homologous functional genes reveal that prokaryotes have a mutational bias which favor deletions over insertions [136].

2.2.5 RNA

RNA is very similar to DNA, but it is usually single-stranded and has a different kind of sugar backbone; RNA use ribose instead of deoxiribose as DNA. Furthermore, in RNA Uracil (U) is used in place of Thymine (T).

RNA molecules have many diverse roles in biological processes, *e.g.*, in the form of messenger RNA (mRNA) it carries the genetic information from the genome to be translated, in the form of transfer RNA (tRNA) it recognizes codons and carries amino acids, and in the form of ribosomal RNA it catalyzes the formation of a peptide bond between two amino acids.

Where the double-helical shape of DNA is very stable, RNA molecules can take a rich variety of shapes by forming hydrogen bonds between bases in the molecule. The functional shape of RNA molecules is essential to their function. For instance, a tRNA is clover-shaped with an exposed anti-codon which can bond with a particular codon. In the mRNA, the breaking of formed bonds leads to translational stalling which can affect the shape of growing protein and the termination sequence may attain a particular functional structure catalyzing termination. Certain mRNA structures play a key role in programmed frameshifts and also in the alternative translation of the non-standard amino acids Pyrrolysine and Selenocysteine.

The structure of an RNA molecule can be characterized in different levels of detail; The primary structure refers to the sequence of bases, the secondary structure specify the hydrogen bonds between the bases, and the tertiary structure specifies the three-dimensional structure in atomic coordinates of the molecule.

2.3 Gene finding

Prokaryotic gene finding have been a motivating case for the application of probabilistic programming in the LoSt project and it is a central theme in this thesis. This section introduces the problem of gene finding and explores different perspectives and different ways of defining and decomposing the problem. It explores contemporary state-of-the-art techniques by examining some of the most prevalent modern gene finders and the assumptions and techniques they use.

A gene finder is a computational/statistical model which attempt to predict (some of) the genes of a genome.

Gene finders exploit some underlying assumptions about what constitute genes and assumptions about factors outside the sequence comprising the gene that may affect the likelihood of a particular sequence being a gene or not. These assumptions are reflected as constraints in the computational gene finder model, and as result, these models may serve as formalized definitions of genes with a probabilistic part which captures the uncertain aspects. Taking this view, a gene finder may be seen as forming a hypothesis; It proposes a simplified view of genes which can be evaluated with respect to accuracy in the prediction of known genes.

A more pressing concern is that computational gene finders are necessary for analyzing the vast amounts of sequence data being generated. Manual annotation and experimental verification of genes is expensive and computational gene finding provides a cheaper (feasible) way of annotating this data. This is feasible only because gene finders can be evaluated, so that qualified estimates about their accuracy can be used to assess the quality of resulting annotations. Conversely, the accuracy of a gene finder correlates with the probability that predicted genes are real genes. This may guide biologists in their work in order to minimize wasted lab effort spent on verifying genes.

State-of-the-art prokaryotic gene finders are very successful and mature, and have made great progress in providing an accurate set of predictions (conversely, eukaryotic gene finders have limited accuracy). It has been stated that prokaryotic gene finding is an almost solved problem. Such a statement is not completely unjustified, since gene finders correctly predict the majority genes in most prokaryotic genomes.

Still, accuracy depends a lot on the organism. Many short and unusual genes elude modern gene finders. Furthermore, accuracy is difficult to assess for genomes which have not been extensively studied and for which the golden standard annotation consist largely of unverified predictions from other gene finders. Hence, there is still room for improvements and computational prokaryotic gene finding is still an important and active and important area of research.

This dissertation include several approaches to gene finding, which aims to further state-of-the-art. In chapter 6, we provide a method to deal with overlapping genes. Chapter 9 explores a novel approach to combine different signals in a gene finder. Chapter 13 demonstrates how the sequence of gene reading frames may be used to improve the accuracy of prokaryotic gene finding. Chapter 14 presents a gene finder for an usual type of genes which may include the non-standard amino acid pyrrolysine.

2.3.1 Constraints and assumptions

This section will briefly discuss some of the main assumptions commonly used in computational gene finding without going into details about specific gene finders.

2.3.1.1 Signals used for gene finding

The way the problem of (prokaryotic) gene finding is traditionally stated is roughly as a crisp classification problem, where some stretches of DNA are classified as either protein coding or not [66]. Due to the requirements of the translation process of genes, this classification problem can be restricted to subsequences which start with a codon which can be translated as methionine (a start codon) and ends with a stop codon. Such a sequence, if it is not interrupted by stop codons, is called an Open Reading Frame (ORF).

More formally, an ORF is a sequence of DNA in the sense strand which can be generated by the following grammar in BNF form:

$$\begin{array}{rcl}
\langle\text{ORF}\rangle & ::= & \langle\text{start}\rangle\ \langle\text{not-stop}\rangle^{*}\ \langle\text{stop}\rangle \\
\langle\text{start}\rangle & ::= & \text{TTG} \mid \text{CTG} \mid \text{ATT} \mid \text{ATC} \mid \text{ATA} \mid \text{ATG} \mid \text{GTG} \\
\langle\text{stop}\rangle & ::= & \text{TAA} \mid \text{TAG} \mid \text{TGA} \\
\langle\text{not-stop}\rangle & ::= & \text{AAA} \mid ... \mid \text{TTT} \quad //\textit{all codons except those in } \langle\text{stop}\rangle
\end{array}$$

The identification of prokaryotic genes may be decomposed into two distinct problems:

1. Identification of ORFs which contain protein coding genes.

2. Identification of the correct start codon within an ORF.

Selection of the correct start codon tends to be the hardest of the two problems. Current state-of-the-art gene finders achieve near perfect accuracy for problem 1, but somewhat lower accuracy for problem 2.

A strong indicator and property of protein coding genes is the statistical composition of nucleic acid sequences. The nucleic sequence composition is markedly different in protein coding genes than it is the non-coding parts of the genome. Genes tend to have a higher G/C content than non-coding regions and since the stop codons have bias towards A/T, the probability of a stop codon arising by chance is lower in high G/C content DNA sequences.

Statistically speaking, a long uninterrupted ORF becomes unlikely to occur by chance and hence long ORFs are likely gene candidates. In early approaches to gene finding it was noted that simply predicting very long ORFs (longer than

400 – 500 bases) as genes is remarkably accurate (in particular with respect to problem 1) [25]. This method is obviously incapable of predicting shorter genes.

In most cases, the correct start codon is an ATG (in approximately 80 percent of genes[9]) which can encode only methionine and alternative start codons occur less frequently.

The context of the ORF can also provide useful indications as to whether it comprises a gene and for the selection of the start codon. Often, a ribosomal binding site — most often the Shine-Dalgarno sequence `AGGAGG`[10] – is present 6-7 bases upstream of the start codon of a gene. Similarly, a RNA transcription termination signal, e.g., a Rho-independent terminator stem-loop, may be found downstream a gene.

Furthermore, the expected extent and size of overlaps between genes can often be used to eliminate candidate ORFs which significantly overlap with a much more promising candidate.

Because of the massive amount of genomes being sequenced, it is often the case that similar genes occur in different genomes. A predecessor gene may have been present in a common ancestor and evolutionarily conserved or genes may have been acquired horizontally. If a gene has been identified (and possible verified) in one genome, the identified similar gene in the other genome is most likely also a gene. Such *homologues* are usually identified by sequence alignment using high-performance heuristic alignment techniques such as BLAST[9]. Hence, conservation is a powerful signal which facilitates the detection a significant amount of genes. Core genes – common genes which are necessary for survival — are usually well-conserved among organisms. Genes which are specialized for individual organisms, are more less likely to be conserved across species. Since most of the core genes have been identified, even as more genomes are sequenced, detection of new genes by conservation is subject to diminishing returns.

2.3.2 Evaluation of gene finders

Gene finders classify stretches of the genome as either coding or non-coding[11]. They are then evaluated on their ability to correctly predict the *coding* parts of the genome. This can be evaluated at both nucleotide and gene level[12]. In the case of the evaluation on the gene level we may consider correctness with regard to the $3'$ end or correctness with regards to both the $3'$ and $5'$ end of the gene.

Gene finders are usually evaluated with regard to the golden standard of experimentally verified genes or a reference annotation which contain a set of known genes (often only a subset of these are experimentally verified).

Alternatively, gene finders can be evaluated by comparing to predictions of other gene finders. This may serve as a way to compare the different signals and

[9]Depending on the genome, e.g., in *E.Coli* it is 83 percent.

[10]Different organisms vary slightly in what compromises a Shine-Dalgarno sequence, e.g., in *E.coli* it is most frequently `AGGAGGT`.

[11]In practice, we may have more fine grained classes, but usually only these two are considered for evaluation purposes.

[12]In the case of Eukaryotic genes, evaluation is often with regard to exons (transcribed parts), which in eukaryotic genes are separated by introns (parts which are not transcribed).

hypotheses embodied in different gene finders. If the same genes are detected by gene finders using distinct signals, then this may indicate that these signals are redundant. If such signals leads to similar predictions, but otherwise have nothing to do with each other, *i.e.*, they are orthogonal, then this may serve as substantial evidence to indicate that a signal is useful.

Unless the golden standard consists of experimentally verified genes, evaluation is biased. If the predicted genes are again used for evaluating other gene finders, then a systematic propagation of bias, *i.e.* ascertainment bias, may occur.

This comparison between predicted genes and a golden standard ("reality") can be viewed in terms of a contingency table:

Reality

Prediction

	coding	non-coding	
coding	TP	FP	$TP + FP$
non-coding	FN	TN	$FN + TN$
	$TP + FN$	$TF + TN$	

True positives TP is the number of true positives (correct predictions with label coding), FP is the number of false positives (incorrect predictions with label coding), FN is the number of false negatives (incorrect predictions with label noncoding) and TN is the number of true negatives (correct predictions with label non-coding).

Results of gene finders are usually reported using sensitivity and specificity measures. Both measures can be reported with regard to predicted bases, gene $3'$ ends and genes $3'$+$5'$ ends.

We can derive sensitivity and specificity from the contingency table. Both measures are relative frequencies which can be intuitively interpreted as probabilities estimates of a models performance.

Sensitivity (SN):

$$SN = \frac{TP}{TP + FN} \approx \hat{P}(prediction(\text{coding})|\text{coding}) \tag{2.9}$$

Sensitivity can be seen as a completeness measure. Sensitivity is a relative measure of the amount of classification instances correctly predicted as *coding* out of the total number of classifications which *should* be predicted as *coding*.

Sensitivity is also sometimes referred to as *recall* in the context of natural language processing and information retrieval [31, 131].

In binary classification problems specificity is traditionally defined [8] as,

$$SP_{traditional} = \frac{TN}{TN + FP}. \tag{2.10}$$

In the gene prediction literature, however, it is usually defined as,

Specificity (SP):

$$SP = \frac{TP}{TP + FP} \approx \hat{P}(coding|prediction(\text{coding})) \tag{2.11}$$

In the context of this thesis, the latter definition of specificity may be assumed.

Specificity is an exactness measure. It is the proportion of classification instances *correctly* predicted as *coding* out of all instances predicted as *coding*. In natural language processing and information retrieval the same measure is usually referred to as *precision*.

Sensitivity and specificity are usually reported together, since the measures are not very informative individually. For this reason, a combination of the measures is also often used. One such combined measure is the correlation coefficient,

$$CC = \frac{(TP \times TN) - (FN \times FP)}{\sqrt{(TP + FN) \times (TN + FP) \times (TP + FP) \times (TN + FN)}} \tag{2.12}$$

Unfortunately, the correlation coefficient has the problem that it is undefined if one of the factors in the square root term of the denominator becomes zero.

In practice, a measure called Average Correlation (AC)[13] is often used instead [26]:

$$AC = \frac{1}{2}\left(\frac{TP}{TP + FN} + \frac{TP}{TP + FP} + \frac{TN}{TN + FP} + \frac{TN}{TN + FN}\right) - 1 \tag{2.13}$$

These integrative accuracy measures, and others, are treated in more detail in [26]. With regard to gene finder results it is almost always the case that $AC \geq CC$ and that AC is within 0.05 of CC 90% of the time [26].

Treating gene finding as a simple classification problem makes it straightforward to evaluate using the above measures. Reporting only the binary classification for each ORF is, however, usually a simplification which is subject to information loss.

For gene finders using a single Viterbi decoding or other methods which result in a set of predictions with no further ordering with regard to the probability of individual predictions, these measures are suitable. However, if the method used reports the probability of each individual prediction, *e.g.*, as is usually the case with posterior decoding [61], there are more informative ways of reporting accuracy.

The probabilities of individual predictions induce an ordering of predictions, *e.g.*, a ranking of predictions from most probable to least probable. Selecting the top-n predictions from this ranking, impose a discrimination threshold which is a function of n, and which give rise to a particular tradeoff between sensitivity and specificity. By measuring sensitivity and specifity for each possible n, we can derive and plot a Receiver Operator Curve (ROC) [64]. An example ROC curve is shown in figure 2.3. A ROC curve has a nice intuitive interpretation – it can be seen as dividing a probability space. The area under the curve (AUC) corresponds to the probability that the classifier will rank a randomly chosen positive example higher than a randomly chosen negative example.

In the lack of a golden standard it is more difficult to evaluate gene finders. One possibility is to consider how well the model fits the training data. For a probabilistic model trained using the maximum-likelihood principle, this is

[13]This measure is sometimes misleadingly described just as "accuracy", perhaps due to the ambiguous abbreviation.

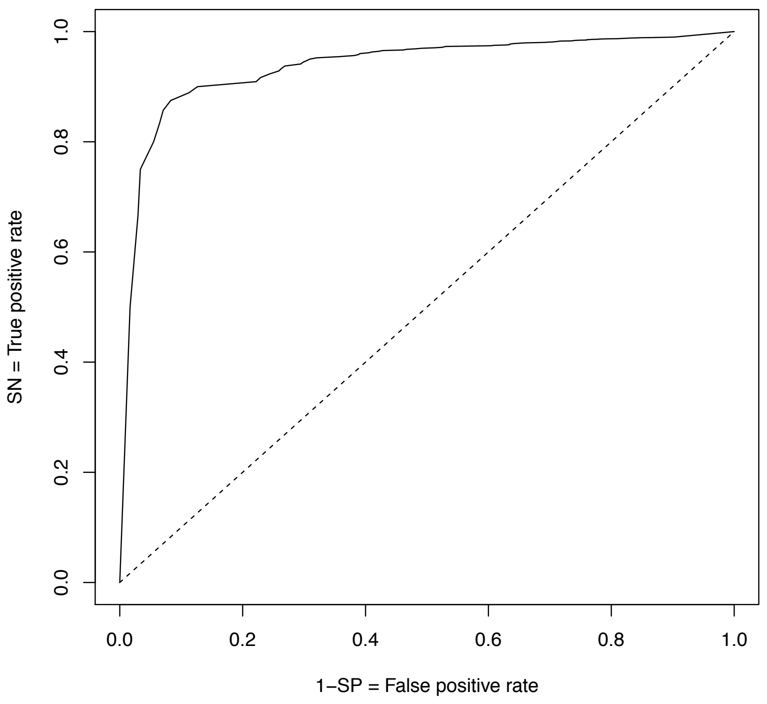


Figure 2.3: Example Receiver Operator Curve. A set of predictions corresponds to a point in the ROC space. Perfect prediction is achieved at the point $(0, 1)$. The dotted line corresponds to random choice of classification – predictions below this line are worse than random and predictions above the line are better than random. The solid curve exemplifies (good) predictions at various discrimination levels.

measured by the likelihood of the training data given the model and its parameters. Increased likelihood, however, can be due to overfitting and provides no guarantee that the gene finder yield accurate predictions for anything but the training data. To ensure a good balance between overfitting and generalization capability a model selection criterion can be used. One such measure is the Bayesian Information Criterion[14],

$$BIC = -2 \times ln(L) + k \times ln(n)$$

where L is the maximized likelihood, k is the number of model parameters and n is the number training examples. BIC is expected to roughly correlate with accuracy and can be used to choose the model with expected best accuracy among alternative models.

[14]This criteria is provided for trained models in PRISM.

2.3.3 Probabilistic methods for prokaryotic gene finding

This section reviews the techniques used by some of the most prevalent prokaryotic gene finders. This includes the modeling techniques that they use, how they estimate their probabilistic models and how they evaluate the results. In this brief and non-exhaustive review we consider techniques which are related to prokaryotic gene finding and may elucidate the relevant concepts.

An *intrinsic*, *de novo* or *ab initio* gene finder is a computational gene finding method which is characterized by only considering the sequenced genome as its input. Such gene finders rely only on data which is “available to” the mechanisms within the organism. Hence, they constitute hypotheses about the internal machinery of the organisms.

A gene finder which also relies on the genomes and/or annotations of other organisms are called *extrinsic* or *comparative.*

Some gene finders rely only on contigs – consensus fragments of genomes from sequencing, whereas most methods use whole chromosomes. A special case is *meta-genomic* gene finders, which deal with contigs or reads from multiple organisms simultaneously.

Some recent gene finders incorporate more experimental evidence sources, such as expressed sequence tags [22] – a sort of complementary RNA primers used to detect expressed mRNAs.

2.3.3.1 Preprocessing and postprocessing

While many gene finders employ a rigorous and statistically well-justified model at their core, they usually also employ certain tricks of the trade to preprocess the input data or post-process the results.

It is common to consider only ORFs that are larger than a particular size, ranging from 60 bp - 200 bp. Most methods have difficulties with such short ORFs (of which there are many) and the statistical significance of predictions for short ORFs is low.

Post processing usually occur after the gene finder has decided on a set of high-probability predictions. In particular, a pair of predictions may be unlikely if the predictions overlap each other to a large extent.

2.3.3.2 Intrinsic methods

A survey of measures of coding potential was described in [66]. The main conclusion from the paper was that oligomer statistics — frequencies of consecutive nucleotide patterns — is the most effective measure for determining coding potential and that other considered measures are to a large degree redundant.

Oligomer statistics can be reflected by Markov models. An example of a Markov chain for DNA is shown in figure 2.4. A Markov chain defines the probability $P(S)$ of a sequence $S = s_1, \ldots, s_n$, which is decomposed as

$$P(S) = P(s_n, \ldots, s_1) = P(S_n|s_{n-1})P(s_{n-1}|s_{n-2}) \ldots P(s_1)$$

by repeated application of the chain rule. In an $n'th$ order Markov chain, the probability depends on the n previous states.

The Genemark [138] gene finder was pioneering in the use of Markov models in gene finding. In this gene finder, a “coding” Markov chain model is applied

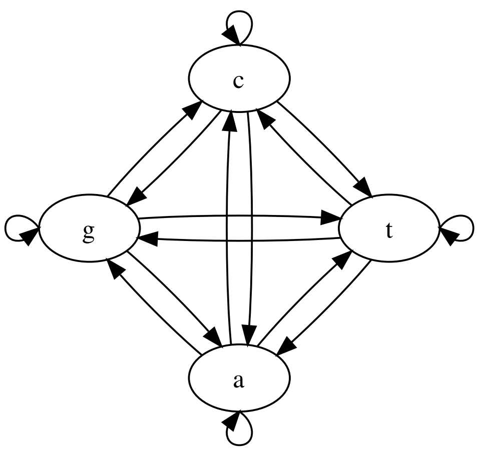


Figure 2.4: A Markov chain for DNA with the states for each base (circles) and possible transitions between states are represented by arrows. Each transition is associated with a probability.

for each reading frame and a reading frame agnostic non-coding Markov model is also applied for the same sequence. These models are applied to calculate the probability of sequences, given the respective model, in windows of a defined size. A naive Bayes model is then applied to select the model which is more probable for the sequence. This serves to classify each window as being coding in a particular reading frame or as being non-coding.

Higher order Markov models can capture longer oligomer patterns, potentially increasing the accuracy of their predictions. The higher the order of the Markov chains, however, the more training data is necessary to reliable estimate the transition probabilities. The Glimmer gene finder [57, 172] use interpolated Markov models (IMMs) to deal with this problem. In this gene finder, Markov models of different orders are interpolated to give a combined transition probability $P^{IMM}(s|c_k)$ of a transition to a state s given different order k-mers[15] $c_k, c_{k-1}, \dots, c_1$, as defined by the equation,

$$P^{IMM}(s|c_k) = \lambda(c_k) \times P(s|c_k) + (1 - \lambda(c_k)) \times P^{IMM}(s|c_{k-1}).$$

The interpolation parameter $\lambda(c_k)$ is a weight assigned the k-mer c_k which reflects the reliability of $P(s|c_k)$ in terms of the amount of training that was used to estimate it.

Later developments include the use of Hidden Markov Models (HMMs) [116, 117, 128, 123] which have been a large influence in the field. HMMs provide for more elegant modeling and rather than explicitly using naive Bayes inference to decide between several individual (Markov chain) models, such

[15] A k-mer is an oligomer of length k

choices are embedded in HMMs using hidden states. A decoding algorithm is used to select a sequence of hidden states for an "observed" input sequence. The corresponding sequence of hidden states serves to classify each symbol in the input sequence. The HMM is constrained by the allowed transitions between states and it is in this way possible to ensure that the sequence of hidden states is grammatically sound with respect to genes. For an introduction to HMMs and variants, see chapter 4.

Different HMM designs calls for different decoding measures. The Viterbi decoding strategy is well-suited for looping HMM designs which for instance specify a grammar alternating between coding states and non-states. In "linear" designs, posterior decoding — which is used to calculated the posterior probability of each state for each sequence position — is a more suitable choice. In looping designs, posterior decoding may lead to an grammatically inconsistent state sequence.

The ECOPARSE program [117] was pioneering in its use of Hidden Markov Models for gene finding. It uses a looping design (see figure 2.5) and uses the Viterbi algorithm for decoding. The gene finder was developed before the *E.coli* genome was fully sequenced and used the Viterbi algorithm for decoding of each available contig[16].

Easygene [123] is a prominent HMM-based prokaryotic gene finder which uses posterior decoding. Easygene is built on a linear design (see figure 2.6) and decoding is done using posterior decoding. One of the merits of Easygene over other gene finders is its capability to reliably detect some shorter genes. Most gene finder approaches exclude genes shorter than a certain size. Easygene incorporates duration modeling the HMM design by applying three identical codon models in series. In conjunction with posterior decoding, this allows for more general length distributions than geometrical[17]. Easygene explicitly models the context near the two ends of a gene, since these may have more distinct patterns [133]. It also explicitly models a possible ribosomal binding site in front of a gene. Even with duration modeling, it is still difficult to reliable predict short genes. To deal with this, Easygene provides a measure of statistical significance for each predicted gene.

Common for most gene finder approaches is the assumption that protein coding genes in a particular genome are characterized by a common codon or nucleotide k-mer frequency pattern and that such a pattern may be used to distinguish them from non-gene parts of the genome. While models based on this assumption may successfully identify most genes, they ignore possible differences in characteristics between genes of the genome. Horizontally transferred genes tend to have a different frequency pattern than host genes [133]. Genemark-Genesis [90] deals with the problem by creating a model for typical genes and one for atypical genes. Easygene models this by having three branches of codons models and can hence reflect three distinct types of gene frequency patterns. However, these approaches assumes common characteristics among atypical genes. RescueNet [129] eliminates this assumption using Self-Organizing Maps (SOMs) to automatically derive multiple coherent amino acid frequency bias models. The RescueNet model, however, is based only on simple amino acid frequency bias as opposed to the HMM based models which

[16]A continuous DNA subsequence resulting from sequencing.

[17]The probability of staying in a state with a self-transition decays geometrically.

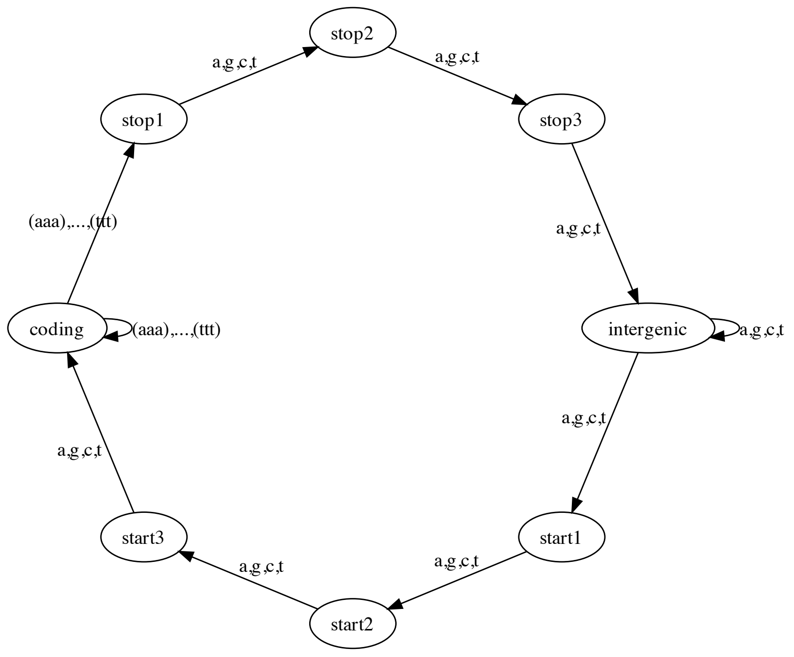


Figure 2.5: The ECOPARSE HMM model. Outgoing transition arrows are labelled with the possible emission from each state. All states except the coding state emits each a, g, c or t. The coding state emit one of 61 possible codons (*i.e.*, excluding stop codons).

also reflect sequential grammatical aspects of the sequence.

Overlapping genes can be problematic for gene prediction approaches. ECOPARSE provide an extension of the basic model, which probabilistically models short overlaps between genes, but relies on heuristics for resolving longer overlaps. Several gene finders, *e.g.*, Genemark, ECOPARSE and RescueNet, use heuristic rules for excluding predictions with long overlaps.

A recent tendency in prokaryotic gene finders, however, is to consider the set of predictions in a genomic context, rather than merely predicting and selecting ORFs individually. In Glimmer 3 [56] and Prodigal [99], a global dynamic programming algorithm is responsible for selecting predictions in a coherent way that respects distance and overlap constraints.

To construct training data, Prodigal implements a dynamical programming algorithm, that selects a subset of preliminary predictions of genes (represented by their containing ORFs) in the genome. Initially, Prodigal screens these ORFs to identify G/C bias in different codon positions. Then G/C bias is used to construct an initial score for Each ORF is given a preliminary score calculated as a function of the relative codon position G/C bias and the ORF length. In a pass of the DP algorithm, Prodigal connects a highest scoring sequence of ORFs subject to constraints on overlaps between these ORFs. A maximal

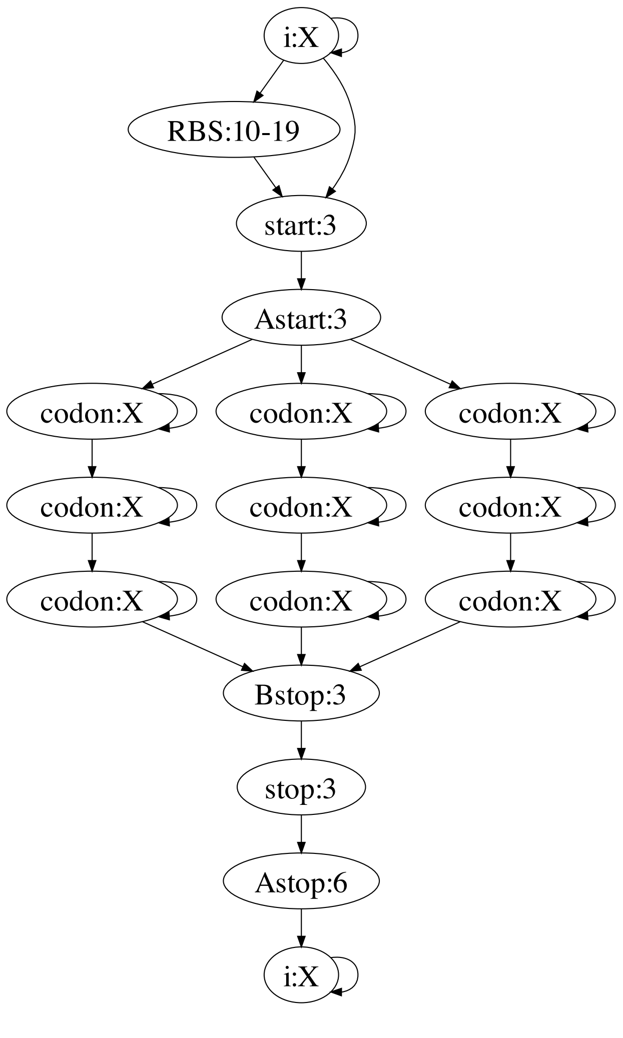


Figure 2.6: The Easygene HMM model. The number of bases emitted from each state is indicated by the number after the colon. The *i* state models intergenic non-coding sequence. The *RBS* state models a possible ribosomal binding site, *start* models a start codon, *Astart* models bases immediately after the start codon, *codon* is a fourth-order gene codon model, *bstop* models base immediately before the stop codon, *stop* models a stop codon and *Astop* models bases immediately after a stop codon.

overlap of 60 bp is allowed between ORFs on the same strand and 200 bp overlap between two ORFs on opposite strands if the overlap is between the $3'$ ends of the ORFs. No overlap is allowed between $5'$ ends of genes on opposite strands.

A very similar DP algorithm is used for prediction. The identified ORFs from the first algorithm are used as training data for a second pass of the DP algorithm where they use a linear combination of hexamer (k-mers of length 6, corresponding to two codons) scores, scores for start codon usage, scores for detected ribosomal binding sites and a score for distance to to an upstream gene. The scores are additionally tweaked to favor particularly long genes. The overlap constraints applied in the first DP algorithm for construction of training data are also applied in the prediction step.

2.3.3.3 Extrinsic techniques

The BLAST tool [9] is one the most commonly used tools in bioinformatics. It is a tool which given a database of sequences (nucleotide or amino acid), finds statistically significant sequence matches to a given (sequence) query, possibly allowing for gaps and mutations.

A recent approach by Poptsova and Gogarten [156] applies BLAST to search for orthologues to annotated genes and large number of missing genes in bacterial genome annotations.

The ORPHEUS gene finder [69] combines frequency pattern statistics with evidence of known proteins. For each ORF, ORPHEUS translates it to a corresponding amino acid sequence and uses database similarity search to find known proteins which are significantly similar.

The CRITICA gene finder [13] computes a codon score for ORFs using statistically significant matches to similar ORFs in the same and other genomes. Inexact matches are scored higher than exact matches, which assures that phylogentically broad conservation is given more weight and that, *e.g.*, repeat regions within the genome are given less weight. The conservation score is combined with a hexamer frequency score.

BioDictionary [194] is a database of statistical oligomer patterns derived from genes from different genomes. This database is used for gene finding by matching patterns against an ORF and determining if the number of matches is statistically significant.

2.3.4 Pipelines and combiners

Annotation pipelines are integrated tools with the purpose of assisting annotation and curation of genomes. Gene finding can be seen as series of dependent tasks – particularly for approaches which incorporate different types of data. Training data must be gathered and the data may need to be preprocessed and prepared before the probabilistic model can be trained and applied to prediction on relevant part of the data. Similarly a range of post-processing steps may be applied, e.g., overlap resolution, start-site correction [209, 150], accuracy calculation or similar. Often, these steps are delegated to several dedicated reusable tools. The annotation pipeline is the glue that tie such tasks together. A variety of dedicated tools and languages have been proposed for this purpose, e.g, [171, 202, 95, 62, 158]. The annotation pipeline may also include several gene finders to provide better coverage (sensitivity). Many annotation pipelines include a visual component – often in the form of a "track viewer" which provide simultaneous overview of different sources evidence, e.g., [201, 195].

A form of pipeline which does not just aggregate predictions and other evidence sources, but combines them to form more accurate predictions is called a combiner. Combiners are very successful in Eukaryotic gene finding [6, 152, 27, 69, 224, 192, 47, 7, 5, 126], but have found limited application in the domain of prokaryotic gene finding, e.g., [231] – perhaps because of the limited gain due to the high accuracy of prokaryotic gene finders as opposed the imprecision of eukaryotic gene finders. Many of the techniques used in combiners for eukaryotic genomes could in theory also be directly applied to prokaryotic genomes, but would probably be less accurate than a single gene finder since they do not reflect the peculiar properties in prokaryotic genomes (*e.g.*, overlapping genes).

2.3.4.1 Training data

There are various methods to bootstrap gene finders with training data which may also be obtained in a variety of ways.

One way is simply to use the annotated genes of the genome and train the gene prediction algorithm on those. In newly sequenced genomes, however, no annotated genes are available.

In practice, this problem is alleviated by using training data obtained from existing annotations of a phylogenetically related genome. Even so, this solution may introduce a bias towards the existing set of genes which may have be obtained primarily using other computational gene prediction methods. To alleviate these problems, some approaches use only genes which are annotated to have a known function and avoids using genes marked as putative or hypothetical in training data. For model organisms such as *E. coli*, a database with experimentally verified genes [108] exists and training only on those is also a possibility for reducing bias introduced in reference annotations by other gene finders.

Self-training has been used to deal with the problem of sparse training data, *e.g*, the gene finder GenemarkS [18] iteratively trains on its own predictions (which is demonstrated to yield improved accuracy).

To completely avoid dependency on existing annotations, several approaches use the unusually long open reading frames — which are statistically likely to contain real genes — as training data. The approach works reasonably well in genomes with low G/C content, whereas long ORFs become more likely to occur by chance in genomes with high G/C content. Some approaches such as [99] attempts to adjust for this.

Easygene uses long conserved ORFs found using database similarity search as training data. Its underlying model, however, still constitutes a hypothesis, since conservation is only used in the training phase, so I still consider it an intrinsic gene finder.

Chapter 3

Overview of Contributions

Each of the following chapters correspond to a particular published or draft paper and the title of the chapter is the same as the title of the paper. A few papers occur in a slightly extended form of the published versions. The aim of this chapter is to provide an overview of the contributions of the papers and to illustrate how these papers fit into the big picture and relate to the research goal. The contributions are divided into to one or more of the following categories:

Abstractions provide a higher level language for working with some sort of problem. The abstractions presented build on the general framework of probabilistic logic programming and extends it toward more specific applications.

Optimizations deals with the complexity of biological sequence analysis by enhancing the underlying algorithms or systems. The optimizations included in this thesis pertain mostly to tabling, a feature which enables the dynamic programming algorithms that is used for inference with probabilistic models.

Applications are tools which can be applied to biological sequence analysis for some particular purpose.

3.1 Abstractions

Several new abstractions are introduced in this thesis. These are summarized in this section.

3.1.1 A Zoo of Hidden Markov Models

Hidden Markov Models are ubiquitous in biological sequence analysis and a variety of different kinds have emerged. In chapter 4 we suggest a unified characterization of various types of Hidden Markov Models using the generic, probabilistic-logic framework PRISM and its generic inference methods. This may promote experiments with new variants of HMMs. Such variants may even involve context dependencies that traditionally are considered beyond reach of HMMs.

A variety of Hidden Markov Models and related models are introduced and details of their implementation in PRISM is described.

The chapter may serve as a starting point to learn about probabilistic sequence analysis with PRISM. It is a self-contained introduction to Hidden Markov Models and the PRISM programming language, although it does assume familiarity with Prolog.

3.1.2 Probabilistic extended regular expressions

Chapter 8 introduces the formalism, *Probabilistic extended regular expressions.* Regular expressions is a familiar and widely used formalism which is integrated in many modern programming languages. Contemporary versions of regular expressions are typically extended variants whose expressive power goes beyond regular languages.

Probabilistic extended regular expressions combine extended regular expressions and probabilistic models. This introduces the possibility to learn the affinity for strings and matches from examples and to assign probabilities to alternative ways of matching a string. The procedural control semantics are replaced by a probabilistic semantics, where the possible matches are ranked by their probability and the most probable match is the one returned.

To deal with cases where the expressive power of extended regular expressions is insufficient, we extend the syntax to integrate external functions, which may be deterministic or probabilistic. It is demonstrated how such extensions can support context-free features and approximate matching of noisy data which may contain insertions, deletions and mutations.

As an abstraction, probabilistic extended regular expressions provide a familiar way to express string matching problems, while at the same time utilizing the power of probabilistic models. This could appeal to programmers, even if they have limited knowledge about probabilistic models or if they are not interested in the details of an underlying probabilistic model. It also simplifies the control logic of traditional extended regular expressions, which is undoubtedly a complex issue that can lead to unintended behavior if the programmer is not careful. The probabilistic variant is more declarative in the sense that the user may disregard issues such as greedy or lazy control – the probabilistic semantics cover cases entailed by both forms of control logic. Another advantage is that the formalism is that less rigorous regular expressions can be made to match what the user intends, if he can provide sufficient training examples to accompany it. This may be useful for instance in cases where the user cannot anticipate all pattern variations in advance.

The formalism is not as expressive (in terms of languages that can me modeled) as, *e.g.*, stochastic definite clause grammars [88], but provides a simple language to express small grammars which are nevertheless capable of modeling a wide range of phenomena. While it can be used for limited parsing, it is primarily intended for more adhoc pattern matching.

3.1.3 Constrained Hidden Markov Models

Chapter 5 describes constrained Hidden Markov Models, which is an extension of Hidden Markov Models with side-constraints. The chapter is an extended version of [41] which includes an example from the preceding workshop paper

[42]. It is demonstrated how side-constraints can simplify modeling and lead to improved performance for certain applications. This is exemplified this using a constraints on a pair HMM for alignment of sequences. Such pair HMMs are widely used for alignment of biological sequences. The side-constraints are added through a `CLP(fd)`-like language which restricts the allowed state sequence and emission sequence. The language allows the expression of well-known global constraints such as `cardinality` and `all_different`. The constraint model considers each time step as variable whose domain is the possible states of the HMM. By adding constraints, the domains of these variables — and hence the state sequence — are restricted. Constrained Hidden Markov Models are implemented as a PRISM-based framework. A constraint store is maintained such that it corresponds to (partial) derivations of the PRISM programs. When the PRISM derivations are extended, the constraint store is updated. If the extension leaves the constraint store inconsistent, the PRISM derivation is forced to fail. Constrained Hidden Markov Models may also be seen as an optimization of Hidden Markov Models. The restriction of possible paths combined with efficient tabling mechanisms may lead to more efficient inference.

In chapter 6 we outline a similar way of adding constraints to Markov chains, but where constraints are instead expressed using a restricted form of Constraint Handling Rules (CHR).

As a different, but related approach, in chapter 7 we consider the expression of the common Viterbi inference algorithm for HMMs in CHR.

3.1.4 Bayesian Annotation Networks and BANpipe

In chapter 9, we introduce Bayesian Annotation Networks (BANs). Bayesian Annotation Networks is a modular methodology, in which complex probabilistic-logic models are defined in terms of separate sub-models, each representing a particular aspect (or "signal") of the input data to be analyzed. The dependencies among the results of analyses performed by these sub-models are described in terms of edges in a Bayesian network. This allows for an implementation based on incremental application of standard methods for prediction and training, one sub-model at a time.

While Bayesian Annotation Networks is an abstraction which allows decomposition of complex models, the proposed inference algorithms for BANs can be viewed as optimizations. The rationale for decomposition and the use of the (approximative) inference algorithms is that inference with a joint model is infeasible.

Chapter 10 introduces BANpipe – a logic-based pipeline scripting language designed to facilitate complex compositions time consuming analyses. Although it is designed to support Bayesian Annotation Networks, it is a general pipeline programming language.

The language supports complex pipelines of Prolog programs, PRISM models and other types of programs through rules which specify dependencies between computations. While existing pipeline scripting languages are not designed for the integration of Prolog and PRISM programs, BANpipe provides for a smooth integration of such programs at the language level.

BANpipe rules implicitly express dependencies between symbolically represented files which are automatically mapped to the underlying filesystem.

The symbolic filenames may include logic variables which enables advanced control mechanisms, i.e., recursion, leading to compactly expressed pipelines. The declarative semantics of the language facilitates goal directed execution, parallel execution, change propagation and type checking.

The applications described in chapter 9, 13 and 14 were implemented using early versions of BANpipe.

3.2 Optimizations

Dealing with biological sequence data requires efficient algorithms. Such algorithms are usually based on dynamic programming which exploits common problem sub-structure to reduce complexity to polynomial rather than exponential. Tabling of structured data is important to support dynamic programming in logic programs. In particular, the algorithms which underpin probabilistic inference in PRISM are based on tabling. The earlier tabling system in B-Prolog and PRISM did not handle structured data in goals efficiently and as result the problems that could be efficiently dealt with in the framework of PRISM was limited.

Early on in the LoSt project, Christiansen and Gallagher [40] deal with one such problem related tabling of annotation arguments. They show that these arguments do not affect control flow and that they can be eliminated during inference and then reconstructed afterwards. They also demonstrate how these arguments can be identified by means of program slicing and implement a program transformation that automatically handles argument elimination and reconstruction. This significantly reduces the running time of PRISM programs with such arguments.

Even with this optimization, scalability in PRISM is still an issue. It is common to introduce constraints in arguments. These affect the control flow, but only in a limited way, *i.e.*, they may lead to failure. Tabling of such arguments, however, leads to significantly decreased efficiency. Tabling of arguments which contain structured data is also inefficient. Since the underlying tabling system does not allow any sharing of such arguments, multiple copies of the arguments are tabled during execution of PRISM programs leading to increased time and space complexity due to argument copying.

To deal with these problems, an approach for (avoiding) tabling of constraint stores and two approaches for efficient tabling of structured data have been devised [86, 227], included as chapter 5, chapter 11 and chapter 12, respectively.

3.2.1 Tabling of constraint stores

In the context of constrained Hidden Markov Models, tabling of a constraints as arguments can significantly decrease efficiency. For such arguments, generalized as constraint stores, we provide a mechanism to allow for incremental constraint checking whilst avoiding tabling of the constraint store (see chapter 5). This leads to huge performance gains for certain types of constraints. The approach is, however, only sound for certain types of constraints and in chapter 6 a mechanism for pruning tabled constraint stores is applied instead.

3.2.2 Efficient tabling of structured data

In chapter 11, we propose simple program transformation which uses an indexing of structured data to allow efficient tabling of structured data.

The transformation results in $O(1)$ time and space complexity for table lookups for programs with arbitrarily large ground structured data as input arguments. A term is represented as a set of facts, each representing a subterm which is referenced by a unique integer serving as an abstract pointer. Matching related to tabling is done solely by comparison of such pointers, independently of the underlying system. The transformation is not specific to PRISM programs, but applies to any Prolog system with tabling.

Benchmarks demonstrate significant speedups of the transformation for all major Prolog systems with tabling and also specifically for PRISM.

The transformation has some limitations, though:

- It uses dynamically asserted facts to achieve constant time lookups - this incurs a considerable constant time overhead.

- It does not allow sharing of tables between different calls.

The paper advocates that similar techniques are integrated directly with underlying tabling systems, where it can be implemented at a lower level where machine address pointers are available.

In a collaboration with Neng-Fa Zhou — the author of B-Prolog — techniques to ensure efficient tabling of structured data was later implemented directly in the tabling system of B-Prolog.

In chapter 12 we apply enhanced hash-consing to improve tabling of structured data in BProlog. While hash-consing can reduce the space consumption when sharing is effective, it does not change the time complexity. We enhance hash-consing with two techniques, called input sharing and hash code memoization, for reducing the time complexity by avoiding computing hash codes for certain terms. The improved system is able to eliminate the extra linear factor in the old system for processing sequences, thus significantly enhancing the scalability of applications such as language parsing and bio-sequence analysis applications.

A comparison with trie based tabling systems reveals that the hash-consing technique is a more attractive method for most sequence analysis programs. Generally speaking, a trie is suitable for sharing prefixes and hash-consing is suitable for sharing suffixes of sequences. Although it is possible to find programs that make prefix sharing arbitrarily better than suffix sharing, it is more common for subgoals of recursive programs to share suffixes than prefixes.

The method is also compared with effects of the program transformation in chapter 11 and is shown to achieve comparable or better results. Unlike the transformation approach, the hash-consing approach fully exploits data sharing. The improvements of the technique is verified using experimental results.

3.3 Applications

3.3.1 Models of repeats in DNA

Finding repeats is an important task in biological sequences analysis. There are countless variations of the problem depending on the type of repeats, the distance between repeats and any combination of such constraints.

Repeats may occur, e.g. through mutational events and can have important functions such as serving as a defense mechanism against invading vira. Oppositely, repeats may also be of a malicious kind which causes diseases in the host organism. A particular form of repeats known as CRISPRs, which are regions of bacterial or archaeaic DNA with short direct repeats, comprised of repeat elements of 24-28 basepairs and interleaved by spacers of around 30 basepairs. The repeat element itself consist of an inverted repeat, *i.e.*, a palindromic structure. The function of CRISPRs is to remember past exposure to exogenous elements such as phages; A protein (CAS) intercepting viral DNA creates a novel spacer and inserts it into to the genome at a CRISPR locus. Once this CRISPR is transcribed as RNA it interacts with proteins which target and inactivate the viral DNA.

In chapter 8 we model simple repeats and CRISPRs using a probabilistic variant of extended regular expressions. By training on existing CRISPRs, a probabilistic extended regular expression model is shown to be capable of predicting repeats which correspond to those indicated by minimal free energy models.

3.3.2 Integration of signals in an experimental prokaryotic gene finder

In chapter 9 we introduce a gene finder for prokaryotes to demonstrate the principle of Bayesian Annotation Networks. Different gene finder architectures and combinations of signals are evaluated. The signals considered are

- a Hidden Markov Model which reflects preferential codon usage in genes,
- a model which incorporates length modeling of genes,
- a conservation model which reflects conservation of genes in related organisms.

These different models are combined in various topologies and evaluated by first training on a random fraction of the genes in *E. Coli* and then predicting other genes in the *E. Coli* genome. Results are assessed in a systematic manner, which allows the evaluation of the impact of the addition of individual signals.

The evaluated gene finders should be regarded as experimental, as they are merely meant to illustrate and evaluate the method of Bayesian Annotation Networks. The obtained results are not on par with state-of-the-art gene finders. The models themselves incorporate useful signals, but it turns out to be difficult to utilize these fully in a combined model using the Bayesian Annotation Network principle.

The gene finder components are implemented as modules in an precursor of BANpipe (chapter 10).

3.3.3 Constraints and Global Optimization for Gene Prediction Overlap Resolution

In this paper, we apply constraints and global optimization to the problem of restricting overlapping of gene predictions for prokaryotic genomes. We investigate existing heuristic methods and show how they may be expressed using Constraint Handling Rules. Furthermore, we integrate existing methods in a global optimization procedure expressed as probabilistic model in the PRISM language. The approach yields an optimal (highest scoring) subset of predictions that satisfy the constraints. Experimental results indicate accuracy comparable to existing heuristic approaches.

3.3.4 A Probabilistic Genome-Wide Reading Frame Sequence Model

In chapter 13 we introduce a probabilistic model, which reflect the sequential composition of reading frames of genes in a genome. A reading frame sequence bias is a general signal that can incorporate gene strand bias, bias due to operonic structures and other potential effects yielding non-random sequence composition. We believe that this is a novel signal to be used in gene finding.

The model is a kind of "delete" HMM (our term) which is used to prune predictions which exhibit unlikely reading frames according to observed reading frame bias in a genome. The predictions are produced by existing state-of-the-art gene finders and ideally, pruned predictions are the false positives of the gene finders.

We demonstrate that our pruning approach can lead to improved specificity with minimal impact to sensitivity. The improvement in accuracy is robust to low quality in the quality of training data and the method works with different underlying gene finders. The reading frame sequence signal is also robust across species – even when the model is trained on distant genomes, improvements in accuracy are achieved.

3.3.5 A gene finder for pyrrolysine incorporating genes

Pyrrolysine is a rare amino acid known as the 22nd amino acid. It is encoded by a codon which is usually stop codon. The mechanisms leading to the translation of stop codons as pyrrolysine are largely unknown.

In chapter 14, we propose an informative method for prediction of pyrrolysine incorporating genes in genomes of bacteria and archaea. Our method clusters open reading frames with possibly pyrrolysine encoding codons based on sequence similarity and rank these clusters according to several features which may influence pyrrolysine translation. The ranking effects of different features are assessed and we propose a weighted combination of these features which best explains the currently known pyrrolysine incorporating genes. From the weighted ranking, we identify a number of potentially pyrrolysine incorporating genes.

Furthermore, we investigate possible factors driving pyrrolysine translation. In particular, the effect of structural conservation is explored. By predicting structures of existing pyrrolysine incorporating genes and by clustering them with regard to the similarity of possible PYLIS structures, we assess the relevance of structure with regard to pyrrolysine translation.

3.4 How chapters of the dissertation contribute to the research goal

The following details how the paper chapters included in this thesis contribute to the research goal.

Chapter 4: The zoo of Hidden Markov models demonstrates the expressiveness of PRISM and the feasibility of expressing various types of Hidden Markov Models which are commonly used in biological sequence analysis. It also touches upon possible ways of integrating constraints, *e.g.*, copying substrings in a pseudo-knot HMM.

Chapter 5: Constrained Hidden Markov models demonstrate that extending Hidden Markov Models with side-constraints can be useful as a means to express constraints from the domain of biology, *e.g.*, the maximal extent of insertion/deletion in pairwise alignment. All the models in the paper are from the domain biological sequence analysis. Furthermore, it is demonstrated that the addition of such constraints make otherwise time-consuming inferences feasible.

From this perspective, the paper addresses all of my three research questions. It demonstrates the use of probabilistic logic programming to create models for biological sequence analysis, it introduces a language for constraints in these models and finally it shows that the introduction of constraints can be a way to deal with the limitation of inefficient inference with these models. This is made possible by introducing an optimization of probabilistic logic programming which avoids tabling of constraint stores.

Chapter 6: This paper demonstrates the combination of probabilistic logic programming extended with constraints expressed using Constraint Handling Rules. This is used to express a model which is extended with constraints about overlapping genes. This model is used to find a optimal (highest-scoring) set of gene finder predictions with respect to the expressed constraints. It hence contributes to my research goal both by demonstrating of the feasibility of using probabilistic logic programming for biological sequence analysis and by making a case for the introduction of (biological) constraints in such models. It also contains an optimization to prune a (tabled) constraint store in order to deal with the limitation of infeasible inference.

Chapter 7: This paper makes a case for expressing Hidden Markov models using Constraint Handling Rules. The paper demonstrates an intuitive Viterbi-like algorithm for HMMs written in CHR and develops optimizations necessary for its efficient execution. The paper exposes a relation between constraint logic programming and probabilistic logic programming. It is a first step towards demonstrating that constraints may form the core of probabilistic logic models and related inference algorithms. It is the only paper in the thesis which is not based on the PRISM system.

Chapter 8: Probabilistic extended regular expressions is an abstraction which allow the modeling of certain types of grammatical constraints. These constraints are demonstrated to be useful in the modeling of different types of repeats in biological sequences.

Chapter 9: Bayesian Annotation Networks are used to deal with a central limitation — the infeasibility of inference (both training and prediction) — of complex biological sequence analysis. This is done decomposing the analysis task into separate models for individual aspects of the sequence and combining the results from these in a structured way. The resulting Bayesian Annotation Network model is an approximation of a hypothetical joint model, but achieves reasonable results as demonstrated on the gene finding task.

Chapter 10: BANpipe is a pipeline scripting language design to support the Bayesian Annotation Network paradigm. It contributes to the goal of making probabilistic logic programming more usable for biological sequence analysis in several ways. First, it provides a means to support bioinformatics workflows with probabilistic logic programming components. Second, it deals with limitations due time-consuming analyses by exploiting to the notion of conditional independence in Bayesian Annotation Networks to provide automatic parallelization of execution when it is thus possible.

Chapter 11: This paper deals with the problem of tabling structured data. This are central to to efficient inference of probabilistic logic programming and the problem leads to non-optimal time and space complexity of inference in the PRISM system (among others). A program transformation is developed to deal with the problem. The result is constant complexity of table lookups. The paper contributes to the research goal by dealing with a central limitation of probabilistic logic programming.

Chapter 12: In this paper, the problem of tabling structured data is dealt with in an low-level implementation in the tabling system of B-Prolog. An new technique called Enhanced Hash-Consing is used for the optimization. This technique does not only guarantee constant complexity, but also enables structure sharing in tabled goals. As the previous chapter, this contributes to the research goal by addressing a central limitation of probabilistic logic programming.

Chapter 13: This chapter introduces a probabilistic model which demonstrates that there is a non-random signal in the sequence of reading frames of genes which can be exploited in gene finding. It contributes to the research goal by demonstrating an application probabilistic logic programming which manage to discover new biological knowledge.

Chapter 14: The chapter demonstrates an application of probabilistic logic programming to disseminate how various features may contribute to the translation of pyrrolysine-incorporating genes. Furthermore, this dissemination is used to build a gene finder pipeline that is used to predict novel pyrrolysine-incorporating genes. It contributes to the research question, by the demonstrating the usefulness of probabilistic logic programming to a real biological problem.

3.5 My contribution to papers in the dissertation

This section more specifically outlines my contributions to the articles included in this thesis.

Chapter 4: Taming the Zoo of Discrete HMM Subspecies and Some of Their Relatives This work was inspired by several activities in the group. I drafted only a few sections of the paper, but participated in revising and proofreading the manuscript.

Chapter 5: Inference with Constrained Hidden Markov Models in PRISM I wrote the implementation, performed the experiments and drafted the sections on these topics. I participated in devising the constraint model and the more theoretical aspects of the paper, but these contributions are mainly due to my coauthors.

Chapter 6: Constraints and Global Optimization for Gene Prediction Overlap Resolution I was solely responsible for this article. Much of the inspiration for the problem and motivating the approach, however, came from discussions in the LoSt group. The approach draws inspiration from Constrained Hidden Markov Models.

Chapter 7: The Viterbi Algorithm expressed in Constraint Handling Rules Henning Christiansen came up with the idea, wrote the implementation and the initial draft of the manuscript. I participated in discussions and revisions of the manuscript.

Chapter 8: Modeling repeats in DNA using Extended Probabilistic Regular Expressions I implemented the system, performed the experiments and drafted the paper, which was revised in cooperation with my coauthor.

Chapter 9: Bayesian Annotation Networks for Complex Sequence Analysis I am responsible for a large part of the implementation of the system and the experiments. I participated in discussions and in the writing of the manuscript.

Chapter 10: A Declarative Pipeline Language for Big Data Analysis I am the principal designer the language and is responsible for its implementation. I drafted the initial manuscript, which was significantly revised in cooperation with my coauthors to in order to cover the formal semantics.

Chapter 11: Efficient Tabling of Structured Data using Indexing and Program Transformation I conceived the idea and implemented the program transformation and experiments described in the paper. I wrote the first draft of the paper, which was revised in cooperation with my coauthor.

Chapter 12: Efficient Tabling of Structured Data with Enhanced Hash-Consing The idea developed through discussions with my coauthor at ICLP 2011 and PADL 2012. My coauthor wrote the implementation and wrote the first draft of the paper. I ran the benchmarks and drafted a first version of the corresponding section. I participated in the proof-reading of the manuscript.

Chapter 13: A Probabilistic Genome-Wide Gene Reading Frame Sequence Model I implemented major parts of the system. Experiments and manuscript writing was done in cooperation and with equal contributions from my coauthor.

Chapter 14: Effects of using Coding Potential, Sequence Conservation and mRNA Structure Conservation for Predicting Pyrrolysine Containing Genes I designed the pipeline in cooperation with my coauthors. I wrote the implementation of the pipeline, ran the most of experiments and actively participated in the data analysis and in the writing of the manuscript.

Chapter 4

Taming the Zoo of Discrete HMM Subspecies & Some of Their Relatives

With Henning Christiansen, Ole Torp Lassen and Matthieu Petit

In Biology, Computation and Linguistics, New Interdisciplinary Paradigms, volume 228 of Frontiers in Artificial Intelligence and Applications, pages 28-42. IOS Press, 2011.

Abstract

Hidden Markov Models, or HMMs, are a family of probabilistic models used for describing and analyzing sequential phenomena such as written and spoken text, biological sequences and sensor data from monitoring of hospital patients and industrial plants. An inherent characteristic of all HMM subspecies is their control by some sort of probabilistic, finite state machine, but which may differ in the detailed structure and specific sorts of conditional probabilities. In the literature, however, the different HMM subspecies tend to be described as separate kingdoms with their entrails and inference methods defined from scratch in each particular case. Here we suggest a unified characterization using a generic, probabilistic-logic framework and generic inference methods, which also promote experiments with new hybrids and mutations. This may even involve context dependencies that traditionally are considered beyond reach of HMMs.

4.1 Introduction

Discrete[1] Hidden Markov Models (HMMs) have become a very popular tool used for the modeling and analysis of a variety of sequential phenomena, e.g., biological sequence data, shallow text analysis and speech recognition; see, e.g., [61, 104] for overview and background. A HMM is a probabilistic finite state machine that produces sequences of symbols from some finite alphabet.

[1]We ignore continuous HMMs completely as they do not fit into the probabilistic-logic framework that we rely on in this paper; see, e.g., [159] for a definition.

In a typical application, the state machine is intended as a (simplified) reconstruction of some sort of system, whose internal details typically cannot be inspected, and the sequence of emission symbols represents the observable behaviour of that system. For DNA analysis, for example, we may consider an actual genome sequence as being produced by an informed typist who, in a probabilistic way, decides when to enter states that corresponds to typing, say promoter sub-sequences, operons, genes etc., or intergenic subsequences whose contents is debated.

One of the advantages of HMMs is that analysis of a sequence can take place in linear time as a function of the sequence length. The price, however, is a lack of sophistication, as the underlying sequence languages are regular [34]. Basically, the only memory available at a given time in the run of a state machine is knowledge about the current state; in its simplest form, a HMM has a probability table for each state to govern transition to the next state and a table for the emission. Subspecies of HMMs may differ by having the probabilities conditioned by one or more previously passed states and-or emissions; in this case we talk about higher-order HMMs. Other distinctive features can be different ways of structuring the state machine, e.g., by product or hierarchical structures of separate machines and other ways that are considered below.

The following three ways of reasoning with HMMs are essential for practical applications:

- Analysis of a sequence S, also called prediction, means to identify the most probable path of states that the machine may pass through in order to produce S; prediction is typically made using some adaptation of the Viterbi algorithm [218], which is a dynamic programming algorithm that reads the sequence symbol by symbol and keeps track of one best path up to each possible state. Its time complexity is known to be $\mathcal{O}(n\,b^2)$ where n is the length of the sequence and b the number of machine states.

- The probabilities which, thus, determine which path is preferred, can be set manually but more interestingly produced by training from known sequences using machine learning techniques, e.g., the EM algorithm [58], that we shall not describe further.

- Finally, sampling is a process of using the HMM to produce sequences that are representative for the distribution defined by the HMM.

In the present paper, we propose a uniform treatment of different subspecies of discrete HMMs, showing how they can be defined in very concise ways using state-of-the-art probabilistic-logic programming as represented by the PRISM system [179, 178]. In addition, we obtain for free, implementations of prediction, training, and sampling, relying on PRISM's generic built-in facilities. We will also show examples of models that go beyond what can be considered as HMMs in a strict sense, such as probabilistic context-free grammars and definite clause grammars as well as context-sensitive extensions to HMMs, can be treated in similar ways. Our reasons for writing this paper can be summarized as follows.

- It can be a bit of a nuisance trying to get an overview of the literature in the field, as there seems to be a trend that each paper defines from scratch its own probability calculations, adaptations of Viterbi and learning algorithms, and implementation principles – and it is left to the reader to figure out that all these anyhow are instances of a common pattern. We hope that the present exposition may contribute to a better overview.

- We want to relieve the researcher who wants to apply, design or experiment with alternative sequential models from wasting valuable time on tiresome programming in imperative languages and on learning the idiosyncrasies of different, specialized software, anyhow doing more or less the same thing.

- We want to emphasize the advantages of using probabilistic-logic programming in teaching, as a uniform theoretical treatment and implementation framework for these inherently related models.

As an example of how this unification can be exploited, it is fairly straightforward to set up a testbed for comparing how well different models perform with respect to precision and recall for the same collections of training and validation as done, e.g., by [137].

We should emphasize that the principle of implementing plain HMMs in PRISM is not our invention but has been used as an example, e.g., in the PRISM User's manual [178]. Our formulation below may be a bit more elegant for the plain HMM case, as this has been a particular goal for us, and most of the variations that we unfold have not been investigated systematically before in PRISM.

We expect a basic knowledge of probability theory and elementary Prolog programming. For reasons of space, we do give all details of all our models, but only highlight the essential fragments; the website [37] lists all models in full text and explains how they can be executed efficiently.

Below, in section 4.2, we give as background a standard definition of HMMs and their probabilities. Section 4.3 provides a compact introduction to PRISM and shows how a plain HMM can be defined in a very concise way together with minor extensions with silent states and duration. The following sections compose a stroll through the HMM Zoo Garden, showing different subspecies properly kept in order by definitions in PRISM; sec. 4.4 shows how transitions and emissions may be conditioned by events in the past, so-called higher-order HMMs; sec. 4.5 shows HMMs that concern several sequences at a time, e.g., for alignment, and different ways that multiple HMMs can form symbioses; sec. 4.6 goes a step further showing how context-dependencies beyond regular and even context-free languages can be characterized as direct extensions of plain HMMs. Finally, section 4.7 mentions other sequence models that can be described in PRISM, but not treated in details here, such as probabilistic versions of context-free and definite clause grammars; sec. 4.8 mentions a few related works, and sec. 4.9 gives a brief summary and conclusion.

4.2 Definition of a Common Ancestor: 1st Order HMMs

We define here the most primitive subspecies of Hidden Markov models, in which transition and emissions probabilities are conditioned by the current state only. For simplicity of the definition, we assume one fixed initial state.

Definition 4.2.1. *A* Hidden Markov Model *(HMM) is a quadruple $\langle \Sigma, A, T, E \rangle$, where*

- *Σ is a finite set of* states, *one of which is distinguished as* initial *and one or more as* final *states, and A a finite set of symbols called the* emission alphabet*;*
- *T is a set of* transition probabilities *$\{p(s_i; s_j)\}$ for pairs of states, with s_i not final, such that for each s_i, $\sum_{s_j} p(s_i; s_j) = 1$;*
- *E is a set of* emission probabilities *$\{p(s_i; e_j)\}$ for pairs of state, with s_i not final, and letters $e_j \in A$ such that for each s_i, $\sum_{e_j} p(s_i; e_j) = 1$.*

A run *for a given HMM is defined is as a pair of sequences of states $s_0 \cdots s_{n+1}$ and letters $e_0 \cdots e_n$ where s_0 (s_{n+1}) is an initial (final) state, $p(s_i; s_{i+1}) > 0$ and $p(s_i; e_i) > 0$; $s_0 \cdots s_{n+1}$ is called a* path *for $e_0 \cdots e_n$.*

The probability *of a given run R is defined as follows, using the notation above.*

$$P(R) = \prod_{i=0}^{n} p(s_i; s_{i+1}) \prod_{i=0}^{n} p(s_i; e_i)$$

Notice that the set of non-zero transition probabilities defines the underlying finite state machine of a specific HMM. For an overview of inference methods for such HMMs, including prediction and training, see, e.g., [61].

In the next section we introduce the generic probabilistic-logic framework PRISM and show how a definition of standard HMMs in PRISM is a realization of the semantics given by def. 4.2.1. For the remaining part of this paper, we will take a specification in the PRISM language as *the* definition of a subspecies.

4.3 Probabilistic-Logic Modelling in PRISM

The PRISM language [179, 178] extends Prolog with so-called multi-valued switches: a call `msw(`*name*`, X)` represents a probabilistic choice of value to be assigned to `X`. A switch is introduced by a declaration of the form

`values(`*name*`, [`$\cdots$ *outcomes* $\cdots$`])`

and defines a family of random variables, one for each execution of `msw(`*name*`,` $\cdots$`)` in a program run. The name can be parametric — as we will show in the example below — in a way that makes it possible to define conditional probabilities.

The probability table associated with an `msw` can be defined by an explicit declaration or be produced by learning as we show below. The overall semantics of a PRISM program is given as a probabilistic Herbrand model, in which any logical atom has a probability of being true, given as the product of the probabilities for the `msw`s applied in a proof tree for that atom. For this semantics to be well-defined, any choice point in the program must be governed by an `msw`.

We use the example below both to show the implementation of a HMM in accordance with def. 4.2.1 and to explain further details about the PRISM system.

4.3.1 Lesson 1: First-Order HMMs in PRISM

Let us consider a play of heads-and-tails in which the cashier has two coins, an honest and a dishonest one. At each time step, he will throw the current coin and decide which next step to take, i.e., to pick one of the coins or close the game. This may be described as a HMM whose states are,

$\{$`honest`, `dishonest`, `close`$\}$

where the initial one is `honest` and the final one `close`; the emissions are $\{$`head`, `tail`$\}$. We can encode this HMM in the following `msw` declarations and probability settings, written as they may appear in a PRISM source file.

```
values(trans(_CurrentState),[honest,dishonest,close]).

values(emit(_CurrentState), [head,tail]).

:- set_sw(trans(honest), [0.4,0.5,0.1]),
   set_sw(trans(dishonest), [0.2,0.7,0.1]),
   set_sw(emit(honest),[0.5,0.5]),
   set_sw(emit(dishonest),[0.7,0.3]).
```

Notice in the `values` declarations, that the names are parameterized by a variable (starting with an underscore) meaning that for whatever term is substituted for that variable, there exists an `msw`. The `set_sw` directives set up distributions for the instances of `trans(_)` that appear in program below, i.e., `trans(honest)` and `trans(dishonest)`, and emission probabilities for the two non-final states. Initial and final states can be defined in terms of Prolog predicates as follows.

```
initial(honest).
final(close).
```

As it may appear, the format shown so far can be applied to encode any individual HMM according to def. 4.2.1. The following lines of PRISM code provide a general definition of a HMM encoded in this way; i.e., if you want to work with another HMM, you just need to change the `values` and `set_sw` declarations. It is tail recursive and it can be understood as a generative specification of all possible runs and their probabilities.

```
hmm(Sequence,Path):- initial(S0), hmm(Sequence,S0,Path).

hmm([],Final,[Final]):- final(Final).

hmm([A|As],S,[S|Ss]):-
   \+final(S),
   msw(emit(S),A),
   msw(trans(S),Snext),
   hmm(As,Snext,Ss).
```

The top predicate `hmm/2` defines a probability distribution for the runs of this HMM, so for example $P($`[head,head]`, `[honest,dishonest,close]`$) = 0.5 \times 0.5 \times 0.1 \times 0.7 = 0.0175$, formed as the product of `msw` probabilities used for generating this particular run. In fact, `[honest,dishonest,close]` is the Viterbi path for `[head,head]`.

The PRISM system provides the necessary tools for inference.

Sampling is the simplest; the program is executed as a plain Prolog program with the outcome of each `msw` determined by pseudorandom numbers. Time complexity is linear in the length of the generated sample.

Prediction can be made using one of PRISM's built-in Viterbi algorithms. The call `viterbig(hmm([head,head],Path))` will instantiate the logical variable `Path` to the path that provides the maximum probability for the call. Time complexity depends on the details of the model; HMMs can run in linear time.[2]

Training: PRISM comes with several built-in algorithms for supervised and unsupervised learning; supervised learning may, e.g., be performed from a large file of ground atoms such as `hmm([head,head], [honest,dishonest, close])` and this will replace the explicit `set_sw` directives shown above; we refer to the PRISM manual for details [178]. Time complexity depends in a non-trivial way on the complexity of the program and the size of the training data. All examples shown in the present paper can be trained in reasonable time from realistic training data.

The main advantage of using a system such as PRISM is that one and the same specification, such as the few program lines shown above, implements the different reasonings in one go.

4.3.2 Minor Variations: Silent States, Duration Modeling

Here we indicate a few local varieties of the basic HMM species that are often encountered in the literature. A *silent state* is one that does not produce any emission symbol. Let us change the heads-and-tails model a little, so it also takes into account that the cashier, when he is using the honest coin may decide to ignore an outcome (that does not fit his interests). This is done as follows, using an additional outcome of the `msw` for `emit(honest)` and modify the code a bit so the "interpretation" of `silent` is to add nothing to the emission sequence.[3] We have used here Prolog's conditional expression *test* `->` *if-true-action* `;` *if-false-action*.

```
values(emit(dishonest), [head,tail]).
values(emit(honest), [head,tail,silent]).
...
hmm(As1,S,[S|Ss]):-
   \+ final(S),
   msw(emit(S),A),
   msw(trans(S),Snext),
   (A=silent -> As1=As ; As1 = [A|As]),
   hmm(As,Snext,Ss).
```

Duration in a HMM means that it can stay for a number of steps in a given state before going to a next state, and with a certain distribution for this number. It is well-known that basic HMMs have problems modeling duration[4] and the literature is rich in contortions to the layout of the state machine to compensate for this using duplicated states, etc.; see, e.g., [61]. In our framework, we can model duration in the most natural way, namely by selecting a number in a probabilistic way, and iterate an

[2] A program transformation technique described in [40] is needed to have this Viterbi computation for a HMM run in linear time; this is explained at our website [37] that contains all program examples shown in the present paper. A forthcoming release of PRISM is expected to incorporate this technique, cf. [228], so we ignore this issue for the remainder of this paper.

[3] The current version of PRISM loops when using the silent state programs for prediction due to the existence of infinite paths. A new, generalized Viterbi algorithm for PRISM is under development which avoids this problem. The website for this paper [37] provides a minor modification of the program that works also under the present system.

[4] Using a possible transition from a state to itself in a plain HMM gives a geometric distribution of the possible durations.

emission from the given state that number of times. For the heads-and-tails example, we may like to model that the cashier uses a chosen coin about 6 times in a row with a little variation, as follows. Notice that we need to prevent transition from a state to itself after the loop.

```
values(duration,[4,5,6,7,8]).
values(trans(honest),[dishonest,close]).
values(trans(dishonest),[honest,close]).
...
:- ... set_sw(duration,[0.15, 0.2, 0.3, 0.2, 0.15]).
...

hmm(As,S,Ss):-
   \+final(S),
   msw(duration,T),
   iterateHmm(T,As,S,Ss).

iterateHmm(0,As,S,Ss):-
   msw(trans(S), Snext),
   hmm(As,Snext,Ss).

iterateHmm(T,[A|As],S,[S|Ss]):-
   T > 0, T1 is T-1,
   msw(emit(S),A),
   iterateHmm(T1,As,S,Ss).
```

More sophisticated patterns can be defined with different length distributions for different states.

4.4 Single Sequence HMMs with Dependencies on a Portion of the Past

The term *higher-order HMM* refers traditionally to different varieties of a subspecies that extends the conditioning of the transition and-or emission probabilities with information about the past.

The basic HMM definition shown in section 4.3.1 can easily mutate into higher-order, adding extra arguments to the `hmm` predicate and extra degrees of freedom in the `msw` definitions. We illustrate this for a so-called 2nd order HMM in which the transition probabilities are now conditioned, not only by current state, but also the previous; the emissions are unchanged in this example but can also be conditioned in a similar way. The symbol `border` imitates a dummy state before the initial state.

```
hmm(Sequence,Path):- initial(S0), hmm(Sequence,S0,border,Path).

hmm([],Final,_,[Final]):- final(Final).

hmm([A|As],S,Sprevious,[S|Ss]):-
   \+final(S),
   msw(emit(S),A),
   msw(trans(S,Sprevious),Snext),
   hmm(As,Snext,S,Ss).
```

When, e.g., the HMM goes through a path $s_0, s_1, \ldots$, the sequence of calls to the recursive `hmm` predicate can be sketched as follows.

$$\texttt{hmm(}\cdots s_0\texttt{,border}\cdots\texttt{)},\texttt{hmm(}\cdots s_1\texttt{,}s_0\cdots\texttt{)},\texttt{hmm(}\cdots s_2\texttt{,}s_1\cdots\texttt{)},\ \cdots$$

While the 1st order is suited to be trained to learn combinations of two emissions letters in a row, an nth order is suited to learn patterns of $n + 1$ letters.

We expect the general principle to be clear by now, so that the reader may experiment with his or her own varieties of higher order HMMs that utilizes different portions of the past in various ways.

4.5 Multi-Sequence and Combined HMMs

4.5.1 HMMs for Sequence Alignment

A HMM can be used to align two or more sequences. This application of HMMs have been particularly successful in computational biology, where an alignment of biological sequences can be used to draw conclusions about the evolutionary process. In the following we sketch a pair HMM [212] (see [61] for a presentation which is easier to compare with other literature in the field), which aligns two sequences, A and B, by simultaneously emitting them. Besides the silent `end` state, the pair HMM has three states; the `match` state aligns the next two symbols from each sequence to each other, the `insert` state aligns the current symbol of sequence A to a gap in sequence B and the `delete` state aligns the current symbol of sequence B to a gap in sequence A. The model is not fully connected as it is not possible to visit the `delete` state immediately after the `insert` state and vice versa.

```
values(trans(match),[match,delete,insert,end]).
values(trans(delete),[match,delete,end]).
values(trans(insert),[match,insert,end]).
initial(match).
final(end).
```

We consider here emissions over the alphabet $\{\texttt{a}, \texttt{c}, \texttt{g}, \texttt{t}\}$. For simplicity, we characterize emissions in a uniform way as pairs, with "-" representing a missing character, e.i., `pair(a,a)` is a typical emission from a match state and `pair(a,-)` a typical emission from an `insert` state. In the `set_sw` directives, we set the probability of `emit(`x`,`y`)`, $x \neq y$, in the `match` state to almost zero, and the probability of `emit(`x`,`y`)`, $y \neq$ "-" to exact zero for `insert`, and analogously for the `delete` state. We simplify the main predicate using an auxiliary programmed in Prolog.

```
first(-,Ls,Ls):- !.
first(L,Ls,[L|Ls]).
```

The recursive specification of the pair HMM is now straightforward as follows.

```
hmm(Seq1,Seq2,Path):- initial(S0), hmm(Seq1,Seq2,S0,Path).

hmm([],[],Final,[Final]):- final(Final).

hmm(As1,Bs1,S,[S|Ss]):-
   \+ final(S),
   msw(emit(S),pair(A,B)),
   first(A,As,As1), first(B,Bs,Bs1),
   msw(trans(S),Snext),
   hmm(As,Bs,Snext,Ss).
```

4.5.2 Hierarchical HMMs

A hierarchical HMM [67] is similar to a 1st order HMM, but instead of emitting probabilistically a letter from each state, a sub-model is selected. Each sub-model is an ordinary HMM, which produces a sequence of letters until the control is given back to the top level. Thus a string produced by a hierarchical HMM is a concatenation of strings produced by different sub-models. The following structure assumes that the states in the different sub-models form disjoint subgraphs; in the choice among sub-models, their initial states serve also the purpose of identifying them. Notice that the `subHmm` predicate carries a state for the top level HMM which is "jumped to" when a sub-model reaches its final state.

```
hmm(Sequence,Path):- initial(S0), hmm(Sequence,S0,Path).

hmm([],Final,[Final]):- top_final(Final).

hmm(As,TopS,[TopS|Ss]):-
   \+ top_final(TopS),
   msw(submodel(TopS),SubInitS),
   msw(trans(TopS), TopSnext),
   subHmm(TopSnext,As,SubInitS,Ss).

subHmm(TopS,As,S,Ss):-
   sub_final(S),
   hmm(As,TopS,Ss).

subHmm(TopS,[A|As],S,[S|Ss]):-
   \+ sub_final(S),
   msw(emit(S),A),
   msw(trans(S),Snext),
   subHmm(TopS,As,Snext,Ss).
```

The `subHMM` predicate is similar to the basic HMM definition of in section 4.3.1 with the only difference that when it reaches its final state, it makes a recursive call to the `topHMM` state referred to by the variable `TopS`, rather than stopping.

A nontrivial application of hierarchical HMMs defined in PRISM for testing gene-finders has been made by [38].

4.5.3 Factorial HMMs

This term, introduced by [79], refers to a subspecies whose finite state machine is defined as the product of two or more state machines. A factorial HMM can be mapped into a plain HMM (whose state set is the product of the individual state sets), but it has the advantages that there are fewer transitions and probabilities that need to be specified: if two sub-machines has n, resp., m states, the factorial HMM includes at most $n^2 + m^2$ different transition probabilities, compared with the corresponding plain HMM that needs up to $n^2 \times m^2$ transition probabilities. The emissions are conditioned by the states of both sub-machines. This is straightforward to incorporate into a PRISM specification, we just need to pass two state variables around and determine the new state for both sub-machines in each step.

```
hmm(Sequence,Path):-
  initial1(S10), initial2(S20), hmm(Sequence,S10,S20,Path).

hmm([],Final1,Final2,[(Final1,Final2)]):-
    final1(Final1) ; final2(Final2).

hmm([A|As],S1,S2,[(S1,S2)|Ss]):-
   \+final1(S1), \+final2(S2),
    msw(emit(S1,S2),A),
    msw(trans1(S1),S1next),
    msw(trans2(S2),S2next),
    hmm(As,S1next,S2next,Ss).
```

4.5.4 Bayesian Coupled HMMs

Recent work on sequence analysis using PRISM for biological sequence analysis [43] shows how PRISM models can be put together in a Bayesian network, such that the resulting analysis from one model is used for conditioning the probabilities of other models. This applies also to the special sort of PRISM models that are HMMs and gives rise to yet another highly specialized HMM subspecies (symbiosis may be a better term) called Bayesian Coupled HMMs. As an example, we consider a system for weather analysis, put together as a network with only two nodes. We assume sequences of observations being either `sun`, `rain`, or `snow`. This model composes two HMMs in the following way.

- `hmm1` is a plain HMM whose states represent temperature plus an end state {`minus`, `aboutzero`, `plus`, `end`};
- `hmm2` has states that represent wind speed plus an end state {`quiet`, `breeze`, `storm`, `end`}; it extends the HMMs that we have seen so far by having the transitions conditioned also by the states produced by another HMM, in this example `hmm1`.

The PRISM code for such a conditioned HMM is as follows.

```
hmm2(Sequence,Condition,Path):-
    initial2(S0),
    hmm2(Sequence,Condition,S0,Path).

hmm2([],_,Final,[Final]):- final2(Final).

hmm2([A|As],[C|Cs],S,[S|Ss]):-
   \+final2(S),
   msw(emit2(S),A),
   msw(trans2(S,C),Snext),
   hmm2(As,Cs,Snext,Ss).
```

The implementation described by [43], which is built on top of PRISM, applies a principle for prediction that runs prediction with PRISM's Viterbi algorithm for each model at time, fixing the path produced before sending it on to the subsequent models. For the example above, we can illustrate this by the following query.

```
?- Seq=[snow,sun], viterbig(hmm1(Seq,P1)),
                   viterbig(hmm2(Seq,P1,P2)).
...
P1 = [aboutzero,aboutzero,end]
P2 = [breeze,quiet,end]
```

The first model, `hmm1`, predicts a path given as `P1`, which then is given to the prediction with `hmm2`; notice that `hmm2` only makes sense as a probabilistic model when this argument is given as a ground list.

Putting different PRISM models together as Bayesian networks in this way yields both a way of decomposing complex models and a way to reduce computational complexity. Bayesian Coupled HMMs can be formed from any number of sub-models and conditioned in a variety of ways, including taking in signals produced by external, not necessarily probabilistic analyses; see the referenced paper for details.

4.6 Adding Context-Sensitive Information

The use of Prolog as a backbone to tie together the different probabilistic choices in our models makes it fairly straightforward to express also some context-sensitive constraints: at any point in a sequence, information can be collected, passed further on via predicate arguments to any other point in the string and applied to restrict, or condition probabilistically, what can occur at that second point. We show here two examples.

4.6.1 Copying Substrings: Pseudoknots

A pseudoknot is a phenomenon that may be observed in the tertiary structure of an RNA molecule. It can be explained syntactically as a pattern in which a certain subsequence, that we call the *glue zone*, appears later with inverted letters, but in the same order. Inversion means to interchange any '`a`' with a '`t`' and any '`c`' with a '`g`' and vice versa. The following is an example of such a pattern; the interesting subsequence and its repeated, inverted version are indicated by boxes.

$$\texttt{aaaaaaaaaaaaaa}\boxed{\texttt{agata}}\texttt{aaaaaa}\boxed{\texttt{tctat}}\texttt{aaaaa}$$

As is well-known from formal language theory, this language exceeds both regular and context-free languages; it is a context-sensitive language. We can describe this as a combination of HMMs and a deterministic predicate that inserts the copy string. In order to keep the predicate arguments simple, we use separate predicates for each indicated substring; the model is put together by the following predicates that are applied in the given order.

`hmm1`, `hmmGlue`, `hmm2`, `copyGlue`, `hmm3`

The first predicate is like a usual 1st order HMM, with the exceptions that when it gets to its final states, it continues with `hmmGlue` instead of stopping.

```
hmm1(As,SFinal,Path):-
   final(SFinal),
   initialGlue(SGlue0),
   hmmGlue(As,SGlue0,Path,[]).

hmm1([A|As],S,[S|Path]):- \+ final(S), ...
```

The second, `hmmGlue`, is structured in the same way, except that it builds a separate list of the letters generated (in inverted form), as to have them ready for the copying later.

```
hmmGlue(As,SGlueFinal,Path,Glue):-
   finalGlue(SGlueFinal),
   initial(S0),
   hmm2(As,S0,Path,Glue).

hmmGlue([A|As],S,[S|Path],Glue):-
   \+ finalGlue(S),
   msw(emitGlue(S),A),
   msw(transGlue(S),Snext),
   addInvertedLetter(Glue,A,Glue1),
   hmmGlue(As,Snext,Path,Glue1).
```

The `addInvertedLetter` predicate adds the indicated letter in inverted form at the end of the currently collected copy string; it can be defined in Prolog as follows.

```
addInvertedLetter(Glue,A,Glue1):-
   invert(A,Ainvert),
   append(Glue,[Ainvert],Glue1).

invert(a,t). invert(t,a). invert(c,g). invert(g,c).
```

There is no reason to show the predicate `hmm2` as it is a mere replicate of `hmm1` except that it continues to `copyGlue`, defined as follows, when it has done its job.

```
 copyGlue(As,Path,[]):-
   initial(S0),
   hmm3(As,S0,Path).

copyGlue([A|As],[copy|Path],[A|Glue]):-
    copyGlue(As,Path,Glue).
```

Note that there are no `msw` calls here as the predicate is deterministic, with the looping controlled by the last arguments that contains the string to be inserted. Finally `hmm3` is a standard HMM. A more sophisticated version of this model could allow mutations in the inserted copy, using a model similar to the pair HMM used for alignment in section 4.5.1.

The website [37] gives the full text of this extended HMM as well as an alternative, and more elegant, version based on difference lists. We refrain from bringing the latter here, as we do not expect our average Zoo guest be familiar with the programming technique of difference lists. An earlier version of this pseudoknot program was given in [35].

4.6.2 Constrained HMMs

The subspecies of Constrained HMMs (CHMMs) are defined as ordinary HMMs with side-constraints on the runs. Such constraints are easily defined in PRISM; let `con(`*sequence*`,`*path*`)` be a predicate considered as constraint which accepts some runs and fails on others. When `hmm` represents an HMM, the following PRISM definition defines a CHMM.

```
chmm(Sequence,Path):- hmm(Sequence,Path), con(Sequence,Path).
```

CHMMs have the interesting property that the sum of probabilities of the runs it produce may be < 1, and the remaining probability mass considered as garbage probability [36]. The disadvantage of the definition just shown is that prediction may be very slow, as a breaking of the constraint is not seen before all `msw`s have been instantiated; furthermore, due to subtle technical reasons, PRISM's Viterbi algorithms cannot handle this sort of specification in a correct way in all cases. Constraints that can be checked incrementally are better suited for prediction as shown in the following example, and will actually work in the current PRISM system. We extend the head-and-tail HMM with the constraint that the dishonest coin may be used at most three times in a game; the new last argument of the recursive predicate counts the number of dishonest states encountered so far.

```
hmm(Sequence,Path):- initial(S0), hmm(Sequence,S0,Path,0).
...
hmm([A|As],S,[S|Ss], Count):-
   \+final(S),
   msw(emit(S),A),
   msw(trans(S),Snext),
   check_count(Snext,Count,Count1),
   hmm(As,Snext,Ss,Count1).

check_count(honest,Count,Count).
check_count(dishonest,3,_):- !, fail.
check_count(dishonest,Count,Count1):- Count1 is Count+1.
check_count(close,_,_).
```

The `check_count` written in plain Prolog takes care of the counting and fails when the count would exceed 3. Constrained HMMs are treated in depth in [41].

4.7 Other Related Species

Probabilistic context-free grammars are another known model for sequences that also can be described in PRISM; it is shown as an example in the PRISM User's Guide [178]. A probabilistic version of popular Prolog based Definite Clause Grammars based on PRISM has been demonstrated by [88]. Such models describe a given sequence using a tree-structured recursion, rather than the tail recursive style we have used here, and it is natural to use the technique of difference lists. In this respect, they are not natural descendants of HMMs but should be considered as their own species.

Notice that PRISM's language, being an extension of Prolog, is Turing-complete, which means that PRISM can describe probabilistic versions of any recursively enumerable language (nothing said here about the efficiency of reasoning).

4.8 Related Work

Dynamic Bayesian Networks (DBNs) [141] are directed graphical models for modeling sequential data. DBNs contain nodes for each time-slice, i.e., discrete time steps, and edges between such nodes signify conditional probabilities. DBNs contain individual variables for the nodes at each time step as opposed to the inductive definition of HMMs where the same random variables are reused. DBNs cannot handle context-dependencies as those we model in section 4.6.

There are various programming libraries available which support development of HMMs, e.g., [191]. Such libraries suffice for a variety of tasks, but are limited to express the kinds of models they were intended to.

PRISM is not the only language available that allows specification of HMMs. Several probabilistic-logic programming languages, which have identical power to PRISM, exist; see [55] for an overview. We have chosen PRISM because it is especially convenient for characterization of recursively defined structures (such as sequences) due to the Prolog backbone and its switches which are perfect for defining conditional probabilities of discrete (interior) events. PRISM provides a good balance between the ease by which HMMs can be expressed and the flexibility to model powerful HMM subspecies and contains generic inference algorithms for working with these models. A probabilistic version of regular expressions is described in [86] which also uses PRISM for its implementation.

4.9 Conclusion

There are countless subspecies of Hidden Markov Models and in this paper we have presented a few of them through probabilistic logic programs expressed in PRISM. PRISM provides a language which caters for experimentation with HMM species and the like and offers powerful inference algorithms that generalize to any model expressed in the language. A general theme should be clear from the examples; the different subspecies of HMMs can be represented with only minor variations of the program code.

Acknowledgement This work is supported by the project "Logic-statistic modelling and analysis of biological sequence data" funded by the NABIIT program under the Danish Strategic Research Council.

Chapter 5

Inference with Constrained Hidden Markov Models in PRISM

With Henning Christiansen, Ole Torp Lassen and Matthieu Petit

Slightly extended version of paper published in Theory and Practice of Logic Programming, 10(4-6), pages 449-464, 2010.

Abstract

A Hidden Markov Model (HMM) is a common statistical model which is widely used for analysis of biological sequence data and other sequential phenomena. In the present paper we show how HMMs can be extended with side-constraints and present constraint solving techniques for efficient inference. Defining HMMs with side-constraints in Constraint Logic Programming have advantages in terms of more compact expression and pruning opportunities during inference. We present a PRISM-based framework for extending HMMs with side-constraints and show how well-known constraints such as `cardinality` and `all_different` are integrated. We experimentally validate our approach on the biologically motivated problem of global pairwise alignment.

5.1 Introduction

Hidden Markov Models (HMMs) are one of the most popular models for analysis of sequential processes taking place in a random way, where "randomness" may also be an abstraction covering the fact that a detailed analytical model for the internal matters is unavailable. Such a sequential process can be observed from outside by its emission sequence (letters, sounds, measures of features, all kinds of signals) produced over time, and an HMM postulates a hypothesis about the internal machinery in terms of a finite state automaton equipped with probabilities for the different state transitions and single emissions. A common inference for a given observed sequence means to compute the "best" state transitions that the HMM may go through to produce the sequence, and thus this represents a best hypothesis for the internal structure or "content" of the sequence. HMMs are widely used in speech recognition and biological sequence analysis [159, 61].

The efficiency of computations on HMMs heavily depends on the Markov property. Decisions made during a process run depends only on a limited past. Dynamic programming algorithms, such as Viterbi and Forward-Backward, are then used to perform efficient inference. However, many problems would require more complex dependencies among elements of the process. For example, it may be interesting to constrain an HMM to visit only different states or limit the number of visits to a given state. It is possible to model the `all_different` constraint for the states visited by extending the underlying finite state automaton, but for the price of a factorial number of new states and with an obvious impact on inference. As an alternative to modifying the HMM structure, we extend instead the HMM with side-constraints [185, 168]. However, classical algorithms, such as Viterbi, must be modified to take care about these side-constraints [30, 42].

In this paper, we extend HMMs with side-constraints, leading to what we call Constrained HMMs (CHMMs). The concept of CHMMs was introduced by Sato et al. in [185], although earlier and unrelated systems have used the same or similar names (discussed in section 5.6). The contribution of this paper is to define CHMMs as constraint logic programs extended with probabilistic choices and to show how to employ this setting for more efficient Viterbi computation, i.e., computation of the most probable explanation of an observation. Moreover, defining HMMs with side-constraints in Constraint Logic Programming have advantages in terms of more compact expression and pruning opportunities during inference. We show how to implement CHMMs in PRISM [179] and how to integrate well-known constraints, such as `cardinality` and `all_different`, into this framework. We validate our approach experimentally on the biologically motivated problem of global pairwise alignment.

The paper is organized as follows: section 5.2 describes background on HMMs. In section 5.3, we formally introduce the constraint model associated with a CHMM. Section 5.4 describes our PRISM-based framework to define CHMMs. Section 5.5 presents an experimental validation. Finally, sections 5.6 and 5.7 present related work and conclusions.

5.2 Background

Here we define Hidden Markov Models (HMM)s and illustrate their application to the problem of pairwise global alignment.

5.2.1 Hidden Markov Models

For simplicity of the technical definitions, we limit ourselves to a discrete Hidden Markov Model with a distinguished initial state.

Definition 5.2.1. *A* Hidden Markov Model *(HMM) is a 4-tuple* $\langle S, A, T, E\rangle$*, where*

- $S = \{s_0, s_1, \ldots, s_m\}$ *is a set of* states *which includes an* initial *state referred to as* s_0*;*
- $A = \{e_1, e_2, \ldots, e_k\}$ *is a finite set of emission symbols;*
- $T = \{(p(s_0; s_1), \ldots, p(s_0; s_m)), \ldots, (p(s_m; s_1), \ldots, p(s_m; s_m))\}$ *is a set of* transition probability distributions *representing probabilities to transit from one state to another;*
- $E = \{(p(s_1; e_1), \ldots, p(s_1; e_k)), \ldots, (p(s_m; e_1), \ldots, p(s_m; e_k))\}$ *is a set of* emission probability distributions *representing probabilities to emit each symbol from one state.*

We define a run *of an HMM as a pair consisting of a sequence of states* $s^{(0)}s^{(1)}\ldots s^{(n)}$*, called a* path *and a corresponding sequence of emissions* $e^{(1)}\ldots e^{(n)}$*, called an* observation*, such that*

- $s^{(0)} = s_0$*;*

- $\forall i, 0 \leq i \leq n-1, p(s^{(i)}; s^{(i+1)}) > 0$ *(probability to transit from* $s^{(i)}$ *to* $s^{(i+1)}$*);*
- $\forall i, 0 < i \leq n, p(s^{(i)}; e^{(i)}) > 0$ *(probability to emit* $e^{(i)}$ *from* $s^{(i)}$*).*

The probability *of such a run is defined as* $\prod_{i=1..n} p(s^{(i-1)}; s^{(i)}) \cdot p(s^{(i)}; e^{(i)})$.

5.2.2 Example HMM 1: a simple gene finder

As a first example of an HMM that we later extend with constraints, we consider the problem of identifying protein coding genes in prokaryotes. A DNA sequence is composed of molecules, called *nucleotides*, represented by the four letters a, c, t and g. Some parts of a DNA sequence code for genes, called *coding regions*, while other parts do not and are called *non coding regions.* Coding regions contain a number of *codons*, triplets of nucleotides, each coding for an amino acid in a protein (to be produced by the gene). For prokaryotes, a coding region is contiguous, and it begins with a specific *start codon*, which is often atg, and ends with a *stop codon*, which is one of taa, tga or tag.

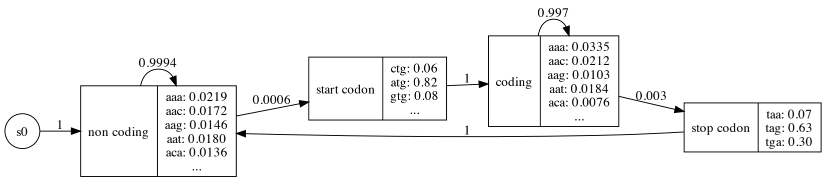


Figure 5.1: A simple HMM for Prokaryote genes prediction

Fig. 5.1 shows a simple HMM for prediction of genes; more advanced HMMs are used in successful gene finders that have been reported in the literature, e.g., Genemark.HMM [128] and EasyGene [123], and they can also be handled by our approach. The emission symbols of this HMM are codons, thus three letters form one symbol. It has four states: **start codon, non coding, coding, stop codon**. From **non coding**, any codon can be emitted. From state **start codon**, only start codons ata, atg, att, ctg, gtg and ctg can be emitted. From **coding**, any codon can be emitted except stop codons, taa, tga and tag. From state **stop codon**, only stop codons taa, tga or tag can be emitted. A consequence of the simplification of only emitting entire codons and not individual letters in this HMM is that we restrict to non coding regions whose length measured in codons is divisible, which is not the case in reality. Transition probabilities have been computed from an already annotated genome, Escherichia coli, K-12 substr. MG1655 (Genbank access NC_000913).

We can illustrate the annotation process as follows; we consider a small piece of E.coli from position 115 to 255.

ctt agg tca cta aat act tta acc aat ata ggc ata gcg cac aga cag ata aaa att aca gag tac aca aca tcc atg aaa cgc att agc acc acc att acc acc acc atc acc att acc aca ggt aac ggt gcg ggc tga,

The Viterbi algorithm computes the most probable path which is indicated as follows:

ctt	···	tcc	atg	aaa	···	ggc	tga
non coding	···	**non coding**	**start codon**	**coding**	···	**coding**	**stop codon**

From the indicated path, we can extract an annotation that states a non coding region from position 115 to 189 and a coding region from position 190 to 255.

5.2.3 A Second Example HMM: Pairwise Global Alignment

As another example of an HMM that we later extend with constraints, we consider the problem of aligning two sequences. Sequence alignment is among the most common tasks in computational biology, where it is used to align sequences assumed to have diverged from a common ancestor. Notice that we here use a so-called pair HMM [61] which emits two sequences at the same time, and which is a straightforward extension of the definition above.

In the global alignment problem, two sequences x and y must be aligned optimally, based on a scoring scheme for comparison of different alignments. In probabilistic modeling, a probability is associated with each pair of symbols emitted from a state and similarly a probability for introducing gaps, δ, and extending gaps, ϵ, in the alignment of the sequences is defined. The probability of an alignment is then the product of probabilistic transitions performed to recognize the alignment. In biology, these probabilities are defined to reflect observed statistics about sequence mutations and conservation.

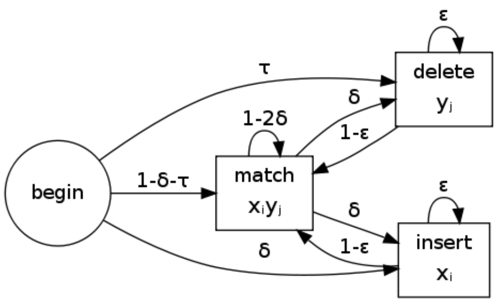


Figure 5.2: A pair HMM for pairwise global alignment of sequences. States, represented by squares for emitting states and circles for silent states, are connected by arrows representing transitions labeled with probabilities.

A Hidden Markov Model for modeling global sequence alignment is depicted in Fig. 5.2. A particular alignment corresponds to a distinct sequence of states in this model. The initial state, `begin`, does not emit symbols. The `match` state emits a pair of symbols (x_i, y_j), one for each sequence corresponding to alignment of the symbol at position i in sequence x and the symbol at position j in sequence y. Emitted symbols can be similar or different. A difference represents a potential mutation between the two sequences. The `insert` state emits only the next symbol of sequence x, effectively aligning position x_i to a gap in y. Oppositely, the `delete` state aligns a symbol y_j to a gap in sequence x.

This HMM is capable of generating a pair of aligned sequences. When given two sequences to align, then a path from the `start` state, such that the model emits the two sequences, will correspond to an alignment.

The following example shows an alignment of two short protein sequences, where the third line indicates the state sequence of this alignment abbreviated with the first letter of the state name:

```
Sequence x:   H G K K G A     A Q V
Sequence y:         K G P K K A Q A
alignment : b i i i m m m d d m m m
```

In this context, a common task is to find the optimal alignment. This means to find a state sequence that can recognize the two sequences and has maximal probability. Another is to calculate the probability to observe an emission sequence, here representing an alignment. A third type of inference is parameter learning, where we are given a set of alignments and estimate the "best" parameters for the model, where best usually means that they maximize likelihood of the alignments.

5.3 A constraint model for HMM with side-constraints

In this section, we give a formal definition of CHMMs and propose a constraint model for CHMM runs. Then, the computation of the most probable path is adapted for CHMMs.

5.3.1 Constrained Hidden Markov Model

A CHMM extends an HMM with constraints that limit the set of valid runs and leave fewer paths to consider for any given sequence.

Definition 5.3.1. *A* constrained HMM (CHMM) *is defined by a 5-tuple $\langle S, A, T, E, C\rangle$ where $\langle S, A, T, E\rangle$ is an HMM and C is a set of constraints, each of which is a mapping from HMM runs into $\{true, false\}$.*

A run *of a CHMM, $\langle path, observation\rangle$ is a run of the corresponding HMM for which $C(path, observation)$ is true.*

Notice that we define constraints in a highly abstract way, independently of any specific constraint language. In the following, constraints over finite domains [217] are used, although other constraint languages such as $CLP(Q)$ and $CLP(R)$ could have been used as well.

5.3.2 Runs of a CHMM as a constraint program

In this section, we propose to model runs of CHMM by a constraint program over finite domains. In this context, a run of CHMM is a solution of the constraint program.

Let $\langle S, A, T, E, C\rangle$ be a CHMM and n the sequence length. A constraint program for runs is given by the following predicate.

$$run([s^{(0)}, S_1, \ldots, S_n], [E_1, \ldots, E_n])$$

where each variable S_i and E_i represents the state and the emission at the step i. The domains of S_i and E_i, are given as $\mathrm{dom}(S_i) = S\backslash\{s_0\}$ and $\mathrm{dom}(E_i) = E$. The *run* predicate is specified as follows.

$run([s^{(0)}, S_1, \ldots, S_n], [E_1, \ldots, E_n])$ is $true$ iff

$$\begin{aligned}
&\exists s^{(1)} \in \mathrm{dom}(S_1), \ldots, \exists s^{(n)} \in \mathrm{dom}(S_n) \text{ and} \\
&\qquad \exists e^{(1)} \in \mathrm{dom}(E_1), \ldots, \exists e^{(n)} \in \mathrm{dom}(E_n), \\
&\qquad C(s^{(0)} s^{(1)} \ldots s^{(n)}, e^{(1)} \ldots e^{(n)}) \text{ is true}, s^{(0)} = s_0 \text{ and} \\
&\qquad p(s^{(0)}; s^{(1)}) \cdot p(s^{(1)}; e^{(1)}) \ldots p(s^{(n-1)}; s^{(n)}) \cdot p(s^{(n)}; e^{(n)}) > 0. \qquad (5.1)
\end{aligned}$$

Formula (5.1) states that $s^{(0)} s^{(1)} \ldots s^{(n)}$ and $e^{(1)} \ldots e^{(n)}$ is a run of the HMM that satisfies C. By the definition of *run/2*, (local) relationships between S_i and S_{i+1} and S_i and E_i can be established, since the probability of a run must be positive. Indeed, valuation of S_i to $s^{(i)}$ and S_{i+1} to $s^{(i+1)}$ is part of a solution of the constraint program whenever $p(s^{(i)}; s^{(i+1)}) > 0$. These relationships between variables of $run/2$ are modeled by the following constraints,

$$trans(S_{i-1}, S_i) \text{ and } emit(S_i, E_i), \text{for all } i,\ 1 \leq i \leq n$$

where S_i, S_{i+1} and E_i are the variables of *run/2*. These constraints are defined as follows.

- $trans(S_i, S_{i+1})$ is $true$ iff $\exists s^{(i)} \in \mathrm{dom}(S_i)$ and $s^{(i+1)} \in \mathrm{dom}(S_{i+1})$ such that $p(s^{(i)}; s^{(i+1)}) > 0$;

- $emit(S_i, E_i)$ is $true$ iff $\exists s^{(i)} \in \mathrm{dom}(S_i)$ and $e^{(i)} \in \mathrm{dom}(E_i)$ such that $p(s^{(i)}; e^{(i)}) > 0$.

Section 5.4 below shows an implementation of this framework such that a solution of the constraint program corresponds to a valid derivation of a PRISM program.

5.3.3 Example 1: A constrained gene finder

We first illustrate the constraint model on the simple gene finder presented subsection 5.2.2. The HMM associated with the simple gene finder is constrained to be in certain states at given positions. For instance, this CHMM allows the inclusion of information about known coding regions during the Viterbi computation.

Consider

$$run([s^{(0)}, S_1, \ldots, S_n], [e^{(1)}, \ldots, e^{(n)}])$$

the constraint model associated with the simple gene finder where $e^{(1)}, \ldots, e^{(n)}$ is a sequence of n codons. A set of variables S_i is constrained to be equal to $State$ with the following constraint:

$$fix(State, Position_1, Position_2)$$

where $State \in S \setminus \{s^{(0)}\}$, $Position_1 \in \{1, \ldots, n\}$, $Position_2 \in \{1, \ldots, n\}$ and $Position_1 \leq Position_2$.

$fix(State, Position_1, Position_2)$ is $true$ iff

$$\exists k \in \mathrm{dom}(Position_1) \text{ and } \exists l \in \mathrm{dom}(Position_2), \forall i, k \leq i \leq l, S_i = State.$$

For example, fix a position of a coding region can be expressed as the conjunction of

$$\begin{aligned}&fix(\textbf{start codon}, P_1, P_1) \wedge P_1 + 1 = P_2 \wedge \\ &\qquad fix(\textbf{coding}, P_2, P_3) \wedge P_3 + 1 = P_4 \wedge fix(\textbf{stop codon}, P_4, P_4).\end{aligned}$$

These constraints on the simple gene finder oblige runs to be in a coding region between the position P_1 and P_4.

5.3.4 Example 2: a Constrained Pairwise Global Alignment

We consider the HMM presented in section 5.2.3 and extend it into a CHMM by the following set of constraints,

$$\begin{aligned}C = \{&\text{cardinality\_atmost}(N_d, [S_1, \ldots, S_n], \text{delete}), \\ &\qquad \text{cardinality\_atmost}(N_i, [S_1, \ldots, S_n], \text{insert})\}.\end{aligned}$$

A constraint cardinality_atmost(N, L, X) is satisfied whenever L is a list of elements, out of which at most N are equal to X. In a biological context, it is reasonable to consider only alignments with a limited number of insertions and deletions given the assumption that the two sequences are related.

As described above, we can consider this CHMM as a constraint program

$$run([s^{(0)}, S_1, \ldots, S_n], [E_1, \ldots, E_n])$$

where $\text{dom}(S_i) \in \{\text{match}, \text{delete}, \text{insert}\}$, $\text{dom}(E_i) \in \{A, C, D, \ldots, W, Y\}$[1] and the constraints C are as described above.

5.3.5 Computation of the most probable path for a CHMM

The Viterbi algorithm [218] is a dynamic programming algorithm for finding a most probable path corresponding to a given observation. The algorithm keeps track of, for each prefix of an observed emission sequence, the most probable (partial) path leading to each possible state, and extends those step by step into longer paths, eventually covering the entire emission sequence. Here, we adapt this algorithm for CHMMs.

Consider a given observation $e^{(1)} \ldots e^{(n)}$, a CHMM $\langle S, A, T, E, C\rangle$, and its constraint program

$$run([s^{(0)}, S_1, \ldots, S_n], [e^{(1)}, \ldots, e^{(n)}]).$$

The most probable path is computed by finding the valuation $s^{(1)}, \ldots, s^{(n)}$ that maximizes the objective function: the run probability.

Computation of the most probable path for CHMM is expressed as a rewriting system on a set of 5-tuples Σ. Each such 5-tuple is of form $\langle s, i, p, \pi, \sigma\rangle$ where π is a partial path ending in state s and representing a path for the emission sequence prefix $e^{(1)} \cdots e^{(i)}$; p is the computed probability for the emissions and transitions applied in the construction of π, and σ is the current constraint

[1]This set of letters refers to the 21 different amino acids from which proteins are composed.

trans_ctr :	$\Sigma := \Sigma \cup \{\langle s', i{+}1, p \cdot p(s;s') \cdot p(s';e^{(i+1)}), \pi\, s', \sigma \wedge S_{i+1} = s'\rangle\}$ whenever $\langle s, i, p, \pi, \sigma\rangle \in \Sigma$, $p(s;s'), p(s';e^{(i+1)}) > 0$ check_constraints($\sigma \wedge S_{i+1} = s'$) and *prune_ctr* does not apply.
prune_ctr :	$\Sigma := \Sigma \setminus \{\langle s, i{+}1, p', \pi', \sigma'\rangle\}$ whenever there is another $\langle s, i{+}1, p, \pi, \sigma\rangle \in \Sigma$ with $p \geq p'$ and sol(σ') $\subseteq$ sol(σ).

Figure 5.3: Rewriting rules for the computation of most probable paths for CHMM

store seen as a conjunction of constraints. Any ground and satisfied constraint will be removed from the constraint store, and *true* refers to the empty conjunction. The set of solutions of a constraint store σ is denoted by sol(σ).

The two rewriting rules in Fig. 5.3 describe an iteration step of the computation of the most probable path.[2] The computation starts from an initial set of 5-tuples

$$\{\langle s^{(0)}, 0, 1, \epsilon, C \wedge trans(s^{(0)}, S_1) \wedge \bigwedge_{1\leq i\leq n-1} trans(S_i, S_{i+1}) \wedge \bigwedge_{1\leq i\leq n} emit(S_i, e_i)\rangle\}. \quad (5.2)$$

The *trans_ctr* rule expands an existing partial path one step in directions that preserve the satisfaction of the constraint store; this satisfiability check is denoted check_constraints (and depends thus on the particular C). The *prune_ctr* rule removes partial solutions that are not optimal for the current observation prefix *and* shares the same set of complete solutions with the better partial solution. The second condition is necessary in case no partial path contained in sol(σ) can be extended into a full path without violating the constraints. We take the following correctness property for granted.

Proposition 5.3.1. *Assume a CHMM H with the notation as above and an observation Obs* $= e^{(1)} \cdots e^{(n)}$. *When the Viterbi algorithm in Fig. 5.3 is executed from an initial set of 5-tuples given the formula (5.2), it terminates with a set of 5-tuples* Σ_{final}. *It holds that*

- *For any* $\langle s, n, p, \pi, true\rangle \in \Sigma_{final}$, π *is a most probable path for Obs ending in s and with probability p.*
- *Whenever there exists a path for Obs ending in s,* Σ_{final} *includes a 5-tuple of the form* $\langle s, n, p, \pi, true\rangle$.

Notice that all the variables of the constraint program are valuated when a final state is reached, and thus any final constraint store is equivalent to *true* (as *trans_ctr* prevents any inconsistent store to arise).

[2]When any reference to constraints and the constraint store are removed from Fig. 5.3, we have a compact representation of one iteration step of the Viterbi algorithm for HMMs.

The classical Viterbi algorithm is guaranteed to run in time linear to the length of the given sequence, whereas our algorithm may in the worst case run in exponential time; this may occur if *prune_ctr* cannot be applied at all. In other words, a representation of the constraint store that allows an efficient comparison as in "$\text{sol}(\sigma') \subseteq \text{sol}(\sigma)$" is essential for the practicability of our algorithm. On the other hand, for those problems that can be formulated as a CHMM with effective and efficient definitions of check_constraints and the comparison test, the Σ states may stay of a reasonable size. Notice that our algorithm is still correct if we use approximations of these tests, more specifically, check_constraints may occasionally return *true* when the correct answer is *f*alse and the opposite for the comparison.

5.4 Implementation of CHMMs in PRISM

After briefly introducing PRISM, we propose a methodology to define CHMMs in this framework.

5.4.1 A brief introduction to the PRISM System

PRISM [185] is a powerful system for working with probabilistic-logic models, based on an extension to Prolog with discrete random variables, called multi-valued switches. We illustrate this with a simple example HMM with two states `s0` and `s1`. A switch declaration,

```
values(x,O).
```

associates the named random variable x with a set of outcomes `O`. Whenever the goal `msw(x,X)` is called from the program, then a probabilistic choice will be made unifying `X` with an element of `O`. Switches can also be defined in a parametric form,

```
values(emit(_),[a,b]).  % symbol emission
values(trans(_),[s0,s1]). % state transition
```

where each declaration defines a family of switches, one for each possible instance of `emit(_)` and `trans(_)` and each instance is given a distinct probability distribution. This parametrization can serve to model dependencies: in our HMM example we define the parameters to be the states `s0` and `s1` (plus `init` for `trans(_)`), thus defining emissions and transitions for each state with the Markov property. Finally, we define a logic program to implement the probabilistic model,

```
hmm(L):- run_length(T), hmm(T,init,L).
hmm(0,_,[]).
hmm(T,State,[Emit|EmitRest]) :-
   T > 0,
   msw(trans(State),NextState),
   msw(emit(NextState),Emit),
   T1 is T-1,
   hmm(T1,NextState,EmitRest).
run_length(10).
```

Here, a derivation of the goal `hmm` corresponds to what we define as a *run* in section 5.2.1. As shown by [173], Prolog's traditional Herbrand model semantics generalizes immediately to a probabilistic semantics when probabilities are given for each random variable (provided that a few restrictions are respected on how `msw` is used in the program). Thus a PRISM program defines a probabilistic model that provides a probability distribution for all goals that can be formulated in the program's logical language. PRISM assigns each possible derivation of a goal a probability defined as the product of the probabilities of the selected switch outcomes of switches used in the derivation. Under normal conditions, it will be the case that the sum of probabilities of all possible derivations of such a goal is unity, but these conditions can be violated in a constrained model. If a program attempts to unify the stochastically selected outcome of a switch with some other value distinct from that outcome, this unification will fail resulting in a failed derivation.

PRISM includes built-in mechanisms for efficient probabilistic inference based on tabling. During inference, once a probabilistic goal has been solved, its answers are put in a global table. Later calls to the same goal will simply lookup the answer in the table in constant time. PRISM utilizes this to provide an efficient generalized Viterbi algorithm that may be used for the computation of the most likely *successful* derivation for a large number of probabilistic models including HMMs. PRISM also includes similar utilities for calculating the probability of a derivation or set of such and machine learning algorithms which produce the most likely probabilities for switch outcomes in order to explain a set of observed goals.

5.4.2 A framework for Constrained HMMs in PRISM

We have implemented a framework for integration of side-constraints in a PRISM program.[3] The framework has been used for adding constraints to HMM based models, but it should be possible to extend to other kinds of models. The underlying idea is that the program is augmented with a constraint store and a constraint checker goal is inserted in a few strategic places of the PRISM program. This constraint checking is the direct implementation of check_constraints of *trans_ ctr*. The *prune_ ctr* implementation is not discussed as we use for free the tabling mechanism of PRISM to prune the search space.

5.4.2.1 Integration of side-constraints in a PRISM program

This section describes how our framework can be integrated in a PRISM program. As an example, we consider an implementation of the HMM from the previous section. Below the central recursive predicate of the implementation is shown extended with constraint checking,

```
hmm(T,State,[Emit|EmitRest],StoreIn) :-
   T > 0,
   msw(trans(State),NextState),
   msw(emit(NextState),Emit),
```

[3]The current implementation of the framework is available via http://akira.ruc.dk/∼cth/chmm

```
   check_constraints([NextState,Emit],StoreIn,StoreOut),
   T1 is T-1,
   hmm(T1,NextState,EmitRest,StoreOut).
```

Integration of side-constraint checking is done by extending relevant predicates with an extra parameter (`StoreIn,StoreOut` in the code above) to accommodate a constraint store and a call to the `check_constraints` goal (line 5), after each distinct sequence of `msw` applications.

If `check_constraints` fails during PRISM inference, then the corresponding PRISM derivation fails, and further extensions of this derivation will not be attempted since it does not constitute a valid run. In effect, inference by PRISM will only consider runs which are guaranteed not to violate any of the constraints declared for the model.

Declaration of constraints and implementation of constraint solvers are conceptually decoupled from the PRISM model. The declaration of side-constraints on the model is done by declaring facts of the form, `constraint(ConstraintSpec)`. The `ConstraintSpec` associates the constraint with a constraint checker implementation and may contain some parameters for this particular instance of the type of constraint.

A satisfiability checker maintains its own constraint store. A satisfiability checker for a particular type of constraint consists of an `init_constraint_store/2` rule and one or more `check_sat/4` rules. The `init_constraint_store/2` rule is used to create a starting point for the constraint store of each declared constraint and is of the form,

```
init_constraint_store(ConstraintSpec, InitialStore).
```

It is given `ConstraintSpec` and must unify `InitialStore` with an initial constraint store matching the `ConstraintSpec`. Additionally, one or more `check_sat` rules of the form,

```
check_sat(ConstraintSpec,StateUpdate,StoreBefore,StoreAfter):-
...
```

must be implemented to check the satisfiability of the constraint.

As an example, consider an implementation of a cardinality_atmost constraint, called `cardinality` in our framework

```
init_constraint_store(cardinality(_,_), 0).
check_sat(cardinality(U,Max), U, VisitsIn, VisitsOut) :-
        VisitsOut is VisitsIn + 1,VisitsOut =< Max.
check_sat(cardinality(X,_),U,S,S) :- X \= U.
```

Each time `check_constraints` is called from the PRISM model, the relevant `check_sat` goals are called for each declared constraint. If any of these fails, so will `check_constraints`. `StateUpdate` and `StoreBefore` are given and `check_cons- traints` is expected to unify `StoreAfter` to an updated constraint store. In our example HMM, the `StateUpdate` will consist of the `[State,Emit]` pattern given to `check_constraints`.

The call to this rule must only succeed if the constraint given by `ConstraintSpec` is not violated by the further information given by the

`StateUpdate`. Constraints are checked incrementally and should only fail if any further updates to the constraint store can only lead to failure.

The constraint stores of individually declared constraints are automatically aggregated in the constraint store exposed to the PRISM model. Individual constraint checkers are unaware of each other and cannot access the individual constraint stores of other constraint checkers. The constraints are checked in the order they are declared, so this order should be optimized to do pruning as early as possible.

5.4.2.2 Efficient inference with a separate constraint store stack

PRISMs tabling mechanism makes Viterbi computation and EM learning efficient, but when extra parameters such as the constraint store are introduced in the probabilistic goals, PRISM considers these as goals with distinct derivations and stores a tabled entry for each version of the goal. This behavior is undesired when the extra parameters are used only for internal bookkeeping. The effect of this excessive tabling is that the dynamic programming advantages are lost with exponential time inference as consequence.

In [40] a related problem concerning tabling of annotations produced by running Viterbi on PRISM programs is approached using a program transformation that removes non-discriminating arguments, which do not affect the control flow. The annotation can be then recovered from the program derivation of the transformed program.

This approach is not applicable for the constraint store argument, because the constraint store implicitly affects control flow by limiting possible future derivation extensions. The constraint store has to be considered in the inference process; otherwise it would be possible to produce invalid derivation paths.

B-Prolog, on which PRISM is based, supports table modes, but this is not directly usable with probabilistic goals in PRISM. It is possible with these modes to declare an argument of a tabled goal as an *output argument*, which means that it will not be used as key in the table lookup, but will be unified with the value of the argument stored in a tabled goal. For our purpose, declaring the constraint store arguments as output arguments would not be feasible since different derivations of the same goal may have differing constraint stores and these determine possible derivation extensions.

As a way to deal with the tabling problem we have introduced a separate constraint store stack, which avoids storing data locally in parameters of probabilistic goals by maintaining the constraint store with assert and retract. This stack is maintained in parallel to the to derivation stack of Prolog. PRISM utilizes Prologs backtracking to explore possible solutions, so the constraint store stack implementation is required to be able to restore a previous constraint store when PRISM encounters failures during inference and performs backtracking to find alternative solutions.

To utilize this functionality, the user should use the goal `check_constraints/1`, which omits the store arguments, rather than `check_constraints/3` as stated above. We then define `check_constraints/1` as

```
check_constraints(StateUpdate) :-
    get_store(StoreBefore),
    check_constraints(StateUpdate,StoreBefore,StoreAfter),
    forward_store(StoreAfter).
```

The new `check_constraints/1` make use of the goal `get_store/1` to retrieve the current version of the constraint store and `forward_store/1` is used to assert the updated store,

```
get_store(S) :- !, store(S).
forward_store(S) :- asserta(store(S)) ; retract(store(S)),fail.
```

If a derivation fails, then PRISM will backtrack to the choice point in the `forward_store` rule and retract the most recently asserted store. Then, when exploring alternative derivation extensions, the previously asserted constraint store will be used as expected.

5.4.2.3 Complexity analysis of our implementation

Due to tabling, PRISM guarantees familiar best known complexity bounds of common inference tasks on a variety of the models that can be expressed in PRISM, including HMMs [181]. This implicitly limits the number of `check_constraints` calls to the same bound. The added complexity of doing constraint checking depends on incremental constraint checking cost of individual constraints checkers and the number of constraints expressed on the model.

Space complexity is influenced by table space usage and maximal length of a derivation at any given point. Since the asserted constraint store contains a constraint store fixpoint for each step of the current derivation, the constraint store stack is bounded by $O(n\max(|c|))$ where n is the length of the sequence and $\max(|c|)$ is maximal size of the constraint store in any derivation step. Note that, the space complexity of the separate constraint store stack is unaffected by time complexity and the number of states in the model. With more complex models like the pair HMM, the table space required for dynamic programming becomes the dominating concern.

5.5 Experimental validation

In this section, we validate our CHMM implementation experimentally with the pair HMM presented in section 5.2.3. The experiments was run on a computer with 16 2.4 GHz, 64 bit Intel Xeon(R) E7340 CPUs and 64 GB of memory. All of the experiments utilized only a single processor at a time.

Our experiments utilize implementations of some common constraints adapted for the CHMM framework: `cardinality(UpdatePatterns,Max)` ensures that entries from the list `UpdatePatterns` occurs at most `Max` times in the derivation sequence. `alldiff` ensures that all updates in a derivation are different; `lock_to_sequence(Seq)` ensures that the sequence of derivation updates is identical to the sequence represented by the list `Seq`; `lock_to_set(Set)` ensures that all updates belong to members of the list `Set`. The operator `forall_subseq(L,C)` applies the constraint C to every subsequence of length L in the derivation sequence and `for_range(From,To,C)` applies C only the range, `To-From`, both inclusive; `state_specific(C)` applies C only to the `State` part of the update.

5.5.1 Running time of constrained alignment

The addition of side-constraints to an HMM adds some computational overhead in order to check the satisfiability of the constraints, but may also reduce the number of possible solutions and in turn the amount of work required for the Viterbi algorithm to find the optimal path. As a practical experiment to demonstrate this we consider global alignment with the pair HMM discussed in section 5.2.3.

The overhead of integrating the constraint checking machinery in the model is demonstrated in the left part of Fig. 5.4, where sequences of increasing length are aligned. It can be observed that the running time penalty is a constant factor and that the polynomial time complexity of the pair HMM is preserved in our framework. Obviously, polynomial time inference presupposes incremental constraint checking to be a constant time operation, which may not be the case for certain types of constraints.

In the right part of figure Fig. 5.5, two sequences of equal length (32) is aligned, but with varying amounts of constraints being enforced: The global cardinality constraint is used to enforce an upper limit, `L`, on the amount of inserts or deletes in the alignment,

```
constraint(state_specific(cardinality([insert,delete],L))).
```

By constraining the alignment (allowing fewer gaps), the space of viable solutions is reduced. The more constrained the alignment is, the more pruning opportunities arise. With an large amount of pruning opportunities, the running time is reduced quite significantly.

Note that, since the imposed constraint is `state_specific`, the number of possible alignments, and hence running time, is unaffected by the structure of the input sequences.

5.5.2 Efficiency of the separate constraint store stack

To verify the efficiency of our constraint store implementation, alignment with a local cardinality constraint was measured for different sizes of input sequences. From the measurements, which are reported in Fig. 5.6, it is apparent that our implementation does not incur the same exponential overhead as the naive implementation where the constraint store is maintained in the goals and hence tabled.

Running times and memory usage for a range of different constraints are reported in Table 5.1 and Table 5.2, respectively. For the sake for completeness, the table also includes measurements for the version where the constraint store is tabled.

In most cases the separate constraint store performs better in terms of both running time and memory consumption. In the cases where performance is worse, it can be attributed to a very small number of possible derivations or a type of constraint which rarely produce changes to the store.

5.6 Related Work

The term "Constrained HMM" is used in [169, 121] and refers to restrictions on the finite automaton associated with an HMM but not as constraint on HMM

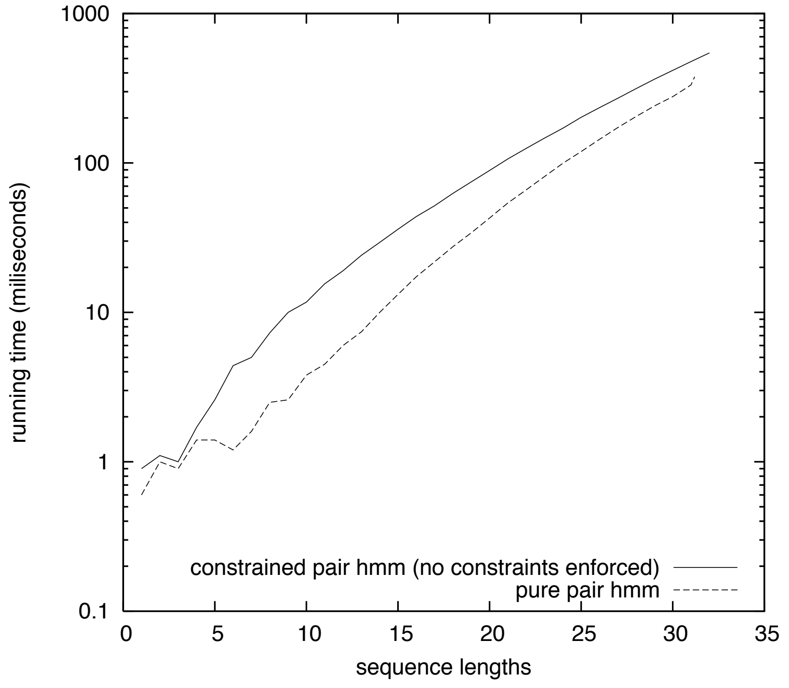


Figure 5.4: Running time of alignment with a pure pair HMM compared to alignment with a CHMM with no constraints enforced.

runs. In [185], CHMMs were introduced to exemplify an EM algorithm, suited for PRISM programs which allow the possibility of derivation failures. Our approach differs, as we augment PRISM programs with side-constraints and use constraint solving techniques to achieve efficient inference.

In [50] relationships between elements of a Bayesian Network are expressed as a constraint logic program, which is similar to the way we define HMMs. However, our paper focus differs as we study the interest of checking satisfiability of side-constraints during inference.

In the natural language processing community, recent work on Constrained Conditional Models feature an approach similar to ours. Indeed, Constraint Conditional Models is a general framework that augments inference and learning of conditional models with declarative constraints [30]. However, inference is expressed as an Integer Linear Programming problem [168]. In this context, more expressive constraints, such as `cardinality` or `all_different`, can not be added on an HMM run. Moreover, our PRISM-based implementation allows us to define the HMM structure (and, e.g., adapt it easily to pair HMMs) separately from the side-constraints and use advanced constraint solving techniques.

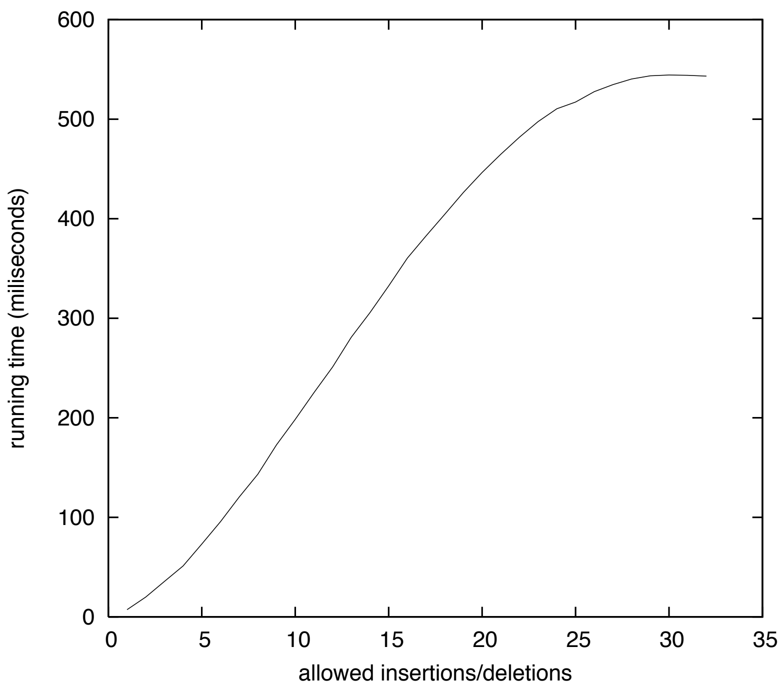


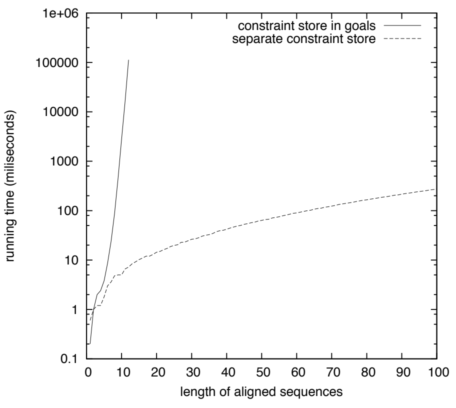
Figure 5.5: Right: Running time of alignment of two sequences of length 32 with varying amounts of allowed insertions and deletions.

Constraint	Sequence lengths	Running time (in ms)	
		in goals	separate
cardinality([insert],20)	50	15460	3176
cardinality([insert],40)	50	29557	3968
for_range(1,50, lock_to_set([match]))	100	24649	4544
for_range(1,90, lock_to_set([match]))	100	20	48
for_range(1,50, lock_to_sequence([match,..,match]))	100	24829	4544
for_range(1,90, lock_to_sequence([match,..,match]))	100	20	48
alldiff	20	100442	28
forall_subseqs(5,alldiff)	10	1664	12

Table 5.1: Running time for alignment with different kinds of constraints.

5.7 Conclusions

In this paper, we propose a framework to define HMMs with side-constraints as a Constraint Logic program extended by probabilistic choices. Constraint Logic Programming have advantages in terms of more compact expression of CHMMs. Inference computations are adapted for CHMMs and conditions for an efficient computation are described. An implementation based on PRISM



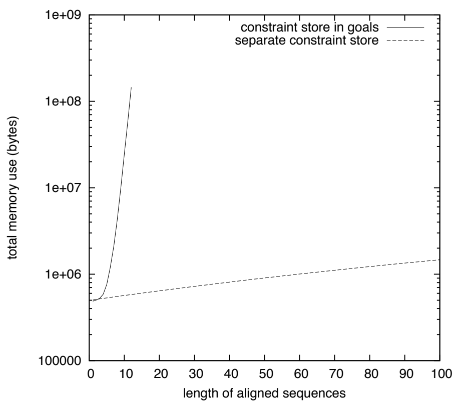


Figure 5.6: A comparison of the running time (left) and memory usage (right) of constrained alignment of two sequences with tabled constraints versus a separate constraint store stack.

Constraint	Sequence lengths	Memory usage (in kb)	
		in goals	separate
cardinality([insert],20)	50	42296	5723
cardinality([insert],40)	50	93845	6703
for_range(1,50, lock_to_set([match]))	100	105498	7137
for_range(1,90, lock_to_set([match]))	100	1641	1198
for_range(1,50, lock_to_sequence([match,..,match]))	100	1641	1198
for_range(1,90, lock_to_sequence([match,..,match]))	100	105498	7137
alldiff	20	85654	256
forall_subseqs(5,alldiff)	10	60098	137

Table 5.2: Memory consumption for alignment with different kinds of constraints.

is proposed and three well-known constraints and operators have been demonstrated for defining CHMMs. Finally, we experimentally validate our approach with a constrained pair HMM used for biological sequence alignment.

As current work, we study how sampling and EM-learning can be adapted for our CHMM framework. Indeed, sampling turns out to be problematic in probabilistic models with a large probability of derivation failure. In [187], Sato et al. address the problem of EM-learning with PRISM programs that can fail and their methods are also applicable for our framework. A first experiment on semi-supervised learning with our CHMM framework gives us encouraging first result.

As further work, we plan to incorporate in the framework more advanced constraint solving techniques such as those used in Weighted CSP [122]. This approach would allow us to combine soft constraints solving and inference and express this as an optimization problem. We also plan to deal with the restriction that individual constraint checkers do not share information in our framework, so that we can benefit from some of the optimization techniques used by other constraint solvers. We are working on extending the library of constraints that can be defined as side-constraints.

Acknowledgment

This work is supported by the project “Logic-statistic modeling and analysis of biological sequence data” funded by the NABIIT program under the Danish Strategic Research Council, and the CONTROL project, funded by Danish Natural Science Research Council.

Chapter 6

Constraints and Global Optimization for Gene Prediction Overlap Resolution

In Proceedings of Workshop on Constraint Based Methods for Bioinformatics, 2011.

Abstract

We apply constraints and global optimization to the problem of restricting overlapping of gene predictions for prokaryotic genomes. We investigate existing heuristic methods and show how they may be expressed using Constraint Handling Rules. Furthermore, we integrate existing methods in a global optimization procedure expressed as probabilistic model in the PRISM language. This approach yields an optimal (highest scoring) subset of predictions that satisfy the constraints. Experimental results indicate accuracy comparable to the heuristic approaches.

6.1 Introduction

Traditionally, gene finding has been considered as a classification task which could be performed without much context [66]. This ignores the problem of the constraints between the set of predicted genes and their placement in the genome. A common problem occurs with overlapping genes. Overlapping genes are rare in prokaryotic genomes, but they do occur [147, 74].

The traditional intrinsic gene finding methods have a tendency to predict too many overlapping genes (particularly in GC rich genomes) because the feature patterns of a gene predicted in one reading frame give rise to similar feature patterns in other reading frames. This effect is known as shadow genes.

Several gene finders deal with the problem of overlapping genes by discarding some of the overlapping predictions in a post-processing step. In this paper we consider and compare such post-processing techniques and give unified presentation using Constraint Handling Rules [72]. We demonstrate how such rules can be formulated as constraints and integrated with a global optimization procedure implemented as a constrained Markov chain in the PRISM system [175].

We adopt a divide and conquer approach to gene finding, which can be seen as composed of two steps:

1. A gene finder supplies a set of candidate predictions $p_1 \dots p_n$, called the *initial set*.

2. The *initial set* is pruned according to certain rules or constraints. We call the pruned set the *final set*.

The present paper is concerned with methods for the second step. The purpose of this step is to repair effects of flawed assumptions in the first step, i.e. leading to over-prediction of overlapping genes, and more specifically to improve accuracy by pruning false predictions. We consider this step as a Constraint Satisfaction Problem (CSP).

Definition 6.1.1. *A Constraint Satisfaction Problem is a triplet $\langle X, D, C \rangle$. X is a set of n variables, $X = x_1, \dots, x_n$, with domains $D = D(x_1), \dots, D(x_n)$. The constraints C impose restrictions on possible assignments for sets of variables. A solution is an assignment of a value $v \in D(x_i)$ to each variable $x_i \in X$, consistent with C.*

We introduce variables $X = x_i \dots x_n$ corresponding to each prediction $p_1 \dots p_n$ in the initial set. All variables have boolean domains, $\forall x_i \in X, D(x_i) = \{true, false\}$ and $x_i = true \Rightarrow p_i \in$ **final set**.

If there are multiple solutions, then we are usually interested in the "best" one. We interpret "best" as meaning a solution that contains as many real genes as possible and as few incorrect predictions as possible. We do not know in advance which predictions are correct, but optimize the probability (or a similar measure) that the predictions are correct. This extends the problem as a constraint optimization problem.

Definition 6.1.2. *A Constraint Optimization Problem (COP) is a CSP where each solution is associated with a cost and the goal is to find a solution with minimal cost*[1].

6.2 Local heuristic methods

An approach taken by many gene finders is to employ local heuristic pruning rules to post-process a set of gene predictions. These rules make pruning decisions based on the context of only a subset of the predictions. Typically, the rules consider overlapping predictions on a case by case basis and deletes inconsistent predictions based on various criteria. The rules essentially work as propagators that reduce the domains of variables, e.g. a deletion corresponds to reducing the boolean domain of the corresponding variable to $false$. The drawback is that the rules are generally not guaranteed to yield a globally optimal solution and that they may produce different solutions depending on the order in which they are applied.

[1]Or equivalently, a solution with maximal negative cost (utility).

These types of rules are conveniently expressed as *simplification* rules in the Constraint Handling Rules (CHR) language. Such rules work on a constraint store, which starts out as the initial set. The simplification rules remove predictions from the constraint store, until no more rules apply. Then, the constraint store represents the final set.

As example, consider the post-processing procedure of the Genemark frame-by-frame gene finder [138] expressed as a single rule in CHR:

```
prediction(Left1,Right1), prediction(Left2,Right2) <=>
        Left1 =< Left2,  Right1 >= Right2
        |  prediction(Left1,Right1).
```

The head of the rule — the part before `<=>` — matches two predictions in the constraint store. The rule replaces both predictions with the first prediction if the first prediction completely overlaps the second prediction. This condition is expressed in the guard of the rule – the part between the head and the `|` character. The rule is applied for all predictions matching the head and the guard, effectively removing all predictions which are completely overlapped by another prediction. With this rule it does not matter in which order the predictions are processed – the final set will be same. This is a consequence since the program consisting of the unique rule presented is *confluent* [1], i.e. it is not sensitive to the order of execution.

As an example of non-confluent rules, consider the scheme used in the ECOPARSE gene finder [118] which addresses partial overlaps and the score of the predictions:

```
prediction(Left1,Right1,Score1), prediction(Left2,Right2,Score2) <=>
    overlap_length((Left1,Right1),(Left2,Right2),OverlapLength),
    length_ratio((Left1,Right1),(Left2,Right2),Ratio),
    length(Left1,Right1,Length1), length(Left2,Right2,Length2),
    OverlapLength > 15, Score1 > Score2
    ((Length1 > 400, Length2 > 400) ; Ratio > 0.5),
    | prediction(Left1,Right1,Score1).

prediction(Left1,Right1,Score1), prediction(Left2,Right2,Score2) <=>
    overlap_length((Left1,Right1),(Left2,Right2),OverlapLength),
    length_ratio((Left1,Right1),(Left2,Right2),Ratio),
    length(Left1,Right1,Length1), length(Left2,Right2,Length2),
    OverlapLength > 15,  Ratio =< 0.5,  Length1 =< Length2
    | prediction(Left1,Right1,Score1).
```

If two predictions overlap by more than 15 bases, then one of them is removed. If the ratio between the longest and shortest of the predictions is more than 0.5, then the lowest scoring is removed (first rule) otherwise the shortest one is removed (second rule). Note how this may lead to different effects depending on the order in which predictions are considered, as illustrated in figure 6.1.

There are other approaches which employ more complex local heuristics. An example is heuristics of the RescueNet gene finder [129] which has rules considering scores, percent overlaps and local overlaps between up to three predictions. These heuristics can be implemented with nine CHR rules (not shown), but the resulting program is not confluent.

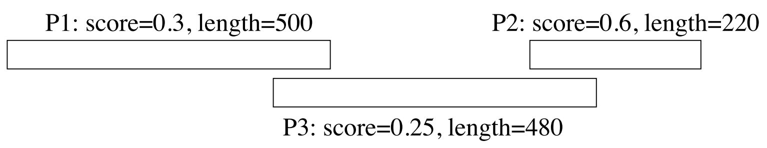

P1: score=0.3, length=500
P2: score=0.6, length=220
P3: score=0.25, length=480

Figure 6.1: ECOGENE post processing: We have two predictions $P1$ and $P2$ that overlap each end of a third prediction $P3$ by more than 15 bases. If $P1$ and $P3$ is are considered before $P2$ and $P3$ then $P3$ will be removed by the first rule. Consequently $P2$ does not overlap and is kept. If they are considered in opposite order, however, then $P2$ will be removed by the second rule and subsequently $P3$ is removed by the first rule.

It is a general theme for the heuristics to be based on two central characteristics of overlapping predictions – the score of the predictions and the (relative) lengths of the predictions and the overlap.

6.3 Global optimization

We would like the final set to reflect the relative confidence scores in the predictions assigned by the gene finder and at the same time be consistent with the overlap constraints. To accomplish this we reformulate the problem as a constraint optimization problem.

Let the scores of $p_1 \dots p_n$ be $s_1 \dots s_n$ and $s_i \in \mathcal{R}^+$. The scores are the confidence scores given by the underlying gene finder, i.e. they reflect the supposed probability that a prediction constitutes a real gene. Such scores are commonly expressed as probabilities, but need not be.

We would like to maximize the sum of the scores $\sum_{i=1}^{n} s_i$ since it is directly related to the criteria of the model that produced the initial set. With this criteria, the inclination to prune a prediction in the final set is inversely proportional to the score which is expected to reflect the underlying models belief that the prediction is a real gene.

To perform global optimization with a set of constraints, we propose to use a constrained first-order Markov chain. We assume that a gene finder has produced initial set of predictions, $p_1 \dots p_n$, and further require these to be sorted by the position of their left-most base, such that $\forall p_i, p_j, i < j \Rightarrow$ left-most$(p_i) \leq$ left-most(p_j). The variables $x_1 \dots x_n$ of the CSP are given the same ordering.

The Markov chain has a *begin* state, an *end* state and two states for each variable x_i corresponding to its boolean domain $D(x_i)$. The state corresponding to $D(x_i) = true$ is denoted α_i and the state corresponding to $D(x_i) = false$ is denoted β_i. In this model, a path from the begin state to the end state corresponds to a potential solution of the CSP. The Markov model is illustrated in figure 6.2. The *begin* state has transitions to α_1 with probability $P(\alpha_1|begin) = \sigma_1$ and β_1 with probability $P(\beta_1|begin) = 1 - \sigma_1$. The last two prediction states, α_n, β_n can only transit to the end state, i.e. $P(end|\alpha_n) = P(end|\beta_n) = 1$. For all other states, we have the transition probabilities,

$$P(\alpha_i|\alpha_{i-1}) = P(\alpha_i|\beta_{i-1}) = \sigma_i \text{ and } P(\beta_i|\alpha_{i-1}) = P(\beta_i|\beta_{i-1}) = 1 - \sigma_i$$

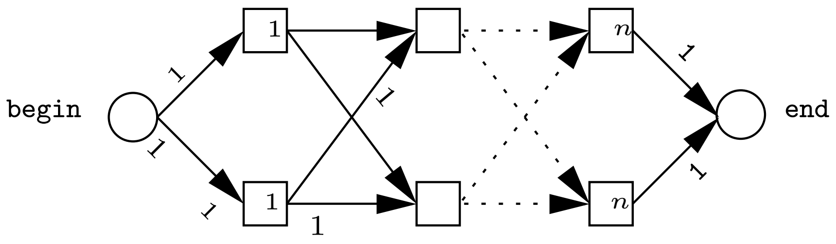


Figure 6.2: Illustration of the Markov chain used. The transitions are marked with their corresponding probabilities. Only the first few and the last states are included - the dotted transition arrows symbolize the omitted $\alpha_3 \dots \alpha_{n-1}$ and $\beta_3 \dots \beta_{n-1}$ states and their transitions, which follows the same principle as the previous.

We normalize the scores to the interval $(0.5, 1]$, yielding the normalized probability scores $\sigma_1 \dots \sigma_n$, in the following way,

$$\sigma_i = 0.5 + \lambda + \frac{(0.5 - \lambda) \times (s_i - min(s_1 \dots s_n))}{max(s_1 \dots s_n) - min(s_1 \dots s_n)}$$

λ is a small pseudo-count to ensure that all σ scores are above 0.5. Since α probabilities are always larger than 0.5, the model prefers α states over their corresponding β states. Hence, a most probable path from the *begin* state to the *end* state will not include any β states. The predictions that maximize the product of the σ scores will also maximize the sum of the original scores, since the normalized σ scores are monotonic to the original scores, $\sigma_i \geq \sigma_j \iff s_i \geq s_j$.

For inference with the model we use the Viterbi algorithm [218], which returns a most probable state sequence $\{begin, S_1, S_2 \dots S_n, end\}|S_i \in \{\alpha_i, \beta_i\}$.

Constraints are defined on states that are not allowed to occur together in a path. These constraints force the Viterbi algorithm to choose a most probable path, consistent with the imposed constraints, i.e. this path may include β states. The constraints are formulated as CHR rules similar to those of the local heuristics, but instead of removing predictions they define conditions for inconsistency. We call these *inconsistency rules*. Inconsistency rules match predictions corresponding to α and β states in the head of the rule. The guard of the rule ensures that the additional criteria for rule application are met and the implication of the rules is always failure. Note that such rules are necessarily confluent.

As example, version 3 of the Glimmer gene finder [56] use a similar approach with a constraint that enforce a maximal length of overlaps (110 for *E.coli*). In our system, this constraint is formulated as,

```
alpha(Left1,Right1), alpha(Left2,Right2) <=>
    overlap_length((Left1,Right1),(Left2,Right2),OverlapLength),
    OverlapLength > 110
    | fail.
```

The Genemark heuristic rule is represented as two inconsistency rules,

```
alpha(Left1,Right1), alpha(Left2,Right2) <=>
    Left1 =< Left2, Right1 >= Right2 | fail.
beta(Left1,Right1), alpha(Left2,Right2) <=>
    Left1 =< Left2, Right1 >= Right2 | fail.
```

The first rule states that one prediction may not completely overlap another and the second says that we cannot include a prediction if a pruned prediction completely overlaps it. Since the heuristic is confluent it may also be applied to the initial set as a filtering algorithm before the process of global optimization. We can reformulate the two ECOGENE rules in the same fashion (guard is omitted, but it is the same as in the heuristic rules),

```
alpha(Left1,Right1), alpha(Left2,Right2) <=>  ... | fail.
beta(Left1,Right1), alpha(Left2,Right2) <=>  ... | fail.
```

Note that the `Score` arguments have been removed. They are now implicitly integrated in the optimization algorithm. The confluence issue is resolved due to the optimization procedure. In effect, the execution strategy that maximizes the score is applied.

6.3.1 Implementation in PRISM

A PRISM program that implements the constrained Markov chain is created from the initial set of predictions and constraints expressed as CHR rules. PRISM is an extension of Prolog with special goals representing random variables. A derivation of the PRISM program corresponds to a path through the Markov chain. The Markov chain is implemented as a recursive predicate, such that in the *i'th* recursive call, the (random) variable x_i is assigned a value corresponding to a Markov chain state; α_i or β_i. After each recursion — an attempted transition in the Markov model — the constraints are checked.

Relevant recent states As part of a derivation we maintain a list of recent states (m_i) sorted by the right-most position of the corresponding predictions. Constraints are only checked for predictions corresponding to elements of m_i. In step i, we construct m_i as the maximal prefix of $x_i + m_{i-1}$, such that $x_j \in m_i \iff$ right-most$(p_j) \geq$ left-most(p_i). If the constraints propagate `fail`, then the PRISM derivation fails and the (partial) path it represents is pruned from the solution space.

The most probable consistent path is found using PRISMs generic adaptation of the Viterbi algorithm for PRISM programs [181].

6.4 Evaluation

In lack of a true golden standard, we use an accepted reference set to define the set of "correct" genes. A slight complication of this approach is that the reference set itself may have incorrect and missing annotations. True positives are gene predictions in the final set which are included (exactly) in the reference set and false positives are those predictions that are not.

Traditionally, in gene finding, accuracy is measured in terms sensitivity and specificity. Sensitivity measures the fraction of reference genes exactly predicted by the approach and specificity measures the fraction of predicted genes that are correct. Since the starting point is the initial set of predictions (which may omit some potential genes) we cannot improve on sensitivity. The goal of a pruning approach is then to improve on specificity with minimal impact to sensitivity.

We consider a pruning approach *successful* wrt. to an initial set when it prunes false positives at a higher rate than it prune true positives. This is reflected by the difference in sensitivity and specificity of the final set compared to the initial set.

We consider constraints *safe* when the constraints prune only false positives. Neither of the examined constraints are safe with respect the RefSeq annotation of *E.coli*, *NC*_000913. Three of the reference genes are completely overlapped by another reference gene. These would be removed by the genemark heuristic and hence it is not safe, although the negative impact of sensitivity would negligible. Similarly with the Glimmer constraint – the reference annotation have four overlaps longer than 110 bases which would be removed by this constraint. There are 93 overlaps longer than 15 bases. All of these would be removed by the ECOGENE constraints, which is therefore expected to have a noticeable negative sensitivity impact.

6.4.1 Experimental validation

We compare the different approaches using the predictions from a very simple codon preference based gene finder – the simplest model described in [43]. The gene finder has been trained on *E.coli NC*_000913 and applied to predict genes in the same genome. It overpredicts quite a lot – a total of 10799 predictions for the genome, which has 4145 known genes.

We ran the constrained Markov chain using the gene finder predictions as initial set, applying our adaptations of the both the Genemark constraint, the ECOGENE constraint and the Glimmer3 constraint. We also tested the local heuristic versions of the Genemark and ECOGENE constraints. The results are summarized in table 6.1.

Both the Genemark and ECOGENE heuristics achieve quite impressive improvement compared to the initial set. Our global optimization achieves better sensitivity than ECOGENE and better specificity than Genemark, but seen as a combination of the measures, the result is not significantly better.

Note that the optimal or highest scoring set of predictions subject to the constraints is not necessarily the most successful, but it is the one that most faithfully reflects the confidence scores assigned by the gene finder.

The purely declarative CHR implementations of genemark or ECOGENE rules are quite slow (hours), e.g. it essentially considers each pair of constraints

Method	#predictions	Sensitivity	Specificity	Time (seconds)
initial set	10799	0.7625	0.2926	na
Genemark rules	5823	0.7558	0.5379	1.4
ECOGENE rules	4981	0.7148	0.5947	1.7
global optimization	5222	0.7201	0.5714	75

Table 6.1: Accuracy of predictions using different overlap resolution approaches. Note that the results for the ECOGENE heuristic may vary depending on execution strategy - in case of above results, predictions with lower left position are considered first.

resulting in $\mathcal{O}(n^2)$ complexity, n being the number of predictions in the initial set. However, with proper control in place (using the relevant recent states optimization described in section 6.3.1), they can be made to run very fast (less than two seconds). The running time for the global optimization is slower – it takes a little more than one minute. This is still acceptable.

6.5 Conclusions

We presented a novel way to post-process gene prediction results based on constrained global optimization. Contrary to the heuristic approaches our approach provides an optimality guarantee – the final set of prediction will be the maximally scoring set that satisfies the imposed constraints. We have incorporated existing heuristic methods with the optimization procedure using inconsistency rules implemented in CHR. Currently, the approach has similar accuracy to the heuristic methods. The results indicate that maximizing the sum of scores have the effect of including more short predictions. This could be addressed weighting the scores by prediction length. We also plan to experiment with different constraints to achieve better accuracy. Our approach is limited to local overlap constraints and is not well-suited for global or long-distance constraints.

We are not the first to use dynamic programming based approaches to post-processing of gene predictions. Version 3 of Glimmer [56] use a custom dynamic programming algorithm which is similar to the present approach, but incorporates only the maximal overlap constraint. Another difference is that our approach is expressed as a declarative PRISM program and can therefore utilize the generalized Viterbi algorithm. Our approach is similar to constrained HMMs in PRISM, which has previously be applied to other biological sequence analysis tasks [42, 41]. A main difference is that we express constraints with CHR rules.

CHRiSM[196] already combines CHR and PRISM and is to our knowledge the first system to do so. CHRiSM assigns probabilistic semantics to CHR rules, which are interpreted as chance rules – e.g. even if a rule head is matched the rule is only applied with a certain probability. The main difference with our approach is that we use ordinary CHR rules in conjunction with a PRISM program, although ordinary CHR rules may be seen as a special case of CHRiSM rules, where the probability of invocation is one. Additionally, the form of the CHR rules we use is restricted (inconsistency rules) and they are only used in the constraint checking part of the PRISM program. It would be interesting to use CHRiSM as a method of incorporating soft constraints with our approach, e.g. redefining the inconsistency rules as CHRiSM chance rules.

Acknowledgement This work is part of the project "Logic-statistic modeling and analysis of biological sequence data" funded by the NABIIT program under the Danish Strategic Research Council.

Chapter 7

The Viterbi Algorithm Expressed in Constraint Handling Rules

With Henning Christiansen and Ole Torp Lassen and Matthieu Petit

In Proceeedings of the 7th International Workshop on Constraint Handling Rules, 2010.

Abstract

The Viterbi algorithm is a classical example of a dynamic programming algorithm, in which pruning reduces the search space drastically, so that an otherwise exponential time complexity is reduced to linearity. The central steps of the algorithm, expansion and pruning, can be expressed in a concise and clear way in CHR, but additional control is needed in order to obtain the desired time complexity. It is shown how auxiliary constraints, called *trigger* constraints, can be applied to fine-tune the order of CHR rule applications in order to reach this goal. It is indicated how properties such as confluence can be useful for showing such optimized programs correct.

7.1 Introduction

Hidden Markov Models (HMMs) are probabilistic finite state machines that for each transition emits a symbol from a finite alphabet, also by probabilistic choice. HMMs are commonly used for modeling and analysis of, e.g., biological sequence data and for speech recognition; see, e.g., [61, 104]. Given a specific HMM and an observed sequence over the alphabet, *prediction* means to find the most probable path, i.e., sequences of states, by means of which the sequence may have been produced; such a path is called a *Viterbi path.* Informally speaking, a Viterbi path represents the most feasible interpretation or explanation of the given sequence; in the biological case, the sequence may be DNA and the Viterbi path indicates the most believable shifts between coding and non-coding regions, and perhaps details concerning introns and exons [61].

HMMs owe much of their popularity to the existence of efficient algorithms for training and, as we consider here, prediction in terms of the classical Viterbi algorithm [218]. It is a dynamic programming algorithm that gradually extends optimal paths so they cover a longer and longer prefix of the sequence, and eventually the entire sequence. The algorithm keeps track of one optimal path ending in each state s for the sub-sequence seen so far; call this set of partial paths Σ. Any non-optimal path is discarded. In the next step, a new set of optimal paths is found among the possible extensions of any $\sigma \in \Sigma$ with one more state. The time complexity is $\mathcal{O}(n \cdot k^2)$ and the space complexity is $\mathcal{O}(n \cdot k)$ where n is the sequence length and k the number of states.

In this paper, we investigate how well the Viterbi algorithm can be expressed in CHR, considering both conciseness and efficiency. CHR, or Constraint Handling Rules [70, 71], was introduced as a declarative language for writing constraint solvers, but has shown to be very useful for a variety of automated reasoning tasks, and attempts have been made to use it as a general language for describing algorithms.

We show that the fundamental steps in the Viterbi algorithm can be exposed very clearly in CHR, but to reach the optimal time complexity, we need to introduce some techniques. We suggest to use *trigger* constraints, by means of which a program's operational behavior can be fine-tuned. For confluent programs, this can be analyzed in a systematic way, and as a more general case, we put forward informally the notion of "relative confluence" based on a more flexible state equivalence (as compared with the usual logical equivalence of states). However, in order to reach the optimal time complexity, we need to make additional transformations that reflect the underlying CHR system's search and matching. This may be less satisfactory from the point of view of declarative programming, but may inspire to the development of new automatic analyses and transformations to be included in CHR implementations.

7.2 A concise Viterbi-like algorithm in CHR

The fundamental parts of the Viterbi algorithm can be expressed in CHR as shown in fig. 7.1; the specific HMM is encoded as a set of constraints of the forms
`trans(`q_1`,`q_2`,`p_1`)` and `emit(`q_3`,`ℓ`,`p_2`)`, where p_1 is the probability to transit from state q_1 to q_2, and p_2 is the probability to emit the letter ℓ in state q_3. For simplicity and wlog, we assume a unique initial state, consistently called `q0`, and that any state serves as a final state. The intuitive meaning of a constraint `path(`E`,`q`,`p`,`π`)` is that there exists a partial path starting in `q0` and ending in q with probability p, and E is the remaining part of the sequence that needs to be analyzed in order to complete a full path; for ease of programming, the argument π represents this partial path in reversed order. The

```
:- chr_constraint path/4, trans/3, emit/3.

expand @ trans(Q,Q1,PT), emit(Q,L,PE), path([L|Ls],Q,P,PathRev) ==>
   P1 is P*PT*PE, path(Ls,Q1,P1,[Q1|PathRev]).

prune @ path(Ls,Q,P1,_) \ path(Ls,Q,P2,_) <=> P1 >= P2 | true.
```

Figure 7.1: A naive Viterbi-like algorithm in CHR

initial query should be stated as "`:-` *HMM*, `path(`L`,q0,1,[])`" where *HMM* is an encoding of the particular HMM and L a sequence to be analyzed.

Termination follows from the fact that the `expand` rule always reduces the length of the first argument in the involved `path` constraint. A correctness proof, which is left out due to space limitations, can be made by induction showing that `prune` will eventually remove any non-optimal `path`, but always leaves an optimal one for any prefix of the sequence, and that `expand` produces all possible extensions of an optimal `path` (for any proper prefix). This proof does not need any assumptions about the order in which the rules are applied.[1]

Let us informally analyze the time complexity of this program. For simplicity we count only the number of constraints that are created during the derivation; for a detailed analysis, we may refer to the methods of [75, 54].[2] Assuming a naive, nondeterministic semantics, we may observe derivations that are exponential in the length of the sequence to be analyzed; this is the case when, e.g., `expand` is applied as long as possible, before any application of `prune`. Our benchmarks (see appendix) confirm the exponential behaviour; interestingly, when swapping the order of the rules (i.e., `prune` first), our tests seem to indicate[3] a time complexity of $\mathcal{O}(n^4)$, although we cannot present a proof for this hypothesis. This is of course far too slow for any interesting application, and also unsatisfactory as it is known that the algorithm can run in linear time when written in an imperative language.

7.3 Fine-grained control by trigger constraints

Linear time complexity requires an optimal interleaving of the `expand` and `prune` rules, so that any `path` constraint, which will be `prune`d sooner or later, is not `expand`ed. We can sketch a class of derivations of linear size by the pseudo-code shown in fig. 7.2. As an attempt to obtain a similar flow of con-

```
seq:= $L$;
while seq $\neq$ [] do
    1) apply expand as long as possible to constraints of
form
            path(seq,$q$,$p$,$\pi$), for any $q$, $p$ and $\pi$;
    2) apply prune as long as possible;
    3) seq:= tail(seq);
```

Figure 7.2: Pseudo-code for optimal control.

trol in CHR, and we introduce what we call *trigger constraints* by means of which we can control the detailed procedural semantics of the underlying implementation. Fig. 7.3 shows an adaptation of the previous version with trigger constraints. The initial query should be stated as
"`:-` *HMM*, `path(`L`,q0,1,[]),trigger(`L`)`". During execution, the `trigger`

[1]Notice that the program of fig. 7.1 is not confluent, although intuitively very close; we consider this in more detail in section 7.5 below.

[2]Our simplified time complexity measure abstracts away the cost of search and matching performed by the CHR system.

[3]This and other estimates for time complexity are made by inspecting higher order differences for the measured runtimes.

```
:- chr_constraint path/4, trans/3, emit/3, trigger/1.

expand @ trans(Q,Q1,PT), emit(Q,L,PE),
         path([L|Ls],Q,P,PathRev), trigger([L|Ls]) ==>
   P1 is P*PT*PE, path(Ls,Q1,P1,[Q1|PathRev]).

prune @ path(Ls,Q,P1,_) \ path(Ls,Q,P2,_) <=> P1 >= P1 | true.

step @ trigger([_|Ls]) <=> trigger(Ls).
```

Figure 7.3: Viterbi with trigger constraints, version 1.

constraint will refer to decreasing remainders of the sequence, and for each such iteration provide the relevant applications of `expand` and `prune`. This preserves the logical meaning of the original program, since 1) the trigger constraints are added only in the head of the original rules, 2) new rules concerning triggers only, e.g., the `step` rule, do not unify any arguments, and 3) no derivation is stopped in a state where the original program would be able to extend the derivation. Notice that such a proof would need to refer to the operational semantics of the underlying implementation as well as to the order of the constraints in the initial query.

We sketch an analysis of the time complexity based on the operational semantics of standard CHR implementations. No rule will execute before `trigger(L)` is called in the initial query, and when this happens, `expand` will apply as long as possible for any path constraint referring to L similarly to what is expressed in line 1 in fig. 7.2, however, interleaved with `prune` (line 2). When this phase is done, the constraint `trigger(`L`)` reaches the `step` rule and mutates into `trigger(`tail(L)`)` and the process repeats for tail(L), and so on, a thus leading to derivations of linear length, as the number of steps in each such iteration is independent of the sequence length. For a fully connected HMM with k states, k^2 new `path` constraints are created in each iteration, so the length of the entire derivation becomes $\mathcal{O}(n \cdot k^2)$. However, the actual time complexity may become higher as we did not count the time for matching of list arguments in the `expand` and `prune` rules, which may, in the worst case, add another factor n to the time complexity, thus $\mathcal{O}(n^2 \cdot k^2)$. In fact, our runtime tests shown in the appendix suggest $\mathcal{O}(n^3)$ for both for a randomly generated sequence and a worst case sequence that repeats a single letter. The latter implies that the comparison of two list arguments always traverses the shortest sequence to the very end; the benchmarks indicate a huge constants factor between the two.

In order to reduce time for matching, we may add a new argument representing the length of the sequence to `path` constraints and let the trigger depend on this length only; the resulting program is shown in fig. 7.4. Assuming an implementation that applies a suitable indexing on the first argument, we would expect this to lead to a linear algorithm in the length of the sequence. However, benchmarks indicate worst case and average complexity of $\mathcal{O}(n^2)$, which we may hypothesize relates to a non-optimal search for `path` constraints. Swapping the `expand` and `prune` rules only changed the figures with a few percent.

```
:- chr_constraint path/5, trans/3, emit/3, trigger/1.

expand @ trans(Q,Q1,PT), emit(Q,L,PE),
         path(N,[L|Ls],Q,P,PathRev), trigger(N) ==>
   P1 is P*PT*PE, N1 is N-1, path(N1,Ls,Q1,P1,[Q1|PathRev]).

prune @ path(N,_,Q,P1,_) \ path(N,_,Q,P2,_) <=> P1 >= P2 | true.

step @ trigger(N) <=> N > 0 | N1 is N-1, trigger(N1).
```

Figure 7.4: Viterbi with trigger constraints, version 2, with length arguments.

To finally overcome these problems and to reach the theoretically best time complexity for Viterbi in CHR, we needed to add explicit passive declarations[4] and additional code to remove non-current path constraints; our experiments showed that both additions were necessary. Such a program is shown in fig. 7.5. We expect that a detailed analysis can prove linear complexity. Indeed, our

```
:- chr_constraint path/5, trans/3, emit/3, trigger/1, zap/1.

expand @ trans(Q,Q1,PT) # Id1, emit(Q,L,PE) # Id2,
         path(N,[L|Ls],Q,P,PathRev) # Id3, trigger(N) ==>
   P1 is P*PT*PE, N1 is N-1, path(N1,Ls,Q1,P1,[Q1|PathRev])
   pragma passive(Id1), passive(Id2), passive(Id3).

prune @ path(N,_,Q,P1,_) \ path(N,_,Q,P2,_) <=> P1 >= P2 | true.

step @ trigger(N) <=> N > 0 | zap(N), N1 is N-1, trigger(N1).
zap(N) \ path(N,_,_,_,_) # Id <=> true pragma passive(Id).
zap(_) <=> true.
```

Figure 7.5: A linear time Viterbi algorithm in CHR; passive declarations and removal of non-current `path` constraints.

benchmarks indicate that it does stay linear until sequence lengths of more than 10,000 and increases significantly from around 20,000 and upwards. While a sequence of length 10,000 can be analyzed in 30 sec., it takes 45 minutes for length 100,000. We expect that this is related to the memory being exhausted due to extreme stack sizes.

7.4 Benchmarks

The different variants of the Viterbi algorithm have been tested for a fixed, fully connected HMM with 4 states (plus a start state that cannot be re-entered) and an emission alphabet of 4 letters. We have measured runtimes as functions

[4]Passive declarations are a low-level device that suppresses certain firings of rules; for details, see, e.g., a manual for any of the major CHR versions in Prolog.

of the sequence lengths. In most cases, we test on a randomly generated sequence, considering it as "typical" or "average". Tests were made with SICStus Prolog 4.0.4 on a Macintosh 2.4 GHz Intel Core 2 Duo with 4GB RAM, and runtimes have been measured using SICStus Prolog's `statistics(runtime, ...)` device that ignores any time spent on garbage collection and other memory management tasks. Runtimes below 10 seconds were taken as average of 10 runs, whereas higher ones were measured by a single run. Space complexity was not considered.

Fig. 7.6 shows runtimes for the naive algorithm (fig. 7.1) for the two alternative ordering of its rules. The top curve to the left, for the `expand` rule first,

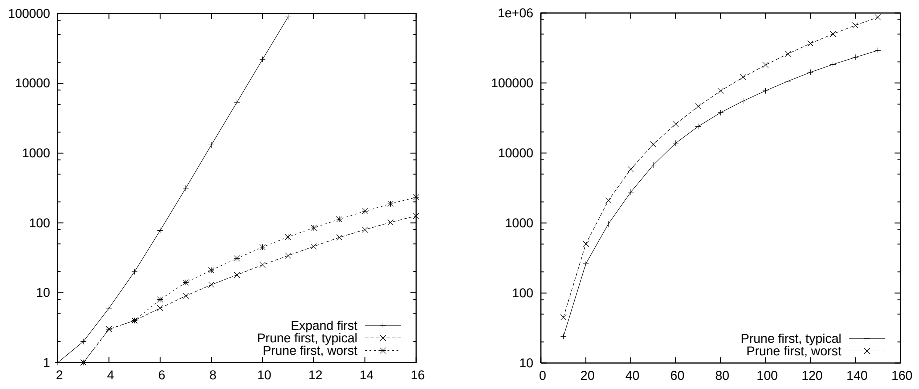


Figure 7.6: Naive algorithm with different rule orders; x-axis n, y-axis ms log-scaled.

confirms our expectation of exponential complexity. Swapping the rules so that `prune` comes first, reduces the complexity drastically. We have measured for random sequences (the "typical") plus the worst case for this algorithm, which are sequences that repeat a single letter. The right part shows the `prune` first version for the two sorts of sequences for n up to 150. An inspection of higher order differences made from the actual figures indicates that $\mathcal{O}(n^4)$ is a reasonable hypothesis for both typical and worst case; the worst case is about 3 times slower than the typical for $n = 150$.

Fig. 7.7 shows runtimes for the versions that use trigger constraints. The two top curves to the left shows typical and worst case for the version where triggers use lists, cf. fig. 7.3. In both cases, differences seem to indicate $O(n^3)$; for $n = 150$ there is a factor 24 between typical and worst case. The measurements when triggers use the list length, as opposed to the actual list, is shown as the lowest curve in the left part and continues as the top curve to the right. Differences suggest $O(n^2)$; the typical and worst case used above provides the same runtimes and swapping the `expand` and `prune` rules changes only a few percent. Finally, the right part of fig. 7.7 shows also runtimes for the fully optimized version of fig. 7.5 with passive declarations and removal of non-current path *constraints.* It stays linear up to above $n = 10,000$, and for larger values, 20,000 and upwards (not shown), the time grows higher, most likely because memory begins to be exhausted due to stack sizes. For $n = 10,000$, the time is about half a minute and for $n = 100,000$, 45 minutes.

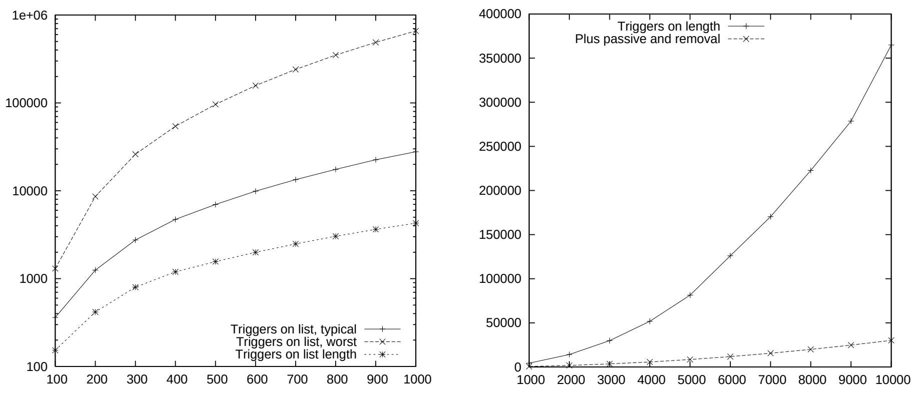


Figure 7.7: Algorithm with triggers based on sequences and lengths; x-axes n, y-axes ms log-scaled in left part, linear in the right part.

7.5 Conclusion: Methodological considerations, future and related work

We have shown an implementation of the Viterbi algorithm in CHR, starting from an abtract and concise specification expansion and pruning. Systematic extensions by triggers and other techniques lead to an implementation with ideal time complexity. The Viterbi algorithm represents a larger class of dynamic programming algorithms for which we believe that our techniques can be applied.

The naive program of fig. 7.1 is not confluent due to the fact that, when two paths exist for the same sub-sequence and with the same probability, `prune` may nondeterministically get rid of an arbitrary one of them, leading to different new states that are not logically equivalent. However, it satisfies a requirement that we may call *relative confluence* based on an application specific state equivalence relation. If, for example, two states differs only by the exchange of `path(`L`,`p`,`q`,`π`)` for `path(`L`,`p`,`q`,`π'`)`, we consider these states equivalent.

Our plans for future work include the formalization of relative confluence and to generalize known results for confluent programs [71, 1] accordingly. We believe that this can be very useful as many interesting non-confluent programs are relatively confluent. For confluent programs, it is possible to show as a general result, that the addition of trigger constraints – satisfying the requirements noticed above for the program of fig. 7.3 – preserves the logical meaning of the program as concerns its original constraints.

Additional optimizations were needed in order to obtain the best time complexity, based on detailed knowledge about the underlying machinery. Our experience in doing the exercise for the Viterbi algorithm may inspire to more advanced, automatic analyses and transformations being applied in CHR implementations in order to promote declarative programming with competitive execution times. We may also consider the ideal of a true separation of logic and control, so that we might do with the naive program of fig. 7.1, complemented by an additional control specification, which may resemble our abstract algorithm in fig. 7.2.

An attempt to obtain such a separation have been done by rule priorities [112]. The priority of each rule is expressed in terms of an arithmetic expression referring to variables in the head of the rules. While the control mechanism appears as decorations to the rules, rather than infiltrating the code as our triggers do, it is also clear that the rules need to be designed in the first place so that the rule heads actually contain the necessary information. For the Viterbi algorithm it seems obvious that the sequence length needs to be present in order to express relevant priorities. We have not tried to express the Viterbi algorithm in CHR with rule priorities, but it seems to require advanced algebraic skills to encode the desired control pattern.

There has been other work studying CHR for expressing algorithms [111]. We will emphasize [197] that gives a detailed analysis of how Dijkstra's shortest path algorithm [59] can be implemented in CHR; specifically, the authors studied the use of priority queues. Similar techniques have been employed by [36] for probabilistic abductive logic programming in CHR, by [19] for soft constraints and by [134] to express imperative control constructs in CHR. We have not seen any earlier, systematic approach for adding detailed procedural control to confluent (or relatively confluent) programs in order to get the best out of the pruning rules. The Viterbi algorithm has been formulated as a constraint problem by [41, 50], but not in CHR.

Finally, we notice that CHR is suited for describing the fundamental steps of interesting algorithms, but it is difficult to consider it as a serious implementation language at present. While the theoretically best complexity often can be reached in CHR, there is typically a huge constant factor due to the overhead in the underlying CHR and Prolog runtime systems. For the Viterbi algorithm, it is thought provoking that it can be implemented very efficiently by a handful of lines of imperative code.

Chapter 8

Modeling Repeats in DNA using Extended Probabilistic Regular Expressions

With Henning Christiansen

In Biology, Computation and Linguistics, New Interdisciplinary Paradigms. Frontiers in Artificial Intelligence and Applications., 228:55-70, 2011.

Abstract

Regular expressions is a familiar and widely used formalism which is integrated in many modern programming languages. Contemporary versions of regular expressions are typically extended variants whose expressive power goes beyond regular languages. Extended regular expressions are inherently nondeterministic and require procedural control such as backtracking. We propose a probabilistic version of extended regular expressions, where the affinity for strings and matches can be learned from examples. The procedural control semantics are replaced by a probabilistic semantics, where the possible matches are ranked by their probability and the most probable match is the one returned. In the present paper, we show how probabilistic extended regular expressions can be used to model repeats in DNA. To deal with cases where the expressive power of probabilistic extended regular expressions is insufficient, we extend the syntax to integrate external functions, which may be deterministic or probabilistic.

8.1 Introduction

Regular expressions are specifications for matching strings of regular languages, i.e. they are compact grammars.

Implementations of regular expressions are available as syntactic extensions in many popular scripting languages such as Perl, Python and Ruby in which they are widely used because they conveniently and compactly allow construction of adhoc parsers for various purposes. Because of their convenience and accessibility, regular expressions have enjoyed a lot of success in both the bioinformatics and natural language processing communities.

Similarly, Markov models — in particular Hidden Markov Models — have become ubiquitous in biological sequence analysis. It is, however, still a considerable task to specify a probabilistic model for a particular purpose, even using one of the available off-the-shelf implementations.

In this paper we propose a new formalism which is a probabilistic variant of regular expressions, combining the convenience and expressivity of extended regular expressions with the power of probabilistic models. In our probabilistic variant, choice operators are given a probabilistic semantics, such that a probabilistic regular expression — in addition to defining the language it recognizes — defines a probability distribution over all strings belonging to this language. Our probabilistic regular expression formalism embraces the operators found in extended regular expressions [29] and extends the syntax to invoke external functions.

We exemplify probabilistic regular expressions by demonstrating how they may be applied to modeling and matching repeats in DNA. Repeats in DNA play a variety of biological roles and can have major phenotypical effects. This makes repeats an important area to study.

8.1.1 Organization of the paper

Section 8.2 introduces regular expressions and extended regular expressions. Section 8.3 describes our probabilistic variant of extended regular expressions. Section 8.4 describes how probabilistic extended regular expressions may be used to model repeats in biological sequences. Section 8.5 presents related work and section 8.6 presents conclusions and perspectives.

8.2 Regular Expressions

Regular expressions date back to Kleene [109] where they were initially used as a formalism to describe a language of inputs to nerve cells. Regular expressions transited from being a mathematical curiosity to being a practically applicable tool for sequential pattern matching due to a compiler written by Ken Thompson [211]. Regular expressions and automata have had a large influence on compiler theory and have been part of computer science curriculum almost as long as the field has existed [49]. For a large part, regular expressions have been popularized through the Unix grep utility [98] and via integration in popular editors and scripting languages.

The syntax of regular expressions have been relatively standardized through the POSIX specification [157], but there are many flavors of regular expressions each with their own idiosyncrasies. Besides syntactic differences there are also semantic differences, and the various implementations do not all have the same expressive power. A distinction which we emphasize is the difference between *regular expressions* which can be compiled to finite automata and *extended regular expressions* which includes backreferences. We borrow this terminology from [29]. It should not be confused with the POSIX distinction between *basic* and *extended* regular expressions. By our definition, both POSIX classes are extended regular expressions.

The brief account of regular expressions provided in the following is based on the syntax and functionality of our own implementation, which is in turn based on the POSIX specification [157].

In the simplest form, regular expressions are sequences of symbols to be matched in the order they are specified. Such strings may, however, contain (possibly parenthesized) subexpressions and special symbols called operators. These operators make it possible to match a large set of sequences using a relatively compact regular expression. The operators are divided into wildcards, repetition, alternation, anchors and backreferences. Regular expressions are composeable, i.e. the concatenation of two regular expressions is itself a regular expression.

Repetition is usually specified using the Kleene star `*` which implies that the preceding expression should be matched zero or more times. For instance, the expression `a*` matches $\{\epsilon, a, aa, aaa, \dots\}$, where ϵ is the empty string.

Variants include the `+` operator which implies that the preceding expression should be matched one or more times and the `?` operator which implies that the preceding expression should be matched zero or one time. Finally, it is also possible to specify a custom range of the number of times an expression should be matched with the `{m,n}`, where `m` is the minimum number of times and `n` is the maximum.

Wildcards are substitutes for a range of different characters. The dot wildcard operator matches any symbol. Range operators, encapsulated by brackets, can be used to specify a range of different symbols to be matched. For instance, `[a-z0-9]` matches a lower case letter or a digit.

The alternation operator — embodied by the bar character `|` — implies matching either the regular expression to the left of the bar or the regular expression on the right. All variants of wildcards can be specified using the alternation operator and all repetition operators can be expressed using the Kleene star and the alternation operator.

In addition to this, *anchors* include `^` which matches any prefix and `$` which matches any suffix. Note that `^` can only occur in the beginning of a pattern and `$` can only occur at the end of a pattern.

8.2.1 Extended Regular Expressions

Parenthesized sub-expressions that do not have an operator following the expression are called capture groups. They are normally used to indicate which parts of the match of a regular expression should be extracted from the sequence being matched.

The matches captured by such a group can also be referenced from within the regular expression using a backreference. A backreference is an expression `\\n` where `n` is an integer referring to the `n`'th capture group. The backreference serves to match the same characters that was matched by the capture group.

This extension to regular expressions is essential to model repeats. Repeats, e.g., of the kind rr, belong in the of group context-sensitive of languages. We can model a limited set of such languages using the backreference operator. For instance, the language rr can expressed using the expression `(.*)\\1`. The first part `(.*)` specifies a capture group; The substring that this capture group matches must then be matched again when we encounter the backreference `\\1`. The addition of capture groups and backreferences does not, however, add capability to recognize all context-free languages [29].

The cost of increased power is exponential rather than linear computational complexity in the worst case. Regular expression engines with support for these operators usually implement them using backtracking which potentially explores an exponential number of solutions.

With capture groups and backreferences, the declarative specification of regular expression may become ambiguous and this necessitates procedural control — i.e. the order in which matches are tried becomes relevant.

Usually, a deterministic regular expression engine provides you with only *one* match (*which* match depends on the semantics). Most engines have greedy semantics where sub-expressions match as much of the input sequence as possible before attempting to match the rest of the string with the next sub-expression. Upon failure, the engine backtracks only as little as necessary. Hence, in the matching returned by such an engine, the left-most sub-expressions will have matched as much of the input sequence as possible. Oppositely, in engines with reluctant semantics, left-most sub-expressions match as little of the input string as possible.

This can make it very difficult to understand the precise meaning of a regular expression, since the expression may can be quite complex and the implementations have different procedural semantics which are often not well-documented.

8.3 Probabilistic Extended Regular Expressions

In addition to defining language membership, a probabilistic regular expression (extended or not) assigns a probability to each string belonging to the language it defines. The probability of a string is defined as the sum of the probabilities for all possible ways to match that string. In the following, a *match* is taken to mean a particular way to match a string. By giving each match a probability, probabilistic regular expressions provide a way to resolve non-determinism of regular expressions, i.e. to rank ambiguous matches of the expression in a string. This is useful for queries beyond mere language membership, e.g., to extract the part of a string which is matched by a capture group.

In the face of non-determinism of a (non-probabilistic) regular expression, the procedural semantics of implementation determines what part of the string is extracted as matching the capture group. This implies that the user needs to know this procedural semantics and craft the regular expression in some ingenious way, exploiting the procedural semantics to extract the relevant part of the string.

Probabilistic regular expressions present an alternative to the usual procedural semantics by making it possible to choose between ambiguous matches by selecting the most probable match for a sequence. For this to make sense, the probabilities must be assigned in a sensible way, e.g. maximizing likelihood of observed examples of strings of the language.

An other benefit of the probabilistic semantics is that is allows random sampling of strings of the language it represents even if the language is infinite. Such sampling does not fit well with a traditional deterministic semantics, where strings much be generated in a particular order.

Any (non-extended) regular expressions can be compiled to a Deterministic

Finite Automaton (DFA). The probabilistic variant of the regular expression can be compiled to a first order Markov chain by labeling each of the transitions in the DFA with probabilities. A Markov chain is a probabilistic automaton which is a characterization of a process with the Markov property; Markov chains have limited (fixed-order) memory of visits to past states, e.g. in a first order Markov chain, the probability of the next state depends solely on the current state.

Since the matching of backreferences requires memory for the capture group visited in the past, extended regular expressions cannot be represented as simple Markov chains, but require unbounded memory of the past.

8.3.1 Implementation

Our implementation of probabilistic regular expressions translates a given expression into a recursive data structure which is used to guide a probabilistic grammar. The implementation is based on the probabilistic logic programming language PRISM [175], which is an extension of Prolog with random variables and efficient algorithms for probabilistic inference.

The regular expression is compiled to a tree structure representation using a Definite Clause Grammar [153]. An internal node in the tree structure corresponds to an operator of the regular expression and the children of the node are the sub-expression(s) to which the operator apply. The leaves of the tree are either concrete symbols or *epsilons*, i.e., matching the empty string. The nodes contained in the internal grammar representation is alternation, Kleene star repetition (augmented with counters), capture groups and backreferences. All other operators are translated into these prior to constructing the internal representation. Choice operators (alternation and repetition) are labelled with a unique identifier in the internal representation which is later used to associate it with a particular random variable governing the choice represented by the operator.

The internal representation is used to guide a probabilistic grammar implemented in PRISM. Inferences with this probabilistic grammar are performed using the mechanisms provided by the PRISM system.

The choice operators are implemented using special rules, in which probabilistic choices are made in the rewriting process.

8.3.2 Probabilistic Semantics

Probabilistic choices are assumed to be independent and hence the probability of a *match* is the product of the probabilities of the individual probabilistic choices made.

A Kleene star is interpreted as a boolean probabilistic choice, either matching the preceding expression and performing the choice again; or matching the empty string ϵ:

$$\textsc{expr}\star \leftarrow \begin{cases} \epsilon & \text{with probability } P(\epsilon) \\ \textsc{expr}\ \textsc{expr}\star & \text{with probability } 1 - P(\epsilon) \end{cases} \tag{8.1}$$

This defines a geometric probability distribution over the number of times that EXPR is matched and implicitly a geometric length distribution over the strings matched by EXPR⋆.

Similarly, an alternation is also modeled as a boolean probabilistic choice:

$$\text{EXPR1|EXPR2} \leftarrow \begin{cases} \text{EXPR1} & \text{with probability } P(\text{EXPR1}) \\ \text{EXPR2} & \text{with probability } 1 - P(\text{EXPR1}) \end{cases} \quad (8.2)$$

Note that this rule recursively covers expressions with multiple alternations, e.g. `a|b|c` which may be interpreted as `a|(b|c)`, and that such expressions defines valid probability distributions, i.e. where the probabilities of alternatives sum to one.

Backreferences are matched symbol by symbol in a non-probabilistic way as in usual extended regular expressions.

Note that in the usual interpretation of regular expressions, the two expressions `(a|b|c)*` and `(a|b)*(a|b|c)*` would be equivalent, i.e., they would reduce to the same minimal DFA. In the probabilistic interpretation they are different: While both of the models are able to generate the same strings as the DFA, they define distinct distributions over these strings. Each *occurrence* of a choice operator is modeled using a *distinct* random variable.

8.3.3 Inference with probabilistic regular expressions

Several forms of probabilistic inference are available using the underlying PRISM system.

Prior to performing any inference, the regular expression must be compiled. In our implementation, this is done using the goal `regex_compile(`**Regex**`)`, where **Regex** is a regular expression. In the inference examples below, we assume that the regular expression `'^(.*)(\\1*)$'` has been compiled.

Training is the process of learning the probabilities associated with the probabilistic choice variables from examples. In the supervised setting, the training algorithm is provided with a few examples of string that the probabilistic regular expression matches and specification of what parts of the string matches which capture groups.

For instance we train our regular expression on the simple sequence `tatatata`:

```
regex_learn([tatatata], [[ta,tatata]]).
```

The predicate is given a list of strings as first argument and a corresponding list of expected matches as the second argument. In this case, the first of these matches corresponds to the repeat unit and the second corresponds to what is matched by the group with the backreference. Note that the expression is ambiguous; the matches list could also have been `[[tata,tata]]` in which case the probabilities learned would be different.

We can also train in an unsupervised setting, where we do not assume knowledge about exactly *how* the expression matches, but only that it matches, by calling the `regex_learn` with only the strings as argument.

PRISM provides several learning algorithms including the well-known EM algorithm which derives probabilities as to maximize the likelihood of the examples.

Matching using probabilistic regular expressions is done using the generalized Viterbi algorithm provided by the PRISM system. Given a pattern and a string, the Viterbi algorithm selects a particular matching such that the product of the probabilities of probabilistic choices made is maximal. Matching is done by calling one of the variants of the `regex_match` predicates. For instance,

```
regex_match(tatatata,M,P).
```

will unify `M` to a list containing the most probable matches for the string, e.g., either the list `[ta,tatata]` or the list `[tata,tata]` depending of the parameters of the model.

Sampling which means generating matching strings of a specified regular expression is usually not possible with non-probabilistic regular expression implementations. The obvious barrier is the procedural control of the generation process. Regular expressions may match an infinite number of different strings. Which of these strings should be generated? With probabilistic regular expressions, this issue is addressed by deciding what string to generate according to its probability. Repeated sampling will result in a collection of different strings from the language, which, in the limit, are representative of the probability distribution of the probabilistic regular expression. Sampling is done by calling the `regex_sample` goal:

```
regex_sample(S,M)
```

This will unify `S` to a repeat sequence matching the regular expression band `M` to a list of matches of the capture groups of the regular expression on that string. It is also possible to do constrained sampling, where the matches `M` are given. Notice, that in this case, the generation process can fail and several attempts may be necessary to sample a string.

8.4 Modeling Repeats in Biological Sequences

Finding repeats is an important task in biological sequences analysis. There are countless variations of the problem depending on the type of repeats, the distance between repeats and any combination of such constraints. In addition to this, there are several classification schemes with overlapping terminology.

The term *repeat* is used here to mean multiple — in some sense similar — sub-sequences, which we call *repeat units*, occurring in a sequence.

To avoid confusion, we briefly mention the *satellite* terminology which seems quite common in biology textbooks, e.g. [24], although in the following the variations of repeats we address is based mostly on the categorization in [103] where that terminology is avoided. The term *satellite DNA* is refers to long repeated sequences of the DNA that when centrifuged form small satellite bands in around the main band, since the repeats have distinct, but monotone GC content and hence weight. Mini- and microsatelittes are usually too small to form visible centrifugation bands, but the name has stuck. Microsatellites have repeat units of less than 13 bp and total length of less than 100kbp and minisatellites are longer repeats of up to 20kbp with repeat units up to 25 bp [24].

A *simple repeat* is a oligonucleotide pattern repeated consecutively in a stretch of DNA. The terminology used describe these is somewhat overloaded, since they are also called tandem repeats, short sequence repeats or microsatellites.

Simple repeats may occur through a mutational event — often, a replication slippage — called tandem duplication, in which the pattern is copied a number of times. The mechanisms leading to such mutations are still subject to study. Simple repeats have a diverse number of effects in various organisms [215]. Tandem repeats in genes are associated to a range of diseases, and repeats in non-coding DNA play key roles in gene regulation. In bacteria, they are known to mediate frameshifts.

Tandem repeats, for repeat units of a certain length, may be identified using the regular expression, where we assume a repeat unit of length two:

Regex 1. `(..)\\1*`

The first capture group identifies a dinucleotide and the second capture group, which contains a backreference to first, may be repeated zero or more times, e.g., this pattern recognizes strings like `CT` and `TATATATA`.

The simple tandem repeat is a special case of a *direct repeat* – a repeat which occurs on the same DNA strand. We can model the general case of non-interrupted direct repeats with arbitrary oligonucleotide lengths using the expression:

Regex 2. `(.*)\\1*`

In this expression the first capture group matches a substring of any length and the backreference in the following capture group matches repetitions of the matched sequence.

Direct repeats may also be separated by spacers. Spacers may be any length of bases and may vary between repeats. A spacer is a modest addition to the previous expression:

Regex 3. `(.*)(.*\\1)*`

There is a distinction between *local repeats* and *global repeats*. In a local repeat the spacer is short and in a global repeat it is much longer, say thousands of bases. However, the distinction somewhat is arbitrary as there are no agreed upon limit on the length of the spacer in local repeats. We may then choose such a limit ourselves; the following expression looks for global direct repeats, where we take global to mean separated by at least a 1000 bases:

Regex 4. `(.*)(.{1000,1000}.*\\1)*`

The expression `.{1000,1000}` matches 1000 consecutive bases and is followed by the `.*` expression which matches any number of bases following that.

Repeats may also be be inverted, e.g. for instance:

$$5' \dots GACTGC \dots GCAGTC \dots 3'$$

The reverse of the complement[1] of the first substring occurs later in the string. Note that this is inherently palindromic and require the power of a context-free grammar. Regular expressions extended with backreferences only cannot

[1]In the double-helical DNA structure A pairs with T and C pairs with G. Complemented means substituting the original base for the base it pairs with.

match such repeats if they are of arbitrary length. If the length of the repeat unit is known in advance, e.g. say four, then this limited palindromicity can be modeled using a somewhat clumsy expression:

Regex 5. `(.)(.)(.)(.).*\\1\\2\\3\\4`

This, however, quickly becomes bothersome and it would be even more bothersome, although possible, to construct an expression which also deals with the situation where second half of the repeat is complemented.

As a way to add additional expressive power to the probabilistic regular expression with only minimal syntax extensions, we have added convenient way for users to integrate their own functions in regular expressions. The function integration operator `#{...}` calls a user defined function. It can, for instance, be used to reverse and complement a sequence:

Regex 6. `(.*).*#{revcmpl(\\1)}`

In this expression the `revmpl` goal is called using the match represented by `\\1` as argument. The function returns the reversed and complemented sequence, which is matched in its place.

A not so common type of repeat is mirror repeats — also called everted repeats — in which the repeat element is copied in the $3' \rightarrow 5'$ direction rather than the $5' \rightarrow 3'$ in which DNA is processed. Such repeats may lead to biologically important tertiary structures. Naturally, we can model such repeats like the usual types of repeats by reversing the sequence.

A complication of repeats in biological sequences is that they do not always occur in integer multiples [10]. For instance, a series of repeated elements may be flanked by a prefix or suffix of the repeat unit. Such half-repeats can also be straight-forwardly modeled using probabilistic regular expression:

Regex 7. `(.*)(.*)(\\1\\2)*\\1?`

In this expression we assume the repeat unit to consist of two parts, each identified by a separate capture group. The third parenthesized expression matched the repeat unit a number of times and finally the fourth sub-expression `(\\1)?` matches a possible flanking prefix.

8.4.1 Approximate Matching

All the examples given above assume that the sequence of repeats is without errors. In real biological data, mutations, insertions and deletions do occur and we would be like to be able to model repeat sequences containing such errors.

We accommodate the need for modeling this by using the function integration operator to call a *non-deterministic, probabilistic* function `noisy`, e.g.

Regex 8. `(.*)#{noisy(\\1)}*`

The function `noisy` probabilistically maps the input argument `\\1` to a "noisy" sequence in which symbols of `\\1` may be mutated, deleted and additional symbols may be inserted. The probabilities of point mutations, deletions and insertions are learned from the examples provided for training the entire probabilistic regular expression.

```
repeat element                              spacer
gttcactgccgtacaggcagcttagaaa   aacctaccgtcttggctagcggttgcagcgaac
gttcactgccgtacaggcagcttagaaa   ggaacaatcttgcaaaggctgtgaaagttggc
gttcactgccgtacaggcagcttagaaa   ttcacaggtaacatactccacccaccat
gttcactgccgtacagacagataaaatg
```

Figure 8.1: The CRISPR NC_009800_1, region 985984-986188.

```
 Folding a (Free energy -7.90)    Folding b (Free energy -8.00)
 guucacugccguacaggcagcuuagaaa     guucacugccguacaggcagcuuagaaa
 .....(((((.....)))))........     .(((.(((((.....)))))....))).
```

Figure 8.2: Minimal energy foldings of the CRISPR repeat element. Note that u (Uracil) is the RNA equivalent of the DNA t (Thymine).

Note that without constraints on the maximal number of mismatches, insertions and deletions this expression matches any sequence. This leads to an infinite number of ways to match the element which is obviously unfeasible. To avoid this, the `noisy` function implicitly limits the maximal number of each type of mutation be equal to the length of the element.

8.4.2 Example: CRISPRs

CRISPRs, short for Clustered Regularly Interspersed Short Palindromic Repeats [200], are regions of bacterial or archaeaic DNA with short direct repeats, with repeat elements of 24-28 basepairs and interupted by a spacer of around 30 basepairs. The repeat element itself exhibit some palindromic structure, i.e. contains a (noisy) inverted repeat.

The function of CRISPRs is to remember past exposure to exogenous elements such as phages; A *CAS* protein intercepting viral DNA creates a novel spacer and inserts it into to the genome at a CRISPR locus. Once this CRISPR is transcribed as RNA it interacts with proteins which target and inactivates the viral DNA.

We consider creating probabilistic regular expression model of a particular case of a CRISPR in a the *HS strain* of *Escherischia Coli* (RefSeq: NC_009800). The CRISPR region we consider is located at 985984-986188 and according to the CRISPR database [82] it contains three inserted spacers and the repeat element is 28 bases long. It is shown in figure 8.1. The online RNA fold server [232] was used to predict two secondary structures for the repeat element, which are shown in figure 8.2.
From the two foldings we can observe two distinct inverted repeats.

The palindromic repeat element We model the repeat element with the expression

Regex 9. `.*(.*).*#{revcmpl(\\1)}.*`

This only matches one inverted repeat, i.e. a hairpin structure. The inverted repeat may be interrupted by bulges — short regions that do not basepair — as is evident from folding b in figure 8.2. This expression is not specific enough to capture inverted repeats with bulges, but it is general enough to match all CRISPR repeat elements since an inverted repeat with a bulge may also be seen as two separate inverted repeats. For instance, we can model an inverted repeat with at most one bulge by slightly extending the expression of regex 9:

Regex 10. `.*(.*).*(.*).*#{revcmpl(\\1)}.*#{revcmpl(\\2)}.*`

A different — and perhaps more elegant way — is by composing the `noisy` function and the `revcmpl` function:

Regex 11. `(.*).*#{noisy(revcmpl(\\1))}`

This expression matches hairpins with an unbounded number of bulges.

Unfortunately, in our probabilistic regular expression implementation, matching and parameter learning with either expression 10 or expression 11 is currently is only feasible for very short sequences. We are working on optimizations to improve this.

Parameter learning Using supervised learning we can incorporate what we have observed from the minimal energy foldings. We train regular expression 9 on the sequence and the repeat element of each of the two distinct inverted repeats observed from the minimal energy foldings as captures.

```
S = gttcactgccgtacaggcagcttagaaa,
regex_learn([S,S],[[ctgcc],[ttc]]).
```

We verify that the learned parameters of the model corresponds to our expectation, i.e. that it matches the repeats that we expect. With the following goal, we extract the five most probable matchings of probabilistic regular expression:

```
| ?- n_regex_match(5,gttcactgccgtacaggcagcttagaaa,M,P).
M = [ctgcc] ,  P = -41.257125721684069 ?;
M = [ttc] , P = -42.609269626583213 ?;
M = [ctgc] , P = -43.014481540778561 ?;
M = [tc] , P = -43.861233634891093 ?;
M = [tgcc] , P = -44.173164286287403 ?;
```

It can be observed that the repeat element `ctgcc` — which occurs in both minimal free energy foldings — is also the most probable repeat element of according to our model. The second match `ttc` is the other repeat element observed in the folding b. The last three matches are subsequences of the repeat elements and represent suboptimal matches from our perspective. Note that the probabilities `P=...` are given in log-space. Matching with an untrained version of the probabilistic regular expression will give a different ranking of matches and assign significantly lower probabilities to the best matches.

If we do not have an authoritative source of correct matches such as the minimal free energy foldings then we need to consider unsupervised learning:

```
regex_learn([gttcactgccgtacaggcagcttagaaa]).
```

The resulting matches are shown below:

```
| ?- n_regex_match(5,gttcactgccgtacaggcagcttagaaa,M,P).
M = [ctgcc] , P = -33.642109806576016 ?;
M = [ctgc] , P = -36.4356138969029 ?;
M = [ctg] , P = -39.229117987229785 ?;
M = [ct] , P = -40.945418474610648 ?;
M = [ctgcc] , P = -41.020430465464123 ?;
```

In this case, `ctgcc` is given much more higher probability and the `ttc` element present in the second best folding is not among the best matches. Note that the learning procedure (Expectation Maximization) does not guarantee that the parameters are not only a local maximum and hence the matches are also not guaranteed to be optimal even the matching procedure (Viterbi) guarantees the match with highest possible probability given the parameters.

The CRISPR repeat element containing inverted repeats is only part of the story; A CRISPR is a sequence direct repeats of such elements separated the inserted spacer elements. Disregarding the palindromicity we can use an expression like expression 3. However, usually it would be of interest to the biologist to extract the spacers between repeat elements. So instead we consider the rather general expression for matching a CRISPRs spacer and its surrounding repeat elements is:

Regex 12. `(.*)(.*)\\1`

This expression serves as a very general description and does not attempt to describe any particular CRISPR. Given this (perhaps too) general description we may attempt to learn the parameters characterizing a particular CRISPR or set of such. Subsequently we may attempt to find (match) similar CRISPRs using the parameters learned from using our examples. However, even though parameter learning will increase the probability that the expression matches the particular type of CRISPR, such an overly general model is likely to have spurious matches. In fact, it matches any direct repeat — for instance a short tandem repeat — and even any sequence, but will assign lower probability to such matches.

Regex 13. `(.{24,28})(.{25,35})\\1`

Note, that here we have arbitrarily chosen to interpret "around 30" as the range $[25-35]$. The resulting expression is more specific as it for instance does not match spurious short tandem repeats and similar unwanted matches. Adding length constraints reduces the number of possible matches. This has two beneficial effects; It potentially reduces running time of inference significantly and since the probabilistic parameters do not have to include probability mass associated with irrelevant solutions, a learning procedure may fit the parameters better. Given that we know the length of the CRISPR we can make a more specific repeat element matcher — i.e. we can specialize the first capture group to instead say `.{28,28}` or we could use 28 dots in its place. These type of constraints are different; The first one assume a common distribution of the symbols in the repeat element whereas each dot of the second method is modeled with a distinct distribution.

After training expression **??** we can match and extract spacer elements, e.g.:

```
| ?- regex_match(...ctgccgtacaggcagcttagaaa...,M,P).
M = [gttcactgccgtacaggcagcttagaaa,
    aacctaccgtcttggctagcggttgcagcgaac]
P = -93.480553848575966
yes
```

Note that neither expression 12 or **??** is capable of matching the last spacer because the last repeat element is noisy. Hence, is necessary to model such sequences with an expression using the noisy function:

```
(.*)(.*)#{noisy(\\1)}
```

8.4.2.1 Why not a complete CRISPR model?

We have sketched how elements of a CRISPR may be modeled using probabilistic regular expressions but no expression for a complete model has been devised.

It is not possible to create an extended regular expression capable of matching and extracting all spacers from CRISPRs of arbitrary size, since the number of capture groups within a regular expression bounded by the size of regular expression.

A similar observation may be made about expression 10 which uses somewhat clumsy way of modeling only a single bulge in a hairpin structure. Using the `revcmpl` function we are not able to model a hairpin loop with unbounded number of bulges. This is a context-free language. So even though we add the capability of modeling palindromicity with the `revcmpl` function, it does not have the capability to model certain context-free languages. Composing `revcmpl` with the non-deterministic `noisy` function as done in expression 11 adds the power to model an unbounded number of bulges, but does not enable us to capture the repeats or bulges in the hairpin.

8.5 Related Work

8.5.1 Probabilistic regular expressions

Ross [167] presents an approach for evolving stochastic regular expressions using genetic programming. The semantics of the stochastic regular expressions they use are similar to those presented in this paper. A notable exception is that their formalism is restricted to regular languages, whereas ours pragmatically embrace some more expressive languages.

The stochastic regular expressions are implemented as an interpreter using the Prolog based DCTG grammar formalism [3]. The implementation augments the grammar with a stochastic semantics and provides a means of inferring the probability of a match of a sequence. Other forms of probabilistic inference, such as finding an optimal match or learning from examples, are not explored in the paper. An interesting feature is that the formalism supports the probabilistic choices in addition to the usual non-deterministic regular expression operators.

An other probabilistic regular expression language is defined in [76] with the purpose of modeling and reasoning about discrete-event systems. Their language is a regular automaton based language, similar to that in [167].

8.5.2 Pattern Matching in Biological Sequences

There are many tools for matching biological sequences using regular expressions, but to our knowledge, none of the surveyed tools employ probabilistic regular expressions.

PatScan [60] employs a special kind of regular expressions with operators for specification of base-pairing. The pattern syntax is a bit different from usual regular expression syntax, but seems quite powerful as it allows the matching some context-free and context-sensitive languages. Additionally, it supports approximate matching and the number of allowed substitutions, insertions and deletions may be specified for individual sub-expressions.

The tool tagc [130] is designed to match motifs in sequences. It uses a combination of regular expression patterns, proximity rules (which specified exact distance between matches) and logical rules which model more complex relationships between several matches. The semantics of these rules seems to be similar to — but perhaps more powerful than — the `{n,m}` operator in regular expressions.

Due large size of genomic data, the computational complexity of finding repeats is an important issue. Biogrep [102] takes the approach of utilizing the well-known GNU grep engine in parallel to increase performance. A considerable effort has be invested in building specialized efficient tools for finding repeats, e.g. [15, 110, 81]. Such tools, however, sacrifice flexibility in terms of modeling capabilities in order to gain high efficiency. In particular, there are very efficient tools and algorithms for finding small tandem repeats, and there are databases of such repeats for various organisms, e.g. [216]. Prosite [97] is an other popular database containing known protein motifs and which provides limited regular expression search for such motifs. The database also contains repeat motifs and regular expressions for searching for these. An other online repeat database for specific for CRISPRs is CRISPRdb [82].

8.6 Conclusions and Future Work

We have presented an account of how repeats in DNA can be modeled using probabilistic extended regular expressions and provided a flavor of our implementation of this formalism. Through the underlying PRISM system we gain access to advanced probabilistic inferences on probabilistic regular expression models of repeats.

It has been shown that regular expressions extended with backreferences comprise a proper subset of context-sensitive languages which is disjoint from context-free languages [29]. The limited context-sensitive power of extended regular expressions is sufficient for modeling direct repeats, whereas inverted repeats require context-free power and cannot be modeled without resorting to an extension of the formalism. In our implementation, a means to integrate external functions provides the extra expressive power. Non-deterministic and probabilistic functions can also be integrated, and we have demonstrated how this can be used to support approximate matching.

We have demonstrated using examples that probabilistic extended regular expressions are applicable to model biological sequence phenomena. That does not imply that they are the best suited for the job. They can be likened to a swiss-army-knife – it is applicable, practical and suffice for a lot of tasks, but is almost never the optimal tool. The appeal of regular expressions is in their succinctness and availability.

Probabilistic extended regular expressions, and probabilistic grammar models in general, add an dimension beyond language membership that models affinity for strings in the language. This extra dimension is useful to quantify effects that we do not yet understand or which is too complex to model exactly. It allows us create models which abstract away the uncertainty of the details of the biological processes, while still providing useful results.

The application to modeling and matching CRISPR is usable, but it is terribly inefficient compared to existing techniques. In particular matching of noisy CRISPRs presents some challenges with regard to efficiency. Our current implementation — which is a prototype — has many unexploited potential optimization possibilities. We are investigating techniques for making this matching more efficient. One of the techniques that we are exploring is to infer constraints from the regular expression and the input string which may be used to further prune the search space early on.

The current viterbi-based matching scheme finds a best match and provides a probability for that match. The scheme does not attempt to estimate the significance of finding a match with some probability or a means to select a sensible threshold. In the future we plan to incorporate ways of estimating the significance of a match based on comparisons with null models.

An application that would be interesting to explore is to use sampling to generate test data other repeat finding methods, as done for gene finders in [38]. Such an approach could potentially be used to uncover weaknesses in existing approaches and provide insights into what kind of repeats they may have missed.

Acknowledgement This work is supported by the project "Logic-statistic modelling and analysis of biological sequence data" funded by the NABIIT program under the Danish Strategic Research Council.

Chapter 9

Bayesian Annotation Networks for Complex Sequence Analysis

With Henning Christiansen, Ole Torp Lassen and Matthieu Petit

In Technical Communications of the 27th International Conference on Logic Programming (ICLP'11), 2011.

Abstract

Probabilistic models that associate annotations to sequential data are widely used in computational biology and a range of other applications. Models integrating with logic programs provide, furthermore, for sophistication and generality, at the cost of potentially very high computational complexity. A methodology is proposed for modularization of such models into sub-models, each representing a particular interpretation of the input data to be analysed. Their composition forms, in a natural way, a Bayesian network, and we show how standard methods for prediction and training can be adapted for such composite models in an iterative way, obtaining reasonable complexity results. Our methodology can be implemented using the probabilistic-logic PRISM system, developed by Sato *et al*, in a way that allows for practical applications.

9.1 Introduction

Analysis of DNA is an important example of a complex sequence annotation task which has motivated our approach. The sheer size of data instances and the degree of ambiguity in such tasks pose great challenges for efficient probabilistic analysis. Furthermore, most systems for DNA-analysis used in practice are implemented in low-level programming languages, optimized and tweaked for very specific procedures, thus leading to systems with an unclear semantics and lack of flexibility for the modeling part. A possible shift to using probabilistic-logic systems and languages provides obvious benefits in terms of clear semantics and flexibility, but also introduces potential problems concerning complexity and scalability. We present here a modular approach, in which complex probabilistic-logic models are defined in terms of separate sub-models, each representing a particular interpretation (or "signal") of the input data to be analyzed. The dependencies among the results of analyses performed by these sub-models are described in terms of edges in a Bayesian network. This allows for an implementation based on incremental application of standard methods for prediction and training, one sub-model at a time, thus possibly leading to acceptable complexity. We refer to such modularized models for sequence analysis as *Bayesian Annotation Networks.* We demonstrate an implementation based on PRISM [175], which is a probabilistic extension of Prolog.

9.2 Probabilistic Annotation Models

Probabilistic-logic models for sequence annotations will be presented in two steps, first the logical part, and then probabilities are added. Notice also, that we abstract away the details of any actual modeling language and the format of probability parameters.

Definition 9.2.1. *An* annotation program*, or just* a program*, is a logic program prog, that defines a set of atoms, each of the form:*

$$prog(s, a, parents),$$

where

- *s is called* the sequence*, and represents the data sequence to be annotated by the program.*
- *a is called an* output annotation*, and*
- *parents represents zero or more* conditioning annotations.

The name "*parents*" anticipates the introduction of Bayesian Annotations Networks in section 9.3 below. They represent annotations produced by other sub-models, serving as conditions for the analysis associated with *prog*.

Definition 9.2.2. *A* probabilistic annotation model

$$m = \langle prog, \theta \rangle$$

consists of a probabilistic annotation program prog and a parameter θ*. The parameter is element of some data domain which is not specified further, but which gives rise to a well-defined conditional probability distribution for atoms* $prog(s, a, parents)$ *as follows:*

$$P(a \mid s, parents, \theta)$$

The intuition is that θ that associates probabilities to the detailed choices made within *prog* to produce the output annotation a, given a specific sequence s and parent annotations. Notice that our framework captures also analyses that are not necessarily written in a probabilistic-logic language. Notice that our framework captures also analyses that are not necessarily written in a probabilistic-logic language.

Definition 9.2.3. *A* deterministic annotation model *is a program*

$$prog(s, a, parents)$$

where, for specific sequence s^0 *and* $parents^0$*, there exists exactly one output annotation* a^0*, i.e.,*

$$P(a^0 \mid s^0, parents^0, \theta) = 1,$$

where θ*, in this case, refers to an (empty) parameter which is ignored.*

The empty parameter is included for uniformity of notation only. A deterministic annotation model with empty parents may represent an analysis provided by an external tool that, e.g., searches for similarities in a database of related sequence data.

9.3 Organizing Annotation Models as a Bayesian Network

Our overall idea for prediction is to evaluate one model at a time, fix its output annotation to a single "best" one which, then, is used as parent for subsequent analyses. This is very similar to the way forward analysis takes place in Bayesian networks, which we thus take as our central paradigm for putting sub-models together to a whole. A Bayesian Network (BN) is defined as a directed acyclic graph as follows [170].

- Its *nodes* are random variables.
- An *edge* from node A to node B indicates that B is directly dependent on A, and A is called a *parent* of B; the notation $parents(B)$ refers to the sequence of parent nodes of B.
- Each node A has an associated *conditional probability distribution*, *CPD*, $P(A \mid parents(A))$.

For many applications of BNs, the CPDs are given in the form of tables, but since the random variables in our case range over huge sets of alternative annotations, this is infeasible, and we use probabilistic models instead.

Definition 9.3.1. *A* Bayesian annotation network *(BAN) is a set of probabilistic annotation models* $\{M_i \mid i = 1, \ldots, n\}$*, with* $M_i = \langle m_i(s, a_i, parents_i), \theta_i \rangle$*, numbered in such a way that* $parents_i \subseteq \{a_1, \ldots, a_{i-1}\}$*.*
The model M_n *is a designated* top model*, and it is assumed that the parent relationship induces a path from any other* M_i *to* M_n*.*

A BAN in itself is not a BN, but it induces a BN in the following way.

- Nodes are labelled a_i, $i = 1, \ldots, n$ and s.
- Whenever $a_j \in parents_i$, there is an edge from a_j to a_i, and there is an edge from s to any a_i.
- The CPD associated with a_i is given by the model M_i, i.e., $P(a_i \mid s, parents_i, \theta_i)$.

For ease of terminology, we refer to a suitable set of annotation programs as a BAN, when below, we talk about training a BAN, i.e., finding parameters such that it actually becomes a BAN, as per the present definition. When presenting a BAN as a graph, we typically leave out s and the n edges going out from it. When doing predictive inference below, the sequence is always fixed, so we can leave it out, assuming instead a particular BN for each sequence.

9.4 Predictive inference

Predictive inference refers here to the process of identifying a best proposal for top output annotation that characterizes a given sequence. The fundamental assumption when using probabilistic models is that quality of a solution is intimately coupled to its probability, in other words, we should be searching for a top output annotation with a relatively high probability, ideally the one with highest probability.

Below, we give first a precise, declarative characterization of the best top output annotation, and then an approximative calculation method which, under certain circumstances, may reduce computational complexity drastically. Examples and detailed arguments for this claim will be given later.

We assume a BAN $\{M_i \mid i = 1, \ldots, n\}$ with $M_i = \langle m_i(s, a_i, \mathit{parents}_i), \theta_i \rangle$ and a fixed sequence s^0 to be analysed. We use Θ to refer to the set of all parameters in the BAN, $\{\theta_1, \ldots, \theta_n\}$. Considering the BAN as an entire model, we can describe the best solution as follows.

$$\mathit{ideal}_n(s^0, \Theta) =_{\text{def}} \operatorname*{argmax}_{a_n} P(a_n \mid s^0, \Theta)$$

where the term inside the argmax can be unfolded as follows.

$$\begin{aligned} P(a_n \mid s^0, \Theta) &= \sum_{\langle a_1, \ldots, a_{n-1} \rangle} P(a_1, \ldots, a_n \mid s^0, \Theta) && (9.1) \\ &= \sum_{\langle a_1, \ldots, a_{n-1} \rangle} \prod_{i=1}^{n} P(a_i \mid s^0, \mathit{parents}_i, \theta_i) && (9.2) \end{aligned}$$

Standard methods for reasoning in Bayesian networks, see, e.g., [68], is of very little use here due to the unmanageable size of the random variables' outcome spaces, which in practice are impossible to iterate over.

We are not aware of any reasonable way to reduce this formula, although we do not have a formal proof that this is not possible. Instead, we propose an approximative, iterative algorithm that fixes one particular best annotation $a_i = \mathit{approx}_i(s_0, \Theta)$ for each sub-model and applies it subsequently in the prediction of those a_j with $a_i \in \mathit{parents}(a_j)$.

$$\mathit{approx}_i(s^0, \Theta) = \operatorname*{argmax}_{a_i} P(a_i \mid s^0, \mathit{approx}_{\mathit{parents}_i}(s^0, \Theta), \Theta), i = 1, \ldots, n \quad (9.3)$$

where $\mathit{approx}_{\mathit{parents}_i}(s, \Theta)$, for some sequence s, stands for the sequence of parent annotations $\mathit{approx}_j(s, \Theta)$ for all $a_j \in \mathit{parents}_i$.

Specifically, we take $\mathit{approx}_n(s^0, \Theta)$ as an approximated value for $\mathit{ideal}_n(s^0, \Theta)$; the possible conditions under which this may be considered a good approximation will be discussed among our conclusions, section 9.8.

Notice, that there is no circularity in this definition and $approx_n(\cdots)$ can be calculated in a single sweep calculating $approx_1(\cdots)$, $approx_2(\cdots)$, ... in that order. The "argmax" in (9.3) may be calculated by existing algorithms as we demonstrate below.

In practical applications of our methodology, we expect the number of sub-models in a BAN to be a relatively small number (say, arbitrarily, < 10), but lengths of sequences and their annotations are expected to be huge. Measured in sequence length, the complexity of approximate prediction with the entire BAN coincides with the complexity for the most complex sub-model.

9.5 Training the network

In order to obtain the probabilistic parameters Θ for a BAN, we rely on existing training algorithms for supervised learning, e.g., as built into the PRISM system [106], [186]. Such algorithms require a sufficiently large and representative collection of ground atoms for each sub-model, each representing a sequence with its correct annotation, which in our motivating application domain means annotations verified in the lab by the biologists.

To this end, we assume the availability of some state-of-the-art training algorithm $T^{supervised}$, described as a function mapping a particular program together with its training data into a parameter. Notice that we are not interested here in the actual details of how the training algorithm works.

For doing supervised training of any sub-model in a BAN, we need in principle ground data that exemplifies the relation between sequence, parent annotations, and output annotation. We define, thus, a *conditional training data set* for program m_i as a set

$$CTD_i = \{m_i(s_i^j, a_i^j, parents_i^j) \mid j = 1, \ldots\}.$$

It is called "conditional" since it includes parent annotations $parents_i^j$ for each output annotation a_i^j.

In practice, however, we cannot expect such conditional training sets always be available as this assumes that the signals represented by the different sub-models has been analyzed consistently for the same set of sequences. In other words, we can only assume that the following sorts of training data are available in a more traditional format without explicit parent annotations.

$$TD_i = \{\langle s_i^j, a_i^j \rangle \mid j = 1, \ldots\}$$

However, if we train the different models one by one in the order M_1, M_2, ..., we can use the already trained models to supply parent annotations. We can thus specify an iterative BAN training algorithm as follows.

$$\theta_i = T^{supervised}(m_i, CTD_i)$$

where

$$CTD_i = \{m(s_i^j, a_i^j, approx_{parents_i^j}(s_i^j, \{\theta_1, \ldots, \theta_{i-1}\})) \mid \langle s_i^j, a_i^j \rangle \in TD_i\}$$

There is no circularity in these equations which may be evaluated in one sweep θ_0, θ_1, ...

This strategy can be adapted to handle cases where training data TD_i are unavailable for some non-top model M_i, i.e., $i < n$. Here we may use unsupervised training, or even set the parameters manually, and still hope for good results. It is not essential that model M_i is a faithful mirror of some physically measurable signal (call this M_i^{true}): the necessary property is whether M_i represents *some* annotation that can help the models M_j of which M_i is a parent to discriminate the details of the sequence under consideration. To see this, notice that such an M_j consistently applies annotations produced by M_i (rather than M_i^{true}) for its own training *and* prediction.

We postulate the following rule of thumb for checking the relevance of a specific model M_i within a given BAN.

(*) – Whether a model M_i contributes an interesting signal to M_j can be checked by inspection of the parameter to check whether different values for a_i provide any significant variation in the magnitude of $P(a_j \mid s^0, a_1, \ldots, a_i, \ldots, a_{j-1})$.[1]

However, we expect that models designed according to biological expert knowledge, that are trained using a sufficient set of authoritative data, and whose position in the hierarchy is based on the same biological expert knowledge, will have the best chance to constitute an interesting signal according to (*). In case of a biologically justified model, for which sufficient amounts of data are available, it will be natural also to check it with standard precision and recall methods.

We can summarize some of the practical consequences of these arguments as follows.

- M_i may for reasons of performance, or to avoid over-training, be programmed in a rather coarse way, which gives only a very rough approximation of M_i.
- We may introduce an arbitrary sub-model in a BAN, be it based on only little or no biological insights; it may be trained unsupervised or the parameter may be set by hand, and we can apply (*) to check whether it is of any use.
- We may introduce alternative models for the same biological signal, and use another model as a voting mechanism to combine the different signals and check its contribution according to (*).
- Having a collection of candidate sub-models, we can experiment with different topologies for dependencies, and validate it internally according to (*) as well as using precision and recall tests for the top model.

We will discuss some these points below in relation to our experiments.

[1]For a trained PRISM model, we may compare the different conditional `msw` probabilities produced by the training of M_j.

9.6 Implementation in PRISM – the LoSt Framework

The methodology described so far is supported by an implementation built on top of the PRISM system [175], which is a probabilistic extension to Prolog which provides a wide range of learning and prediction algorithms.

In this section, we first explain our own system, called the *LoSt-framework* [39], which is basically a collection of scripts that control the ordering of different runs of PRISM for prediction and training plus a file management system that keeps track of the different models, their parameters, the connections that tie them together to a BAN as well as all data files involved (catalogues of sequences, training data, files of predicted annotations, etc.). We also show a simplified, implemented example that illustrates different aspects of our methodology.

9.6.1 Embedding BANs in PRISM

The PRISM system [175] realizes a probabilistic extension to Prolog and is equipped with a comprehensive collection of facilities for prediction and training.

The PRISM language extends Prolog with so-called multi-valued switches: a call `msw(`*name*`, X)` represents a probabilistic choice of a value to be assigned to `X`.[2] The semantics of a PRISM program is given as a probabilistic Herbrand model, determined by a parameter which is a file of probability declarations for the individual switches. For this semantics to be well-defined, any choice point in the program must be governed by an msw.

The program part of a sub-model $m(s, a, parents)$ may be represented by a PRISM program with a main predicate

`m(`*s*`, `*a*`, `*parents*`)`

where *parents* are a arguments corresponding to the number of parents of m_i. A typical sequence model is implemented as a recursive predicate which relates the s and a arguments in a probabilistic fashion conditioned by given parent annotations and involving myriads of `msw` calls.

PRISM contains algorithms for training based on suitable generalizations of EM learning and Variational Bayesian learning [186] which can be used for both supervised and unsupervised learning; the LoSt environment keeps track of training data and generated parameter files for the individual sub-models.

Prediction using a PRISM program, representing a trained sub-model, can be performed using one of PRISM's generalized Viterbi algorithms. Specifically, we use a minor extension to PRISM, described in [40], which makes it possible to analyse longer sequences in reasonable time. The following call,

```
S= ··· , A1= ··· , A2= ··· , viterbiAnnot( m(S,A,A1,A2, ···)),
```

will instantiate `A` to the annotation that provides the highest probability of the goal `m(S,A,A1,A2, ···)`, thus implementing the argmax in equation (9.3) above.

[2]To be exact, a switch introduced by a declaration `values(name, [···` *outcomes* `···])` defines a family of random variables, one for each execution of `msw(`*name*`, ···)` in a program run.

The scripts in the LoSt environment implement the correct ordering of sub-model processing as prescribed by our incremental prediction and training algorithms described in sections 9.4 and 9.5 above, however, avoiding computations that have been made before and whose results are available on files.

9.6.2 Example: Gene-finding in DNA

We illustrate our methodology showing experiments with BANs that represent gene-finders for DNA sequences. A piece of DNA is a sequence of letters $\{a, c, g, t\}$; it can be viewed as a sequence of triplets, each called a codon. Codons are separated into specific start-codons, stop-codons and other codons; a gene is a specific subsequence matching the codon structure such that it must begin with a start codon and it will definitely end at the next stop-codon; such a syntactic pattern is called an open reading frame (ORF). Our BAN models are designed to annotate ORFs, where the annotation task is to find out whether an ORF contains a gene and, if so, where in the sequence the gene starts.

We define sub-models for different signals — codon preference, gene length and conservation — which are are expected to have influence on whether a sequence is a gene or not. The resulting annotation from such models is a sequence that for each position in the original sequence contains a `1` if the position is predicted as part of a gene and a `0` if it is not.

All our probabilistic models are output HMMs with a gene-state and a non-gene state, which can emit symbols of the annotations of the parent nodes. The transitions between the states reflect the described ORF pattern.

The *codon preference* model **m1** reflects preferential codon usage in the gene and non-gene state. The states can each emit one of the 64 possible codons.

The *gene length* annotation is obtained by using a deterministic model **m2** that annotates each potential start codon with a symbol representing the distance to the upstream stop codon.

Conservation describes a degree to which the codons of a DNA sequence are conserved across species. To detect conservation, each ORF matched to a database of genome sequences of distantly related organisms [3] using the tblastn tool, which produce a gapped alignment of the matches. Only statistically significant matches (evalue $< 10^{-34}$) and only one match per organism are reported. The conservation model **m3** emits identity positions of reported matches to ORFs.

In the following we discuss and assess a number of BAN topologies built using these three signals as basic building blocks. The considered models are **m1**, **m3**, **m1** conditioned on **m2** – **m1(m2)**, **m1** conditioned on **m3** – **m1(m3)**, and **m1** conditioned on both **m2** and **m3** – **m1(m2,m3)**.

We train and predict on the well-annotated *Escherischia Coli* genome and its curated gene annotations from refseq (NC_000913). We have randomly divided the ORFs of the genome into a training and a test set. Supervised training is done using only the former and the method for supervised training algorithm described in section 9.5. We report prediction accuracy results for both sets. Accuracy is measured as *Sensitivity*$(SN) = \frac{TP}{TP+FN}$ and

[3] The sequences are from refseq: NC_004547, NC_008800, NC_009436, NC_009792, NC_010067, NC_010694 and NC_011283.

Specificity$(SP) = \frac{TP}{TP+FP}$, with respect to annotation of start and stop codons. The results are summarized in table 9.1.

	Training set (114429 ORFs, 2075 genes)			
BAN	SN_{start}	SP_{start}	SN_{stop}	SP_{stop}
m1	0.7701	0.2935	0.9711	0.3701
m3	0.0636	0.0322	0.8255	0.4183
m1(m2)	0.6723	0.5011	0.9345	0.6965
m1(m3)	0.4405	0.2243	0.8255	0.4204
m1(m2,m3)	0.4361	0.2228	0.8255	0.4217
	Test set (114404 ORFs, 2065 genes)			
BAN	SN_{start}	SP_{start}	SN_{stop}	SP_{stop}
m1	0.7564	0.2920	0.9719	0.3751
m3	0.0140	0.0072	0.8412	0.4298
m1(m2)	0.6489	0.4896	0.9433	0.7117
m1(m3)	0.4315	0.2216	0.8416	0.4323
m1(m2,m3)	0.4174	0.2149	0.8416	0.4333

Table 9.1: Accuracy of predictions using different BAN topologies.

It can be observed from table 9.1 that all our models have good generalization capabilities, since the performance is very similar on both the training and test set.

The best model seems to be **m1(m2)**, which achieve a significant increase in specificity with only slightly degraded sensitivity, e.g. it predicts fewer genes but its predictions are more reliable. By them selves, both **m1** and **m3** have reasonable stop specificity, but **m3** has consistent tendency to predict too long genes, leading to severely decreased start specificity. Interestingly, conditioning **m1** on the conservation additional signal **m3** does not improve prediction accuracy much. It does lead to slightly better stop specificity but it tends to degrade the start specificity. Additionally, conditioning on the length signal as done in **m1(m2,m3)** does not seem to help, even though the impact observed in **m1(m2)** was quite significant. It seems that the **m3** signal dominates decisions about which ORFs should be predicted as coding. This effect is apparent from model parameters and it is possible to get an intuition of the problem from inspection of the prediction accuracies.

The **m3** model has a (stop) false negative rate of $1 - SP_{stop} = 1 - 0.4183 \approx 0.58$. The vast majority of ORFs $\sim 98\%$ does not contain genes. The probability that an ORF contains a gene but **m3** classifies it as non-gene is thus relatively small, $1 - 0.98 \times 0.58 \approx 0.11$. In the conditional distribution defined by **m1(m3)** (given predictions of **m3**), it becomes virtually impossible for the viterbi algorithm to classify an ORF as a gene if **m3** has not, since the probability of the gene hypothesis is scaled by ~ 0.11 and the non-gene hypothesis by ~ 0.89.

Part of the explanation is that maximizing the likelihood of observed data (as we do in training) is not equivalent to maximizing prediction accuracy; it may have an adverse effect when selecting predictions as most probable explanations as done by the viterbi algorithm. An other part of the explanation is in our model assumptions; namely, **m1(m3)** is an output HMM that has joint emissions of both codon and the signal from **m3**, and these are dominated by **m3** as explained above. Alternative HMM structures with different constraints and independence tradeoffs might avoid the dominating effect of **m3**. We are still investigating how this is best done.

9.7 Related work

Our method is closely related to *Dynamic Bayesian Networks* (DBNs) of [142]. By our definition of a BAN, the detailed dependencies between individual models in the network are left abstract, but a concrete instantiation of a BAN may indeed be a DBN. However, as the nodes in a BAN may be arbitrary probabilistic models, for instance context-free grammars, not all BAN instantiations can be represented as DBNs. Oppositely, we only define BANs for discrete models but DBNs may include continuous-valued nodes.

In the realm of classification techniques, it is common to combine the results of different classifiers of the same phenomena in ways such that the combined classifications outperform the individual constituent classifiers. Such methods are generally known by the name *ensemble methods*, which covers a wide range of different ways to the combine classifiers [166]. Our method is related but quite different; this is not just because we consider sequence annotation rather than classification, but also because constituent models of a BAN may model very different phenomena.

In biological sequence analysis, the most successful genome annotation programs are *combiners* [83]; programs which combines different sources of annotation evidence using some sort of weighting scheme. Evidence may come in diverse forms, including comparative analysis sources [156], but are typically predictions (e.g. annotations) from other annotation programs (e.g. gene finders). Brent [22] makes a distinction between combiners and joint models, where joint models are described as models which consider the full joint probability distributions evidence and combiners as probabilistic models of the the relative accuracy of evidence sources they are combining. Using our approximate inference algorithm we have a situation similar to combiners in that predictions of parents are combined by child nodes.

While many combiners use non-probabilistic combinations methods, several are explicitly based on principles of (dynamic) Bayesian networks [152, 126]. A main difference is that our framework allows multi-layered and branching topologies where the combiners are usually just single layered probabilistic models.

Our approach also has analogies to *annotation pipelines* [158, 28] where a complex sequence of analysis steps are performed in a possibly branching topology and perhaps synthesized (e.g. by a combiner) in a final annotation as the last step. Opposed to combiners, such pipelines usually allows complex topologies like our framework. However, such pipelines are usually just practical and pragmatic ways of combining existing tools and incorporate probabilistic modeling only to a very limited degree.

There are other declarative approaches to combining evidence in biological sequence analysis. In GAZE [96], a configurable XML-based specification describes a particular composition of evidence sources. However, GAZE integrates existing tools, where our PRISM based approach allows for much more modelling flexibility and have clear and well-defined semantics.

9.8 Conclusions

We have proposed a Bayesian framework, *Bayesian Annotation Networks*, which allows the representation and composition of models for complex sequence analysis. In a modular way, it supports experimentation with and evaluation of models and signals and it is a practically useful tool for modeling and analyzing sequences. In particular, its applicability to biological sequence analysis has been motivated. We have shown that reasonable complexity can be achieved by the use of tractable, incremental algorithms for inference and training, which can be implemented by successive calls to PRISM, and shown that these algorithms may produce useful annotations.
In general, we have no good analytical or sampling-based principles for analyzing the quality of the approximated annotations compared with the ideal ones. By assumption, the ideal annotations provided by a BAN for a given sequence is too complex to be evaluated, so we need to rely on standard validation techniques based on authoritative test data. However, we will list a few observations which may be used as guidelines.
The crux in our approximate inference algorithm is, in each iteration step, to select a most probable annotation $approx_i$ for each annotation node a_i and take it as a representative for the distribution of all possible a_i values. In the detailed calculations, this means that we use $P(a_j \mid s, \cdots approx_i \cdots)$, for some a_j with $a_i \in parents_j$, as a replacement of a weighted sum over all possible a_i values of $P(a_j \mid s, \cdots a_i \cdots)$.
In the trivial case, where all freedom of choice is implemented in the top node of the Bayesian Network, the approximate algorithm coincides to the ideal. Beyond the trivial case, however, it is difficult (impossible in general) to give sufficient conditions for which the approximate inference method will yield good results.
The relation between the quality of an annotation and its probability is assumed implied by the purpose of a probabilistic annotation model; e.g. it should assign high probability to good annotations. In our definitions of BANs, we define a child model to be dependent on its parent model. In a concrete BAN, however, individual models typically have more fine grained interdependencies, e.g. enforce their inherent independence assumptions. If these assumptions are not faithful to the actual data dependencies, a discordance between annotation quality and probability may arise. Similarly, in the case of approximate inference, we are concerned in particular with the degree of validity of the assumption about independence of parent distributions given the most probable individual elements of those distributions.
Perhaps surprisingly, the annotation quality achieved by the approximate method may be positively affected by correlation between assumed independent nodes of the network. Redundant (correlated) signals does not generally result in better annotations, if the ideal inference method is used. However, such overlapping signals may indeed compensate for the information lost due to the (possibly) unjustified independence assumptions imposed by the approximation method or inherent in constituent models. For instance, information contained in the distribution of a particular parent node, but not reflected by the best an-

notation from that distribution, may be reflected through the best annotation of some other (correlated) parent.
In practice, we are satisfied with the approximation if the annotations are judged as good using an external measure of quality (e.g. sensitivity/specificity) and we have used cross-validation to build confidence about generality, as demonstrated in section 9.6.2. Obviously, this may require a considerable amount of, possibly unavailable, labelled training data. A second consequence, also observed in section 9.6.2, is that the measure optimized by the training algorithm does not necessarily coincide with the external measure of quality. Model constraints and independence assumptions play a key role affecting the correlation between these measures.

Chapter 10

A Declarative Pipeline Language for Big Data Analysis

With Henning Christiansen, Ole Torp Lassen and Matthieu Petit

In pre-proceedings of 22nd International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR), 2012.

Abstract

We introduce BANpipe – a logic-based scripting language designed to model complex compositions of time consuming analyses. Declarative semantics are described together with alternative operational semantics facilitating goal directed execution, parallel execution, change propagation and type checking.

10.1 Introduction

Computations for biological sequence processing are often complex compositions of time consuming analyses, including calls to external resources and databases. The expected output, intermediate results and original input data are often huge files. To facilitate such computations, we introduce a new declarative scripting language called BANpipe. The language supports complex pipelines of Prolog programs, PRISM [179] models and other types of programs through rules which specify dependencies between computations.

BANpipe is a general pipeline programming language, but it is designed to support a special kind of annotation pipelines which we call Bayesian Annotation Networks (BANs) [43, 124]. A Bayesian Network is a directed acyclic graph where the nodes are conditional probability distributions and the edges represent conditional dependencies. A BAN is a Bayesian Network where nodes are instead (probabilistic) annotation programs and edges are input/output dependencies between programs. Inference in BANs is performed iteratively by evaluating each program at a time and using its output annotation as input for dependent programs. This is not only similar to the way forward analysis takes place in Bayesian networks, but also a nice fit to the pipeline paradigm.

Existing pipeline scripting languages, however, are not designed for the integration of Prolog and PRISM programs. As consequence, integration would have to rely on shell commands rather than providing a smooth integration at the language level. An other reason for designing a new language is that existing pipeline languages lack the desired balance between declarativity and expressiveness, cf. section 10.7. Declarative pipeline languages typically sacrifice expressiveness, resulting in verbose languages with limited capabilities for modeling dynamic aspects of pipelines. Inversely, languages which favor expressiveness are often locked into a procedural semantics which exclude possibilities for automated parallelism, change propagation and management of result files.

BANpipe rules express dependencies between symbolically represented files that are automatically mapped to the underlying filesystem. The symbolic filenames may include logic variables which enables advanced control mechanisms, leading to compactly expressed pipelines. The declarative semantics of the language allow for useful extensions.

Execution of a pipeline script is goal directed, where only the desired result is specified and the system then executes programs necessary to achieve the result as entailed by the dependencies in the script. Computations entailed by multiple goals are only performed once and subsequently shared by the goals. The language enables incremental change propagation, where all depending files are recomputed recursively after a change to a component in the pipeline.

The language also lends itself to parallel execution; programs which can run in parallel are inferred from the conditional independencies in a script.

Additionally, many aspects of scripts can be statically checked, e.g., the language is extended with a type system which can be used to infer compatibility between file formats of intermediate files.

10.2 Syntax and informal semantics of BANpipe

The BANpipe language consists of scripts for controlling collections of programs that work on data files. We add another layer on top of the traditional file system, so that an arbitrary ground Prolog term can be used for identifying a file. We assume a *(local) file environment* that maps such file identifiers into files, via their real file name in the local file system or a remote resource. The syntax is embedded in Prolog so we inherit its notions of terms and variables[1] (written with capital letters). Our implementation use a convention that a Prolog constant whose name indicates a protocol is treated as an URL, and the files for terms are determined through the local file environment. For example:

- `'file:///a/b/c/datafile'`: refers to the file with the real name `datafile` in the `/a/b/c` directory in the local file system,
- `'http://a.b.c/file'`: refers to a file referenced using the http protocol.
- `f(7)`: may refer to a file in the local file system.

For ease of usage, we refer to Prolog terms expected to denote files as *file names*.

[1]Terms with variables may become ground through a substitution as result of execution a script.

The programs referred to in BANpipe scripts are placed in modules, and a program defines a function from zero or more file to one or more files; a program may take options as additional arguments that modify the function being calculated. Programs are referred to in the body of BANpipes *dependency rules*, exemplified as follows.

$$\texttt{file1, file2 <- m::prog([file3,file4],op1,op2).} \tag{10.1}$$

Here `prog` is a program in module `m`, taking two files `file3` and `file4`, plus options `op1` and `op2` as input. The rules explains how two output files `file1` and `file2` depend on `file3`, `file4`, namely being the result of applying the function (or *task*) given by `m::prog([−,−],op1,op2)`.

The sets of input files, output files and options are fixed for a given program.

File names in rules can be parameterized as shown in this rule:

$$\texttt{f(N) <- m::prog(g(N)).} \tag{10.2}$$

For any ground instance of `N`, this rule explains the dependency between two files, e.g., between `f(7)` and `g(7)` or `f(h(a))` and `g(h(a))`. Rules can be recursive as shown in the following example.

$$\texttt{f(0) <- file::get('file:///data').} \tag{10.3}$$
$$\texttt{f(N) <- N > 0, N1 is N-1 | m::prog(f(N1)).} \tag{10.4}$$

Here, rule (10.3) applies a built-in module that takes care of simple file handling including the `get` facility that provides a copy of a file as shown; rule (10.4) includes a *guard*, which may precede the program call and is used for rule selection and for instantiating variables not given by the matching in the head. The recursion works as expected, and the evaluation of a `query f(2)` involves the calculation of files named `f(2)`, `f(1)` and `f(0)` from the local file with the real name `data`.

A BANpipe *script* is a sequence of rules defined by the following syntax, perhaps extended with definitions of Prolog predicates to be used in rule guards.

$$\langle\text{rule}\rangle ::= \langle\text{head}\rangle \texttt{ <- } \langle\text{body}\rangle \tag{10.5}$$
$$\langle\text{head}\rangle ::= \langle\text{file}\rangle_1\texttt{,} \ldots \texttt{,}\langle\text{file}\rangle_m \quad m \geq 1 \tag{10.6}$$
$$\langle\text{file}\rangle ::= \textit{any Prolog term, as described above} \tag{10.7}$$
$$\langle\text{body}\rangle ::= \{\, \langle\text{guard}\rangle \texttt{|} \,\}\ \langle\text{program call}\rangle \tag{10.8}$$
$$\langle\text{guard}\rangle ::= \textit{sequence of one or more Prolog calls} \tag{10.9}$$
$$\langle\text{program call}\rangle ::= \langle\text{module}\rangle\texttt{::}\langle\text{program name}\rangle\texttt{(} \langle\text{file}\rangle_1\texttt{,} \ldots \texttt{,}\langle\text{file}\rangle_n, \textit{options}\texttt{)} \quad n \geq 0 \tag{10.10}$$

The following syntactic restrictions must hold for any BANpipe script.

- File names given as URLs are only allowed in program calls and cannot occur in rule heads.
- When a rule head contains file names with variables, any file name in that head must contain the same variables.

We assume that any predicate call in the guard of rule r is terminating, whenever the variables in the head of rule r are ground.

The selection of a rule for the evaluation of a query (a ground file name) must be unique. We capture the essential properties in the following definition and explain afterwards how such a selection function is implemented in practice.

Definition 10.2.1. *A selection function for a BANpipe script S is a partial function σ_S from non-URL ground Prolog terms to ground instances of rules of S such that if*

$$\sigma_S(f) = \big(f_1^{out}, \ldots, f_n^{out} \texttt{ <- } guard \mid m :: prog(f_1^{in}, \ldots f_m^{in})\big) \tag{10.11}$$

then guard evaluates to true, and it holds that

$$f = f_i^{out} \text{ for some } i = 1, \ldots, n, \tag{10.12}$$

$$\sigma_S(f_i^{out}) = \sigma_S(f) \text{ for all } i = 1, \ldots, n. \tag{10.13}$$

Any such instance is called a selection instance *for S.*

To simplify notation later, we may leave out the guard when referring to a selected instance, as it has made its duty for testing and variable instantiation, once the selection is effectuated.

Condition (10.12) states that the chosen rule is actually relevant for f, and condition (10.13) indicates that whenever a rule is applied, it calculates the unique results for all files mentioned in its head, independently of which request for a file that triggered the rule.

In the implemented system, the rules are checked in the order they appear in the script and the file names in their heads from left to right. If such a head file name unifies with the given f, and the guard succeeds, the rule is a candidate for selection. However, if the execution of a guard leads to a runtime error or does not instantiate all remaining variables in the rule, the search stops and no rule is selected. Condition (10.13) of def. 10.2.1 is undecidable, but it is straightforward to define sufficient conditions that can be checked syntactically; we do not consider this topic further here.

The evaluation of a query Q can be done in a standard recursive way, which will be described in more detail in section 10.4.

10.2.1 Defining programs and modules

As mentioned, the tasks activated from a BANpipe script are defined by programs that are grouped into named modules. How these modules are structured is not important for the understanding of the BANpipe script language, so we give here just a brief overview.

A module m must contain a designated interface file that defines each task through a Prolog predicate of the following form,

$$task\texttt{(}\texttt{[}in\text{-}file_1, \ldots, in\text{-}file_n\texttt{]}\texttt{,}\, opts\texttt{,}\, \texttt{[}out\text{-}file_1, \ldots, out\text{-}file_m\texttt{]}\texttt{)} \tag{10.14}$$

that will be matched by a program call $m\texttt{::}task\texttt{(}\cdots\texttt{)}$ in a script as described above. In accordance with the precise semantics specified below, the file names (whether URLs or arbitrary Prolog ground terms) encountered by the script are mapped into references to actual files, which then are given to task predicates that access the files through standard input/output built-ins.

The interface file may contain all the code that implements the tasks but, typically, a module contains a number of source files, which may be shared by the different tasks. The execution of a task is done by the PRISM system, which is an extension to B-Prolog, and thus PRISM probabilistic inference and ordinary Prolog code is readily available. Calls to programs in other languages or web services are facilitated as well.

The system includes a sort of dynamic types that are specified in the interface files and not visible in the BANpipe scripts. This is described in more details in section 10.5.

10.3 Declarative semantics of BANpipe

Raw data and results of analyses are represented as data files. The language relies on no assumptions about the detailed structure of those files. We assume an unspecified domain

$$\mathit{DataFile} \tag{10.15}$$

including a $\perp$ element, which has the intuitive interpretation of a recognized unsuccessful result (rather that no results). Notationwise, we consider a tuple $\langle\perp,\ldots,\perp\rangle$ equivalent with $\perp$.

Program calls in a script denote *tasks* that are mappings from a (perhaps empty) sequence of data files into another sequence of data files. Thus

$$\mathit{Task} = \sum_{i=0,1,\ldots;j=1,2,\ldots} \mathit{Task}_{i,j} \tag{10.16}$$

$$\mathit{Task}_{i,j} = \mathit{DataFile}^i \rightarrow \mathit{DataFile}^j \tag{10.17}$$

Tasks are assumed to be strict in the sense that if any component of an input argument is $\perp$, the result is $\perp$. A task may also result in $\perp$ reflecting a runtime error or a Prolog failure.

Definition 10.3.1. *A* program semantics *is a function* $[\![-]\!]$ *from triples of module name, program name, and ground values for possible option parameters into tasks. For module mod, program prog (with n input and m output files), and option values opts, this function is indicated as*

$$[\![\mathit{mod} :: \mathit{prog}(\mathit{opts})]\!] \in \mathit{Task}_{n,m}. \tag{10.18}$$

Ground file names are used as synonyms for variables ranging over the domain *DataFile*; for a ground file name f, the corresponding unique variable, called a *file variable*, is denoted $\widehat{f}$, and this notation is extended to sets, $\widehat{F} = \{\widehat{f} \mid f \in F\}$; whenever f is an URL, $\widehat{f}$ is called an *URL variable.* (Partial) answers to queries are represented below as substitution for file variables into *DataFile* and are typically indicated by the letter Φ with possible subscripts. We recognize a special form of *URL substitutions* for URL variables only. For ease of notation, an URL substitution is assumed to provide a value for any URL variable which might be $\perp$. The notation Φ_0 typically refers to an URL substitution. A substitution Φ is considered equivalent to the set of equations $\{f \doteq d \mid \Phi(f) = d\}$.[2]

The declarative meaning of a BANscript is given by a recursive systems of equations defined as follows.

Definition 10.3.2. *Given a BANscript S, a* defining equation for *a non-URL file name f is of the form*

$$\langle \widehat{f_1^{out}}, \dots, \widehat{f_m^{out}} \rangle \doteq [\![mod :: prog(opts)]\!] \langle \widehat{f_1^{in}}, \dots, \widehat{f_n^{in}} \rangle \tag{10.19}$$

where $f = \widehat{f_i^{out}}$ for some $i = 1, \dots, m$, and S has a selection instance for f,

$$f_1^{out}, \dots, f_m^{out} \texttt{ <- } m :: prog(f_1^{in}, \dots, f_m^{in}). \tag{10.20}$$

Given such S and $[\![-]\!]$, the defining set of equations for a query *Q, denoted $Eq(Q,S)$ is defined as the smallest set E of defining equations such that*

- *E contains a defining equation for any $q \in Q$,*
- *for any equation in E whose righthand side contains a non-URL variable $\widehat{f}$, E contains a defining equation for f.*

We say that a BANscript S is well-behaved *for a query Q if $Eq(Q,S)$ exists, is finite, and contains no circularities.*[3]

Notice that $Eq(Q,S)$ is defined independently of program semantics, so this definition is equally relevant for a standard semantics (i.e., the intended computations on real data files) as for different abstract semantics reflecting different program properties.

The solution to a set of equations is given as usual, as a substitution that maps variables to values, such that the left and right hand sides of each equation become identical when all functions are evaluated. Whenever a script S is well-behaved for a query Q, and Φ_0 is an URL substitution for the URL variables of $Eq(Q,S)$, there exists a unique solution for $Eq(Q,S) \cup \Phi_0$. To prove this, first convert each equation on tuples into equations of the form $\widehat{f_i} \doteq \dots$ by projections, and then notice that all variables in the righthand sides can be eliminated in a finite number of steps. Condition (10.13) of def. 10.2.1 ensures that the resulting set of equations with variable-free righthand sides is unique. We can thus define:

Definition 10.3.3. *Let $[\![-]\!]$ be a program semantics, S a BANscript which is well-behaved for a query Q and Φ_0 an URL substitution, and let Φ be the solution to $Eq(Q,S) \cup \Phi_0$. The* answer *to Q (with respect to S, $[\![-]\!]$ and Φ_0) is the restriction of Φ to $\widehat{Q}$; the substitution Φ is referred to as the* full answer *to Q.*

The query is failed *whenever the solution assigns $\perp$ to any variable in $\widehat{Q}$.*

[2] We use symbol "$\doteq$" to distinguish equations that are explicit syntactic objects from the normal use of "$=$" as meta-notation.

[3] "No circularities" can be formalized by separating variables into disjoint, indexed strata, such that for an equation $\cdots V \cdots = \cdots V' \cdots$, that the stratum number for V' is always lower than the stratum number for V.

Alternatively, this semantics could have been formulated in terms of a fixed point or a least model, which is straightforward due to the well-behavedness property.

Well-behavedness is obviously an undecidable property as an arbitrary Turing machine can be encoded through recurrence of variables in the terms that represent file names. In practice, however, we expect the recursion patterns through file names to be rather simple, so a straightforward depth-first algorithm for calculation of $Eq(Q, S)$ is sufficient. The error message "(perhaps) not well-behaved" is then issued if this algorithm exceeds a certain recursion depth, or the selection of a defining equation for a particular file name fails.

10.4 Operational semantics

We present a number of alternative operational semantics for BANpipe as abstract algorithms. A script is executed in a state that contains a substitution mapping file variables into the *DataFile* domain, and which grows during the execution of a query.

10.4.1 Bottom-up operational semantics with memoization

The following algorithm defines an operational semantics that works in a bottom-up fashion, calculating all involved files from scratch. It ensures that any intermediate file needed to obtain the final results is evaluated exactly once, even if used in different program calls.

```
Algorithm 1: Bottom-up operational semantics for BANscript
Input: A query $Q$, a BANscript $S$, program semantics $[\![-]\!]$
        and initial substitution $\Phi_0$;
Output: A substitution;

$\Phi := \Phi_0$;
while $Eq(Q,S)$ contains an equation
            $\langle\widehat{f_1^{out}},\ldots,\widehat{f_m^{out}}\rangle \doteq [\![P]\!]\langle\widehat{f_1^{in}},\ldots,\widehat{f_n^{in}}\rangle$
      for which $\Phi(f_i^{out})$ is undefined for all $i = 1,\ldots,m$,
            and $\Phi(f_j^{in})$ is defined for all $j = 1,\ldots,n$
do $\Phi := \Phi[\widehat{f_1^{out}}/df_1,\ldots,\widehat{f_m^{out}}/df_m]$
            where $\langle df_1,\cdots,df_m\rangle = [\![P]\!]\langle\Phi(\widehat{f_1^{in}}),\ldots,\Phi(\widehat{f_n^{in}})\rangle$;
return $\Phi$;
```

Theorem 1. *Given a program semantics $[\![-]\!]$, a BANscript S which is well-behaved for a query Q and an URL substitution Φ_0, Algorithm 1 returns the full answer Φ to Q. The solution to Q is a found as a the restriction of Φ to $\widehat{Q}$.*

Having the algorithm to return the full substitution produced, makes it possible to use it also for incremental maintenance of solutions, as we will see below in section 10.4.2.

Sketch of proof. Each step performed in the while loop in Algorithm 1 corresponds to a variable elimination step in $Eq(Q, S) \cup \Phi_0$. Furthermore, each such step that processes an equation of the form $\langle \widehat{f_1^{out}}, \ldots, \widehat{f_m^{out}} \rangle \doteq [\![P]\!] \langle \widehat{f_1^{in}}, \ldots, \widehat{f_n^{in}} \rangle$, will bind variables $\widehat{f_1^{out}}, \ldots, \widehat{f_m^{out}}$ to their final values in the resulting solution. $\square$

This algorithm provides an abstract operational semantics for BANscripts which can be transformed into a running implementation by adding suitable data structures for representing the defining equations and the file environment (appearing in the algorithm as substitutions).

Algorithm 1 can also be applied for symbolic program executions in which the standard program semantics is replaced by one that calculates program properties, but the evaluation of the guards and patterns of recursion will be the same. Such symbolic executions are expected to run essentially faster than runs with a standard semantics. This principle is used below for predicting change propagation, section 10.4.2, and type inference, section 10.5.

10.4.2 Operational semantics for incremental change propagation

Our BANpipe language is intended for time consuming computations and will be used by researchers in an experimental style, with frequent modifications of the involved programs and data files. We describe here an extension of the operational semantics that accomodates such changes, and which reuses previous results where possible.

It involves an alternative program semantics for measuring change propagation, based on the domain $\mathit{DataFile}^{prop} = \{\mathit{changed}, \mathit{unchanged}, \bot\}$ and program semantics $[\![-]\!]^{prop}$ defined as follows: whenever the program *prog* (with n input and m output files) in module *mod* has been modified, or one of its input arguments (x_i below) has the value *changed*, we set

$$[\![\mathit{mod} :: \mathit{prog}(\mathit{opts})]\!]^{prop} \langle x_1, \ldots, x_n \rangle = \langle \underbrace{\mathit{changed}, \ldots, \mathit{changed}}_{m \text{ times}} \rangle; \tag{10.21}$$

otherwise (i.e., program not modified, input = $\langle \mathit{unchanged}, \ldots, \mathit{unchanged} \rangle$), the program call returns $\langle \mathit{unchanged}, \ldots, \mathit{unchanged} \rangle$.

We consider the difference between two substitutions Φ^{before} and Φ^{after}, intended to represent correct values for all file variables before and after the modification. This is characterized by a *propagation substitution* defined as follows.

$$\mathit{Diff}(\Phi^{before}, \Phi^{after})(\widehat{f}) = \begin{cases} \mathit{unchanged} & \text{whenever } \Phi^{before}(\widehat{f}) = \Phi^{after}(\widehat{f}) \\ \mathit{changed} & \text{otherwise} \end{cases} \tag{10.22}$$

Changes in the set of URL files (i.e., where the ultimate input comes from) is characterized by a propagation substitution for URL variables.

We can now define an algorithm that predicts which files that need to be re-evaluated to obtain a consistent state following particular modifications of the programs and URL files. It is used as a helper in the algorithm for incremental re-evaluation later.

```
Algorithm 2: Change prediction for BANscript
Input: A query $Q$, a BANscript $S$ and URL change substitution $\Phi_0^{prop}$;
Output: A substitution of variables into $\{changed, unchanged\}$;
--------------------------------------------------------------------
$\Phi^{prop}$ := run Algorithm 1 for $Q$, $S$, $[\![-]\!]^{prop}$ and $\Phi_0^{prop}$;
return $\Phi^{prop}$;
```

We state the following weak correctness statement for Algorithm 2.

Theorem 2 (Soundness of the change prediction algorithm)**.** *Let Φ_0^{before} and Φ_0^{after} be URL substitutions for the same set of variables into DataFile, and assume two program semantics $[\![-]\!]^{before}$ and $[\![-]\!]^{after}$. Let, furthermore,*

Φ_1^{before} be the full answer for Q wrt S, $[\![-]\!]^{before}$ and Φ_0^{before},

Φ_1^{after} the full answer for Q wrt S, $[\![-]\!]^{after}$ and Φ_0^{after}, and

Φ_1^{prop} the result of running Algorithm 2 for Q, S and $Diff(\Phi_0^{before}, \Phi_0^{after})$.

Then it holds for any ground file name f, that if $\Phi_1^{prop}(\widehat{f}) = unchanged$, then $\Phi_1^{before}(\widehat{f}) = \Phi_2^{before}(\widehat{f})$.

Sketch of proof. According to theorem 1, Φ_1^{before} (resp. Φ_1^{after}) can be characterized as the result of running Algorithm 1 for Q, S, $[\![-]\!]^{before}$ and Φ_0^{before} (resp. $[\![-]\!]^{after}$ and Φ_0^{after}). We can thus construct three synchronized runs of algorithm 1, calculating Φ_1^{before}, Φ_1^{after} and Φ_1^{prop} selecting the same equations in the same order. The theorem is easily shown by induction over these runs. □

Due to the sort of complex programs and data involved in our intended applications, we find it unlikely that an output of a program accidentally happens to be the same after a modification of input data or program text. It can be shown that algorithm 2 is optimal under this assumption, in the sense that if $\Phi_1^{prop}(\widehat{f}) = changed$, we will indeed have $\Phi_1^{before}(\widehat{f}) \neq \Phi_2^{before}(\widehat{f})$.

We can now give the algorithm for incremental maintenance of the solution for a given query and a script, when input data and programs called are modified. It uses algorithm 2 to identify which files that must be recomputed; their values are set to undefined in the current file substitutions, and a run of Algorithm 1 starting from this substitution leads to correctly updated file substitutions, i.e., the full answer for the query under the new circumstances, with as few program calls as possible.

```
Algorithm 3: Incremental maintenance for BANscript
Input: A query $Q$, a BANscript $S$, a substitution $\Phi_1$ produced
       by Algorithm 1 from some $Q$, $S$, some $[\![-]\!]^{before}$ and $\Phi_0^{before}$,
       and an URL substitution $\Phi_0^{after}$;
Output: A substitution;
--------------------------------------------------------------------
```

```
$\Phi^{prop}$ := **run** Algorithm 2 for $Q$, $S$ and $\mathit{Diff}(\Phi_0^{before}, \Phi_0^{after})$;
$\Phi_2 := \Phi_1 \setminus \{ (\widehat{f}/\Phi_1(\widehat{f})) \mid \Phi^{prop}(\widehat{f}) = \mathit{changed}\}$;
$\phi_3$ := **run** Algorithm 1 for $Q$, $S$, $[\![-]\!]^{after}$ and $\Phi_2$;
**return** $\Phi_3$;
```

We leave out the correctness statement, which is straightforward to formulate and follows easily from the previous theorems.

10.4.3 A parallel operational semantics

BANpipe scripts are obvious candidates for parallel execution, as we can illustrate with this fragment of a script.

```
f0 <- m0::p0(f1,f2,f3).
f1 <- m1::p1('file:///data').
f2 <- m2::p2('file:///data').
f3 <- file::get('http://server/remoteFile').
```

Here `f1` and `f2` can be computed independently in parallel, and at the same time `f3` can be downloaded from the internet. When they all have finished, `m0::p0` can start running, taking as input the files thus produce, but not before.

Here we describe the structure of a parallel operational semantics as modification of Algoritm 1. We assume a *task manager* that maintains a queue of defining equations waiting ready to be executed. Whenever the sufficient resources are available, e.g., a free processor core plus a suitable chunk of memory, it can take an arbitrary equation from the queue and start its evaluation in a new process. When the processing of an equation is finished, it sends a signal about this, returning the result. Seen from the calling control algorithm, the task manager can receive messages of the form

enqueue(e), e being a defining equation,

and sends messages back of the form

finished($e, \mathit{df}_1, \mathit{df}_2, \ldots$), e being a defining equation, $\mathit{df}_1, \mathit{df}_2, \ldots \in \mathit{DataFile}$.

Such a message should guarantee that the task referred to in e has been applied correctly in order to produce the resulting file values $\mathit{df}_1, \mathit{df}_2, \ldots$ according to the standard semantics $[\![-]\!]$. A parallel operational semantics can now be given by the following abstract algorithm.

```
**Algorithm 4:** Parallel operational semantics for BANscript
**Input:** A query $Q$, a BANscript $S$, program semantics $[\![-]\!]$
        and initial substitution $\Phi_0$;
**Output:** A substitution;

$\Phi := \Phi_0$;
$E := \mathit{Eq}(Q,S)$;   // Equations not yet enqueued
$F := \emptyset$;          // Equations that have been processed
```

```
while $F \neq Eq(Q,S)$ do
    while there is an $e \in E$ of the form
            $\langle \widehat{f_1^{out}}, \ldots, \widehat{f_m^{out}} \rangle \doteq [\![P]\!] \langle \widehat{f_1^{in}}, \ldots, \widehat{f_n^{in}} \rangle$
        for which $\Phi(f_i^{out})$ is undefined for all $i = 1, \ldots, m$,
            and $\Phi(f_j^{in})$ is defined for all $j = 1, \ldots, n$
    do  enqueue($e$);
        $E := E \setminus \{e\}$;
    await message finished($e', df_1, df_2, \ldots$);
    $\Phi := \Phi[\widehat{f_1'^{out}}/df_1, \ldots, \widehat{f_{m'}'^{out}}/df_{m'}]$
        where $e' = \big(\langle \widehat{f_1'^{out}}, \ldots, \widehat{f_{m'}'^{out}} \rangle \doteq \cdots\big)$;
    $F := F \cup \{e'\}$;
return $\Phi$;
```

Correctness is straightforward as this algorithm performs exactly the same file assignments as algorithm 1. We refrain from a formal exposition.

This algorithm is implemented in our system for a multicore computer, but it should also work for other architectures such as grids and clusters. We are considering an enhanced implementation that combines Algorithms 3 and 4 such that each time a resource is reported modified, the maximum number of processors are put to work to restore consistency. This may involve stopping active processes, or removing them from the queue, when input files are outdated.

10.5 Types and type inference for BANpipe scripts

The system includes a dynamic type system such that, for a given program call, the output files are assigned types based on the types of the input files. These types are programmer-defined and may not indicate anything about the internal structure of the file. It is up to the programmer to associate a meaning with the types, and they are only used by the system for checking the overall sensibility of a script. Types are not visible in a script, but are managed through optional declarations in the interface files (cf. section 10.2.1) and are checked separately by a symbolic execution of the program as explained in the following.

A type can be any Prolog term. An URL file has a default type `file`, which may be coerced into a more specific type by a variant of the `file::get` task, illustrated as follows.

```
f <- file::get(['http://server/file.html'],type(text(html))).
```

A task specified as in (10.14) may have an associated, polymorphic type declaration of the following form, specifying requirements for its input and output files.

$$\texttt{type}(\textit{prog}, \texttt{(}\texttt{[}\textit{Type}_1^{in}, \ldots, \textit{Type}_n^{in}\texttt{]}\texttt{,}\, \textit{options}\texttt{,}\, \texttt{[}\textit{Type}_1^{out}, \ldots, \textit{Type}_m^{out}\texttt{]}\texttt{)} \quad (10.23)$$

Each $\mathit{Type}_i^{in/out}$ and *options* are Prolog terms, possibly with variables, and any variable in a Type_j^{out} must occur in some Type_k^{in} or *options*. Thus, if the types for n actual input files plus actual option values simultaneously unifies with $\mathit{Type}_1^{in}, \ldots, \mathit{Type}_n^{in}, \mathit{opts}$, unique ground instances are created for $\mathit{Type}_1^{out}, \ldots, \mathit{Type}_m^{out}$, which are then assigned as types for m output files.

Correct typing of a well-behaved BANscript S with respect to a given query Q are formalized through a type semantics $[\![-]\!]^{type}$. For each task with n input and m output files, assuming a type declaration as in (10.23) above, we define,

$$\begin{aligned} &[\![\mathit{mod}::\mathit{prog}(\mathit{opts})]\!]^{type}\langle x_1, \ldots, x_n\rangle = \langle \mathit{Type}_1^{out}, \ldots, \mathit{Type}_m^{out}\rangle\rho \\ &\qquad \text{where } \rho \text{ is the unifier of } \langle x_1, \ldots, x_n, \mathit{opts}\rangle \\ &\qquad\qquad \text{and } \langle \mathit{Type}_1^{in}, \ldots, \mathit{Type}_n^{in}, \mathit{options}\rangle \end{aligned} \tag{10.24}$$

If the mentioned unifier does not exists, the result is instead $\perp$.

We can now use Algorithm 1 with this semantics for type checking a BANpipe script with respect to a given query as a symbolic execution of the program. The initial substitution Φ_0^{type} maps any URL variable to the type `file`.

```
Algorithm 5: Type inference for BANscript
Input: A query $Q$, a BANscript $S$;
Output: A substitution of variables into types;

$\Phi^{type}$ := run Algorithm 1 for $Q$, $S$, $[\![-]\!]^{type}$ and $\Phi_0^{type}$;
return $\Phi^{type}$;
```

If, in the resulting substitution, any file is mapped to $\perp$, we say that type checking of S failed for Q; otherwise type checking succeeded. Taking $[\![-]\!]^{type}$ as definition of correct typing, correctness of this algorithm is a consequence of theorem 1.

10.6 Examples

We exemplify BANpipe using examples drawn from biological sequence analysis and machine learning. In the first example we present a simple gene prediction pipeline and in the second example we show how such a pipeline can be extended with recursive rules used to implement self-training.

10.6.1 A basic gene prediction pipeline

The following is an example of a simple gene prediction pipeline, corresponding to experiments previously reported [43]. The premise is to train a gene finder expressed as a PRISM model using some of the known genes of the *Escherischia Coli* genome (the training set) and verify its prediction accuracy on a different set of known genes (the test set). First we get some initial data files; namely a genome sequence (`fasta_seq`) and a list of reference genes (`genes_ptt`),

```
fasta_seq <- file::get(['ftp://ftp.ncbi.nih.gov/.../NC_000913.fna']).
genes_ptt <- file::get(['ftp://ftp.ncbi.nih.gov/.../NC_000913.ptt']).
```

The fetched files are parsed into suitable format (Prolog facts),

```
genome <- fasta::parse([fasta_seq]).
genes(reference) <- ptt::parse([genes_ptt]).
```

We then extract all open reading frames (`orfs`) from the genome, and divide them into a training set and a test set,

```
orfs <- sequence::extract_orfs([genome]).
orfs(training_set), orfs(test_set) <- file::random_split([orfs],seed(42)).
```

Slightly simplified, open reading frames are subsequences of the genome which may contain genes. The `random_split` program divides the `orfs` randomly into the two files `orfs(training_set)` and `orfs(test_set)`. The process is deterministic due to the `seed(42)` option, i.e. it will split `orfs` in the same way if it is rerun. Next, we extract known genes corresponding to each set,

```
genes(Set) <- ranges::intersect([genes(reference),orfs(Set)]).
```

The `ranges` module contains tasks which deal with a files containing particular facts which besides representing sub-sequences also includes their positions in a genome. The `intersect` task finds all facts from `genes(reference)` where the represented sub-sequences are completely overlapped[4] by a member of `orfs(Set)`. It is used here to find the reference genes that belong to some `Set`, i.e., either the `training_set` or the `test_set`. This concludes the preparation of data files and we turn to the rules for the gene finder,

```
params <- genefinder::learn([genes(training_set)]).
predictions <- genefinder::predict([orfs(test_set), params]).
report <- accuracy::measures([genes(test_set), predictions]).
```

The `genefinder` module contains a PRISM based gene finder and the `learn` task bootstraps and invokes PRISMs machine learning procedure from the facts in the file `genes(traning_set)`. The resulting `params` file is a parameterization for the PRISM model. Using this parameterization, the task `predict` probabilistically predicts which orfs in `orfs(test_set)` represent genes, resulting in the file `predictions`. Finally, the accuracy of the `predictions` are evaluated with regards to the reference genes, `genes(test_set)`, by the task `accuracy::measures` which calculates, e.g., sensitivity and specificity.

10.6.2 Self-training

Self-training has been demonstrated to yield improved gene prediction accuracy [18]. A self-training gene finder can be expressed by mutually recursive rules,

```
known_genes <- ...
self_learn(1) <- genefinder::learn([known_genes]).
self_learn(N) <- N > 1, N1 is N-1 | genefinder::learn([predict(N1)]).
predict(N) <- genefinder::predict([self_learn(N)]).
```

To elaborate: `known_genes` (obtained somehow) is the starting point for training and `self_learn(1)` is the parameter file resulting from training on `known_genes`. The second `self_learn(N)` rule is the recursive case, which learns parameters from the predictions of the the previous iteration. The goal

[4]For the biologically inclined; we require this overlap to be in the same reading frame.

`predict(N)` produces a set of gene predictions based on the parameters of obtained from `self_learn(N)`. For instance, the goal `predict(100)` corresponds to predictions after 100 iterations of self-training.

It is straight-forward to extend this example to more advanced self-training, e.g., co-training [20] where multiple models generate training data for each other.

10.7 Related work

Computational pipelines are ubiquitous. The classic example is Unix pipes, which feed the output of one program into another. Declarative pipeline languages with non-procedural semantics goes back at least to the make utility [65].

The importance of pipelines for biological sequence analysis has been acknowledged [146] and there are a variety of biological pipeline languages to choose from, e.g., EGene[62], BioPipe[95], DIYA[202], and SKAM[139]. The first three are configured using a static XML format with limited expressivity. DIYA targets annotation of bacterial genomes which has also been a motivating case for us. SKAM is a Prolog based pipeline with a syntax that resembles makefiles, but which borrows Prologs expressive power and provides built-in support for iteration. Dependencies are realized through nesting of functors, but are specified between tasks rather than files.

The family of concurrent logic languages [193] have syntactical and semantic similarities to BANpipe. Rules have guards and the successful execution of a guard implies a committed choice to evaluate the body of a rule. In *flat* variants of the these languages, guards are restricted to a predefined set of predicates as opposed to arbitrary used defined predicates. BANpipe allows user defined predicates, but for the semantics to be well-defined, these are subject to obvious restrictions, e.g. they should terminate. BANpipe rules have a single goal in the rule body, whereas concurrent logic languages typically have conjunctions of goals. These languages execute in parallel and synchronize computations by suspension of unification, which may be subject to certain restrictions. For instance, in Guarded Horn Clauses [214], the guard is not allowed to bind variables in the head and the body may not bind variables in the guard. BANpipe have more restricted assumptions; the guard never binds variables in the head and the body never binds variables in the head or the guard.

10.8 Conclusions

BANpipe is a declarative logic-language for modeling of computational pipelines with time consuming analyses. We gave abstract and operational semantics for BANpipe, which was extended for change propagation and parallelism and type checking. The language was illustrated with examples inspired from gene finding.

Our implementation of BANpipe supports all the concepts and extensions presented in this paper. The language is implemented as an interpreted domain specific language in B-Prolog/PRISM and the implementation is mature enough to handle pipelines of considerable size. Future versions may support other Prolog systems and adapt to a distributed setting by realizing tasks as web services.

Acknowledgement This work is part of the project "Logic-statistic modeling and analysis of biological sequence data" funded by the NABIIT program under the Danish Strategic Research Council.

Chapter 11

Efficient Tabling of Structured Data using Indexing and Program Transformation

With Henning Christiansen

In Proceedings of Practical Aspects of Declarative Languages (PADL'12), Springer LCNS, volume 7149, 2012.

Abstract

Tabling of structured data is important to support dynamic programming in logic programs. Several existing tabling systems for Prolog do not efficiently deal with structured data, but duplicate part of the structured data in different instances of tabled goals. As a consequence, time and space complexity may often be significantly higher than the theoretically optimal. A simple program transformation is proposed which uses an indexing of structured data that eliminates this problem, and drastic improvements of time and space complexity can be demonstrated. The technique is demonstrated for dynamic programming examples expressed in Prolog and in PRISM.

11.1 Introduction

Tabling in logic programming systems is an established technique which can give a significant speed-up of program execution and make it easier to write efficient programs in a declarative style. Basically, tabling means that the system maintains a table of calls and their answers and each time a new call is entered, it is checked if it (or a perhaps more general call, cf. [208]) is stored in the table already; if so, there is no need to execute it again and a previously found solution is used. It is included in several recognized Prolog systems such as B-Prolog [226], XSB [205] and YAP [164].

However, we can demonstrate that these systems may waste unnecessary time and space for copying and matching structures in situations where operations on single pointers could have been used instead. This can be the case when a program is called with a huge, ground structure as one of its arguments, and this argument is decomposed into sub-structures which are tabled independently.

In addition to pointing out the problem, we can show how it can be bypassed by a straightforward program transformation and a few auxiliary predicates that can be written in plain Prolog. A significant speed-up is demonstrated for selected test programs. In a longer perspective, we advocate such techniques be incorporated into logic programming systems with tabling such as those mentioned, where it can be implemented at a lower level where machine address pointers are available rather that using a high-level "simulation" of pointers as we do here.

Our own background for working with this problem is work on analysis of biological sequence data using the probabilistic-logic system PRISM [180] which is implemented on top of B-Prolog and which is heavily dependent on its tabling mechanism. Together with another general program transformation based technique that we have developed [40], which improves the performance of tabling in the presence of non-discriminatory arguments, the technique described in the present paper increases significantly the size of sequences that can be meaningfully analyzed by means of PRISM programs.

Section 11.2 introduces the problem with tabling of structured data through an example. Section 11.3 describes an indexed representation of structured data that circumvents the problem, and section 11.4 demonstrates the effect for two problems, a dynamic programming problem in Prolog in section 11.4.1 and a PRISM program in section 11.4.2. Section 11.5 describes a general and automatic program transformation. Section 11.6 discuss limitations of our approach. Section 11.7 describes related work and section 11.8 sums up and discuss future work.

11.2 The Trouble with Tabling Structured Data

In this section we empirically demonstrate that all major Prolog tabling systems have a problem with structured data. Through the benchmarking of an implementation of the `last/2` predicate — which traverses a list to find the last element — we observe that when this predicate is tabled, time and space complexity is far worse than without tabling.

The following is a straight-forward implementation of the `last/2` predicate.

```
last([X],X).
last([_|L],X) :-
    last(L,X).
```

If `last/2` is called with a list `L` of length N, e.g. `last(L,_)`, then the expected time-complexity of this implementation is clearly $O(N)$. However, if the predicate is tabled, then the tabling system may have to store N partial copies of the list, e.g. the first copy will be the full list, the second copy will just store $N-1$ elements, and so on until every possible tail down to the last element of the list has been tabled. This results in $O(N^2)$ tabled list elements.

Naive copying of the lists hence make the tabled version of `last/2` (at least) quadratic — with regard to both time and space consumption — rather than linear as in the non-tabled version. Tabling systems do employ some advanced techniques to avoid the expensive copying and which may reduce memory consumption and/or time complexity. For instance, B-Prolog uses hashing of goals [230], XSB uses a trie data structure [205] and Yap [164] uses a trie structure, which in [160] is refined into a so-called global trie which applies a sharing strategy for common subterms whenever possible. This can reduce space consumption, but since there is no sharing between the trie and the actual arguments of an active call, each execution of a call may typically involve a full traversal of its arguments.

Nevertheless, as can be witnessed from Figure 11.1, all tabling systems pay a price for structured data in either time or space. The figure shows time and space consumption for `last(L,_)` with varying sizes of `L`, where `L` is either a list of consecutive ones or a list of random numbers generated using the following simple random number generator.

```
random_list(0,_,[]).
random_list(N,Prev,[X|L]):-
    B is (9381*Prev + 12345) mod 32768,
    X is B mod 12,
    N1 is N-1,
    random_list(N1,B,L).
```

The nature of the data seems highly relevant. For instance, YAP and XSB performs better with repeated data and B-Prolog performs better with random data. As can be observed from Figure 11.1 plots a and c, time complexity is larger than linear in all cases, but varies depending the type of data. Space consumption is linear for repeated data in XSB and YAP, but for B-Prolog it is linear regardless of the type of data. The best time complexity is observed for B-Prolog with random data but as can be observed in plot c it is still super-linear. XSB and YAP show a different pattern where the time complexity seems to be more closely coupled to space complexity. For repeated data they are more time-efficient than B-Prolog but still significantly slower than B-Prolog with random data and still distinctively super-linear.

11.3 A Workaround and Its Implementation in Prolog

We present here a workaround that results in $O(1)$ time and space complexity for table lookups for programs with arbitrarily large ground structured data as input arguments. A term is represented as a set of facts, each representing a subterm which is referenced by a unique integer serving as an abstract pointer. Matching related to tabling is done solely by comparison of such pointers, independently of the underlying system. The representation is given by the following predicates which all together can be understood as an abstract datatype.

store_term($+$***ground-term*****,** ***pointer*****)**
: The *ground-term* is any ground term, and the *pointer* returned is a unique reference (an integer) for that term.

retrieve_term($+$***pointer*****,** **?*****functor*****,** **?*****arg-pointers-list*****)**
: Returns the functor and a list of pointers to representations of the substructures of the term represented by *pointer*.

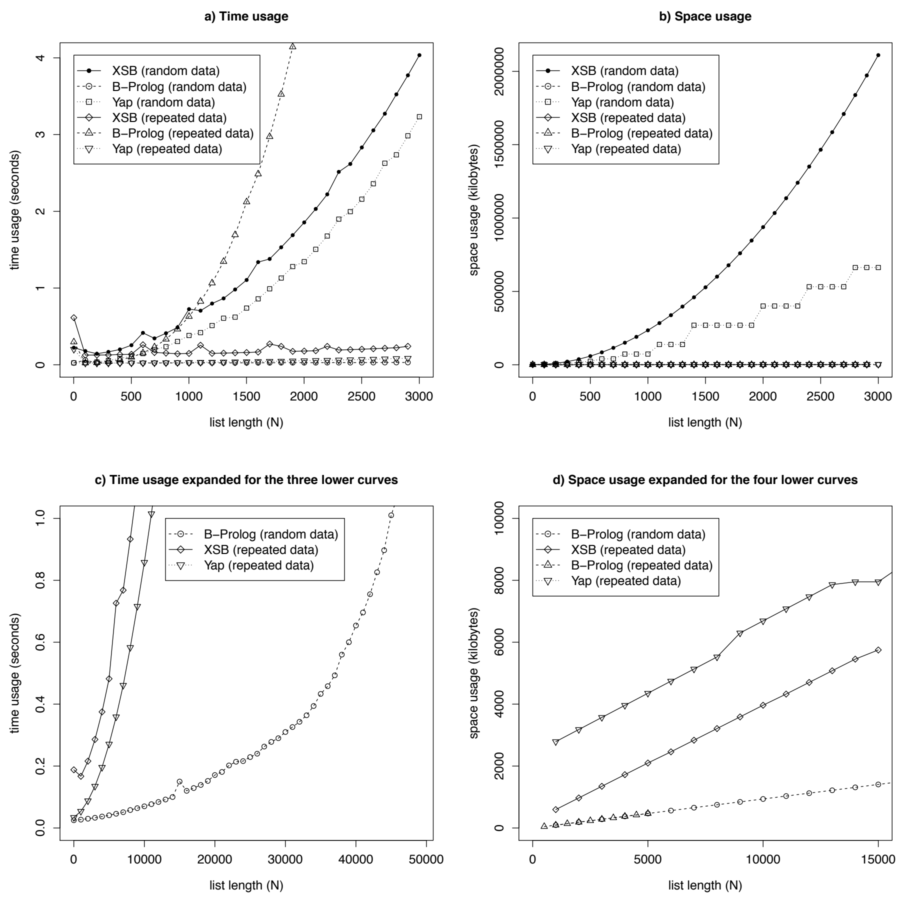


Figure 11.1: Plot a) shows the time consumption of different Prolog engines for running tabled `last/2` with lists of length N and plot b) shows the table space usage. Table space usage is measured using the `statistics/1` predicate (which is different for each Prolog). In Yap it includes no specific measurement of "table space" and we measure instead the "program space" which is taken to include the table space. *Random data* means that the list contained random integers and in the *repeated data* means that the lists containing the same integer repeated N times. Plot c) and d) shows time and table space usage for the curves in a) and b) that looks flat because of the scale, but expanded for larger values of N. The curve of B-Prolog for repeated data in plot d) is truncated at 5000 due to its long running time for larger values.

full_retrieve_term(**+*pointer*, ?*ground-term*)**
Returns the term represented by *pointer*.

More precisely, it must hold for any ground term s, that the query

store_term(s, P), full_retrieve_term(P, S),

assigns to the variable S a value identical to s. Furthermore, it must hold for any ground term s of the form $f(s_1, \ldots, s_n)$ that

store_term(s, P), retrieve_term(P, F, Ss),

assigns to the variable F the symbol f, and to Ss a list of ground values $[p_1, \ldots, p_n]$ such that additional queries

full_retrieve_term(p_i, $\mathtt{S}_i$), $i = 1, \ldots, n$

assign to the variables $\mathtt{S}_i$ values identical to s_i.

Example 11.3.1. *The following call converts the term* `f(a,g(b))` *into its internal representation and returns a pointer value in the variable* `P`.

```
store_term(f(a,g(b)),P).
```

After this, the following sequence of calls will succeed.

```
retrieve_term(P,f,[P1,P2]),
retrieve_term(P1,a,[]),
retrieve_term(P2,g,[P21]),
retrieve_term(P21,b,[]),
full_retrieve_term(P,f(a,g(b))).
```

Example 11.3.2. *One possible way of implementing the predicates introduced above is to have* `store_term/2` *asserting facts for the* `retrieve_term/3` *predicate using increasing integers as pointers. Then the call* `store_term(f(a,g(b)),P)` *considered in example 11.3.1 may assign the value* `100` *to* `P` *and as a side-effect assert the following facts.*

```
retrieve_term(100,f,[101,102]).
retrieve_term(101,a,[]).
retrieve_term(102,g,[103]).
retrieve_term(103,b,[]).
```

Notice that Prolog's indexing on first arguments ensures a constant lookup time.

Example 11.3.3. *In an application for which large numbers of identical subterms are expected, the representation can exploit this for sharing, so for example the term* `h(very(large,sub(term)), very(large,sub(term)))` *may be represented by the pointer value* `200` *and the following facts.*

```
retrieve_term(200,g,[201,201]).
retrieve_term(201,very,[...]).
...
```

This will increase the time complexity for `store_term/2` *but the advantages are i) storage consumption is reduced, and more importantly ii) an additional – and for the right sort of application programs drastic – speed-up may be obtained from the improved utilization of tabling that this automatically implies.*

Finally we introduce a utility predicate which may simplify the use of the representation in application programs. It utilizes a special kind of terms, called *patterns*, which are not necessarily ground and which may contain subterms of the form `lazy(`*variable*`)`.

lookup_pattern($+$***pointer*****,** $+$***pattern*****)**

The *pattern* is matched in a recursive way against the term represented by the pointer p in the following way.

– `lookup_pattern(`p`,X)` is treated as `full_retrieve_term(`p`,X)`.

– `lookup_pattern(`p`,lazy(X))` unifies `X` with p.

– For any other *pattern* `=..` `[F,X`$_1$`,...,X`$_n$`]` we call

`retrieve_term(`p`, F, [P`$_1$`,...,P`$_n$`])`

followed by `lookup_pattern(P`$_i$`,X`$_i$`)`, $i = 1, \ldots, n$.

Example 11.3.4. *Continuing example 11.3.2, we get that*

```
lookup_pattern(100, f(X,lazy(Y)))
```

leads to `X=a` *and* `Y=102`*.*

The `lookup_pattern/2` predicate simplifies the program transformation introduced in section 11.5 although further efficiency can be gained by compiling it out for each specific pattern.

11.4 Examples

The impact of indexing for ground arguments with tabled execution is evaluated through two experiments. Firstly, we compare the performance of existing Prolog systems with tabling for a simple edit distance problem. The second experiment is related to our driving motivation – biological sequence analysis in PRISM, exemplified for probabilistic inference with Hidden Markov Models. All experiments were run on a MacBook Pro with a 2.53 GHz Intel core 2 Duo processor, 4 GB memory and Mac OS X version 10.6.8 (Snow Leopard).

11.4.1 Example: Edit Distance

We consider a minimal edit-distance algorithm written in Prolog which is dependent on tabling for any non-trivial problem. Time and space consumption are measured for increasing problem sizes in the three major tabling systems with and without our indexed representation.

Edit-distance is the textbook example dynamic programming. In the classic imperative formulation of the problem, a matrix with N^2 values is calculated, such that the calculation of the value for each cell is a constant time operation that depends on at most three other cells. The theoretical best time complexity of edit distance has been proven to be $O(N^2)$ [222]. Dynamic programming problems exhibit *optimal sub-structure* which implies that partial solutions can be reused rather than recomputed [14]. Tabling supports dynamic programming since resolved goals are kept in a table and reused rather than re-derived if the tabled goals are called again. The following Prolog program implements *minimal edit distance* between two lists; given two lists L_1 and L_2, the call `edit(`L_1`, `L_2`,D)` will return the minimal number of edits (substitutions,insertions and deletions) needed to change one of the lists into the other.

```
:- table edit/3.

edit([],[],0).

edit([],[Y|Ys],Dist) :-
   edit([],Ys,Dist1),
   Dist is 1 + Dist1.

edit([X|Xs],[],Dist) :-
   edit(Xs,[],Dist1),
   Dist is 1 + Dist1.

edit([X|Xs],[Y|Ys],Dist) :-
   edit([X|Xs],Ys,InsDist),
   edit(Xs,[Y|Ys],DelDist),
   edit(Xs,Ys,TailDist),
   (X==Y ->
       Dist = TailDist
       ;
       % Minimum of insertion, deletion or substitution
       sort([InsDist,DelDist,TailDist],[MinDist|_]),
       Dist is 1 + MinDist).
```

Without tabling the `edit/3` predicate, the same subgoals are derived again and again leading to exponential blowup, but it can be shown that the number of distinct calls are quadratic, which is the actual complexity we may hope for with optimal tabling.

The program has been transformed manually for this experiment based on the pointer based representation shown in example 11.3.2 above, simplified slightly for lists. The `retrieve_term` predicate is applied to resolve pointers during program execution. For completeness, we include a suitable implementation of `store_term/2` and `retrieve_term/2`.

```
store_term([],Index) :- assert(retrieve_term([],Index)).

store_term([X|Xs],Idx) :-
   Idx1 is Idx + 1,
   assert(retrieve_term(Idx,[X,Idx1])),
   store_term(Xs,Idx1).
```

The transformed version of the edit distance program is now as follows.

```
:- table edit/3.
```

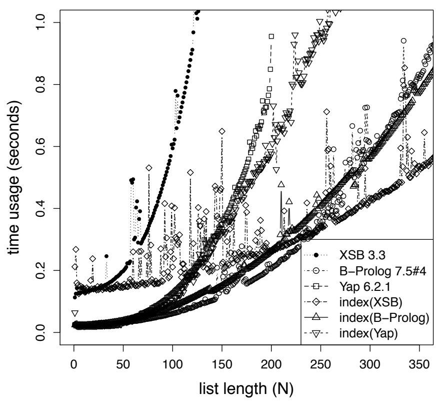
```
edit(XIdx,YIdx,0) :-
   retrieve_term(XIdx,[]),
   retrieve_term(YIdx,[]).

edit(XIdx,YIdx,Dist) :-
   retrieve_term(XIdx,[]),
   retrieve_term(YIdx,[_,YIdxNext]),
   edit(XIdx,YIdxNext,Dist1),
   Dist is Dist1 + 1.

edit(XIdx,YIdx,Dist) :-
   retrieve_term(YIdx,[]),
   retrieve_term(XIdx,[_,XIdxNext]),
   edit(XIdxNext,YIdx,Dist1),
   Dist is Dist1 + 1.

edit(XIdx,YIdx,Dist) :-
   retrieve_term(XIdx,[X,NextXIdx]),
   retrieve_term(YIdx,[Y,NextYIdx]),
   edit(XIdx,NextYIdx,InsDist),
   edit(NextXIdx,YIdx,DelDist),
   edit(NextXIdx,NextYIdx,TailDist),
   (X==Y ->
       Dist = TailDist
       ;
       sort([InsDist,DelDist,TailDist],[MinDist|_]),
       Dist is 1 + MinDist).
```

The program is tested for randomly generated sequences of increasing lengths. We measure the total time for the different Prolog engines to load the program file, generate two different random sequences of a particular length, assert these lists using `store_term/2` and compute edit distance between these sequences, as follows.

```
run(N) :-
   random_list(N,117,L1), % Generate random list L1 with seed 117
   random_list(N,42,L2),  % Generate random list L1 with seed 42
   store_term(L1,P1),
   store_term(L2,P2),
   edit(P1,P2,_Dist).
```

The results, shown in Figure 11.2, demonstrate that all tested Prolog systems use more time for the unmodified tabled edit distance program than for the transformed program when applied to large data instances. For XSB and Yap the major factor impacting time complexity seems to be space consumption. The transformation has a positive impact space complexity regardless of the underlying tabling strategy. For B-Prolog, space consumption is much closer to the theoretical $O(N^2)$. Even though B-Prolog is very space efficient, the transformed program still uses less memory.

For larger problem instances the transformation has a significant impact on time complexity. XSB seems to benefit greatly from the transformation, although it starts out the slowest, it catches up for longer sequences, where it outperforms the two other Prologs in time efficiency. Yap seems to gain a modest boost from the transformation strategy and still seems to have a rather high time complexity although it is significantly faster than without the transformation. For B-Prolog, the two versions perform more or less the same for sequences of length up to 350, but for longer sequences (not shown in the figure) the transformed version is significantly faster: for example, with length 1000, the execution times are 7.5 seconds for the transformed and 21.5 seconds for the original version.



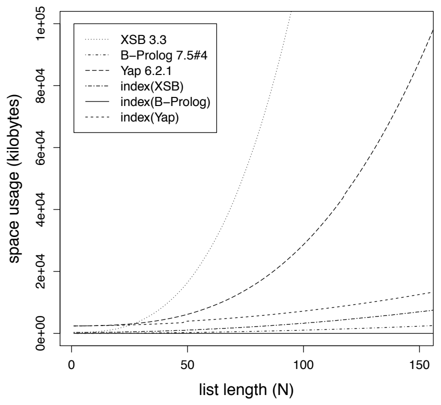


Figure 11.2: The first plot shows the time consumption of different Prolog engines for edit distance with two lists of length N. The second is a plot of the space consumption for the same calls. The plots for both normal tabled in execution and execution of a transformed program that uses indexing as described in section 11.3 are shown, e.g. the plot $index(X)$ shows the performance of the transformed version for Prolog implementation X.

11.4.2 Example: Hidden Markov Model in PRISM

PRISM [180] is an extension of Prolog with special goals representing random variables. A global declaration such as `values(coin,[head,tail])` introduces a so-called multivalued switch which means that an occurrence of the subgoal `msw(coin,C)` represents a probabilistic choice of assigning either `head` or `tail` to `C`. The semantics of PRISM is defined in terms of probabilistic Herbrand models, which means that a program specifies a probability of any goal G to be true determined from the possible combinations of `msw` outcomes that happen to make G true.

The PRISM system supports various probabilistic inferences, such as finding an optimal derivation, computing the probability for a goal or deriving `msw` probabilities by learning from a set of goals. The algorithms behind these inferences are dynamic programming algorithms and PRISM is implemented in B-Prolog [226], relying heavily on tabling for the efficiency of the probabilistic inferences.

We consider the example of a Hidden Markov Model (HMM) in PRISM taken from the PRISM manual [190] and adapted here to accommodate variable length sequences. In general, an HMM is a probabilistic model for sequential phenomena based on a finite automaton, which chooses state transitions and emissions by probabilistic choices; see [159] for a general introduction to HMMs and [44] for an account on how different HMMs are expressed in PRISM. Our example program is the following.

```
values(init,[s0,s1]).
values(out(_),[a,b]).
values(tr(_),[s0,s1]).

hmm(L):-
    msw(init,S),
    hmm(S,L).
```

```
hmm(_,[]).

hmm(S,[Ob|Y]) :-
    msw(out(S),Ob),
    msw(tr(S),Next),
    hmm(Next,Y).
```

The `init`, `out(−)` and `tr(−)` switches determine initial state, state transitions and emissions. Notice that two last ones are parameterized meaning that they define a switch for whatever value is substituted in for the parameter, which in this program always is the present state.

Using the same list encoding as in the previous example, the recursive predicate is rewritten as follows.

```
hmm(S,ObsPtr):-
   retrieve_term(ObsPtr,[]).

hmm(S,ObsPtr) :-
   retrieve_term(ObsPtr,[Ob,Y]),
   msw(out(S),Ob),
   msw(tr(S),Next),
   hmm(Next,Y).
```

The rewritten program can be shown to be semantics preserving wrt a standard Prolog semantics as well as PRISM's probabilistic semantics, and thus running PRISM's utilities for probability calculations should yield the same results.

When calculating the probability of a given goal, PRISM iterates over all possible ways to execute the goal using tabling to avoid enumerating the exponential number of different derivations. The same principle applies for PRISMs version of the Viterbi algorithm which is a dynamic programming algorithm that finds the most probable derivation. Assuming optimal execution of tabling, these algorithms should in principle run in linear time.

We measured running times of probability calculations (`prob` in PRISM lingo) for both the original and the transformed version of the PRISM HMM program with sequences of increasing lengths from 100 to 5000. The actual sequences used are instances of the pattern `[a,b]` repeated a number of times. The results are shown in Figure 11.3. It is apparent from the figure that indexed lookups results in approximately linear running time while the running time is at least quadratic for the unmodified program. The reported times are measured using `prism_statistics(infer_time,Time)`, which is a PRISM built-in predicate.

We did not measure running times of sequences longer than 5000 for the unmodified program, but the transformed program scales up to sequences much longer than this, for instance, the time for probability calculation for a sequence of length 100000 takes less than 5 seconds.

11.5 Automatic Program Transformation

The indexed versions of the example programs shown in section 11.4 can be produced automatically by a straightforward program transformation. The user must declare modes for which arguments predicate arguments that should be indexed. For the HMM program of section 11.4.2 this may look as follows; plus means transform the argument, minus means keep it unchanged.

```
table_index_mode(hmm(+))
table_index_mode(hmm(-,+))
```

The correctness of the transformation depends on the following properties of the program.

- The arguments indicated for indexing must be called with ground data only.
- Variables that occur in an indexed argument in the head of a clause, cannot occur in the body of that clause in both an indexed and a non-indexed argument.
- Any argument of a goal within a clause body which is declared to be indexed, must be given as a variable that also occurs in an indexed argument in the head of that clause.

Each clause whose head predicate is covered by a table mode declaration is transformed using the procedure outlined in algorithm 1, and all other clauses are left untouched. The transformation moves any term appearing in an indexed position in the head of a clause into a call to the `lookup_pattern` predicate, which is added to the body. Variables in such terms are marked lazy when they do not occur in any non-indexed argument inside the clause.

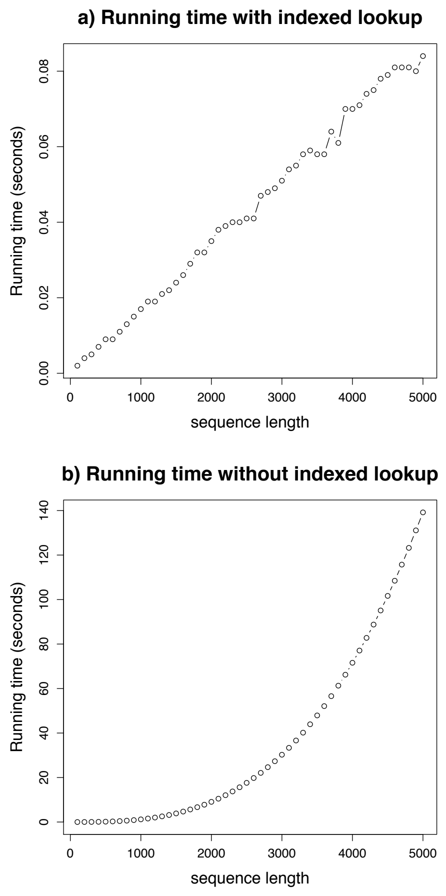


Figure 11.3: The running time of the a) the transformed PRISM program and b) the unmodified PRISM program. While a) shows a linear development as function of sequence length, the development in b) is a higher-degree polynomial. Notice also the different scales on the vertical axes.

This transformation can be shown to be semantics preserving for programs satisfying the requirements given above.

The translation can be further enhanced by an unfolding of `lookup_pattern` calls into specialized calls to `retrieve_term` as shown in the examples in the previous section. This last step gave a speed-up of a factor of 5 for these examples when comparing with implementations using `lookup_pattern` directly.

11.6 Limitations

Our transformation assumes ground input arguments. As illustrated by the examples, this has applications to a lot of interesting problems, in particular dynamic programming problems. With regard to PRISM, our transformation is useful for ordinary probability calculation, Viterbi decoding and supervised

```
for each clause H:-B in original program do
    if table_index_mode (M) matching H then
        for each argument $H_i \in$ H, $M_i \in$M do
            if $M_i =$'+' then
                $H'_i \leftarrow$ MarkLazy$(H_i, B)$
                $B \leftarrow (lookup\_pattern(V_i, H'_i), B)$
                $H_i \leftarrow V_i$
        end
end
```

where MarkLazy is defined as

```
MarkLazy($H_i$,B) :
  $PotentialLazy$ = variables in all goals $G \in$ B
    where $G$ has table_index_mode declaration
  $NonLazy$ = variables in all goals $G \in$ B
    where $G$ has no table_index_mode declaration
  $Lazy = PotentialLazy \setminus NonLazy$
  for each variable $V \in H_i$ do
      if $V \in Lazy$ then
          $V \leftarrow lazy(V)$
  end
```

Algorithm 1: Program transformation.

learning. For other probabilistic inferences such as sampling, posterior decoding, unsupervised and semi-supervised learning, arguments containing variables are required. Sampling is of minor concern, since this can be done in linear time using the original program.

We currently have no optimization for structured terms in output arguments – they must be handled by the usual tabling mechanism. Structured terms in output arguments have the same consequences for complexity, which can be observed for instance with the well-known `append/3` predicate. Suppose that `append/3` is tabled and transformed using our approach, e.g. with `table_index_mode(+,+,-)`. Using our workaround, the space complexity for the input lists will be kept linear rather than quadratic, but the answers for the third list is tabled in the usual way which leads to quadratic space complexity nevertheless. Output arguments that do not contain structured data — as in the case of edit distance — do not present such a problem since the output argument is of constant size.

A drawback of our transformation is that it, by replacing the patterns in the head of rules with pointers, circumvents Prolog own indexing mechanism. As result, indexing cannot use the pattern of the arguments to determine which clauses to try. Instead, when multiple clauses with same name and arity exist, Prolog will have to try each of them in order and creates a choice point each time it tries a clause. This adds a constant factor — corresponding to the number of such clauses — to the running time of the program. It most practical programs it is realistic to assume that this factor will be fairly low, e.g. in the edit distance program only four such clauses exist.

11.7 Related work

The hashing employed by B-Prolog and the global trie of YAP [160] address a related problem. Both methods reduce space consumption and this may lead to reductions in running time since less copying is needed. However, even with these mechanisms complexity is sub-optimal as shown in section 11.2. Furthermore, the methods have the drawback that the running time depends on the type of data. In comparison, our approach is data invariant and yields optimal complexity.

Due to restrictions in the Mercury language, input arguments are always ground, and the tabling system provides an option which identifies arguments by their pointers [199] (see also more detailed explanations in the reference manual [92]). This yields constant time storing and comparison of tabled arguments, similar to how any standard tabling mechanism will work for the programs produced by our program transformation.

The problem with tabling of structured data has addressed in applications with methods similar to our approach. In particular, in chart-parsing with DCGs supported by tabling, position indexed facts has been used [203]. A similar approach has been applied to PCFG parsing in PRISM [184]. This works by splitting the input list, $t_1 \ldots t_N$ into facts, `{pos(`$1, t_1, 2$`), ..., pos(`$N-1, t_n, N$`)}`. XSB Prolog have special constructions for tabled DCGs, where the standard `'C'/3'` predicate is replaced by a special version that instead of using difference lists, utilize position indexed facts constructed from the original input list [204]. The position indexed difference list approach is quite similar to our approach, but is specific for difference lists. Our approach is more generally applicable and can be used with various kinds of structured data.

11.8 Conclusion

We have demonstrated that major Prolog implementations do not efficiently handle tabling of structured data and we have provided a program transformation that ensures $O(1)$ time and space complexity of tabled lookups of goals with structured data in input arguments and is applicable regardless of inefficiencies with structured data in the underlying tabling implementation. We have demonstrated the applicability of our transformation using examples from dynamic programming in Prolog and PRISM. The transformation makes it possible to scale to much larger problem instances.

Our program transformation should be seen as workaround, until such optimizations find their way into the tabling systems. We hope that Prolog implementors will pick up on this and integrate such optimizations directly in the tabling systems, so that the user does not need to transform his program, and need not worry about the underlying tabled representation and its implicit complexity.

Acknowledgements. This work is supported by the project “Logic-statistic modelling and analysis of biological sequence data” funded by the NABIIT program under the Danish Strategic Research Council.

We would like to thank Neng-Fa Zhou and Yoshitaka Kameya for their encouraging comments on a very early draft of this paper. We would also like to thank to the anonymous reviewers for insightful and constructive reviews.

Chapter 12

Efficient Tabling of Structured Data with Enhanced Hash-Consing

With Neng-Fa Zhou

In Theory and Practice of Logic Programming, Volume 12, Special Issue 4-5, July 2012.

Abstract

Current tabling systems suffer from an increase in space complexity, time complexity or both when dealing with sequences due to the use of data structures for tabled subgoals and answers and the need to copy terms into and from the table area. This symptom can be seen in not only B-Prolog, which uses hash tables, but also systems that use tries such as XSB and YAP. In this paper, we apply hash-consing to tabling structured data in B-Prolog. While hash-consing can reduce the space consumption when sharing is effective, it does not change the time complexity. We enhance hash-consing with two techniques, called *i*nput sharing and *h*ash code memoization, for reducing the time complexity by avoiding computing hash codes for certain terms. The improved system is able to eliminate the extra linear factor in the old system for processing sequences, thus significantly enhancing the scalability of applications such as language parsing and bio-sequence analysis applications. We confirm this improvement with experimental results.

12.1 Introduction

Tabling, as provided in logic programming systems such as B-Prolog [229], XSB [205], YAP [51], and Mercury [199], has been shown to be a viable declarative language construct for describing dynamic programming solutions for various kinds of real-world applications, ranging from program analysis, parsing, deductive databases, theorem proving, model checking, to logic-based probabilistic learning. The main idea of tabling is to memorize the answers to subgoals in a table area and use the answers to resolve their variant or subsumed descendants. This idea of caching previously calculated solutions, called *m*emoization, was first used to speed up the evaluation of functions [135]. Tabling can get rid of not only infinite loops for bounded-term-size programs but also redundant computations in the execution of recursive programs. While Datalog programs require tabling only subgoals with atomic arguments, many other programs such as those dealing with complex language corpora or bio-sequences require tabling structured data. Unfortunately, none of the current tabling systems can process structured data satisfactorily. Consider, for example, the predicate is_list/2:

```
:-table is_list/1.
is_list([]).
is_list([_|L]):-is_list(L).
```

For the subgoal is_list([1,2,...,N]), the current tabled Prolog systems demonstrate a higher complexity than linear in N: B-Prolog (version 7.6 and older) consumes linear space but quadratic time; YAP, with a global trie for all tabled structured terms [160], consumes linear space but quadratic time; XSB is quadratic in both time and space. The nonlinear complexity is due to the data structure used to represent tabled subgoals and answers and the need to copy terms into and from the table area.

The inefficiency of early versions of B-Prolog in handling large sequences has been reported and a program transformation method has been proposed to index ground structured data to work around the problem [87]. In old versions of B-Prolog, tabled subgoals and answers were organized as hash tables, and *i*nput sharing was exploited to allow a tabled subgoal to share its ground structured arguments with its answers and its descendant subgoals. Input sharing enabled B-Prolog to consume only linear space for the tabled subgoal is_list([1,2,...,N]). Nevertheless, since the hash code was based on the first three elements of a list, the time complexity for a query like is_list([1,1,...,1]) was quadratic in the length of the list. B-Prolog didn't support *o*utput sharing, i.e. letting different answers share structured data. Therefore, on the tabled version of the permutation program that generates all permutations through backtracking, B-Prolog would create $n \times n!$ cons cells where n is the length of the given list.

This problem with tabling structured data has been noticed before and several remedies have been attempted. One well known technique used in parsing is to represent sentences as position indexed facts rather than lists. XSB provides tabled grammar predicates that convert list representation to position representation by redefining the built-in predicate 'C'/3.[1] The position representation is also used for PCFG parsing in PRISM [185]. A program transformation method has been proposed to index ground structured data to work around the quadratic time complexity of B-Prolog's tabling system [87]. Nevertheless, these remedies have their limitations: the position representation disallows natural declarative modeling of sequences and the program transformation incurs considerable overhead. Have and Christiansen advocate for native support of data sharing in tabled Prolog systems for better scalability of their bio-sequence analysis application [87].

We have implemented full data sharing in B-Prolog in response to the manifesto. In the new version of B-Prolog, both input sharing and output sharing are exploited to allow tabled subgoals and answers to share ground structured

[1]Personal communication with David S. Warren, 2011.

data. *H*ash-consing [63], a technique originally used in functional programming to share values that are structurally equal [80, 11], is adopted to memorize structured data in the table area. This technique avoids storing the same ground term more than once in the table area. While hash-consing can reduce the space consumption when sharing is effective, it does not change the time complexity. To avoid the extra linear time factor in dealing with sequences, we enhance hash-consing with input sharing and hash code memoization. For each compound term, an extra cell is used to store its hash code.

Our main contribution in this paper is to apply hash-consing to tabling and enhance it with techniques to make it time efficient. The resulting system demonstrates linear complexity in terms of both space and time on the query `is_list(L)` for any kind of ground list `L`. As another contribution, we also compare tries with hash consing in the tabling context. As long as sequences are concerned, a trie allows for sharing of prefixes while hash-consing allows for sharing of ground suffixes. While we can build examples that arbitrarily favor one over the other, for recursively defined predicates such as `is_list`, it is more common for subgoals to share suffixes than prefixes. The enhanced hash-consing greatly improves the scalability of PRISM on sequence analysis applications. Our experimental results on a simulator of a hidden Markov model show that PRISM with enhanced hash-consing is asymptotically better than the previous version that supports no hash-consing.

The remainder of the paper is structured as follows: Section 12.2 defines the primitive operations on the table area used in a typical tabling system; Section 12.3 presents the hash tables for subgoals and answers, and describes the copy algorithm for copying data from the stack/heap to the table area; Section 12.4 modifies the copy algorithm to accommodate hash-consing; Section 12.5 describes the techniques for speeding up computation of hash codes; Section 12.6 evaluates the new tabling system with enhanced hash-consing; Section 12.7 gives a survey of related work; and Section 12.8 concludes the paper.

12.2 Operations on the Table Area

A tabling system uses a data area, called *t*able area, to store tabled subgoals and their answers. A tabling system, whether it is suspension-based SLG [33] or iteration-based linear tabling [229], relies on the following three primitive operations to access and update the table area.[2]

Subgoal lookup and registration: This operation is used when a tabled subgoal is encountered in execution. It looks up the subgoal table to see if there is a variant of the subgoal. If not, it inserts the subgoal (termed a *p*ioneer or *g*enerator) into the subgoal table. It also allocates an answer table for the subgoal and its variants. Initially, the answer table is empty. If the lookup finds that there already is a variant of the subgoal in the table, then the record stored in the table is used for the subgoal (called a *c*onsumer). Generators and consumers are dealt with differently. In linear tabling, for example, a generator is resolved using clauses and a consumer is resolved using answers; a generator is iterated until the fixed

[2]The interpretation of these operations may vary depending on implementations.

point is reached and a consumer fails after it exhausts all the existing answers.

Answer lookup and registration: This operation is executed when a clause succeeds in generating an answer for a tabled subgoal. If a variant of the answer already exists in the table, it does nothing; otherwise, it inserts the answer into the answer table for the subgoal. When the lazy consumption strategy (also called local strategy) is used, a failure occurs no matter whether the answer is in the table or not, which drives the system to produce the next answer.

Answer return: When a consumer is encountered, an answer is returned immediately if any. On backtracking, the next answer is returned. A generator starts consuming its answers after it has exhausted all its clauses. Under the lazy consumption strategy, a top-most looping generator does not return any answer until it is complete.

12.3 Hash Tables for Subgoals and Answers

The data structures used for the table area are orthogonal to the tabling mechanism, whether it is suspension-based or iteration-based; they can be hash tables, tries, or some other data structures. In this section, we consider hash tables and the operations for the table area without data sharing.

A hash table, called a *s*ubgoal table, is used for all tabled subgoals. For each tabled subgoal and its variants, there is a record in the subgoal table, which includes, amongst others, the following fields:

`Answer`Table:	Pointer to the answer table for the subgoal
`sym`:	The functor of the subgoal
`A1`...An:	The arguments the subgoal

When a tabled predicate is invoked by a subgoal, the subgoal table is looked up to see if a variant of the subgoal exists. If not, a record is allocated and the arguments are copied from the stack/heap to the table area. The copy of the subgoal shares no structured terms with the original subgoal and all of its variables are numbered so that they have different identities from those in the original subgoal.

The record of a subgoal in the subgoal table includes a pointer to another hash table, called an *a*nswer table, for storing answers produced for the subgoal. For each answer and its variants, there is a record in the answer table, which stores amongst others a pointer to a copy of the answer. When an answer is produced for a subgoal, the subgoal's answer table is looked up to see if a variant of the answer exists. If not, a record is allocated and the answer is copied from the stack/heap to the table area. The answers in a subgoal's answer table are connected from the oldest one to the newest one such that they can be consumed by the subgoal one by one through backtracking.

In the implementation, a hash table is represented as an array. To add an item into a hash table, the system computes the hash code of the item and uses the hash code modulo the size of the array to determine a slot for the item. All items hashed to the same slot are connected as a linked list, called a *h*ash chain. A hash table is expanded when the number of records in it exceeds the size of the array.

The WAM representation [219] is used to represent both terms on the heap and terms in the table area except that variables in tabled terms are numbered. A term is represented by a word containing a value and a tag. The tag distinguishes the type of the term. It may be `REF` denoting a reference, `ATM` an atomic value, `STR` a structure, `LST` a cons, or `NUMVAR` a numbered variable. A `STR`-tagged reference to a structure $f(t_1, \ldots, t_n)$ points to a block of $n + 1$ consecutive words where the first word points to the functor f/n in the symbol table and the remaining n words store the n components of the structure. An `LST`-tagged reference to a list cons $[H|T]$ points to a block of two consecutive words where the first word stores the car H and the second word stores the cdr T.

Figure 12.1 gives the definition of the function `copy_term` that copies a numbered term from the stack/heap to the table area. The hash function is designed in such a way that the hash code of a non-ground term is always 0. The function call `seq_hcode(code1,code2)` gives the combined hash code of the two hash codes from two components:

```
int seq_hcode(int code1, int code2){
    if (code1==0) return 0;
    if (code2==0) return 0;
    return code1+31*code2+1;
}
```

If either code is 0, then the resulting code is 0 too.[3]

It is assumed that all the variables in a subgoal have been numbered before the arguments are copied. In the real implementation, variables are numbered inside the function `copy_term`. The function call `copy_subgoal_args(src,des,arity)` copies the arguments of a numbered subgoal to the table area where `(src-i)` points to the `i`th argument on the stack and `(des+i)` is the destination in the table area where the argument is copied to. In the TOAM architecture [226] on which B-Prolog is based, arguments are passed through the stack and the stack grows downward from high addresses to low ones. That is why `(src-1)` points to the first argument and `(src-arity)` points to the last argument of the subgoal. A similar function is used to copy answers to the table area.

The function `copy_term` is not tail recursive and can easily cause the native C stack to overflow when copying large lists. In the real implementation, an iterative version is used to copy a list and compute its hash code. For a cons, the function needs to compute the hash codes of the car and the cdr before computing its hash code. The function does this in two passes: in the first pass it reverses the list and in the second pass it computes the hash codes while reversing the list back.

The function `copy_term` exploits no sharing of data. Consider, for example, the following program and the query `is_list([1,2])`. After completion of the query, the subgoal table contains three tabled subgoals, `is_list([1,2])`, `is_list([2])`, and `is_list([])`, and each subgoal's answer table contains an answer that is just a copy of the subgoal itself. No data are shared among the copies of the terms. So there are two separate copies of `[1,2]` and two separate copies of `[2]` in the table area. In the WAM representation of lists, a cons re-

[3]Note that this way of combing hash codes is for hash consing terms. For the subgoal and answer tables, hash codes are combined in a different way.

```
int copy_subgoal_args(TermPtr src, TermPtr des, int arity){
   hcsum = 0;
   for (i=1;i<=arity;i++){
        hcode = copy_term(*(src-i), des+i);
        hc_sum = seq_hcode(hc_sum,hcode);
   }
   return hc_sum;
}

int copy_term(Term t, TermPtr des){
   deref(t);
   switch (tag(t)){
   case NUMVAR:
        *des = t;
        return 0;
   case ATM:
        *des = t;
        return atomic_hcode(t);
   case LST:
        p1 = untag(t);
        p2 = allocate_from_table(2);
        car_code = copy_term(*p1, p2);
        cdr_code = copy_term(*(p1+1), p2+1);
        hcode = seq_hcode(car_code,cdr_code);
        t1 = add_tag(p2,LST);
        *des = t1;
        return hcode;
   case STR:
        p1 = untag(t);
        sym = *p1;
        arity = get_arity(sym);
        p2 = allocate_from_table(arity+1);
        hcode = *p2 = sym;
        for (i=1;i<=arity;i++)
            hcode = seq_hcode(hcode, copy_term(*(p1+i), p2+i));
        t1 = add_tag(p2,STR);
        *des = t1;
        return hcode;
   } /* end switch */
} /* end copy_term */
```

Figure 12.1: Copy data to the table area with no sharing.

quires two words to store, so 12 words are used in total. In general, the query is_list([1,2,...,N]) consumes $O(\mathtt{N}^2)$ space in the table area.

12.4 Hash-Consing of Ground Compound Terms

Hash-consing, like tabling, is a memoization technique which uses a hash table to memorize values that have been created. Before creating a new value, it looks up the table to see if the value exists. If so, it reuses the existing value, otherwise, it inserts the value into the table. The concept of hash-consing originates from implementations of Lisp that attempt to reuse cons cells that have been constructed before [80]. This technique has also been suggested for Prolog (e.g., for sharing answers of findall/3 [149]), but its use in Prolog implementations is unknown, not to mention its use in tabling.

Let's call the hash table used for all ground terms *t*erms-table. Figure 12.2 gives an updated version of copy_term that performs hash-consing. If the term is a list or a structure, the function copies it into the table area first. If the term is ground, it then calls the function hash_consing(t1,hcode) to look up the terms-table to see if a copy of t1 already exists in the table.

```
int copy_term(Term t, TermPtr des){
   deref(t);
   switch (tag(t)){
   case NUMVAR:
       *des = t;
       return 0;
   case ATM:
       *des = t;
       return atomic_hcode(t);
   case LST:
       p1 = untag(t);
       p2 = allocate_from_table(2);
       car_code = copy_term(*p1, p2);
       cdr_code = copy_term(*(p1+1), p2+1);
       hcode = seq_hcode(car_code,cdr_code);
       t1 = add_tag(p2,LST);
       if (is_ground_hcode(hcode)){
           t2 = hash_consing(t1,hcode);
           if (t1 != t2){
               deallocate_to_table(2);
               t1 = t2;
           }
       }
       *des = t1;
       return hcode;
   case STR:
       p1 = untag(t);
       sym = *p1;
       arity = get_arity(sym);
       p2 = allocate_from_table(arity+1);
       hcode = *p2 = sym;
       for (i=1;i<=arity;i++)
           hcode = seq_hcode(hcode, copy_term(*(p1+i), p2+i));
       t1 = add_tag(p2,STR);
       if (is_ground_hcode(hcode)){
           t2 = hash_consing(t1,hcode);
           if (t1 != t2){
               deallocate_to_table(arity+1);
               t1 = t2;
           }
       }
       *des = t1;
       return hcode;
   } /* end switch */
} /* end copy_term */
```

Figure 12.2: Copy data with hash-consing.

If so, hash_consing(t1,hcode) returns the copy; otherwise, it inserts `t1` into the terms-table and returns `t1` itself. If an old copy in the terms-table is returned (`t1 != t2`), the function deallocates the memory space allocated for the current copy.

With hash-consing, the query ?-is_list([1,2]) only creates one copy of [1,2] in the table area and the list is shared by the subgoals and the answers. As [2] is the cdr of [1,2], no separate copy is stored for it. So, only 4 words are used in total for the list. The number of words used for hashing the two lists varies, depending on if there is a collision. If no collision occurs, two slots in the terms-table are used; otherwise, one slot in the terms-table is used and one node with two words is used to chain the two lists. So in the worst case, 7 words are needed in total.

12.5 Enhanced Hash-Consing

With hash-consing, the tabled subgoal `is_list`([1,...,N]) consumes only linear table space now. Nevertheless, its time complexity remains quadratic in `N`. This is because for each descendant subgoal `is_list`([K,...,N]) (`K`>1) the hash code of the list `[`K,...,N] has to be computed and the terms-table has to be looked up. We enhance hash-consing with two techniques to lower the time complexity of `is_list`([1,...,N]) to linear.[4]

12.5.1 Hash code memoization

The first technique is to table hash codes of structured terms in the table area. For each structure or a list cons in the table area, we use an extra word to store its hash code. The WAM representation of terms is not changed. The word for the hash code of a compound term is located right before the term. So assume `p` is the untagged reference to a structure or a list cons, then `p-1` references the hash code.

Figure 12.3 gives a new version of `copy_term` that tables hash codes. Tabled hash codes are used for two purposes. Firstly, when searching for the term `t1` in the hash chain, the function `hash_consing`(t1,hcode) always compares the hash codes first and only when the codes are equal will it compare the terms. Secondly, the system reuses the tabled hash codes of terms when it expands a hash table and rehashes the terms into the new hash table.

With tabled hash codes, the subgoal `is_list`([1,...,N]) still takes quadratic time since the list `[`1,...,N] resides on the heap and for each descendant subgoal, the hash code of the argument is not available and hence has to be computed. To avoid this computation, we introduce input sharing.

12.5.2 Input Sharing

Input sharing amounts to letting a subgoal share its ground terms with its answers and descendant subgoals. Consider the tabled subgoal `is_list`([1,2,3]). The answer is the same as the subgoal, so it shares the term `[`1,2,3] with the subgoal in the table area. The direct descendant subgoal is `is_list`([2,3]). Since the list `[`2,3] is a suffix of `[`1,2,3], the descendant subgoal should share it with the original subgoal in the table area.

To implement input sharing, we let the copying procedure set the frame slot of an argument of a tabled subgoal to the address of the copied argument in the table area if the argument is a ground structured term. So for the tabled subgoal `is_list`([1,2,3]), the frame slot of the argument initially references the list `[`1,2,3] on the heap. After the subgoal is copied to the table area, the frame slot is set to reference the copy of the list in the table area. In this way, the list will be shared by answers and the descendant subgoals. For programs that do not use destructive assignments, which is the case for tabled programs, updating frame slots this way causes no problem.

The function `copy_subgoal_args` shown in Figure 12.4 implements input sharing. When an argument is found to be ground, the function lets the

[4]The worst case time complexity is still quadratic in theory if a poorly designed hash function is used.

```
int copy_term(Term t, TermPtr des){
   deref(t);
   switch (tag(t)){
   case NUMVAR:
       *des = t;
       return 0;
   case ATM:
       *des = t;
       return atomic_hcode(t);
   case LST:
       p1 = untag(t);
       if (!is_heap_reference(p1)){
           *des = t;
           return *(p1-1); /* return the tabled hash code */
       }
       p2 = allocate_from_table(3);
       p2++;
       car_code = copy_term(*p1, p2);
       cdr_code = copy_term(*(p1+1), p2+1);
       hcode = seq_hcode(car_code,cdr_code);
       *(p2-1) = hcode;
       t1 = add_tag(p2,LST);
       if (is_ground_hcode(hcode)){
           t2 = hash_consing(t1,hcode);
           if (t1 != t2){
               deallocate_to_table(3);
               t1 = t2;
           }
       }
       *des = t1;
       return hcode;
   case STR:
       p1 = untag(t);
       if (!is_heap_reference(p1)){
           *des = t;
           return *(p1-1); /* return the tabled hash code */
       }
       sym = *p1;
       arity = get_arity(sym);
       p2 = allocate_from_table(arity+2);
       p2++;
       hcode = *p2 = sym;
       for (i=1;i<=arity;i++)
           hcode = seq_hcode(hcode, copy_term(*(p1+i), p2+i));
       *(p2-1) = hcode;
       t1 = add_tag(p2,STR);
       if (is_ground_hcode(hcode)){
           t2 = hash_consing(t1,hcode);
           if (t1 != t2){
               deallocate_to_table(arity+2);
               t1 = t2;
           }
       }
       *des = t1;
       return hcode;
   } /* end switch */
} /* end copy_term */
```

Figure 12.3: Tabling hash codes while copying with hash-consing.

stack slot of the argument reference its copy in the table area. The function copy_term (in Figure 12.3) tests the reference to a compound term to see if the term needs to be copied. If it is not a heap reference, then the referenced term must reside in the table area and thus can be reused.

Note that our input sharing scheme has its limitation in the sense that it fails to facilitate sharing of ground components in non-ground arguments. Consider, for example, the subgoal is_list([X,2,3]). The suffix [2,3] will not

```
int copy_subgoal_args(TermPtr src, TermPtr des, int arity){
   hcsum = 0;
   for (i=1;i<=arity;i++){
        hcode = copy_term(*(src-i), des+i);
        if (is_ground_hcode(hcode)) *(src-i) = *(des+1);
        hc_sum = seq_hcode(hc_sum,hcode);
   }
   return hc_sum;
}
```

Figure 12.4: Input sharing by updating frame slots.

be shared through input sharing in our implementation since the argument is not ground. It will eventually be shared through hash-consing, but its hash code needs to be computed again when it occurs in a descendant subgoal or an answer.

12.6 Evaluation

The improved tabling system described in this paper has been implemented and made available with B-Prolog version 7.7 (BP7.7). We evaluate the proposed approach by comparing BP7.7 with YAP (version 6.3.2) and XSB (version 3.3.6), and also the previous version of B-Prolog, version 7.6 (BP7.6), which did not have enhanced hash-consing. We also compare it with indexed programs produced by the transformation proposed in [87] running on B-Prolog 7.6 (*i*ndexed). We use the `is_list/1` predicate, the `edit_distance/3`[5] program, and a PRISM program to show the effectiveness of the proposed techniques. We also test on a program that favors prefix sharing with tries more than suffix sharing with hash-consing. In addition, we also show results for the CHAT suite and the ATR parser, the traditional benchmarks used to evaluate tabling systems.

The results are obtained on a Linux machine with 16 2.4 GHz, 64 bit Intel Xeon(R) E7340 processor cores and 64 GB of memory. For this evaluation, only a single processor core is utilized. CPU times (in seconds) and table space (in kilobytes) consumptions are measured using the `statistics/1` built-in for BP and XSB, and `table_statistics/1` for YAP.

Table 12.1 shows the results on the query `is_list([1,1,...,1])` where N is the number of 1s in the list. All the systems except for BP7.6 demonstrate a close-to-linear complexity. The higher time complexity of BP7.6 is due to that fact that BP7.6 only uses the first three elements of a list as the key and hashing degenerates into linear search for the query because of hash collision. The difference in time among BP7.7, YAP and XSB is at least a large constant factor. As mentioned above, a trie allows for sharing of prefixes while hash-consing allows for sharing of suffixes as long as lists are concerned. For a list that contains repeated data, there are an equal number of prefixes and suffixes, and hence both types of sharing are equally favored. The difference between BP7.7 and *i*ndexed is only a small constant factor.

Table 12.2 shows the results on the query `is_list(L)` where `L` is a list of

[5]The source code is available in [87].

Table 12.1: Results on `is_list`([1,1,...,1])

	BP7.7		BP7.6		*i*ndexed		YAP		XSB	
N	time	space	time	space	time	space	time	space	time	space
500	0.000	33	0.098	43	0.001	39	0.007	90	0.003	399
1000	0.001	66	0.776	86	0.003	78	0.033	180	0.010	567
1500	0.001	99	2.608	128	0.004	117	0.073	269	0.019	735
2000	0.002	131	6.169	171	0.005	156	0.134	359	0.037	903
2500	0.001	164	12.034	214	0.006	195	0.186	449	0.058	1071
3000	0.002	197	20.777	257	0.008	234	0.282	539	0.078	1239
3500	0.002	229	32.975	300	0.009	273	0.384	629	0.108	1407
4000	0.003	264	49.204	343	0.011	312	0.498	719	0.139	1575
4500	0.003	297	70.048	386	0.011	351	0.571	809	0.177	1743
5000	0.003	330	96.112	429	0.013	390	0.729	898	0.217	1911

Table 12.2: Results on `is_list(L)` where `L` contains random data.

	BP7.7		BP7.6		*i*ndexed		YAP		XSB	
N	time	space	time	space	time	space	time	space	time	space
500	0.000	33	0.000	43	0.002	39	0.008	90	0.024	9990
1000	0.001	66	0.001	86	0.002	78	0.032	180	0.063	39236
1500	0.001	99	0.001	128	0.004	117	0.082	270	0.142	87991
2000	0.001	132	0.002	171	0.005	156	0.134	360	0.252	156269
2500	0.001	164	0.003	214	0.007	195	0.218	450	0.387	244071
3000	0.002	197	0.003	257	0.008	234	0.341	540	0.559	351401
3500	0.002	229	0.004	300	0.010	273	0.401	630	0.766	478260
4000	0.003	264	0.005	343	0.011	312	0.537	719	0.978	624640
4500	0.003	297	0.006	386	0.012	351	0.703	809	1.244	790555
5000	0.004	330	0.008	429	0.013	390	0.894	899	1.504	975990

random constants.[6] BP consumes linear space and linear time; YAP consumes linear space thanks to the global trie for terms but takes quadratic time; XSB is quadratic in both time and space. For random lists, suffix sharing with hash consing is clearly more effective than prefix sharing with tries.

Tables 12.3 and 12.4 show the results on the `edit`_distance program with repeated data and random data, respectively. The main predicate `edit`(L1,L2,D) in the program computes the distance between `L1` and `L2`, i.e., the number of substitutions, insertions and deletions needed to transform `L1` to `L2`. The tabled version finds all solutions. BP7.7 is significantly faster than BP7.6 on the type of queries that use repeated data. BP7.7 also outperforms YAP and XSB in both time and space on both types of queries. Similar to the `is`_list benchmark, enhanced hash-consing is asymptotically more effective than tries on random data.

Table 12.5 compares BP7.7 and BP7.6 on the PRISM program that simulates a two-state hidden Markov model [207]. For our benchmarking purpose, the training data of the form `hmm`([a,b,a,b,...]) are used, and only the time and space required to find all the explanations are measured. While BP7.7 consumes slightly more space than BP7.6 due to the overhead of hash-consing, it outperforms BP7.6 in time by a linear factor.

Although it is more common for subgoals of recursive programs to share suffixes than prefixes, it is possible to find programs on which prefix sharing

[6]A random number generator is used to generate the lists. For each size, the same list was used for all the systems.

Table 12.3: Results on `edit([1,1,...,1],[1,1,...,1],D)`.

	BP7.7		BP7.6		*i*ndexed		YAP		XSB	
N	time	space	time	space	time	space	time	space	time	space
30	0.000	60	0.026	97	0.003	90	0.005	213	0.006	1273
60	0.003	233	0.726	378	0.016	348	0.034	819	0.057	4341
90	0.007	519	5.189	841	0.036	776	0.107	1820	0.235	9435
120	0.015	917	21.216	1487	0.064	1372	0.266	3214	0.736	16554
150	0.022	1427	63.536	2316	0.102	2137	0.517	5002	1.635	25698
180	0.031	2051	156.072	3328	0.142	3071	0.942	7183	3.041	36868
210	0.047	2786	334.190	4523	0.208	4173	1.533	9759	5.035	50064
240	0.060	3634	646.550	5900	0.267	5445	2.367	12728	7.662	65285
270	0.074	4595	1159.182	7460	0.339	6885	3.081	16090	11.327	82531
300	0.095	5668	1955.331	9204	0.448	8493	4.401	19847	15.664	101803

Table 12.4: Results on `edit(L1,L2,D)` where `L1` and `L2` contain random data.

	BP7.7		BP7.6		*i*ndexed		YAP		XSB	
N	time	space	time	space	time	space	time	space	time	space
30	0.001	61	0.000	97	0.004	90	0.005	214	0.011	4148
60	0.003	234	0.006	378	0.020	348	0.045	822	0.099	27706
90	0.010	521	0.016	841	0.038	776	0.118	1823	0.313	89645
120	0.017	919	0.033	1487	0.067	1372	0.298	3218	0.759	209183
150	0.027	1430	0.057	2316	0.105	2137	0.591	5007	1.501	404752
180	0.038	2054	0.094	3328	0.148	3071	1.058	7190	2.771	695363
210	0.056	2790	0.156	4523	0.217	4173	1.695	9766	4.271	1099906
240	0.073	3639	0.219	5900	0.282	5445	2.687	12736	6.247	1637354
270	0.092	4600	0.297	7460	0.352	6885	3.782	16100	8.787	2327276
300	0.114	5674	0.435	9204	0.466	8493	5.248	19857	11.954	3187340

Table 12.5: Results on the PRISM program HMM.

	BP7.7		BP7.6	
N	time	space	time	space
2000	0.002	222	1.164	179
3000	0.005	333	3.911	269
4000	0.006	444	9.249	359
5000	0.008	555	18.044	449
6000	0.010	666	31.150	539
7000	0.011	776	49.441	628
8000	0.013	889	73.774	718
9000	0.015	1000	105.049	808
10000	0.018	1111	144.140	898

Table 12.6: Results on create_list(N,L).

	BP7.7		BP7.6		YAP		XSB	
N	time	space	time	space	time	space	time	space
500	0.035	2417	0.107	990	0.039	3965	0.023	290
1000	0.201	9564	0.827	3937	0.201	15742	0.043	348
1500	0.654	21635	2.989	8831	0.523	35332	0.095	407
2000	0.969	37926	7.245	15679	0.962	62734	0.169	465
2500	2.151	60082	14.130	24480	1.699	97949	0.264	524
3000	2.660	85890	24.343	35249	2.630	140976	0.378	583
3500	3.276	116011	38.397	47956	3.739	191816	0.517	641
4000	4.011	150192	57.217	62616	5.071	250468	0.675	700
4500	7.319	194310	80.994	79229	6.978	316933	0.853	758
5000	8.316	238885	110.631	97796	9.267	391211	1.051	817

Table 12.7: Results on the CHAT benchmarks and the ATR parser.

	BP 7.7		BP 7.6		YAP		XSB	
Benchmark	time	space	time	space	time	space	time	space
cs_o	0.015	198	0.0129	11	0.009	26	0.011	285
cs_r	0.025	332	0.026	11	0.019	27	0.022	286
disj	0.008	108	0.009	11	0.005	23	0.007	277
gabriel	0.011	111	0.012	9	0.006	20	0.008	272
kalah	0.008	90	0.008	15	0.006	35	0.008	304
pg	0.006	69	0.006	7	0.004	15	0.006	263
read	0.057	987	0.058	23	0.099	46	0.030	327
atr	0.509	15111	0.543	5947	0.325	52520	0.280	45400

with tries is more effective than suffix sharing with hash-consing. The following gives such a program:

```
:-table create_list/2.
create_list(N,L):-
    between(1,N,I),
    range(1,I,L).
```

The query create_list(N,L) creates `N` lists [1], [1,2], ..., and [1,2,...,N] that have only common prefixes. As shown in Table 12.6, XSB consumes linear space, while BP and YAP consume quadratic space. YAP tables all suffixes into the global trie for terms and there are $O(\mathtt{N}^2)$ suffixes. BP7.7 consumes more table space than BP7.6 since all the terms are hash-consed but none is shared. BP7.6 is slower than BP7.7 since the hash function used in BP7.6, which is based on the first three elements of a list, results in more collisions than BP7.7.

Table 12.7 compares the systems on the CHAT benchmark suite and the ATR parser. There is almost no difference between BP7.7 and BP7.6 in time and the space overhead incurred by hash-consing is noticeable. Hash-consing has no positive effect on these programs because the sequences used in the programs are very short.

12.7 Related Work

Since structure sharing [21] was discarded and the Warren Abstract Machine (WAM) [219] triumphed as the implementation model of Prolog, there has been little attention paid to exploiting data sharing in Prolog implementations.[7] In his Diploma thesis [144], Ulrich Neumerkel gave several example Prolog programs that would consume an-order-of-magnitude less space with data sharing than without sharing. He proposed applying hash-consing and DFA-minimization to sharing terms including cyclic ones. The proposed approach would incur considerable overhead if every compound term is hash-consed when created, and hence it is infeasible to incorporate the approach into the WAM. Following Appel and Goncalves's hash-consing garbage collector for SML/NJ [11], Nguyen and Demoen recently built a similar garbage collector for hProlog [145]. The garbage collector hash-conses compound terms on the heap in one phase and performs absorption in another phase such that for the replications of a compound term only one copy is kept and all the others are garbage collected. Their experiment basically confirms the disappointing result reported in Appel and Goncalves's paper: the overhead outweighs the gain except for special programs.

Hash-consing can be applied to the built-in predicate `findall`/3, as suggested by O'Keefe [149], to avoid repeatedly copying the same term in different answers. Currently, B-Prolog is the only Prolog system that supports hash-consing for `findall`/3. It employs a hash table for ground terms in the findall area. The algorithm and memory manager developed for the table area is reused for the findall area. With hash-consing, the system copies a ground term only once when copying answers from the findall area to the heap. Input sharing is exploited in the same way as for tabled subgoals. For a `findall` call, the compiler converts it into a call to a temporary predicate such that each argument of the generator occupies one slot in the stack frame. At runtime, the system first copies the arguments of the generator from the stack/heap to the findall area before the generator is executed. When an argument of the generator is found to be a ground compound term, its frame slot is set to reference the copy in the findall area. In this way, the argument and its subterms can be reused by the answers and the descendant calls. Nguyen and Demoen's implementation of input sharing for `findall`/3 [145] distinguishes between old terms that are created before the generator and new terms that are generated by the generator, and have answers share the old terms. Their scheme can exploit sharing of not only ground arguments but also ground terms in non-ground arguments. Their scheme may not be suited for tabled data since, unlike data in the findall area which live and die with the generator, tabled data are permanent. Also, their implementation does not exploit output sharing.

A trie has been a popular data structure for organizing tabled subgoals and answers [161]. It is adopted by all the tabled Prolog systems except B-Prolog. As far as lists are concerned, a trie facilitates sharing of the prefixes while hash-consing allows for sharing of the suffixes. So for the two lists `[1,2]` and `[1,2,3]`, the former shares the same path as the latter in the trie, but they are treated as separate lists when hash-consed; for the two lists `[2,3]` and `[1,2,3]`,

[7]A lot of work has been done on indexing Prolog terms, but indexing is a different kind of sharing since it does not consider reuse of terms from different sources.

however, a trie allows for no sharing while hash-consing allows for complete sharing.

Another advantage of tries is that they can be used to perform both variant testing and subsumption testing, and thus can be used in both variant-based and subsumption-based tabling systems. Hash-consing, on the other hand, can be used to perform equivalence testing only and thus cannot directly be used for subsumption-based tabling.

Terms stored in a trie have a different representation from terms on the heap. For example, in the YAP system, tries are represented as trie instructions [51]. For this reason, when an answer is returned, it must be copied from its trie in the table area to the heap even if it is ground. In our system, structured ground terms in the table area have exactly the same representation as on the heap, so when they occur in an answer they do not need to be copied when the answer is returned.

In the original implementation of XSB and YAP, one trie is used for all tabled subgoals, and for each subgoal one trie is used for the answer table. To enhance sharing, Raimundo and Rocha propose using a global trie for all tabled structured terms [160]. Due to the necessity of copying answers from the table area to the heap, the time complexity remains the same even when the space complexity drops.

To some extent, the idea of representing sentences as position indexed facts [87, 206] is similar to hash-consing in the sense that a hash-consed term always is associated with a hash code. The translation from a program that deals with sequences represented as lists into one that uses position representation is not trivial. When difference lists are involved, the translation is even more complicated. The program obtained after translation may lose sharing opportunities. Therefore, hash-consing is a more practical solution to sharing than program transformation.

As far as we know, our implementation is the first attempt to apply hash-consing to tabling. Our implementation enhances hash-consing with input sharing and hash code memoization to speed-up computation of hash codes. The extra cell used to store the hash code of a compound term is overhead if the term is never shared. Nevertheless, while the increase of space is always a constant factor, the gain in speed can be linear in the size of the data.

12.8 Conclusion

We have presented an implementation of hash-consing for tabling structured data. Hash-consing facilitates sharing of structured data and can eliminate the extra linear factor of space complexity commonly seen in early tabling systems when dealing with sequences. Hash-consing alone does not change the time complexity. We have enhanced it with input sharing and hash code memoization to eliminate the extra linear factor of time complexity in dealing with sequences. The resulting tabling system significantly improves the scalability of language parsing and bio-sequence analysis applications.

Our work will shed some light on the discussion on what data structure to use for tabled data. A trie is suitable for sharing prefixes and hash-consing is suitable for sharing suffixes of sequences. Although it is possible to find programs that make prefix sharing arbitrarily better than suffix sharing, it is more common for subgoals of recursive programs to share suffixes than prefixes. Therefore, hash-consing is in general a better choice than tries as a data structure for representing tabled data. Hash-consing as it is in our implementation is not suitable for subsumption-based tabling. It is future work to adapt hash-consing to subsumption testing.

Acknowledgements

The PRISM system has been the motivation for this project and we thank Taisuke Sato and Yoshitaka Kameya for their discussion. We also thank the anonymous referees for their detailed comments on the presentation. Neng-Fa Zhou was supported in part by NSF (No.1018006) and Christian Theil Have was supported by the project "Logic-statistic modelling and analysis of biological sequence data" funded by the NABIIT program under the Danish Strategic Research Council.

Chapter 13

A Probabilistic Genome-Wide Gene Reading Frame Sequence Model

With Søren Mørk

Submitted to PLOS ONE (in review).

Abstract

We introduce a new type of probabilistic sequence model, that model the sequential composition of reading frames of genes in a genome. Our approach extends gene finders with a model of the sequential composition of genes at the genome-level – effectively producing a sequential genome annotation as output. The model can be used to obtain the most probable genome annotation based on a combination of i: a gene finder score of each gene candidate and ii: the sequence of the reading frames of gene candidates through a genome. The model — as well as a higher order variant — is developed and tested using the probabilistic logic programming language and machine learning system PRISM - a fast and efficient model prototyping environment, using bacterial gene finding performance as a benchmark of signal strength. The model is used to prune a set of gene predictions from an underlying gene finder and are evaluated by the effect on prediction performance. Since bacterial gene finding to a large extent is a solved problem it forms an ideal proving ground for evaluating the explicit modeling of larger scale gene sequence composition of genomes.

We conclude that the sequential composition of gene reading frames is a consistent signal present in bacterial genomes, that can be effectively modeled with probabilistic sequence models.

13.1 Introduction

Automated genome annotation is essential for exploiting the enormous amounts of genome sequence data currently being generated [22]. The initial steps of genome annotation relies heavily on probabilistic nucleotide sequence models, for generating sets of predicted genes. Such models typically estimate the probability that each open reading frame (ORF) is a gene. This estimate is usually based on only a limited context comprising the ORF nucleotide sequence and perhaps a few hundred bases upstream and downstream to include signals such as promoters and ribosomal binding sites. The subsequent steps to assemble a genome annotation typically involves selecting the highly scoring predictions using a significance criteria or threshold. In some recent gene finders [56, 99, 89] the selection of predictions is done as a genome-wide optimization where the predictions are chosen to form a coherent genome annotation by taking into account the extent of overlap between genes.

In a similar vein, we introduce a probabilistic sequence model which select the set of predictions that form the genome annotation, but which is based on sequential composition of gene reading frames, which we believe is a novel signal to be explored in gene finding. Our purpose is not to build the next state-of-the-art gene finder, but to present a class of simple models which clearly demonstrates the efficacy of exploiting the gene-reading-frame-sequence bias.

13.1.1 The gene reading frame sequence bias

The existence of a gene-strand bias in prokaryotes is well established [23]. One source for this bias is a tendency for genes to be placed on the leading strand due to replication efficiency consequences of co-directional and head-on collisions of the replication and transcription apparatus [155]. It has also been argued that the preferential placement of genes in the leading strand is driven by essentiality rather than expression [163].

A gene-reading-frame-sequence bias is a general signal that can incorporate gene-strand bias, bias due to clusters of orthologous genes [165], operonic structures [221], phase preference for overlapping genes [46] and other potential effects yielding non-random sequence composition.

The gene-strand bias account for a large proportion of the gene-reading-frame-sequence bias, but a pronounced bias is detectable even within the strands. Furthermore, the gene-reading-frame-sequence bias seems to be symmetric for the two strands, cf. Table 1. This is a convenient property, especially considering the arbitrary designation of which is the forward and which is the reverse strand.

13.2 Methods

Our gene-reading-frame-sequence model are implemented in PRISM, a probabilistic logic programming language and machine learning system with generic algorithms for parameter estimation and decoding [175]. We use PRISM as a convenient model comparison platform, since it is powerful enough to express the different models and enables a level execution provided by its generic machine learning routines. The use of probabilistic logic programming for evaluating sequence models as the heart of contemporary gene finders has recently been demonstrated in [137].

13.2.1 The Frameseq model

The basic *Frameseq* model is a variant of a fully connected Hidden Markov Model (HMM) [159] with a state for each of the six possible reading frames —

the *frame states* — and a delete state. Given a sequence of gene predictions sorted by position, a path through the model capable of emitting this prediction sequence represents a classification of predictions into presumed true positives emitted from the frame states and presumed false positives emitted from the delete state. A path with optimal probability represents a best hypothesis about the classification of predictions into positives and negatives. This path can be calculated using the Viterbi algorithm which is provided by PRISM.

Each state emits a score symbol and a frame for each gene prediction. Frame states only emit predictions with a corresponding frame, whereas the delete state may emit predictions of any frame. The score symbol is a symbolic value representing a range of confidence scores for the predictions of the input gene finder. The emission probabilities thus reflect the prediction confidence scores in the training set.

Traditionally, the transition probabilities of an HMM are conditioned only on the previous state (the Markov property). In our model the transition probability is conditioned on the previous *frame state* rather than just the previous state. The frame state transition probabilities are thus assumed to reflect the probability of a seeing a gene in a particular reading frame given the reading frame of the previous gene.

Higher ordered Markov models have generally shown to be an improvement over standard models for the nucleotide sequence models used in bacterial gene finding (*e.g.* as used in Genemark and Glimmer). To explore the possibility that the same might be true for the gene reading frame sequence, we have also employed a second order version of Frameseq, *i.e.*, which conditions transitions on the two previous frame states.

The transition probabilities between the *frame states* are estimated as the relative frequency of observed adjacent genes in the various frames observed in the set of verified genes.

The probability of a transition to the delete state (from any state) reflects the probability that a gene finder prediction is a false positive,

$$P(delete) = 1 - \frac{TP}{TP + FP}$$

where TP is the number of true positives predicted by the gene finder and FP is the number of false positives. This probability is directly related to gene finder specificity and may be tweaked for different sensitivity/specificity trade-offs. We exploit this in experiments reported below.

The frame state transition probabilities are estimated as relative frequencies, which have the interpretation of conditional probabilities given that a transition to the delete state did not occur. We normalize each of these transition probabilities by multiplying them by $1 - P(delete)$.

Each state is capable of emitting a finite set of i symbols $\delta_1 \ldots \delta_n$ corresponding to ranges of prediction scores, *i.e.*n the states emit a discretized symbol corresponding to the confidence score of a prediction as supplied by the gene finder. The ranges are selected to ensure that each score symbol correspond to an equal proportion of gene finder predictions. The number of ranges, n, is a configurable parameter; when n is high the model can better exploit the scores from the gene finder, but the estimated emission probabilities become more fragile, *i.e.*, more data is needed to reliably estimate them. The emission probabilities of the delete state are estimated as the relative frequency of each of the possible score symbols for all false positives predictions, *i.e.*,

$$P(\delta_i|state = delete) = \frac{FP_{\delta_i}}{FP}$$

where FP_{δ_i} is the number of false positives with a confidence score within the range symbolized by δ_i.

Similarly, the emission probabilities of frame states are estimated as the fraction of true positive predictions belonging to a particular range within the corresponding frame, *i.e.*,

$$P(\delta_i|state = frame_j) = \frac{TP_{\delta_i}^{frame_j}}{TP^{frame_j}}$$

where TP^{frame_j} is the total number true positive predictions in reading frame j and $TP_{\delta_i}^{frame_j}$ is the number of true positive predictions in reading frame j with a confidence score within the range symbolized by δ_i.

A illustration of the states and transitions and of the model is shown in figure 13.2.1.

Instead of using exact empirical frequency counts as described above, we use a variational Bayes version of the EM algorithm [186] provided with PRISM. This algorithm puts Dirichlet priors (pseudo-counts) on random variables ensuring that all estimated probabilities are non-zero.

13.3 Results and discussion

13.3.1 The phylogenetic reach of the gene reading frame bias

To test the generalization capability and potential phylogenetic reach of our model, we train models on five different prokaryotic genomes and use them to filter predictions for the *E. coli* genome. We expect *E. coli* to have the most reliable genome annotation and by using it for validation we obtain the most reliable validation results. By training on distant organisms, we show the robustness of our approach with regard to both training set quality and phylogenetic distance. Good performance on *E. coli* should also imply that we can train our model on a well annotated genome and filter gene finder predictions in other genomes with increased reliability. To validate this we also do this experiment in reverse, *i.e.*, we also train our model on *E. coli* to predict on each of the other genomes. For all models trained, we set the number of score ranges to $n = 15$.

The five genomes, listed here in ascending order of phylogenetic distance from *E. coli*:

- *Escherichia coli* [REFSEQ:NC_000913],
- *Salmonella enterica* [REFSEQ:NC_004631.1],
- *Legionella pneumophila* [REFSEQ:NC_002942],
- *Bacillus subtilis* [REFSEQ:NC_000964]
- *Thermoplasma acidophilum* [REFSEQ:NC_002578].

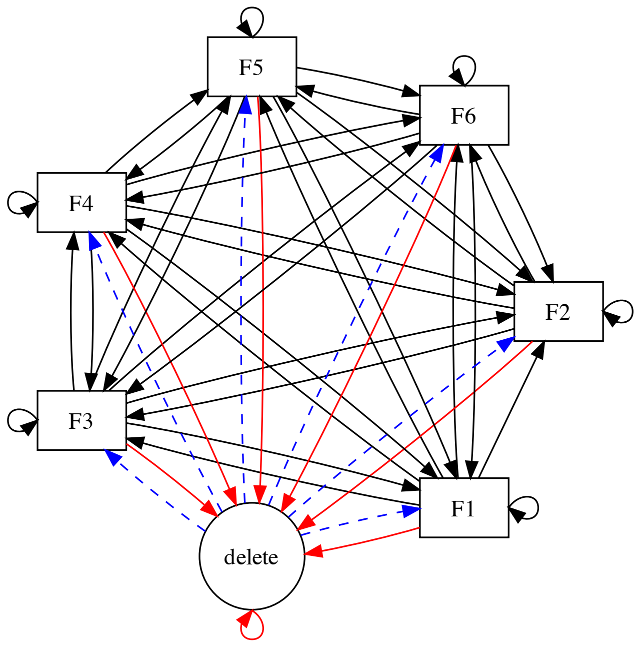


Figure 13.1: The Frameseq delete-HMM model. All frame states F1 . . . F6 have transitions to each other and to themselves. Transitions to the delete state are symbolized by red arrows, to indicate that they share the transition probability, $P(delete)$. The dashed blue arrows illustrate transitions from the delete state to a frame state – the probability of which depend on the last frame state visited before the delete state. Furthermore, the delete state is drawn as circle rather than a box to convey that it resembles a silent state – it does produce emissions (predicted false positives) but we are only interested in emissions from the frame states (predicted true positives). To minimize visual clutter, a begin and end state have been omitted.

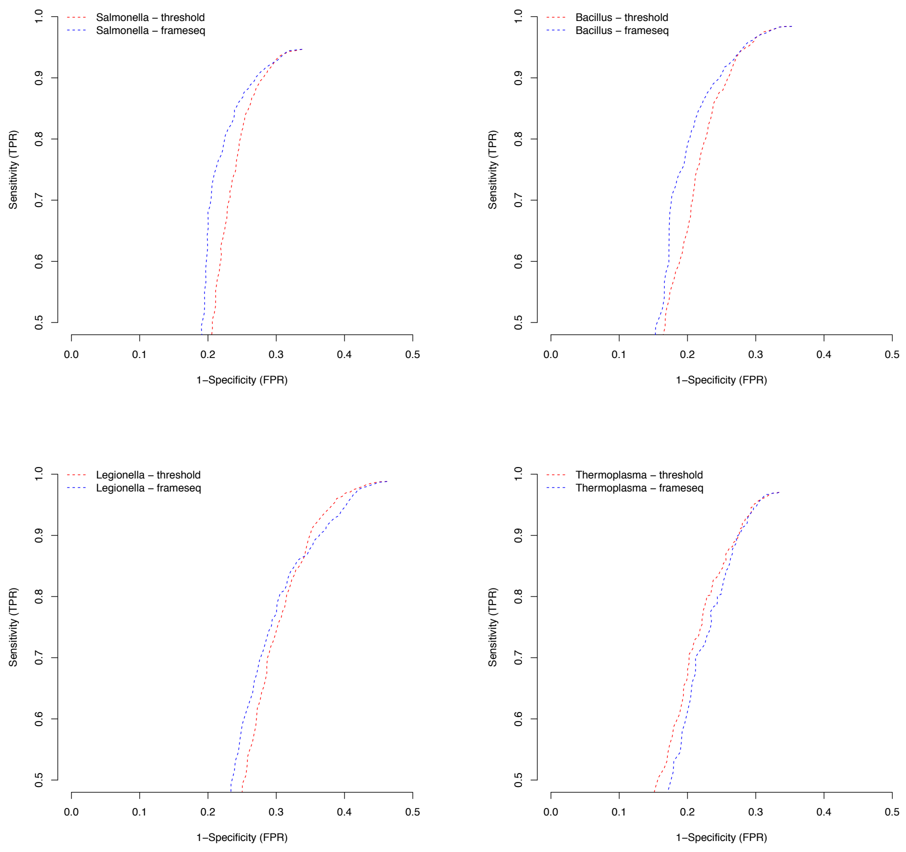


Figure 13.2: ROC curve illustrating phylogenetic reach (inverse). The figure shows ROC curves for filtering of all Genemark 2.5 predictions with score > 0.1 for different organisms. The black curve shows selection using a threshold and the colored curves show filtering using Frameseq. The Frameseq model is trained the *E. coli* for all organisms. The experiment shows that it is possible to train Frameseq on a well-known and well-annotated organism and apply it to filter predictions on phylogenetically distant organisms with improved accuracy. Accuracy is improved for both *S. enterica* and *B. subtilis* and in part for *L. pneumophila*, but not for the phylogenetically distant *T. acidophilum*.

We use Genemark 2.5 [17] which is available as a web-service to produce a large initial set of candidate genes.

Genemark is currently available in newer versions (GeneMarkS and GenemarkHMM) with improved prediction performance.

However, as our main interest is to introduce a new type of genome-sequence model, not to improve gene finding, the older version of Genemark provides a number of advantages for our purposes that are not present in other available single-sequence gene finders: Genemark 2.5 use a very simple scoring model and do not employ any post-scoring prediction selection algorithm, but is capable of producing a large set of predictions simply by enforcing a (low) score cut-off. As it does not otherwise prune predictions, we eliminate factors which could affect and reduce the pruning potential available to our model. Obviously, the accuracy of this gene finder is slightly below what is now state-of-the-art.

The full dataset offered by being able to produce a large set of predictions provides a better evaluation of the contribution of the reading frame signal than pruning a small optimal prediction set or (for completeness we do include such more limited experiments for state-of-the-art gene finders below).

We set the configurable score cut-off as low as possible, *i.e.*, to 0.1, to allow as many false positive predictions as possible. The gene finder predictions are preprocessed to contain only the best scoring prediction for each distinct stop codon. For each genome, we train using the preprocessed Genemark predictions and use the RefSeq annotation as golden standard. By inspection of transition probabilities, we observe that the gene-reading-frame-sequence bias tends to be almost symmetric for the strands, see table 13.3.1.

We test the performance of each model on the Genemark predictions for the target genome by measuring sensitivity and specificity in terms of predicted stop codons with respect to the RefSeq annotation.

We repeat this process with incrementally increasing delete state probabilities resulting in a range of sensitivity/specificity trade-offs. These are plotted in Figure 13.3.1 (predictions on *E. coli*) and Figure **??** (predictions on the other genomes) as a Receiver Operator Characteristic (ROC) curves.

For comparison we provide a baseline ROC curve, produced via incrementally increasing a cut off value of the scores for the Genemark predictions for the target genome.

For all organisms except the phylogenetically very distant *T. acidophilum*, Frameseq improves accuracy and the margin of the improvement correlates with phylogenetic distance. The pronounced improvement in the accuracy which can be observed in ROC curves for the frame-bias model as compared to the baseline demonstrates that for comparable levels sensitivity, Frameseq achieves a lower false positive rate.

13.3.1.1 A higher order signal?

It is plausible that the gene-reading-frame-sequence bias is more complex than just pairwise dependencies between the frames of genes. More complex dependencies on previous gene reading frames can be modeled using a higher order model.

We test this hypothesis by using a second order HMM based Frameseq model which is trained and applied on *E. coli.* We compare this to the basic Frameseq model which uses a first order HMM. We also investigate the phy-

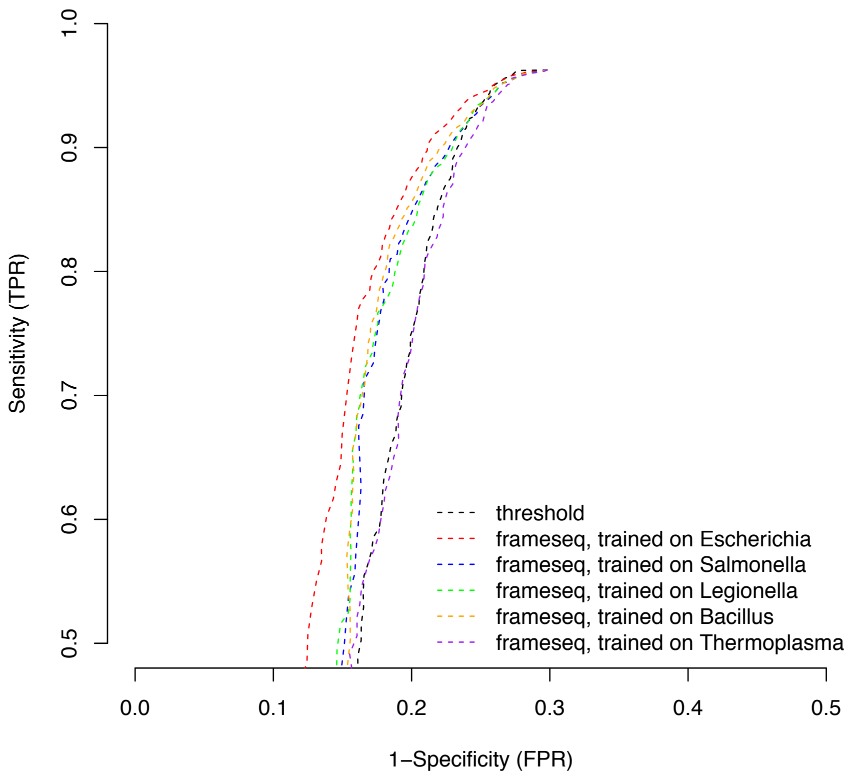


Figure 13.3: ROC curves illustrating phylogenetic reach. The figure shows ROC curves for filtering of all Genemark 2.5 predictions with score > 0.1 for *E. coli*. The black curve shows selection using a threshold and the colored curves show filtering using Frameseq. Note that the ROC curve does not extend all the way to the right; this is due to the 0.1 Genemark cutoff which still eliminates a lot of candidate predictions.

logenic of conservation of a possible higher order signal by training the same model on *S. enterica* and decoding on *E. coli*. In both cases, the we use predictions from the Genemark 2.5 gene finder, with a score cut-off of 0.1. As in the previous experiments we set the number of score ranges to $n = 15$.

We derive and compare ROC curves for threshold selection and Frameseq selection like in the previous experiments, but here for both the first order and second order models. We provide a separate plot with the *E. coli* trained models (Figure 13.3.1.1) and the *S. enterica* trained models (Figure 13.3.1.1).

In the case where we train on *E. coli*, the second order Frameseq model results in significantly better accuracy than with the first order model. The improvement degrades quite a bit when we instead train the model on *S. enterica*, but there is still a detectable improvement in accuracy for the second

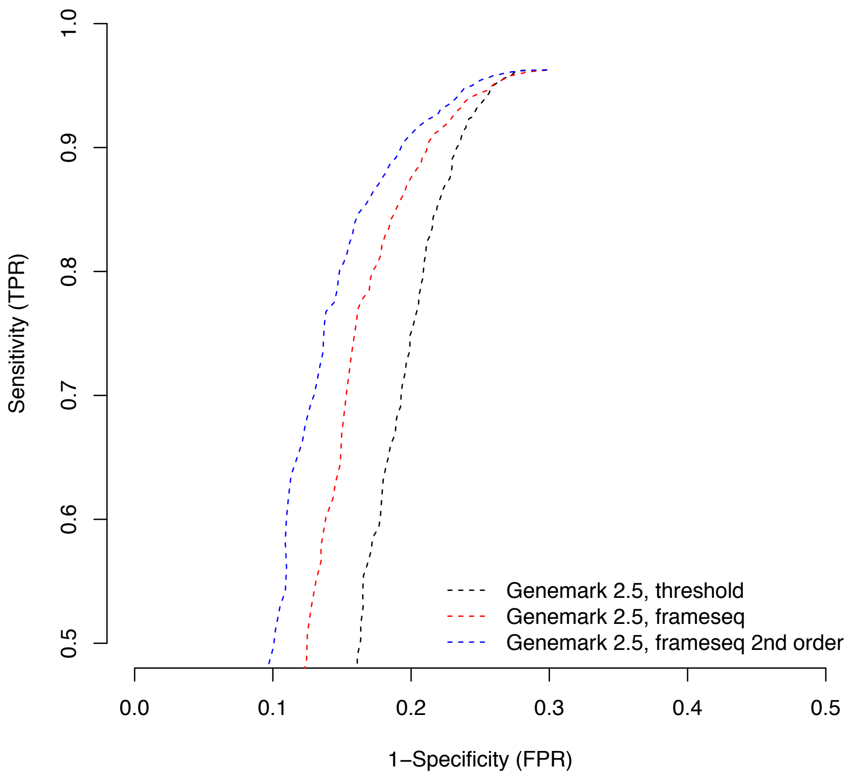


Figure 13.4: ROC curves illustrating potential accuracy gain with second order model. This ROC curve compares the basic first order Frameseq model to a second order model. The black curve indicate threshold selection, the red curve is the first order model and the blue curve is the second order model. Both the first order model and the second order model are trained on *E. coli* and applied to filter the predictions of the same genome. The second order model results in markedly better accuracy than with the first order model.

order model.

It should be noted that, higher order models effectively increase the amount of transition probabilities involved, but the amount of training data used to estimate these are fixed in our case. This means that increasing the order of the model results in less reliable transition probabilities. This may explain some of the loss of accuracy when observed when training on *S. enterica* as compared to training on *E. coli*.

On the other hand, the experiment with the second order Frameseq model trained on *E. coli* demonstrates the maximal potential of utilizing a higher order signal.

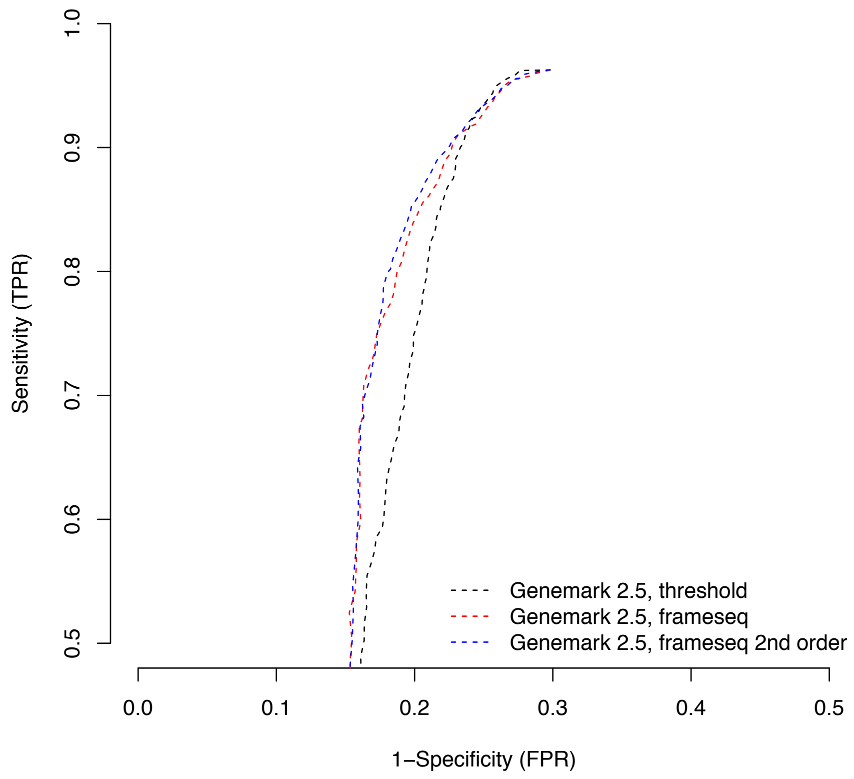


Figure 13.5: Phylogenetic robustness of a second order signal. This ROC curve compares the basic first order Frameseq model to a second order model. The black curve indicate threshold selection, the red curve is the first order model and the blue curve is the second order model. Both the first order model and the second order model are trained on *S. enterica* and applied to filter the predictions of *E. coli*. The second order model results in marginally better accuracy.

13.3.1.2 Effect on state-of-the-art gene finders

In this section we explore using Frameseq with Glimmer 3 and Prodigal 2.50 — to evaluate the contribution of a reading frame sequence signal for state-of-the art gene finders.

Genemark 2.5 which was used in the previous experiments, scores each open reading frame individually and does not attempt to stitch such individual predictions together into a more coherent set of predictions for the genome.

The algorithms employed by the two other gene finders have some similarities to the delete-HMM of Frameseq. Both gene finders use custom dynamic programming algorithms to achieve a more coherent set of predictions for a genome. Prodigal use several features including hexamer scores, ribosomal binding site detection, maximal overlap and distance between predictions combined using a custom dynamic programming algorithm. Similarly, Glimmer 3 also use a dynamic programming algorithm which restricts the size of overlaps between predictions. Neither of these algorithms utilize the gene frame bias.

Our algorithm is quite simplistic in comparison since it only considers one signal – the gene-reading-frame-sequence bias. It could undoubtedly be improved by considering other signals and constraints inherent between predictions such as distance and overlaps. In being simplistic, however, it clearly demonstrates the utility of the gene-reading-frame-sequence bias without the inherent noise from the impact of other features – which is our purpose.

In these experiments, we use Frameseq to filter the predictions of the state-of-the-art gene finders in order to explore the potential beneficial effect they could achieve by incorporating the gene-reading-frame-sequence bias. For each gene finder — Prodigal 2.50 and Glimmer 3 — we apply the second order Frameseq model trained on *E. coli* to filter their respective predictions on also on *E. coli.* The number of score ranges is in this experiment set to $n = 100$ to better capture the more detailed variations of the scores. The results are shown in figure 13.3.1.2 (Glimmer) and figure 13.3.1.2 (Prodigal).

In all cases the filtered predictions have significantly improved specificity for comparable levels of sensitivity. The effect of Frameseq seems most pronounced with reduced sensitivities which could indicate that the scores of the gene finders are more reliable for the top-scoring predictions.

These experiments do not conclusively prove that all the gene finders could achieve improved specificity for the desired levels of sensitivity (close to one) by incorporating the gene-reading-frame-sequence bias. It should be noted that we slightly over-fit the model by training on *E. coli* and by doing this we get more impressive results than would have been the case if the models where trained using other organisms. Training Frameseq on, *e.g.*, *S. enterica* and filtering predictions for these gene finders does not result in significantly improved accuracy (data not shown). We believe this to be mainly a problem of sparsity of the training data, but also due to the reduced margins for possible improvement as compared to Genemark 2.5. We demonstrated the phylogenetic reach using Genemark 2.5, but the margin for possible improvement is significantly smaller with Glimmer and Prodigal. Due to this, a slightly under-fitted model will generalize sufficiently to improve Genemark 2.5 results, but insufficiently with the state-of-the art gene finders.

The experiment here, however, does show that the gene-reading-frame-sequence bias signal provides useful information which is complementary to the signals used by the contemporary methods.

13.4 Conclusions

We have demonstrated the feasibility of modeling the sequential composition of genes in a genome with simple sequential reading frame models. We obtain surprisingly good results when predicting on one organism with models trained on phylogenetically distant genomes, which implies both the generality of the approach and the potential importance of gene reading frame sequence structure across taxa.

The impact of our method is most pronounced for reduced levels of sensitiv-

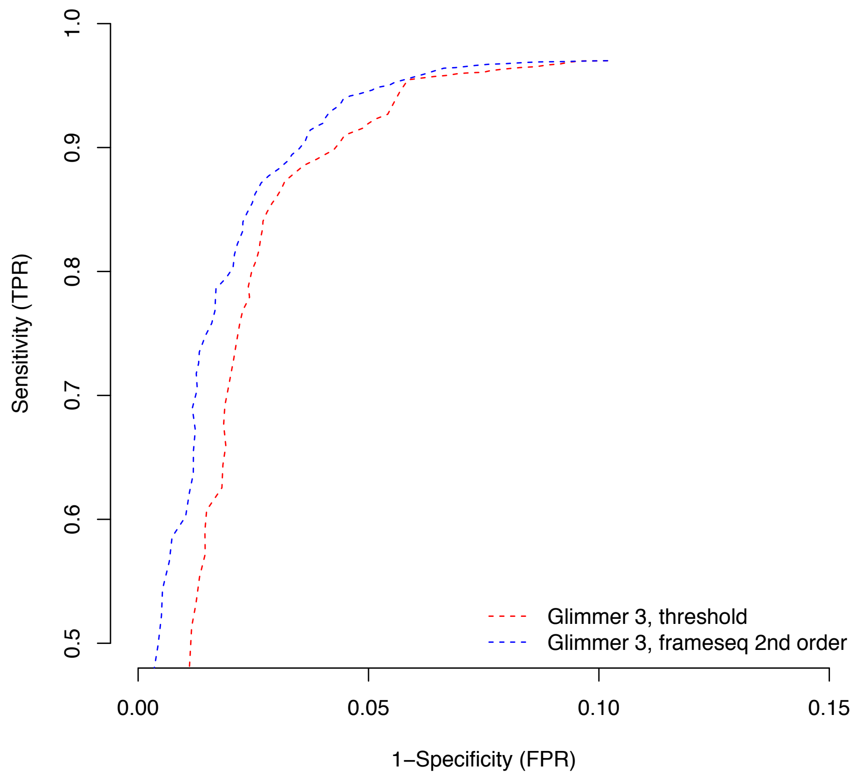


Figure 13.6: Frameseq with Glimmer. The red ROC curve shows threshold selection with Glimmer 3 predictions on *E. coli* and blue curve shows results of filtering these predictions with a second order Frameseq model trained on *E. coli*.

ity. Ideally we would like to achieve as significant improvements in specificity for a higher level of sensitivity, but improved specificity with a lower sensitivity is still a good result with important implications; It means that our approach is capable of supplying a larger set of gene predictions with a specified upper bound on the false positive rate, than is possible with any other gene finder. This may be useful when selecting candidate genes for experimental verification and can reduce the likelihood of wasted lab effort.

We also believe that the gene-reading-frame-sequence bias signal can be useful for improving automated computational genome annotation, but in order to achieve this, it will need to be integrated with the algorithms of state-of-the-art gene finders instead of the relatively superficial augmentation we do here.

In order to clearly illustrate the gene-reading-frame-sequence bias, we engineered our method to be as simple as possible, which in effect have several

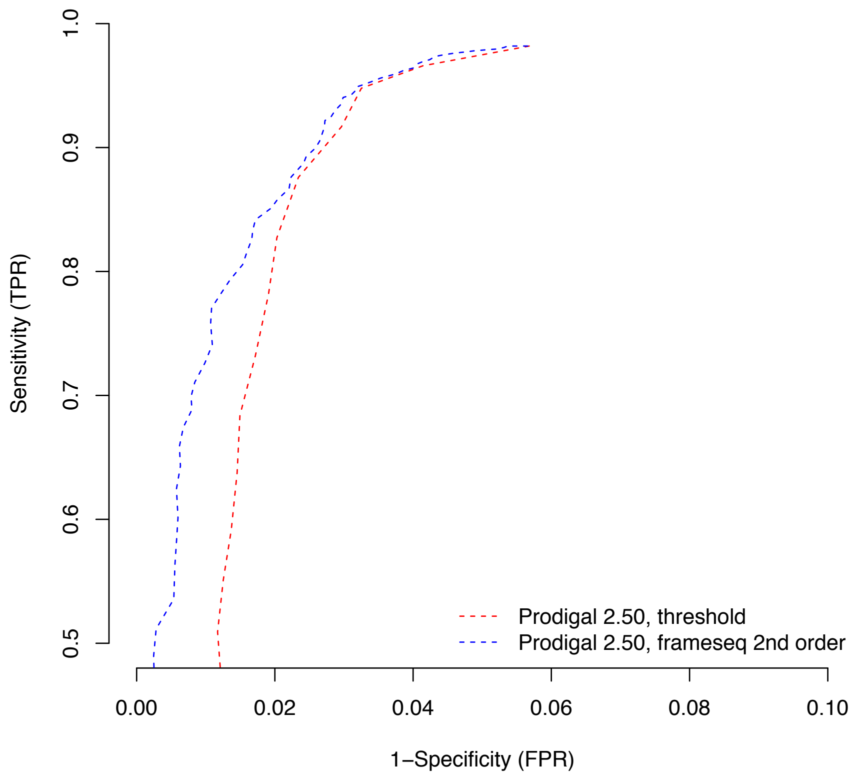


Figure 13.7: Frameseq with Prodigal. The red ROC curve shows threshold selection with Prodigal 2.50 predictions on *E. coli* and blue curve shows results of filtering these predictions with a second order Frameseq model trained on *E. coli*.

limitations:

- It relies on gene finder scores rather integrating with the algorithm of the gene finder, thereby missing out on exploiting possible correlations with signals incorporated in the gene finder.
- It relies on discretization of gene finder scores, *i.e.*, it summarizes of the information contained in prediction scores and hence cannot fully exploit these. The discretization procedure could be improved by using variable sized bins, *e.g.*, as in [113], or by instead using a continuous Hidden Markov Model.
- We do not fully exploit the nature of the gene finder score distribution for parameter smoothing. We do apply limited parameter smoothing by using the variational Bayes EM algorithm, but we could probably achieve

better generalization by fitting a suitable function to the gene finder score distribution.

- We use only a single organism as training data which become too sparse; This results in slightly over-fitted models when training on the same genome and slightly under-fitted models when training on an other genome. This situation could be amended by training on several genomes.

Despite these limitations, our method achieves good results which illustrate the usefulness of the signal, yet still leaves room for potential improvement.

We choose the problem of bacterial gene finding to exemplify the gene-reading-frame-sequence bias and its use. This problem has the nice property that it is almost solved, which enables us to use reference annotations to validate the approach. It should be noted, however, that many reference annotations are unverified results from the gene finders that we try to improve upon. This bias gives our method a slight disadvantage.

Lastly, we believe that the gene-reading-frame-sequence bias signal could have applications beyond gene finding. For instance, it may potentially benefit next generation sequencing and genome assembly where a complete model of the overall gene content of a genome would be applicable.

Availability The source code of the model and accessory scripts are freely available at:
http://github.com/frameseq/frameseq

13.5 Acknowledgements

This work is part of the project "Logic-statistic modeling and analysis of biological sequence data" funded by the NABIIT program under the Danish Strategic Research Council.

Table 13.1: Estimated transition probabilities between frame states. A cell indicates the probability that a gene in the frame indicated by the row is followed by the gene in the frame indicated by the column. Note that the strands have almost symmetrical probabilities.

from \ to	1	2	3	4	5	6
1	0.18	0.20	0.28	0.13	0.1	0.1
2	0.29	0.2	0.2	0.12	0.09	0.1
3	0.22	0.3	0.17	0.1	0.11	0.1
4	0.11	0.1	0.1	0.19	0.23	0.27
5	0.11	0.9	0.1	0.29	0.19	0.22
6	0.09	0.08	0.1	0.23	0.30	0.19

E. coli

from \ to	1	2	3	4	5	6
1	0.19	0.23	0.30	0.09	0.09	0.09
2	0.29	0.20	0.25	0.08	0.10	0.07
3	0.23	0.29	0.18	0.09	0.11	0.10
4	0.10	0.09	0.10	0.21	0.22	0.28
5	0.11	0.10	0.10	0.28	0.18	0.22
6	0.11	0.10	0.10	0.20	0.30	0.21

S. enterica

from \ to	1	2	3	4	5	6
1	0.18	0.23	0.26	0.10	0.11	0.13
2	0.27	0.18	0.22	0.12	0.09	0.12
3	0.24	0.27	0.18	0.10	0.12	0.10
4	0.09	0.11	0.11	0.18	0.20	0.31
5	0.12	0.11	0.11	0.26	0.17	0.23
6	0.09	0.09	0.11	0.24	0.29	0.18

L. pneumophila

from \ to	1	2	3	4	5	6
1	0.22	0.24	0.25	0.09	0.10	0.10
2	0.29	0.20	0.23	0.11	0.08	0.09
3	0.25	0.25	0.22	0.08	0.10	0.11
4	0.09	0.10	0.06	0.23	0.24	0.28
5	0.11	0.09	0.09	0.26	0.22	0.23
6	0.10	0.07	0.08	0.26	0.28	0.21

B. subtilis

from \ to	1	2	3	4	5	6
1	0.18	0.20	0.23	0.17	0.09	0.13
2	0.25	0.16	0.24	0.13	0.13	0.09
3	0.22	0.26	0.21	0.11	0.10	0.10
4	0.12	0.14	0.12	0.18	0.23	0.20
5	0.15	0.16	0.06	0.24	0.18	0.20
6	0.10	0.11	0.15	0.20	0.22	0.21

T. acidophilum

Chapter 14

Effects of using Coding Potential, Sequence Conservation and mRNA Structure Conservation for Predicting Pyrrolysine Containing Genes

With Sine Zambach and Henning Christiansen

Submitted to BMC Bioinformatics (in review).

Abstract

Background: Pyrrolysine (the 22nd amino acid and) is in certain organisms and under certain circumstances encoded by the amber stop codon, UAG. The circumstances driving pyrrolysine translation are not well-understood. The involvement of a predicted mRNA structure in the region downstream the UAG has been suggested, but the structure does not seem to be present in all pyrrolysine incorporating genes.

Results: We propose a strategy to predict pyrrolysine encoding genes in genomes of archaea and bacteria. We cluster open reading frames interrupted by the amber codon based on sequence similarity and rank these clusters according to several features which may influence pyrrolysine translation. The ranking effects of different features are assessed and we propose a weighted combination of these features which best explains the currently known pyrrolysine incorporating genes. We devote special attention to the effect of structural conservation and provide further substantiation to support that structural conservation may be influential – but is not a necessary factor. Finally, from the weighted ranking, we identify a number of potentially pyrrolysine incorporating genes.

Conclusions: We propose a method for prediction of pyrrolysine incorporating genes in genomes of bacteria and archaea leading to insights about the fac-

tors driving pyrrolysine translation and identification of new gene candidates. The method predicts known conserved genes with high recall and predicts several other promising candidates for experimental verification. The method is implemented as a computational pipeline which is available upon request.

Background

Over the past two decades, the standard genetic code has been revised to include the two new amino acids, selenocysteine and pyrrolysine. These amino acids are, under certain circumstances, encoded by codons which are normally stop codons. Translation of these codons can be influenced by the mRNA structure, which is the case for selenocysteine where a *cis*-acting mRNA structure (SECIS) drives translation of the opal stop codon (UGA) as selenocysteine. Similarly, a structure (PYLIS) has been identified in some genes where pyrrolysine is encoded by the (usual) stop codon UAG [223]. The structure lies in the region between the UAG codon and approximately 100 bp downstream. The role of the structure in translation is unclear and it is only conserved among some pyrrolysine incorporating genes [225]. Zhang *et al* [225] suggest that either a complete recoding of the UAG codon as pyrrolysine occurs or alternatively that UAG serves a dual function in pyrrolysine incorporating organisms; termination and translation competes leading to "statistical proteins" where both terminated and elongated products occur, but amounts of protein products may depend on circumstances.

The latter possibility is substantiated in an in vitro study [127] where the components necessary for pyrrolysine synthesis is inserted into *E.Coli.* The study shows that the PYLIS structure is not essential for translation of pyrrolysine incorporating genes, but also concludes that the presence of the structure results in a higher amount of pyrrolysine incorporating protein product and that synonymous codon mutations in the the PYLIS sequence results in lesser amounts.

The translation of pyrrolysine is associated with methane metabolism. All known organisms with methane metabolism have pyrrolysine incorporating methyltransferases, which initiate the transfer of methyl groups from methyl amines and into a process of which methane is the result [120, 78]. Three distinct methyl transferases have been identified — methyltransferase (mtmB), dimethyltransferase (mtbB), and trimethyltransferase (mttB) — each of which allow metabolism of different kinds of methyl amines [12]. Not all methyltransferases are present in all methane-producing organisms. It has been hypothesized that the availability of methyl amines regulates translation of UAG as pyrrolysine [225].

While selenocysteine is translated in a broad variety of organisms including archaea, bacteria and eukaryotes, pyrrolysine translation has so far been known to occur only in a few microbes, although it has recently been detected in a somewhat larger number of genomes within archea and bacteria [77]. So far, approximately 16 species are known to have pyrrolysine-containing genes.

Identification of selenocysteine encoding genes based on detection of the SECIS structural motif is quite successful [119]. Such approaches might successfully identify genes with a PYLIS structure, but would have difficulties predicting pyrrolysine incorporating genes without the consensus structure. Pre-

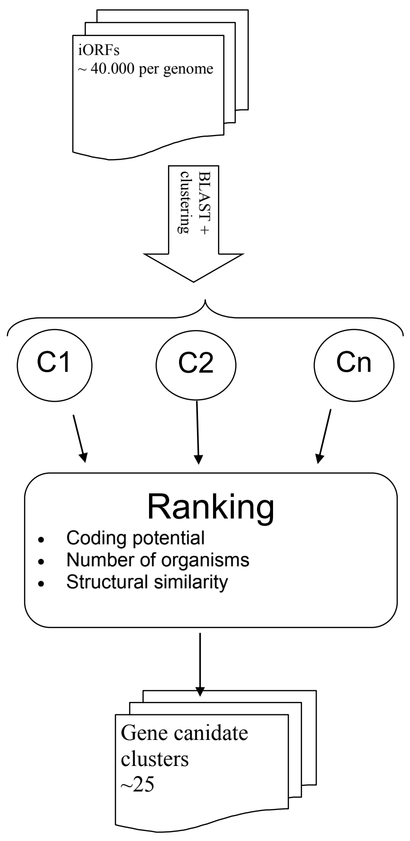


Figure 14.1: Illustration of the pipeline for identifying pyrrolysine containing genes. The process extracts iORFs which are then clustered using BLAST. Finally, the clusters are ranked according to several features.

vious computational methods for detection of pyrrolysine genes, e.g., [32, 73], has been based on homology search. Little attention has been paid to the use of structural conservation and codon sequence composition of downstream region for predicting pyrrolysine incorporating genes. In this paper we introduce an approach which takes all these factors in account. Unlike approaches like [119], we do not a assume a particular consensus structure to be present. Our model is, however, capable of taking conserved structure in the region downstream UAG into account.

14.1 Methods

14.1.1 Identification of relevant organisms

We identify organisms of interest by 1) searching for the tRNA^{pyl} synthetase using BLAST [9] and 2) verifying the results by creating a structure profile of the tRNA^{pyl} using ClustalW [210] and RNAalifold [16] and using this profile for screening the genomes with Infernal [143]. The genomes verified to have the tRNA and for which complete assembled genomes are available are used for further investigations. These are, in alphabetical order:

- *Acetohalobium arabaticum* [RefSeq:NC_014378.1]
- *Desulfitobacterium hafniense* [RefSeq:NC_011830.1]
- *Desulfobacterium autotrophicum* [RefSeq:NC_012108.1]
- *Desulfosporosinus orientis* [RefSeq:NC_016584.1]
- *Methanococcoides burtonii* [RefSeq:NC_007955.1]
- *Methanohalobium evestigatum* [RefSeq:NC_014253.1]
- *Methanohalophilus mahii* [RefSeq:NC_014002.1]
- *Methanosalsum zhilinae* [RefSeq:NC_015676.1]
- *Methanosarcina acetivorans* [RefSeq:NC_003552.1]
- *Methanosarcina barkeri* [RefSeq:NC_007355.1]
- *Methanosarcina mazei* [RefSeq:NC_003901.1]
- *Thermincola potens* [RefSeq:NC_014152.1]

A summary of the genome screening can be found in table 14.1.1.

Extraction of interrupted ORFs

We adopt the terminology *Interupted ORFs* (iORFs) from Chaudhuri and Yeates [32] and in a similar vein we extract iORFs from the genomes of interest. Interrupted ORFs are like traditional ORFs except that they contain an UAG codon between the first potential start codon and the following stop codon. Such iORFs are described by the following grammar in extended BNF notation,

$$\begin{array}{rcl}
\langle\text{iORF}\rangle & ::= & \langle\text{start}\rangle\ \langle\text{not-stop}\rangle^*\ \langle\text{amber}\rangle\ \langle\text{not-stop}\rangle^*\ \langle\text{stop}\rangle \\
\langle\text{start}\rangle & ::= & \text{TTG} \mid \text{CTG} \mid \text{ATT} \mid \text{ATC} \mid \text{ATA} \mid \text{ATG} \mid \text{GTG} \\
\langle\text{stop}\rangle & ::= & \text{TAA} \mid \text{TAG} \mid \text{TGA} \\
\langle\text{amber}\rangle & ::= & \text{TAG} \\
\langle\text{regular}\rangle & ::= & \text{AAA} \mid \ldots \mid \text{TTT} \quad \textit{// all codons except those in } \langle\text{start}\rangle \textit{ and } \langle\text{stop}\rangle \\
\langle\text{not-stop}\rangle & ::= & \langle\text{start}\rangle \mid \langle\text{regular}\rangle
\end{array}$$

An iORF is any subsequence of nucleotides specified by this the grammar, in either the sense strand or the reverse complemented anti-sense strand. The

Table 14.1: Table of potential Pyl-coding organisms and their pyrrolysine connection

Organism	Family	Known Pyl	tRNA^{Pyl}	pylS	MtxB-gene found
Methanosarcina acetivorans	*Archaea (Methanosarcinaceae)*	Yes	Yes[2]	Yes	Yes
Methanococcoides burtonii	*Archaea (Methanosarcinaceae)*	Yes	Yes[2]	Yes	Yes
Methanosarcina barkeri	*Archaea (Methanosarcinaceae)*	Yes	Yes[2]	Yes	Yes
Methanosarcina mazei	*Archaea (Methanosarcinaceae)*	Yes	Yes[2]	Yes	Yes
Methanohalophilus mahii	*Archaea (Methanosarcinaceae)*	Yes	Verified by Infernal	Yes	Yes
Methanohaophilus evestigatum	*Archaea (Methanosarcinaceae)*	Yes	Verified by Infernal	Yes	Yes
Methanosarcina thermophila	*Archaea (Methanosarcinaceae)*	Yes	Verified by Infernal	Yes	Yes
Methanosalsum zhilinae	*Archaea (Methanosarcinaceae)*	Yes	Verified by Infernal	Yes	Yes
Desulfitobacterium hafniense	*Bacteria (Clostridia)*	Yes	Yes[2]	Yes	Yes
Desulfitobacterium autotrophicum	*Bacteria (Deltaproteobacteria)*	Yes	Yes[2]	(Yes)	(Yes)
Desulfotomaculum acetoxidans	*Bacteria(Clostridia)*	Yes	Verified by Infernal	Yes	Yes
Bilophila wadsworthia	*Bacteria (Deltaproteobacteria)*	Yes	-	Yes	Yes
Acetohalobium arabaticum	*Bacteria (Clostridia)*	Yes	Verified by Infernal	Yes	Yes
Thermincola potens	*Bacteria (Clostridia)*	Yes	Verified by Infernal	Yes	Yes
Desulfosporosinus orientis	*Bacteria (Clostridia)*	-	Verified by Infernal	Yes	Yes
Desulfotomaculum gibsoniae	*Bacteria (Clostridia)*	-	-	Yes	Yes
Desulfosporosinus meridiei	*Bacteria (Clostridia)*	-	-	Yes	Yes
Geodermatophilus obscurus	*Bacteria (Actinobacteria)*	-	Verified by Infernal*	-	-

* No UAG-antisense

star notation in the first rule indicates a repetition zero or more times, and the vertical bar is used for alternatives.

We extract only iORFs which has at least 100 bases downstream the UAG. This ensures that the iORF can accommodate a PYLIS structure. A obvious consequence of this restriction is that we do not consider iORFs where the PYLIS structure possibly extends beyond the stop codon or where a hypothetical PYLIS structure occurs upstream the amber codon.

14.1.2 Reciprocal blast

Presumably, the PYLIS structure region is subject to purifying selective pressure due to both its protein function and the possible importance of a putative structure.

We identify homologous putative PYLIS sequences (100 bp downstream the UAG) conserved at amino acid level using a reciprocal BLAST search. For each iORF, we translate the region 100 bp downstream the UAG to its amino acid sequence. Then, using TBLASTN with an e-value threshold of 10^{-6} we search for this amino acid sequence in all the candidate genomes. We disregard hits to the query iORF itself and also hits to non-iORF regions.

The result of this search is a set of pairwise matches between some of the iORFs. Transitively, matched iORFs form clusters of similar hits. We disregard clusters which include an iORF where a) the region 100 bp downstream is completely overlapped by a known gene[1] in a different reading frame or b) the region partially overlaps a known functional RNA genes.

In particular, requirement a) excludes a lot of shadow ORFs which may arise when a protein sequence in a different reading frame is conserved and which as a side-effect seems like conservation in the reading frame of the iORF. Since we only consider overlaps with genes in different reading frames, iORFs for known pyrrolysine incorporating genes are not eliminated since those will be in the same reading frame as the iORF. It happens that such genes are erroneously annotated as two genes[2], where the first uses the UAG as stop codon, but these are still in the same reading frame as the iORF.

After these pruning steps, 1789 clusters remain.

14.1.3 Feature extraction

Coding potential is a measure of how likely a stretch of DNA may encode a protein. Protein coding genes exhibit a non-random sequence of codons which turn out to be a strong indicator of coding potential. Many contemporary gene finders use variants of Hidden Markov Models (HMM) to statistically model the codon sequences of genes. We apply a HMM where the hidden states correspond to amino acids which can emit the codons that encode the amino acid with distinct probabilities [137]. Additionally the HMM incorporates length modeling of genes. As an adaptation to be able to model iORFs as well as usual ORFs, the states may emit the UAG codon (with no effect on probability) in addition to the usual probabilistic codon emission. This adaptation

[1]RefSeq annotated genes, except genes where one of the words "pseudo, predicted, putative, unknown, possible, hypothetical or probable" occur in the gene product description.

[2]This is prevalent for instance in *M. Mazei.*

means that we are able to train the model on non-pyrrolysine incorporating genes, but are able decode also on iORFs[3].

We train the model on the RefSeq annotated genes of each genome resulting in maximum-likelihood parameters Θ_g for each genome g.

The trained models are used assign a probability to each iORF i from genome g. In effect, the probability reflects how much the iORF i resembles the known genes of the genome g in sequence composition and length. A log-odds ratio is calculated with the probability score of an i.i.d. nucleotide sequence model[4] as null model:

$$\mathrm{HMM}_g(i) = log\ P(i|model_{HMM}, \Theta_g) - log\ P(iORF|model_{i.i.d.}).$$

We define the coding potential of a cluster ω of size n to be the average of iORF coding potential scores within a cluster,

$$f_\omega^{coding} = \frac{1}{n} * \sum_{i \in \omega} \mathrm{HMM}_g(i), i \text{ is an iORF from genome } g.$$

Only 958 of the initial 1789 have $f_\omega^{Coding} < 0$. We only consider these 958 clusters for further investigation.

The number of homologues may be indicative of functional importance. We define a feature f_ω^{size} which measure the number of hits in a cluster, $f_\omega^{size} = \parallel \omega \parallel$. The f_ω^{size} feature does not distinguish between paralogues and orthologues. Since, orthologous conservation is a stronger indicator of important function we also define a feature $f_\omega^{Organisms}$ which is the number of unique organisms present in a cluster ω.

We expect clusters which contain real PYLIS regions to be relatively more diverse in their nucleic sequence than their amino acid sequence, whereas this may not be the case for spurious hits. On the other hand, primary sequence variation can have a degrading effect on protein function and for paralogues genes the variation may be minimal. We model diversity using the $f^{diversity}$ feature, which is calculated as the average distance between the PYLIS regions of all n iORFs in a cluster,

$$f_\omega^{diversity} = \frac{1}{n^2 - n} \sum_{\langle s,t \rangle \in \omega, s \neq t} \textsc{dist}_3(s_{pyl}, t_{pyl})$$

where s_{pyl} and t_{pyl} are the regions 100 bp downstream of the in-frame UAG, of s and t, respectively. $\textsc{dist}_m(s_{pyl}, t_{pyl})$ is the edit distance — the number of insertion, deletions or mutations needed to transform s_{pyl} into t_{pyl} — disallowing gaps which are not in multiples of m. Note that DIST_m is symmetric, i.e., $\textsc{dist}_m(a, b) = \textsc{dist}_m(b, a)$, but for convenience of notation the feature includes all pairwise distances.

Since primary sequence variation may have a degrading effect on protein, we also consider the average number of synonymous codons, f^{syn_codons}, which is defined as,

$$f_\omega^{syn\_codons} = \frac{1}{n^2 - n} \sum_{\langle s,t \rangle \in \omega, s \neq t} \frac{1}{\textsc{dist}_1(s'_{pyl}, t'_{pyl})}$$

[3]The adaptation corresponds to removing the in-frame UAG codon before decoding.

[4]A model which assumes that all nucleotides occur with the same frequency.

where, s'_{pyl} and t'_{pyl} are the amino acid sequences translated from s_{pyl} and t_{pyl}.

Note that $f^{diversity}$ and f^{syn_codons} are inversely correlated except in cases where diversity is preferentially in third codon position such that it leads to synonymous codons.

The iORF extraction step ensures that iORFs have at least 100 bases downstream an in-frame UAG. In many clusters, however, iORFs have only a few bases upstream the UAG and a start codon just upstream the UAG. In such short upstream regions it becomes more likely that the UAG and upstream start codon occurs due to chance. To address this we define the features $f^{upstream}$ and $f^{downstream}$,

$$f_{\omega}^{upstream} = \frac{1}{n}\sum_{i\in\omega} \| \, i_{start} \ldots i_{uag} \, \|$$

where $\| \, i_{start} \ldots i_{uag} \, \|$ is the distance in nucleotides from the start codon to the in-frame UAG codon and

$$f_{\omega}^{downstream} = \frac{1}{n}\sum_{i\in\omega} \| \, i_{uag} \ldots i_{stop} \, \|$$

where $\| \, i_{uag} \ldots i_{stop} \, \|$ is the distance in nucleotides from the UAG codon to the stop codon.

Assuming that the structure of PYLIS region may be important, we model structural similarity within a cluster ω with the feature $f_{\omega}^{structure}$ defined as follows. We measure similarity based on alignment of base-pairing probabilities of the sequences, which is independent of any predicted structure. We compute the base-pairing probabilities using RNAfold [93] and align these using the PMCOMP [94] with default settings. The $f^{structure}$ score is the average PMCOMP score for each pair of pylis regions in a cluster,

$$f_{\omega}^{structure} = \frac{1}{n^2 - n} \sum_{\langle s,t\rangle \in\omega, s\neq t} \text{PMCOMP}(s_{pyl}, t_{pyl}).$$

Normalization

We normalize features to the interval $[0, 1]$; $\widehat{f_i^j}$ is the normalized value for the j'th feature in the i'th cluster, defined as

$$\widehat{f_i^j} = \frac{f_i^j - \min(f^j)}{\max(f^j) - \min(f^j)}$$

where f_i^j is the value for the j'th feature for the i'th cluster, $min(f^j)$ is the minimum value for feature in any cluster and vice versa for $max(f^j)$.

14.1.3.1 Complex features

In addition to the basic features we derive two combined features based on our intuitions and on observed correlations (see Figure 14.1.3.1). f^{coding} and $f^{upstream}$ are inversely correlated in general, but positively correlated for the known clusters. The negative correlation occurs, e.g. with an iORF with a long gene just downstream the UAG (high f^{coding}) but a only a few bases upstream the UAG. The correlation effect observed between f^{coding} and $f^{downstream}$ is

Table 14.2: Raw feature values and ranking of known true positive clusters based on single features and combined ranking using regression weights. P-values for significant features ($p > 0.05$) are marked in bold.

	mttB	mtbB	mtmB	trans-posase$_1$	trans-posase$_2$	TetR	p-value
$f^{organisms}$	11	9	10	2	1	2	-
$f^{diversity}$	32.2	18.8	25.2	5.4	0.0	18.0	-
$f^{structure}$	19.0	37.6	22.5	41.3	42.4	23.2	-
f^{coding}	111.8	107.1	95.2	103.2	56.0	58.3	-
$f^{upstream}$	1016.6	992.1	605.4	293.2	325.7	297.5	-
$f^{downstream}$	495	325.4	773.2	1150	1149.7	391	-
f^{syn_codons}	90.6	96	93.7	98.4	100	91.0	-
f^{size}	18	18	19	16	3	2	-
$rank(f^{organisms})$	1	3	2	84	508	85	**0.0006**
$rank(f^{diversity})$	32	222	87	639	890	243	0.13
$rank(f^{structure})$	895	344	858	196	147	840	0.72
$rank(f^{coding})$	22	26	38	31	80	76	**0.00006**
$rank(f^{upstream})$	6	7	13	46	32	41	**0.000027**
$rank(f^{downstream})$	6	90	18	602	888	255	0.064
$rank(f^{syn\_codons})$	793	385	596	252	99	762	0.51
$rank(f^{size})$	8	9	6	10	305	498	**0.0013**
$rank(\sqrt{\widehat{f}^{coding} \times \widehat{f}^{upstream}})$	4	5	10	16	18	19	**0.000017**
$rank(\sqrt[3]{\widehat{f}^{structure} \times \widehat{f}^{diversity} \times \widehat{f}^{syn\_codons}})$	211	6	132	504	876	417	0.13
$rank(regression)$	2	3	4	15	23	19	**0.000015**

less pronounced. This motivates the addition of a combined feature, $\sqrt{\widehat{f}^{coding} \times \widehat{f}^{upstream}}$ which is the geometric mean of $\widehat{f}^{coding}$ and $\widehat{f}^{upstream}$.

Structural similarity may arise by chance and without sequence diversity it is difficult to judge the significance of a structurally similar cluster. To penalize diversity due to overhangs in blast hits and diversity which has a degrading on the amino acid sequence, we also take into account the number of synonymous codons. This leads to a combined feature $\sqrt[3]{\widehat{f}^{structure} \times \widehat{f}^{diversity} \times \widehat{f}^{syn\_codons}}$.

14.1.3.2 Feature significance

For each of the features we calculate a p-value to assess its statistical significance. We obtain p-values in the following way: We sample without replacement 10^5 means of n random ranks out of the 958 possible, where n is the number clusters containing known pyrrolysine-incorporating genes[5]. Building on the central limit theorem which guarantees that the distribution of means is normal, we fit the sampled means to normal distribution ($\mu \approx 479, \sigma \approx 112$). P-values can then be calculated from the estimated normal distribution in the usual fashion. We calculate the p-value of the mean rank — when ranked according to each feature — of the clusters known to include pyrrolysine-incorporating genes.

Features for which the null hypothesis cannot be rejected (P-VALUE > 0.05) are not used for the final ranking, i.e. we use a subset of features indexed by h : P-VALUE$(rank(f^h)) > 0.05$.

The p-values of each feature are reported in table 14.1.3.2.

[5]There are six of clusters which contain known pyrrolysine-incorporating genes.

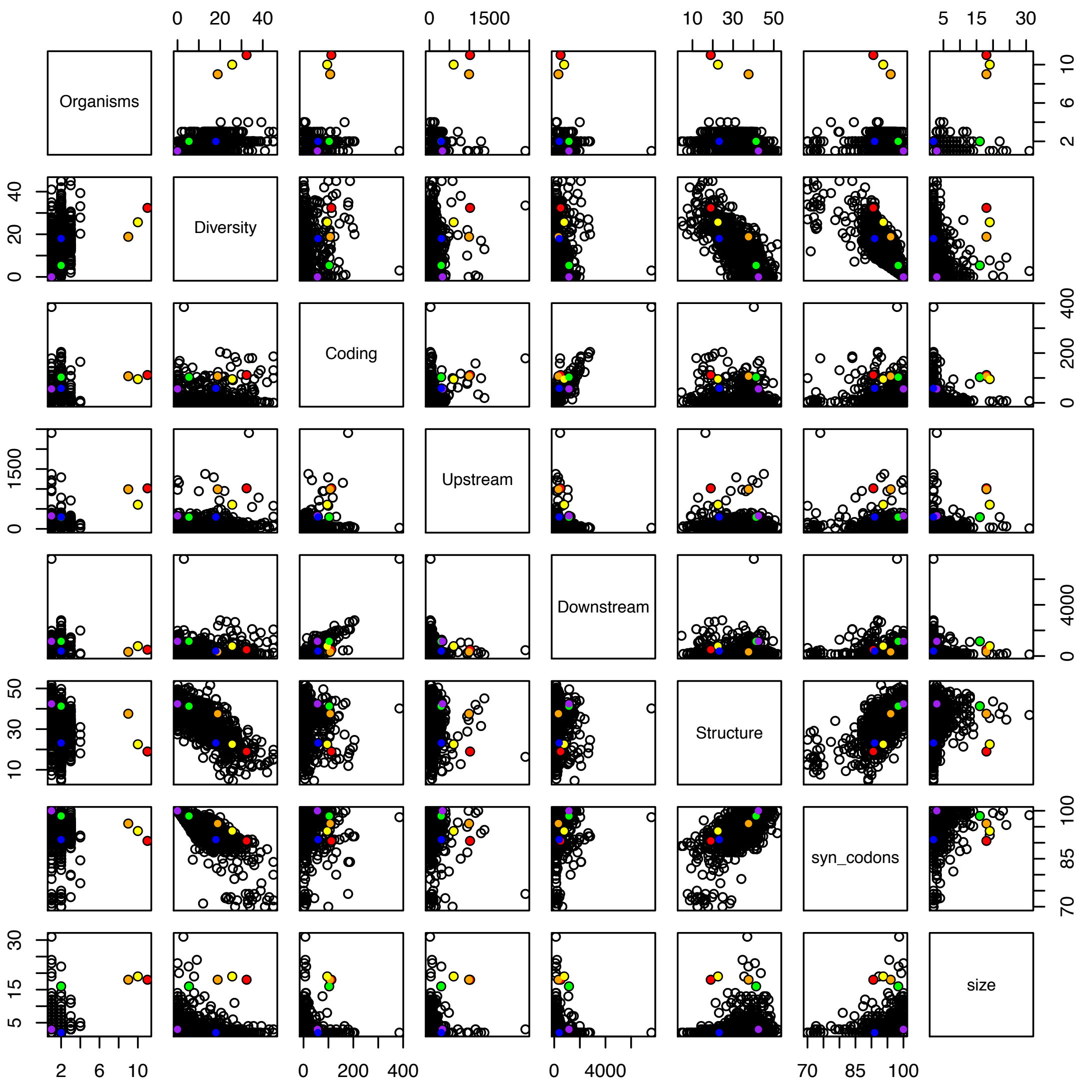


Figure 14.2: Correlations between cluster features. Known pyrrolysine gene clusters are marked with colors: *mttB* (red),*mtbB* (orange), *mtmB* (yellow), $transposase_1$ (green), $transposase_2$ (blue) and *TetR* (purple). Unknown clusters are white.

14.1.4 Ranking of clusters

Based on significant normalized feature values $\widehat{f}_i^h$ we calculate a combined cluster score which is used for ranking,

$$score_i = \sum_h w^h \widehat{f}_i^h$$

where w^h is a unique weight which we associate to feature $\widehat{f}_i^h$. We estimate weights for each feature using gradient descent to minimize the sum of ranks of positive examples,

$$\arg \min_{w^1 \ldots w^n} \sum_{e \in E^+} rank^e.$$

The set of positive examples, denoted E^+, are clusters which include known pyrrolysine-incorporating genes. There are six of these: mtbB, mttB, mtmB, two clusters with transposases only annotated in *M. Acetivorans* and transcriptional regulator of the TetR family also only annotated in *M. Acetivorans.* In the genomes we consider, there are five other (RefSeq) annotated genes with in-frame UAGs, but these are not conserved and as result they are not present in reciprocal blast clusters.

The ranking scheme is based on a simple a linear combination features, where the weights are be estimated by regression over rankings of known positive examples. It is possible to devise a more precise but complex ranking function, but we have opted for this simple scheme because we only have a few positive examples and there is a large potential for overfitting a more complex function. With the few positive examples available, we have no real means of doing cross-validation and even this simple function may slightly overfit. With the discovery of additional pyrrolysine incorporating genes, the generality of the approach can improve.

The individual rankings as well as the combined ranking are shown in table 14.1.3.2.

14.1.5 Hierarchical clustering of PYLIS structures

To assess evolutionary relationships between known pyrrolysine incorporating genes we group the PYLIS regions of these genes into clusters which are similar in structure as follows. We consider all genes annotated with UAG as well as other members of the reciprocal blast clusters which include annotated genes. We perform a hierarchical clustering of the genes using a form of neighbor joining (rapidNJ [132]) resulting in a phylogenetic tree that depicts structural conservation relationships. The clustering method relies on a distance measure between a pair of sequences, e.g., derived from a structural alignment. Rather than relying on alignment of predicted structures, our distance measure is calculated using PMCOMP [94] which is based on alignment of base-pairing probabilities of the sequences. We compute the base-pairing probabilities using the RNAFOLD tool. The distance between two sequences is the inverse of the alignment score, $score_{A-B}$, from PMCOMP:

$$dist_{A-B} = \frac{1}{score_{A-B}}.$$

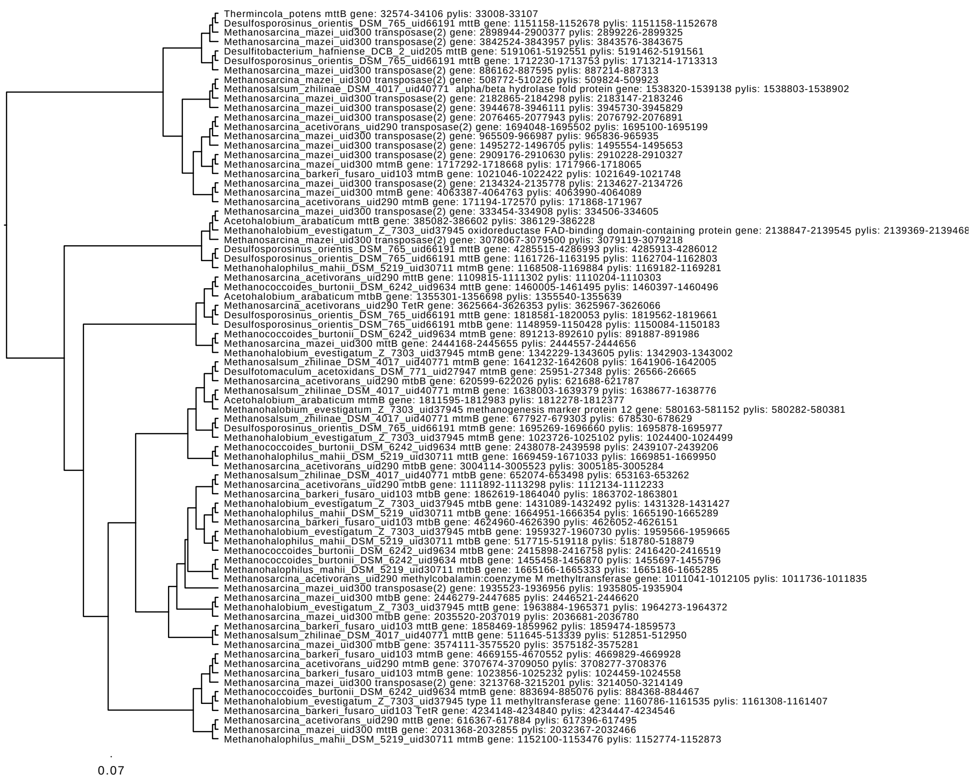

Thermincola_potens mttB gene: 32574-34106 pylis: 33008-33107
Desulfosporosinus_orientis_DSM_765_uid66191 mttB gene: 1151158-1152678 pylis: 1151158-1152678
Methanosarcina_mazei_uid300 transposase(2) gene: 2898944-2900377 pylis: 2899226-2899325
Methanosarcina_mazei_uid300 transposase(2) gene: 3842524-3843957 pylis: 3843576-3843675
Desulfitobacterium_hafniense_DCB_2_uid205 mttB gene: 5191061-5192551 pylis: 5191462-5191561
Desulfosporosinus_orientis_DSM_765_uid66191 mttB gene: 1712230-1713753 pylis: 1713214-1713313
Methanosarcina_mazei_uid300 transposase(2) gene: 886162-887595 pylis: 887214-887313
Methanosarcina_mazei_uid300 transposase(2) gene: 508772-510226 pylis: 509824-509923
Methanosalsum_zhilinae_DSM_4017_uid40771 alpha/beta hydrolase fold protein gene: 1538320-1539138 pylis: 1538803-1538902
Methanosarcina_mazei_uid300 transposase(2) gene: 2182865-2184298 pylis: 2183147-2183246
Methanosarcina_mazei_uid300 transposase(2) gene: 3944678-3946111 pylis: 3945730-3945829
Methanosarcina_mazei_uid300 transposase(2) gene: 2076465-2077943 pylis: 2076792-2076891
Methanosarcina_acetivorans_uid290 transposase(2) gene: 1694048-1695502 pylis: 1695100-1695199
Methanosarcina_mazei_uid300 transposase(2) gene: 965509-966987 pylis: 965836-965935
Methanosarcina_mazei_uid300 transposase(2) gene: 1495272-1496705 pylis: 1495554-1495653
Methanosarcina_mazei_uid300 transposase(2) gene: 2909176-2910630 pylis: 2910228-2910327
Methanosarcina_mazei_uid300 mtmB gene: 1717292-1718668 pylis: 1717966-1718065
Methanosarcina_barkeri_fusaro_uid103 mtmB gene: 1021046-1022422 pylis: 1021649-1021748
Methanosarcina_mazei_uid300 transposase(2) gene: 2134324-2135778 pylis: 2134627-2134726
Methanosarcina_mazei_uid300 mtmB gene: 4063387-4064763 pylis: 4063990-4064089
Methanosarcina_acetivorans_uid290 mtmB gene: 171194-172570 pylis: 171868-171967
Methanosarcina_mazei_uid300 transposase(2) gene: 333454-334908 pylis: 334506-334605
Acetohalobium_arabaticum mttB gene: 385082-386602 pylis: 386129-386228
Methanohalobium_evestigatum_Z_7303_uid37945 oxidoreductase FAD-binding domain-containing protein gene: 2138847-2139545 pylis: 2139369-2139468
Methanosarcina_mazei_uid300 transposase(2) gene: 3078067-3079500 pylis: 3079119-3079218
Desulfosporosinus_orientis_DSM_765_uid66191 mttB gene: 4285515-4286993 pylis: 4285913-4286012
Desulfosporosinus_orientis_DSM_765_uid66191 mttB gene: 1161726-1163195 pylis: 1162704-1162803
Methanohalophilus_mahii_DSM_5219_uid30711 mtmB gene: 1168508-1169884 pylis: 1169182-1169281
Methanosarcina_acetivorans_uid290 mttB gene: 1109815-1111302 pylis: 1110204-1110303
Methanococcoides_burtonii_DSM_6242_uid9634 mttB gene: 1460005-1461495 pylis: 1460397-1460496
Acetohalobium_arabaticum mtbB gene: 1355301-1356698 pylis: 1355540-1355639
Methanosarcina_acetivorans_uid290 TetR gene: 3625664-3626353 pylis: 3625967-3626066
Desulfosporosinus_orientis_DSM_765_uid66191 mttB gene: 1818581-1820053 pylis: 1819562-1819661
Desulfosporosinus_orientis_DSM_765_uid66191 mtbB gene: 1148959-1150428 pylis: 1150084-1150183
Methanococcoides_burtonii_DSM_6242_uid9634 mtmB gene: 891213-892610 pylis: 891887-891986
Methanosarcina_mazei_uid300 mttB gene: 2444168-2445655 pylis: 2444557-2444656
Methanohalobium_evestigatum_Z_7303_uid37945 mtmB gene: 1342229-1343605 pylis: 1342903-1343002
Methanosalsum_zhilinae_DSM_4017_uid40771 mtmB gene: 1641232-1642608 pylis: 1641906-1642005
Desulfotomaculum_acetoxidans_DSM_771_uid27947 mtmB gene: 25951-27348 pylis: 26566-26665
Methanosarcina_acetivorans_uid290 mtbB gene: 620599-622026 pylis: 621688-621787
Methanosalsum_zhilinae_DSM_4017_uid40771 mtmB gene: 1638003-1639379 pylis: 1638677-1638776
Acetohalobium_arabaticum mtmB gene: 1811595-1812983 pylis: 1812278-1812377
Methanohalobium_evestigatum_Z_7303_uid37945 methanogenesis marker protein 12 gene: 580163-581152 pylis: 580282-580381
Methanosalsum_zhilinae_DSM_4017_uid40771 mtmB gene: 677927-679303 pylis: 678530-678629
Desulfosporosinus_orientis_DSM_765_uid66191 mtmB gene: 1695269-1696660 pylis: 1695878-1695977
Methanohalobium_evestigatum_Z_7303_uid37945 mtmB gene: 1023726-1025102 pylis: 1024400-1024499
Methanococcoides_burtonii_DSM_6242_uid9634 mttB gene: 2438078-2439598 pylis: 2439107-2439206
Methanohalophilus_mahii_DSM_5219_uid30711 mttB gene: 1669459-1671033 pylis: 1669851-1669950
Methanosarcina_acetivorans_uid290 mtbB gene: 3004114-3005523 pylis: 3005185-3005284
Methanosalsum_zhilinae_DSM_4017_uid40771 mtbB gene: 652074-653498 pylis: 653163-653262
Methanosarcina_acetivorans_uid290 mtbB gene: 1111892-1113298 pylis: 1112134-1112233
Methanosarcina_barkeri_fusaro_uid103 mtbB gene: 1862619-1864040 pylis: 1863702-1863801
Methanohalobium_evestigatum_Z_7303_uid37945 mtbB gene: 1431089-1432492 pylis: 1431328-1431427
Methanohalophilus_mahii_DSM_5219_uid30711 mtbB gene: 1664951-1666354 pylis: 1665190-1665289
Methanosarcina_barkeri_fusaro_uid103 mtbB gene: 4624960-4626390 pylis: 4626052-4626151
Methanohalobium_evestigatum_Z_7303_uid37945 mtbB gene: 1959327-1960730 pylis: 1959566-1959665
Methanohalophilus_mahii_DSM_5219_uid30711 mtbB gene: 517715-519118 pylis: 518780-518879
Methanococcoides_burtonii_DSM_6242_uid9634 mtbB gene: 2415898-2416758 pylis: 2416420-2416519
Methanococcoides_burtonii_DSM_6242_uid9634 mtbB gene: 1455458-1456870 pylis: 1455697-1455796
Methanohalophilus_mahii_DSM_5219_uid30711 mtbB gene: 1665166-1665333 pylis: 1665186-1665285
Methanosarcina_acetivorans_uid290 methylcobalamin:coenzyme M methyltransferase gene: 1011041-1012105 pylis: 1011736-1011835
Methanosarcina_mazei_uid300 transposase(2) gene: 1935523-1936956 pylis: 1935805-1935904
Methanosarcina_mazei_uid300 mtbB gene: 2446279-2447685 pylis: 2446521-2446620
Methanohalobium_evestigatum_Z_7303_uid37945 mttB gene: 1963884-1965371 pylis: 1964273-1964372
Methanosarcina_mazei_uid300 mtbB gene: 2035520-2037019 pylis: 2036681-2036780
Methanosarcina_barkeri_fusaro_uid103 mttB gene: 1858469-1859962 pylis: 1859474-1859573
Methanosalsum_zhilinae_DSM_4017_uid40771 mttB gene: 511645-513339 pylis: 512851-512950
Methanosarcina_mazei_uid300 mtbB gene: 3574111-3575520 pylis: 3575182-3575281
Methanosarcina_barkeri_fusaro_uid103 mtmB gene: 4669155-4670552 pylis: 4669829-4669928
Methanosarcina_acetivorans_uid290 mtmB gene: 3707674-3709050 pylis: 3708277-3708376
Methanosarcina_barkeri_fusaro_uid103 mtmB gene: 1023856-1025232 pylis: 1024459-1024558
Methanosarcina_mazei_uid300 transposase(2) gene: 3213768-3215201 pylis: 3214050-3214149
Methanococcoides_burtonii_DSM_6242_uid9634 mtmB gene: 883694-885076 pylis: 884368-884467
Methanohalobium_evestigatum_Z_7303_uid37945 type 11 methyltransferase gene: 1160786-1161535 pylis: 1161308-1161407
Methanosarcina_barkeri_fusaro_uid103 TetR gene: 4234148-4234840 pylis: 4234447-4234546
Methanosarcina_acetivorans_uid290 mttB gene: 616367-617884 pylis: 617396-617495
Methanosarcina_mazei_uid300 mttB gene: 2031368-2032855 pylis: 2032367-2032466
Methanohalophilus_mahii_DSM_5219_uid30711 mtmB gene: 1152100-1153476 pylis: 1152774-1152873
0.07

Figure 14.3: Structural clustering of PYLIS regions from iORFs in the clusters for known pyrrolysine incorporating genes.

The resulting dendrogram of PYLIS regions is shown in figure 14.1.5.

14.2 Results and discussion

To ensure generalization capability and to minimize model complexity we systematically assess the ranking features using a criteria of statistical significance. In effect, this assessment leads to a deeper understanding of the factors influencing pyrrolysine translation. We inspect and discuss the the list of candidate genes ranked by our method and we discuss structural conservation for known pyrrolysine incorporating genes.

14.2.1 Ranking of gene clusters

Our method to automatically identify pyrrolysine coding candidate genes is unique in utilizing both coding potential, structural conservation and amino acid conservation. Additionally, we take the number of organisms with homologous PYLIS regions and the length of up and downstream the potential in-frame UAG into account.

The ranking is systematically modeled from the known genes taking several factors into account and weigh the different features with respect to the distance of known genes using pyrrolysine (Table 14.1.3.2).

The method has some limitations due to the assumptions that we make. We assume that the PYLIS region is 100 bases downstream the UAG and is well-conserved due to the presumptive presence of a PYLIS structure. Our method cannot detect pyrrolysine containing genes which have divergent PYLIS regions with no significant conservation in homologues or genes with only non-pyrrolysine incorporating homologues.

Our approach is similar to an earlier approach called read-through Similarity Analysis [32]. As in our approach, the authors extract iORFs from candidate genomes and perform a reciprocal blast analysis. The query is a 100bp window pivoting around the read-through codon. In the case of pyrrolysine incorporating genes this means that the downstream region is shorter than in our case and may not hold the entire PYLIS structure. They calculate an alignment score for the region downstream the read-through codon and a measure of statistical significance by aligning shuffled sequences. Hits with sufficiently high significance are examined further. These hits are expanded using PSI-BLAST and manually checked for non read-through codons lining up with read-though codons. Their driving assumption is that read-through genes will have non read-through homologues.

Another approach sets out to discover unknown amino acids by conservation of iORFs [73]. The approach is also capable of detecting pyrrolysine incorporating genes. It also begins with iORF extraction and uses BLAST to detect homologous sequences. The authors use a query (80bp) centered around the stop codon. Similar to our approach the BLAST search results in a number of clusters which is then reduced by pruning rules. Unlike our approach, they only examine clusters with interspecies matches. To distinguish adjacent genes, they exploit the synteny, by looking for blast hits to the N-terminal and the C-terminal of candidates in other genomes. If there are distinct, but closely arranged hits in other genomes, this is taken to indicate evidence of adjacent genes rather than a single pyrrolysine incorporating gene. Furthermore, they filter iORFs clusters based on purifying selection, i.e. they prune hits with significantly high incidents of non-synonymous codon usage in the sequences flanking the read-through codon.

However, unlike [32] we do not require non-pyrrolysine homologues, and unlike [73] our method can detect genes with only paralogoue conservation. This is reasonable, since some annotated pyrrolysine containing genes are only found in several copies in the same genome.

Candidate genes

As result of our method we obtain ranking of the clusters in which the known pyrrolysine incorporating genes occur within the top 25 clusters. The ranked list is supplied as Additional File 1. Several other high ranking clusters seems to be false positives, but there are also some interesting candidates.

- An example, which is probably a false positive, is the cluster with rank 11. Although it is conserved in three organisms and is on average 711 bases long, it has the problem that in Thermincola potens it overlaps with a much longer gene in another reading frame. Either the long gene (CRISPR-assosiated protein Cas2) is wrongly annotated, or the short one (CRISPR-associated protein Cas1) “shadows” by incidence the long one. However, in the two other organisms D. hafniense and D. orientis the possibility of a direct translation of UAG is more probable since the two annotated genes are in same reading frame and within the predicted iORF.

- An example of a probable gene participating in neutral evolution (drift) is the cluster with rank 7. It consists of only two Thermincola potens genes, which are almost identical. However, they are both long and the genes are flanking another gene (putative anaerobic sulfite) which is in the same reading frame. (a similar neutral evolution has been believed to occur in the known genes in M. acetivorans as in cluster 19 and 23 [91]). This type of pyrrolysine usage does usually not create new genes with new functions.

- In cluster 16, a less neutral selection is probable in the evolution towards using UAG in-frame, since is has occurred in three different Archaea. The organisms are closely related and the iORF covers a gene (sensory transduction histidine kinase) and a hypothetical protein in same reading frame in M. acetivorans, a full pseudogene in M. mahii, and a shadow part of a sensory transduction histidine kinase in M. barkeri. Likewise are the three methyl transferases a product of selection towards a function of producing methane.

All candidates need to be verified experimentally before they can be determined as pyrrolysine encoding. In addition to this, the clusters containing the three known methyl transferases include several genes annotated as pseudogenes, as two genes or not at all. Only 21 of the 46 methyl transferases in the 12 investigated organisms are annotated correctly, i.e., as one non-pseudo gene with with an in-frame UAG stop codon. These are likely deficiencies in the existing annotations and should be corrected/included after further investigation.

14.2.2 Structural conservation and the evolution of genes

It is possible to create a relatively canonical secondary structure[6] for the PYLIS region of mtmB and mtbB. The same region in mttB does not show any sign of common structure. However, since mttB is the gene present in most different organisms and has a high degree of sequence diversity, an elevated variance in the structure is possible. See Figure 14.2.2 for predicted structures.

A search using INFERNAL reveals that using predicted structures has a positive impact on the recall and precision of detecting the pyrrolysine containing genes mtmB and mtbB, when compared to searching without the predicted structure (data not shown).

Our assessment of ranking features indicate that the majority of clusters have a degree of potential structural similarity which is comparable to the known genes. Consequently, neither the $f^{structure}$ feature nor the complex $\sqrt[3]{\widehat{f}^{structure} \times \widehat{f}^{diversity} \times \widehat{f}^{syn\_codons}})$ feature is not adequate for recovering

[6]This structures have more than 90 percent canonical base pairs, a free energy on less than -15 and base pair probability above 0.6.

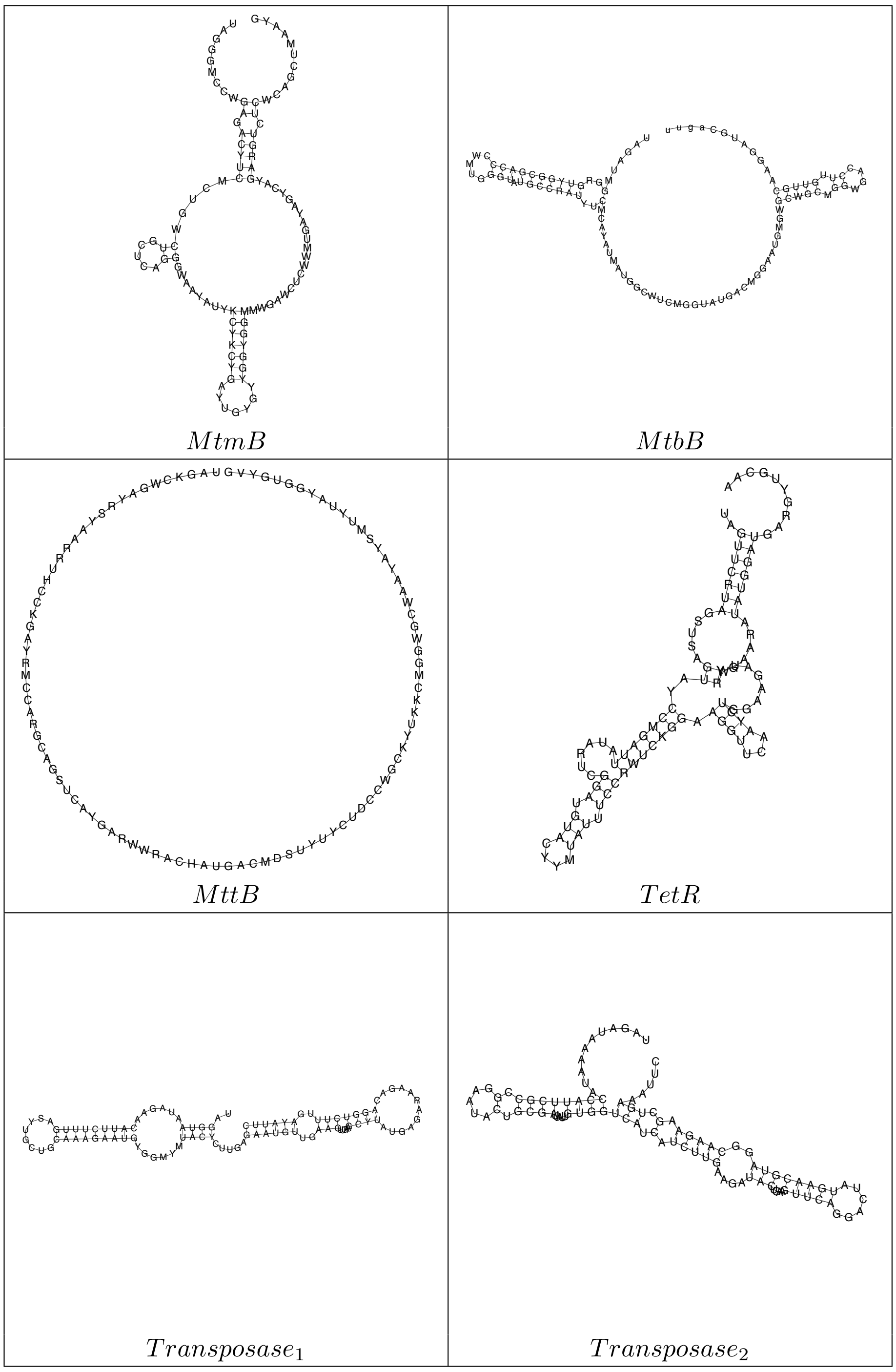


Figure 14.4: Predicted structures for each of the known pyrrolysine incorporating clusters. All structures are predicted using ClustalW[210]+RNAalifold[16] through the WAR web service [213].

known genes. The known genes seem to have either high structural similarity and low primary sequence diversity (as is the case for the two transposases) or they have low structural similarity but high primary sequence diversity. We see a different picture if we instead consider only the three methyl transferase clusters when calculating feature significance; Then, the p-value for $rank(\sqrt[3]{\widehat{f}^{structure} \times \widehat{f}^{diversity} \times \widehat{f}^{syn\_codons}})$ is 0.012 which is significant within the 0.05 limit. This may suggest that the methyl transferases have important structures although the statistical significance does not necessarily imply biological relevance. It is more difficult to say if there is an important structure in the transposases because of the high sequence similarity.

On the other hand the structural clustering of the PYLIS sequences of known pyrrolysine incorporating genes (see Figure 14.1.5) reveals relatively compact clusters. One cluster clearly corresponds to $transposase_2$, but with few elements from the mttB and mtmB clusters. The majority of PYLIS sequences from the mtmB cluster falls into one of two fairly coherent groupings which are dominated by mtmB genes. One of these groups, however, include the methanogenesis marker protein 12 gene from *M. evestigatum* and the other mtmB grouping includes one of the TetR genes. The other TetR gene is in a smaller grouping which include elements from all methyl transferases. The majority of mtbB members also form a coherent subgroup. The mttB members does not seem to form any coherent grouping.

While it is possible to observe general trends of the clustering, chance similarities distort the clustering accuracy to an undesired degree and outliers should not be given too much significance. It is, however, clear that the UAG downstream regions of Pyrrolysine incorporating genes do not all share similar structures. There are distinct sub groupings which may correspond to distinct structures and hence it seems that there are several PYLIS structures.

Our findings support that pyrrolysine has different ways to evolve in the genomes containing the $tRNA^{pyl}$ as suggested in [91]. For the methyl transferases, a selection for producing methane may have conserved the structures as well as the amino acid residue. In other cases a neutral evolution is believed to occur, allowing for a single mutation leading to an in-frame amber stop codon [91]. In our list of candidate genes, a few high ranking candidates had multiple paralogues of the UAG in-frame codon either within only one genome or among a few only. These genes are conserved within a given species and have even accomplished duplication events. However, as the mutation is not conserved among different species, these may be relatively recent adaptations.

These different gene evolution models constitute a challenge for the approach to select the clusters that represent true positives among clusters of iORFs.

14.3 Conclusions

In this work we presented a method for predicting pyrrolysine coding genes. The method clusters genes with homologue sequences downstream the in-frame UAG. The clusters are ranked according to observed properties of existing homologous pyrrolysine incorporating genes so that top ranking candidates correspond to known pyrrolysine incorporating gene families or to promising new candidates.

Our method is successful in recovering conserved pyrrolysine containing genes and additionally detects several promising candidates which are not currently annotated. We provide a ranked list of potential pyrrolysine coding gene candidates.

In addition our method provides insights into the features that characterize pyrrolysine incorporating genes. We find evidence of conserved structures only within MtmB and MtbB and provide substantiation to suggest that pyrrolysine genes may also arise due to neutral evolution.

Author's contributions

CTH designed and wrote the pipeline, collected the data, performed the data analysis, and participated in the drafting of the manuscript. SZ participated in designing the pipeline, participated in the data analysis, performed the manual analysis of candidate genes clusters and participated in the drafting of the manuscript. HC supervised and financed the study, participated in designing the pipeline, and participated in the drafting of the manuscript. All authors approved the final manuscript.

Acknowledgements

We would like to thank Anders Krogh and Ole Skovgaard for their valuable feedback on an early draft of this paper. This work is part of the project "Logic-statistic modeling and analysis of biological sequence data" funded by the NABIIT program under the Danish Strategic Research Council.

Chapter 15

Conclusions and Future Work

15.1 Conclusions

This thesis presents a collection of papers which contribute to the research goal of answering the questions,

- *To what extent is it possible to use probabilistic logic programming for biological sequence analysis?*
- *How can constraints relevant to the domain of biological sequence analysis be combined with probabilistic logic programming?*
- *What are the limitations with regard to efficiency and how can these be dealt with?*

The first question is perhaps the most central one, but it is difficult to answer precisely. It is trivial to answer whether it is possible to use probabilistic logic programming for biological sequence analysis with a simple proof of concept. The extent is more difficult to quantify, but it is intricately related to the next two questions; It depends on how accurately and conveniently constraints from the domain of biological sequence analysis can be combined with probabilistic logic models and it depends on how efficiently this can be done.

The dissertation addresses the questions by providing successful cases of applications, abstractions and optimizations. This approach results in a non-exhaustive answer to the questions, but demonstrates the potential of the approach and shows what is at least possible.

15.1.1 To what extent is it possible to use probabilistic logic programming for biological sequence analysis?

It is clear that probabilistic logic programming is a powerful abstraction which is eminently suited for modeling biological sequences. Common types of probabilistic sequence models can be easily and concisely expressed and also extended to non-standard cases (as demonstrated, e.g., in chapter 4).

Applications of probabilistic logic programming to deal with real biological problems presented are presented in chapter 13 and chapter 14. This shows that probabilistic logic programming is not merely a powerful abstraction, but that it is a valuable and practical tool for biological sequence analysis.

15.1.2 How can constraints relevant to the domain of biological sequence analysis be combined with probabilistic logic programming?

When constraints are expressed through a high level language or abstraction which is closer to the domain, then they enable a user to express what is on his mind without having to worry about complicated implementation details.

Probabilistic logic programming is a suitable language for building higher level abstractions which can facilitate more convenient expression of constraints from the domain. This is demonstrated, e.g., in chapter 5, 6 and 8. The abstractions presented in this dissertation are motivated by applications to biological sequence analysis, but are general abstractions which may also be useful in other domains.

The notion of constraints is difficult to separate from notion of a model. Indeed, models are typically based on assumptions which may themselves be seen as inherent constraints. There is a good reason, however, for separating the two concepts. It is convenient to be able to express standard sequence models and then later add constraints as a form of rationalization. The benefits are in a sense similar to those obtained by the separation of logic and control where general algorithms may be reused for different types of models. With constraints as a separate notion, models may be reused with different constraints.

The use of probabilistic logic models with constraints has been demonstrated in a variety of ways. Constraints can serve as the basis of a high-level language to express biological knowledge and assumptions as demonstrated in chapter 5, 6 and 8. Constraints have also been demonstrated to be useful for dealing with efficiency limitations of probabilistic logic models by reducing the search space (chapter 5). Furthermore, a step towards demonstrating that constraints (CHR) may serve as the core foundation to express probabilistic models and related inference algorithms has been taken in chapter 7.

15.1.3 What are the limitations with regard to efficiency and how can these be dealt with?

While the separation of logic and control enables general inference algorithms to be used for wide range models, challenges remain with the control part.

The PRISM system for probabilistic logic programming includes brilliant generalizations of probabilistic inference algorithms which in theory guarantees identical complexity to the best known algorithms for the types of models being expressed. Unfortunately, they are not always as efficient in practice as they could be. Fortunately, because of the separation between logic and control, models automatically benefit from improvements of algorithms. This separation also enables approaches based on program transformation as demonstrated by chapter 11 and [40]. Such transformations would be difficult if logic and control aspects were thoroughly entangled.

In this dissertation, key limitations regarding *efficiency* which are problematic when using probabilistic logic programming for biological sequence analysis have been addressed. Efficiency issues related to tabling — both with regard to structured data (chapter 11 and 12) and constraints (chapter 5 and 6) — have been addressed and models for many kinds of interesting problems can now be implemented with sufficient efficiency.

When the LoSt project started out, we were able to use PRISM-based HMMs to analyze sequences of at most a few hundred bases within a few minutes. Later optimizations, *i.e.*, pertaining to non-discriminating arguments [40], made it possible analyze sequences of a few thousand bases within the same timeframe. With the tabling optimizations presented in this thesis, it is now possible to analyze hundreds of thousands of bases within seconds. For simple HMMs, efficiency is no longer an issue. The empirical evaluations in chapter 11 and chapter 12 indicate that inference with tested models occurs with close to optimal time complexity. At the scale of analyzing hundred of thousands of bases, other limitations regarding floating point calculations begin to show up in PRISM even if probabilities are represented in log-scale.

For more complex models, other techniques to improve efficiency have been devised. Model decomposition using Bayesian Annotation Networks (chapter 9) can be useful when joint-model inference is still not feasible. The BANpipe pipeline programming language (chapter 10) — which supports this paradigm — provides automatic parallelization. This may significantly reduce the time spent on inferences on multicore computers. The incorporation of constraints has also been demonstrated to be able to reduce running time under certain conditions (chapter 5).

15.2 Future work

While the research question has to a large degree been resolved, a lot of open ends remain. Many ideas and opportunities have arisen from working with the domain and remain to be explored. In this section I describe a few of these, which I think would be worthwhile to investigate further.

Tabling The explored approaches to efficient tabling of structured result in efficient inference for a wide range of dynamic programming problems, but the approaches only work for ground data and variant based tabling. It would be useful to have techniques which allow for efficient tabling of structured data with non-ground data. For instance, unsupervised learning in PRISM normally works by using variables in input data. This scenario would not benefit from the approaches given in this thesis. Currently, B-Prolog and PRISM use variant-based tabling, which is restricted to exact matching of terms to tabled terms. Subsumption-based tabling have been shown to be quite useful, for instance to develop algebraic variants of probabilistic logic programming frameworks, *i.e.*,[162]. This framework essentially enables the programmer to replace the underlying semantics of the language, with *e.g.*, possibilistic logic programming rather than probabilistic logic programming. Subsumption-based tabling would also be interesting to explore for Constrained HMMs, where it could support integration of advanced types of constraints through tabling-based subsumption testing.

BANpipe The banpipe language has recently been reimplemented in Logtalk and now supports most Prolog systems. BANpipe modules can be written for different Prolog systems, but may still be integrated in the same BANpipe script. It has thoroughly documented and is available at `http://banpipe.org`.

While BANpipe is functional and can be used to develop realistic pipelines for bioinformatics projects, some work needs to be done to make it sufficiently usable and accessible for the general bioinformatics community. The most pressing concerns are documentation with regard to specific sequence analysis components.

A key issue is portability and integration with existing tools used for bioinformatics, in particular in the context of logic programming. A holistic framework for bioinformatics in Prolog — blipkit [140] — has previously been introduced, but its components are very different from the ones in the BANpipe framework. Since BANpipe now runs on most Prolog systems, it should be relatively easy to integrate parts of blipkit as banpipe modules.

Nicos Angelopoulos has argued[1] for a piece-meal approach to bioinformatics which sounds like a more promising way collaborate and reuse logic programming based bioinformatics components. Instead of creating holistic frameworks, it should be easy to integrate smaller tools with specific purposes. BANpipe provides a way to do this, if components are made available as BANpipe modules, i.e., provides an interface file.

BANpipe is currently restricted to parallel invocation tasks residing on the same system. In the future, BANpipe may be adapted to support a distributed setting.

Better genome models and combiners Two of the papers in this thesis provide two different way to optimize a set of predictions from an underlying gene finder on the genome level. One is the approach of using global optimization and constraints for overlap resolution (chapter 6) and the other is utilizing the sequence of gene reading frames (chapter 13). By themselves, these methods have only a little impact in improving accuracy. I believe, however, that integrating these two techniques may result in significantly better accuracy and I would like to investigate this.

A limitation of the current implementation of the gene reading frame model is that it relies on discretization of the continuous confidence scores from gene finders. This discretization either leads to information loss or to a situation with sparsity of training data. Recent work on continuous variables in PRISM [100] seems to be a promising alternative to discretization. This work extends PRISM, including the main inference algorithms, to support multi-valued switches which have continuous-valued rather than symbolic outcomes.

Further experiments and investigation needs to be done in order to assess the possibility and impact of using this approach with the gene reading frame model.

Furthermore, there is no reason why predictions from only one gene finder should be considered. Combiners have been very successful for eukaryotic gene finding and I believe that they could also be useful in prokaryotic gene finding given this approach.

Alternative Viterbi search strategies for PRISM programs PRISMs Viterbi inference is usually quite efficient, but with the addition of constraints, the search may become prohibitively slow. In essence, PRISMs existing search strategy is depth-first search with a failure-driven loop and using tabling to

[1]In his WCB 2012 talk.

avoid recomputations. This means that in programs with constraints, a lot of programs branches which will eventually fail will be faithfully (but unnecessarily) explored.

I would like to explore alternative search strategies for PRISM, which may more efficiently deal with programs with constraints. I believe that a combination of branch-and-bound and a-star search could be much more efficient than the current approach for such programs. With branch and bound search, certain low-probability branches could be pruned if (as soon as) their probability is lower than that of a previously found solution. Similar pruning can occur with a-star search, but it may be difficult to devise admissible heuristics which are required by this kind of search. Inspiration may also be drawn from strategies used in Answer Set Programming. Conflicting sets of constraints (no-goods) observed in the search may be added as new constraints to further restrict search. Similarly, the search may be restarted with different strategies, that may be more efficient when enforcing no-goods and low-probability pruning.

CHR and probabilistic grammars Chapter 7 presents a CHR based version of the Viterbi algorithm for HMMs. The use of CHR for expressing probabilistic models has been a relatively hot topic in the LoSt group and for me. An early (unfinished and unpublished) approach to constrained HMMs tackled the problem by transforming a CHR representation of unconstrained HMMs into constrained variants. We have also experimented with other related CHR based algorithms for working with probabilistic grammars, including an implementation of the inside-outside algorithm and an algorithm for transforming PCFGs to HMMs (approximations). This work — which has been conducted in cooperation with Ole Torp Lassen — is still unfinished. However, from our experiments so far, it seems that these kinds of algorithms can be quite elegantly expressed with CHR.

Probabilistic regular expressions The current implementation of probabilistic extended regular expressions works, but only for relatively small examples. I have several ideas on how the implementation can be made much more scalable and I plan to eventually make a more efficient version available as stand-alone tool similar to grep. A problem with the current implementation is that — with backreferences — it may explore a lot of paths which fail. The engine would generally benefit from a better (Viterbi) search strategy, but I am also considering to use a suffix tree representation to efficiently identify and prune paths where it is obvious that the backreference cannot be matched.

Suffix trees can be conveniently represented using tabling, but for inexact matching a form of subsumption based tabling would be useful.

A lot of other techniques, e.g., partial evaluation, have been proposed for efficient regular expression matching. Quite possibly, many of these could be adopted to probabilistic logic programming and applied to probabilistic regular expressions as well. This, of course, needs further investigation.

Bibliography

[1] S. Abdennadher, T. W. Frühwirth, and H. Meuss. On confluence of constraint handling rules. In E. C. Freuder, editor, *CP*, volume 1118 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 1996.

[2] T. Abe, T. Ikemura, J. Sugahara, A. Kanai, Y. Ohara, H. Uehara, M. Kinouchi, S. Kanaya, Y. Yamada, A. Muto, and H. Inokuchi. trnadb-ce 2011: trna gene database curated manually by experts. *Nucleic Acids Research*, 39(suppl 1):D210–D213, 2011.

[3] H. Abramson and V. Dahl. *Logic Grammars.* Symbolic Computation. Springer-Verlag, 1989.

[4] B. Alberts, A. Johnson, J. Lewis, M. Raff, K. Roberts, and P. Walter. *Molecular biology of the cell.* Garland Science Taylor & Francis Group, 4 edition, 2002.

[5] J. E. Allen, W. H. Majoros, M. Pertea, and S. L. Salzberg. Jigsaw, genezilla, and glimmerhmm: puzzling out the features of human genes in the encode regions. *Genome Biol*, pages 1–13, Aug 2006.

[6] J. E. Allen, M. Pertea, and S. L. Salzberg. Computational gene prediction using multiple sources of evidence. *Genome Research*, pages 1–7, Dec 2003.

[7] J. E. Allen and S. L. Salzberg. Jigsaw: integration of multiple source of evidence for gene prediction. *BIOINFORMATICS*, 21(18):3596–3603, Aug 2005.

[8] D. Altman and J. Bland. Diagnostic tests. 1: Sensitivity and specificity. *BMJ: British Medical Journal*, 308(6943):1552, 1994.

[9] 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 Research*, 25(17):3389–3402, 1997.

[10] M. A. Andrade, C. Perez-Iratxeta, and C. P. Ponting. Protein repeats: structures, functions, and evolution. *Journal of structural biology*, 134(2-3):117–31, 2001.

[11] A. Appel and M. Conçalves. *Hash-consing garbage collection.* Princeton University, Department of Computer Science, 1993.

[12] J. Atkins and R. Gesteland. The 22nd amino acid. *Science(Washington)*, 296(5572):1409–1410, 2002.

[13] J. Badger and G. Olsen. Critica: coding region identification tool invoking comparative analysis. *Molecular biology and evolution*, 16(4):512–524, 1999.

[14] R. Bellman. *Dynamic Programming.* Princeton University Press, 1957.

[15] G. Benson. Tandem repeats finder: a program to analyze DNA sequences. *Nucleic acids research*, 27(2):573–80, Jan. 1999.

[16] S. Bernhart, I. Hofacker, S. Will, A. Gruber, and P. Stadler. Rnaalifold: improved consensus structure prediction for rna alignments. *BMC Bioinformatics*, 9(1):474, 2008.

[17] J. Besemer and M. Borodovsky. Genemark: web software for gene finding in prokaryotes, eukaryotes and viruses. *Nucleic Acids Res.*, 33:451–454, 2005.

[18] J. Besemer, A. Lomsadze, and M. Borodovsky. GenemarkS: a self-training method for predicition of gene starts in microbial genomes. implications for finding sequence motifs in regulatory regions. *Nucleic Acids Research*, 29:2607–2618, 2001.

[19] S. Bistarelli, T. W. Frühwirth, M. Marte, and F. Rossi. Soft constraint propagation and solving in constraint handling rules. *Computational Intelligence*, 20(2):287–307, 2004.

[20] A. Blum and T. Mitchell. Combining labeled and unlabeled data with co-training. In *Proceedings of the eleventh annual conference on Computational learning theory*, pages 92–100. ACM, 1998.

[21] R. Boyer and J. Moore. The sharing of structure in theorem-proving programs. *Machine intelligence*, 7:101–116, 1972.

[22] M. R. Brent. Steady progress and recent breakthroughs in the accuracy of automated genome annotation. *Nature Reviews Genetics*, 9(1):62–73, 2008.

[23] B. J. Brewer. When polymerases collide: replication and the transcriptional organization of the e. coli chromosome. *Cell*, 53:679–686, 1988.

[24] T. A. Brown. *Genomes.* Oxford: Wiley-Liss, second edition, 2002.

[25] C. Burge and S. Karlin. Finding the genes in genomic dna. *Current opinion in structural biology*, 8(3):346–354, 1998.

[26] M. Burset, R. Guigo, et al. Evaluation of gene structure prediction programs. *Genomics*, 34(3):353–367, 1996.

[27] B. L. Cantarel, I. Korf, S. M. Robb, G. Parra, E. Ross, B. Moore, C. Holt, A. S. Alvarado, and M. Yandell. Maker: An easy-to-use annotation pipeline designed for emerging model organism genomes. *Genome Research*, 18:188–196, Dec 2008.

[28] 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 Research*, 18(1):188–196, 2008.

[29] B. Carle, P. Narendran, and C. Scheriff. On extended regular expressions. In *Language and Automata Theory and Applications: Third International Conference, LATA 2009, Tarragona, Spain, April 2-8, 2009. Proceedings*, volume 76, page 279. Springer-Verlag New York Inc, Nov. 2009.

[30] M.-W. Chang, L.-A. Ratinov, and D. Rizzolo, N. Roth. Learning and inference with constraints. In *Proc. of AAAI Conference on Artificial Intelligence*, pages 1513–1518, Chicago, USA, July 2008.

[31] E. Charniak. Statistical techniques for natural language parsing. *AI Magazine*, 18(4):33–44, 1997.

[32] B. Chaudhuri and T. Yeates. A computational method to predict genetically encoded rare amino acids in proteins. *Genome Biology*, 6(9):R79, 2005.

[33] W. Chen and D. Warren. Tabled evaluation with delaying for general logic programs. *Journal of the ACM (JACM)*, 43(1):20–74, 1996.

[34] N. Chomsky. On certain formal properties of grammars. *icalt*, 2:137–167, 1959.

[35] H. Christiansen. Logic-statistic modeling and analysis of biological sequence data: a research agenda. In *Pre-Proceedings of the 2007 International Workshop on Abduction and Induction in Artificial Intelligence (AIAI'07)*, pages 42–49, 2007.

[36] H. Christiansen. Implementing probabilistic abductive logic programming with constraint handling rules. In T. Schrijvers and T. W. Frühwirth, editors, *Constraint Handling Rules*, volume 5388 of *Lecture Notes in Computer Science*, pages 85–118. Springer, 2008.

[37] H. Christiansen. Taming the Zoo of Discrete HMM Subspecies & Some of their Relatives; example programs and guidelines, Established March 2011. Website; `http://www.ruc.dk/∼henning/hmmzoo`.

[38] H. Christiansen and C. M. Dahmcke. A machine learning approach to Test Data Generation : A Case Study in Evaluation of Gene Finders. *Machine Learning*, pages 742–755, 2007.

[39] H. Christiansen et al. Lost on the Web. http://lost.ruc.dk.

[40] H. Christiansen and J. P. Gallagher. Non-discriminating arguments and their uses. In P. M. Hill and D. S. Warren, editors, *Logic Programming, 25th International Conference, ICLP 2009, Pasadena, CA, USA, July 14-17, 2009. Proceedings*, volume 5649 of *Lecture Notes in Computer Science*, pages 55–69. Springer, 2009.

[41] H. Christiansen, C. Have, O. Lassen, and M. Petit. Inference with constrained hidden markov models in prism. *Theory and Practice of Logic Programming*, 10(4-6):449–464, 2010.

[42] H. Christiansen, C. T. Have, O. T. Lassen, and M. Petit. A constraint model for constrained hidden markov models: a first biological application. In *Proc. of the International Workshop on Constraint Based Methods for Bioinformatics*, pages 19–26, Lisbon, Portugal, September 2009.

[43] H. Christiansen, C. T. Have, O. T. Lassen, and M. Petit. Bayesian Annotation Networks for Complex Sequence Analysis. In J. Gallagher and M. Gelfond, editors, *Technical Communications of the 27th International Conference on Logic Programming (ICLP'11)*, volume 11 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 220–230, Dagstuhl, Germany, 2011. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[44] H. Christiansen, C. T. Have, O. T. Lassen, and M. Petit. Taming the zoo of discrete HMM subspecies & some of their relatives. In *Biology, Computation and Linguistics, New Interdisciplinary Paradigms*, volume 228 of *Frontiers in Artificial Intelligence and Applications*, pages 28–42. IOS Press, 2011.

[45] W. F. Clocksin and C. S. Mellish. *Programming in Prolog: Using the ISO Standard*. Springer, 5th ed. edition, 2003.

[46] P. J. A. Cock and D. E. Whitworth. Evolution of relative reading frame bias in unidirectional prokaryotic gene overlaps. *Molecular Biology and Evolution*, 27(4):753–756, 2010.

[47] 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, Jul 2007.

[48] A. Colmerauer. Metamorphosis grammars. In L. Bolc, editor, *Natural Language Communication with Computers*. Springer-Verlag, 1978.

[49] R. L. Constable. The Role of Finite Automata in the Development of Modern Computing Theory*. In H. J. K. Jon Barwise and K. Kunen, editors, *The Kleene Symposium*, volume 101 of *Studies in Logic and the Foundations of Mathematics*, pages 61–83. Elsevier, 1980.

[50] V. Costa, D. Page, and J. Cussens. CLP(BN): Constraint logic programming for probabilistic knowledge. *Probabilistic Inductive Logic Programming*, LNAI 4911:156–188, 2008.

[51] V. Costa, R. Rocha, and L. Damas. The yap prolog system. *Theory and Practice of Logic Programming, Special Issue on Prolog Systems*, 12(1-2):5–34, 2012.

[52] F. Crick. Central dogma of molecular biology. *Nature*, 227(5258):561–563, 1970.

[53] J. Cussens. Parameter estimation in stochastic logic programs. *Machine Learning*, 44(3):245, 2001.

[54] L. De Koninck. *Execution Control for Constraint Handling Rules.* PhD thesis, K.U.Leuven, Belgium, Nov. 2008.

[55] L. De Raedt, B. Demoen, D. Fierens, B. Gutmann, G. Janssens, A. Kimmig, N. Landwehr, T. Mantadelis, W. Meert, R. Rocha, V. Santos Costa, I. Thon, and J. Vennekens. Towards digesting the alphabet-soup of statistical relational learning. In D. Roy, J. Winn, D. McAllester, V. Mansinghka, and J. Tenenbaum, editors, *Proceedings of the 1st Workshop on Probabilistic Programming: Universal Languages, Systems and Applications*, Whistler, Canada, December 2008.

[56] A. L. Delcher, K. A. Bratke, E. C. Powers, and S. L. Salzberg. Identifying bacterial genes and endosymbiont DNA with Glimmer. *Bioinformatics*, 23:673–679, 2007.

[57] A. L. Delcher, D. Harmon, S. Kasif, O. White, and S. L. Salzberg. Improved microbial gene identification with glimmer. *Nucleic Acids Research*, 27(23):4636–4641, Oct 1999.

[58] A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society*, 39:1–38, 1977.

[59] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[60] M. Dsouza, N. Larsen, and R. Overbeek. Searching for patterns in genomic data. *Trends in genetics: TIG*, 13(12):497, 1997.

[61] R. Durbin, S. Eddy, A. Krogh, and G. Mitchison. *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids.* Cambridge University Press, 1999.

[62] A. M. Durham, A. Y. Kashiwabara, F. T. G. Matsunaga, P. H. Ahagon, F. Rainone, L. Varuzza, and A. Gruber. Egene: a configurable pipeline generation system for automated sequence analysis. *Bioinformatics*, 21(12):2812–2813, 2005.

[63] A. Ershov. On programming of arithmetic operations. *Communications of the ACM*, 1(8):3–6, 1958.

[64] T. Fawcett. An introduction to ROC analysis. *Pattern Recognition Letters*, 27(8):861–874, June 2006.

[65] S. I. Feldman. Make – A program for maintaining computer programs. *Software – Practice and Experience*, 9(3):255–265, Mar. 1979.

[66] J. W. Fickett and C.-S. Tung. Assessment of protein coding measures. *Nucl. Acids Res.*, 20(24):6441–6450, 1992.

[67] S. Fine, Y. Singer, and N. Tishby. The hierarchical Hidden Markov Model: Analysis and applications. *Machine Learning*, 32(1):41–62, 1998.

[68] Finn V. Jensen and Thomas D. Nielsen. *Bayesian Networks and Decision Graphs*. Springer, 2 edition, 2007.

[69] D. Frishman, A. Mironov, H.-W. Mewes, and M. Gelfand. Combining diverse evidence for gene recognition in completely sequenced bacterial genomes. *Nucleic Acids Research*, 26(12):2941–2947, 1998.

[70] T. Frühwirth. Theory and practice of constraint handling rules, special issue on constraint logic programming. *Journal of Logic Programming*, 37(1–3):95–138, Oct. 1998.

[71] T. Frühwirth. *Constraint Handling Rules*. Cambridge University Press, 2009.

[72] T. W. Frühwirth. Constraint handling rules. In A. Podelski, editor, *Constraint Programming*, volume 910 of *Lecture Notes in Computer Science*, pages 90–107. Springer, 1994.

[73] M. Fujita, H. Mihara, S. Goto, N. Esaki, and M. Kanehisa. Mining prokaryotic genomes for unknown amino acids: a stop-codon-based approach. *BMC Bioinformatics*, 8(1):225, 2007.

[74] Y. Fukuda, Y. Nakayama, and M. Tomita. On dynamics of overlapping genes in bacterial genomes. *Gene*, 323:181 – 187, 2003.

[75] H. Ganzinger and D. A. McAllester. A new meta-complexity theorem for bottom-up logic programs. In R. Goré, A. Leitsch, and T. Nipkow, editors, *IJCAR*, volume 2083 of *Lecture Notes in Computer Science*, pages 514–528. Springer, 2001.

[76] V. Garg, R. Kumar, and S. I. Marcus. Probabilistic Language Formalism for Stochastic Discrete Event Systems. *IEEE Trans. Automatic Control*, 44:23–29, 1997.

[77] M. A. Gaston, R. Jiang, and J. A. Krzycki. Functional context, biosynthesis, and genetic encoding of pyrrolysine. *Curr. Opin. Microbiol.*, 14:342–349, Jun 2011.

[78] M. A. Gaston, L. Zhang, K. B. Green-Church, and J. A. Krzycki. The complete biosynthesis of the genetically encoded amino acid pyrrolysine from lysine. *Nature*, 471(7340):647–50, Mar. 2011.

[79] Z. Ghahramani and M. I. Jordan. Factorial Hidden Markov Models. *Machine Learning*, 29(2-3):245–273, 1997.

[80] E. Goto. Monocopy and associative algorithms in extended Lisp. Technical report, Technical Report TR-74-03, University of Tokyo, 1974.

[81] I. Grissa, G. Vergnaud, and C. Pourcel. CRISPRFinder: a web tool to identify clustered regularly interspaced short palindromic repeats. *Nucleic acids research*, 35(Web Server issue):W52–7, July 2007.

[82] I. Grissa, G. Vergnaud, and C. Pourcel. The CRISPRdb database and tools to display CRISPRs and to generate dictionaries of spacers and repeats. *BMC bioinformatics*, 8:172, Jan. 2007.

[83] R. Guigó, P. Flicek, J. F. Abril, A. Reymond, J. Lagarde, F. Denoeud, S. Antonarakis, M. Ashburner, V. B. Bajic, E. Birney, R. Castelo, E. Eyras, C. Ucla, T. R. Gingeras, J. Harrow, T. Hubbard, S. E. Lewis, and M. G. Reese. Egasp: the human encode genome annotation assessment project. *Genome Biology*, 7(Suppl 1):S2, 2006.

[84] K. Hansen, S. Zambach, and C. Have. Ontology-based retrieval of biomedical information based on microarray text corpora. In *ESSLLI 2010 Student Session*, 2010.

[85] C. Have. Logic-statistic models with constraints for biological sequence analysis. In *ICLP 2009*, pages 549–550. Springer Berlin/Heidelberg, 2009.

[86] C. Have and H. Christiansen. Modeling repeats in dna using probabilistic extended regular expressions. *Biology, Computation and Linguistics, New Interdisciplinary Paradigms. Frontiers in Artificial Intelligence and Applications.*, 228:55–70, 2011.

[87] C. Have and H. Christiansen. Efficient tabling of structured data using indexing and program transformation. *Practical Aspects of Declarative Languages*, pages 93–107, 2012.

[88] C. T. Have. Stochastic definite clause grammars. In G. Angelova, K. Bontcheva, R. Mitkov, N. Nicolov, and N. Nikolov, editors, *RANLP '09: Proceedings of International Conference, Recent Advances in Natural Language Processing*, pages 139–144, Borovets, Bulgaria, September 2009.

[89] C. T. Have. Constraints and global optimization for gene prediction overlap resolution. In A. D. Palu, A. Dovier, and A. Formisano, editors, *Proceedings of Workshop on Constraint Based Methods for Bioinformatics*, 2011.

[90] W. Hayes and M. Borodovsky. How to interpret an anonymous bacterial genome: machine learning approach to gene identification. *Genome research*, 8(11):1154–1171, 1998.

[91] I. U. Heinemann, P. O'Donoghue, C. Madinger, J. Benner, L. Randau, C. J. Noren, and D. Soll. The appearance of pyrrolysine in tRNAHis guanylyltransferase by neutral evolution. *Proc. Natl. Acad. Sci. U.S.A.*, 106(50):21103–21108, Dec 2009.

[92] F. Henderson, T. Conway, Z. Somogyi, D. Jeffery, P. Schachte, S. Taylor, C. Speirs, T. Dowd, R. Becket, M. Brown, and P. Wang. *The Mercury Language Reference Manual. Version 11.01*, 2011. Available at http://www.mercury.cs.mu.oz.au/information/documentation.html.

[93] I. Hofacker, W. Fontana, P. Stadler, L. Bonhoeffer, M. Tacker, and P. Schuster. Fast folding and comparison of rna secondary structures. *Monatshefte für Chemie/Chemical Monthly*, 125(2):167–188, 1994.

[94] I. L. Hofacker, S. H. F. Bernhart, and P. F. Stadler. Alignment of rna base pairing probability matrices. *Bioinformatics*, 20(14):2222–2227, 2004.

[95] S. Hoon, K. Ratnapu, J. Chia, B. Kumarasamy, X. Juguang, M. Clamp, A. Stabenau, S. Potter, L. Clarke, and E. Stupka. Biopipe: A flexible framework for protocol-based bioinformatics analysis. *Genome Research*, pages 1904–1915, 2003.

[96] K. L. Howe, T. Chothia, and R. Durbin. GAZE: A Generic Framework for the Integration of Gene-Prediction Data by Dynamic Programming. *Genome Research*, 12(9):1418–1427, 2002.

[97] N. Hulo, A. Bairoch, V. Bulliard, L. Cerutti, E. De Castro, P. S. Langendijk-Genevaux, M. Pagni, and C. J. A. Sigrist. The PROSITE database. *Nucleic Acids Research*, 34(Database issue):D227–D230, 2006.

[98] A. Hume. A tale of two greps. *Software Practice and Experience*, 18(11):1063–1072, 1988.

[99] D. Hyatt, G. Chen, P. LoCascio, M. Land, F. Larimer, and L. Hauser. Prodigal: prokaryotic gene recognition and translation initiation site identification. *BMC bioinformatics*, 11(1):119, 2010.

[100] M. A. Islam, C. R. Ramakrishnan, and I. V. Ramakrishnan. Parameter learning in prism programs with continuous random variables. *Theory and Practice of Logic Programming*, 12:681–700, 2012.

[101] J. Jaffar and J. Lassez. Constraint logic programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 111–119. ACM, 1987.

[102] K. Jensen, G. Stephanopoulos, and I. Rigoutsos. Biogrep: a multi-threaded pattern matcher for large pattern sets. *Trends in Genetics*, 16(6):276–277, 2002.

[103] L. J. Jensen, C. Friis, and D. W. Ussery. Three views of microbial genomes. *Research in microbiology*, 150(9-10):773–7, 1999.

[104] D. Jurafsky and J. H. Martin. *Speech and Language Processing (2nd Edition) (Prentice Hall Series in Artificial Intelligence)*. Prentice Hall, 2 edition, 2008.

[105] Y. Kameya and T. Sato. Efficient EM learning with tabulation for parameterized logic programs. In J. W. Lloyd, V. Dahl, U. Furbach, M. Kerber, K.-K. Lau, C. Palamidessi, L. M. Pereira, Y. Sagiv, and P. J. Stuckey, editors, *CL*, volume 1861 of *Lecture Notes in Computer Science*, pages 269–284. Springer, 2000.

[106] Y. Kameya and T. Sato. Efficient em learning with tabulation for parameterized logic programs. In J. W. Lloyd, V. Dahl, U. Furbach, M. Kerber, K.-K. Lau, C. Palamidessi, L. M. Pereira, Y. Sagiv, and P. J. Stuckey, editors, *Computational Logic*, volume 1861 of *Lecture Notes in Computer Science*, pages 269–284. Springer, 2000.

[107] Y. Kameya, N. Ueda, and T. Sato. A graphical method for parameter learning of symbolic-statistical models. In S. Arikawa and K. Furukawa, editors, *Proceedings of the 2nd International Conference on Discovery*

Science (DS-99), volume 1721 of *LNAI*, pages 264–276, Berlin, Dec. 6–8 1999. Springer.

[108] I. Keseler, J. Collado-Vides, S. Gama-Castro, J. Ingraham, S. Paley, I. Paulsen, M. Peralta-Gil, and P. Karp. Ecocyc: a comprehensive database resource for escherichia coli. *Nucleic acids research*, 33(suppl 1):D334–D337, 2005.

[109] S. C. Kleene. *Representation of events in nerve nets and finite automata*, volume 34, pages 3–41. Princeton University Press, 1956.

[110] R. Kolpakov. mreps: efficient and flexible detection of tandem repeats in DNA. *Nucleic Acids Research*, 31(13):3672–3678, July 2003.

[111] L. D. Koninck, T. Schrijvers, and B. Demoen. The correspondence between the logical algorithms language and CHR. In V. Dahl and I. Niemelä, editors, *ICLP*, volume 4670 of *Lecture Notes in Computer Science*, pages 209–223. Springer, 2007.

[112] L. D. Koninck, T. Schrijvers, and B. Demoen. User-definable rule priorities for chr. In M. Leuschel and A. Podelski, editors, *PPDP*, pages 25–36. ACM, 2007.

[113] P. Kontkanen and P. Myllymäki. MDL histogram density estimation. *Journal of Machine Learning Research - Proceedings Track*, 2:219–226, 2007.

[114] R. Kowalski. Algorithm = logic + control. *Commun. ACM*, 22(7):424–436, July 1979.

[115] R. Kowalski and D. Kuehner. Linear resolution with selection function. *Artificial Intelligence*, 2(3-4):227–260, 1972.

[116] A. Krogh, M. Brown, I. Mian, K. Sjolander, and D. Haussler. Hidden markov models in computational biology: Applications to protein modeling. *Journal of molecular biology*, 235(5):1501–1531, 1994.

[117] A. Krogh, I. Mian, and D. Haussler. A hidden markov model that finds genes in e. coli dna. *Nucleic Acids Research*, 22(22):4768–4778, 1994.

[118] A. Krogh, I. S. Mian, and D. Haussler. A hidden Markov model that finds genes in E.coli DNA. *Nucl. Acids Res.*, 22(22):4768–4778, 1994.

[119] G. V. Kryukov, S. Castellano, S. V. Novoselov, A. V. Lobanov, O. Zehtab, R. Guigó, and V. N. Gladyshev. Characterization of mammalian selenoproteomes. *Science (New York, N.Y.)*, 300(5624):1439–43, May 2003.

[120] J. A. Krzycki. Function of genetically encoded pyrrolysine in corrinoid-dependent methylamine methyltransferases. *Current Opinion in Chemical Biology*, 8(5):484 – 491, 2004.

[121] N. Landwehr, T. MielikŁinen, L. Eronen, H. Toivonen, and H. Mannila. Constrained hidden markov models for population-based haplotyping. *BMC Bioinformatics*, 8(S-2), 2007.

[122] J. Larrosa and T. Schiex. Solving weighted CSP by maintaining arc consistency. *Artificial Intelligence*, 159(1-2):1–26, 2004.

[123] T. Larsen and A. Krogh. Easygene - a prokaryotic gene finder that ranks orfs by statistical significance. *BMC Bioinformatics*, 4(21):21, 2003.

[124] O. T. Lassen. *Compositionality in probabilistic logic modelling for biological sequence analysis.* PhD thesis, Roskilde University, 2011.

[125] V. Lifschitz. What is answer set programming? In D. Fox and C. P. Gomes, editors, *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence, AAAI 2008, Chicago, Illinois, USA, July 13-17, 2008*, pages 1594–1597. AAAI Press, 2008.

[126] 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.

[127] D. G. Longstaff, S. K. Blight, L. Zhang, K. B. Green-Church, and J. A. Krzycki. In vivo contextual requirements for uag translation as pyrrolysine. *Molecular Microbiology*, 63(1):229–241, 2007.

[128] A. Lukashin and M. Bordovsky. GeneMark.hmm: new solutions for gene finding. *Nucleic Acids Research*, 26(4):1107–1115, February 1998.

[129] S. Mahony, J. O. McInerney, T. J. Smith, and A. Golden. Gene prediction using the self-organizing map: automatic generation of multiple gene models. *BMC Bioinformatics*, 5:23, 2004.

[130] H. J. Mangalam. tacg – a grep for DNA. *BMC Bioinformatics*, 3(1):8, 2002.

[131] C. Manning, P. Raghavan, and H. Schütze. *Introduction to information retrieval*, volume 1. Cambridge University Press Cambridge, 2008.

[132] T. M. Martin Simonsen and C. N. S. Pedersen. Rapid neighbour joining. In *Proceedings of the 8th Workshop in Algorithms in Bioinformatics (WABI)*, volume LNBI 5251, pages 113–122. Springer Verlag, 2008.

[133] C. Médigue, T. Rouxel, P. Vigier, A. Hénaut, and A. Danchin. Evidence for horizontal gene transfer in escherichia coli speciation. *Journal of molecular biology*, 222(4):851–856, 1991.

[134] M. Meister. *Advances in Constraint Handling Rules.* PhD thesis, Universität Ulm, Germany, 2008.

[135] D. Michie. Memo functions and machine learning. *Nature*, 218(5136):19–22, 1968.

[136] A. Mira, H. Ochman, and N. Moran. Deletional bias and the evolution of bacterial genomes. *Trends in Genetics*, 17(10):589–596, 2001.

[137] S. Mørk and I. Holmes. Evaluating bacterial gene-finding hmm structures as probabilistic logic programs. *Bioinformatics*, 28(5):636–642, 2012.

[138] A. M.Shmatkov, A. A.Melikyan, F. L.Chernousko, and M. Borodovsky. Finding prokaryotic genes by the frame-by-frame' algorithm: targeting gene starts and overlapping genes. *Bioinformatics*, 15(11):874–886, 1999.

[139] C. Mungall. Skam - skolem assisted makefiles. `http://skam.sourceforge.net/`.

[140] C. Mungall. Experiences using logic programming in bioinformatics. In *ICLP*, pages 1–21, 2009.

[141] K. Murphy. *Dynamic Bayesian Networks: Representation, Inference and Learning*. PhD thesis, UC Berkeley, Computer Science Division, Berkeley, USA, July 2002.

[142] K. P. Murphy. Dynamic bayesian networks. *Probabilistic graphical models*, 41(November):515–29, 2003.

[143] E. P. Nawrocki, D. L. Kolbe, and S. R. Eddy. Infernal 1.0: inference of rna alignments. *Bioinformatics*, 25(10):1335–1337, 2009.

[144] U. Neumerkel. *Garbage collection in Prolog systems (in German)*. PhD thesis, Technical University of Vienna, 1989.

[145] P. Nguyen and B. Demoen. Representation sharing for prolog. *Theory and Practice of Logic Programming*, 2012.

[146] W. S. Noble. A quick guide to organizing computational biology projects. *PLoS Comput Biol*, 5(7):e1000424, 07 2009.

[147] S. Normark, S. Bergstrom, T. Edlund, T. Grundstrom, B. Jaurin, F. P. Lindberg, and O. Olsson. Overlapping genes. *Annual Review of Genetics*, 17:499–525, 1983.

[148] H. Ochman, J. Lawrence, E. Groisman, et al. Lateral gene transfer and the nature of bacterial innovation. *Nature*, 405(6784):299–304, 2000.

[149] R. O'KEEFE. O(1) reversible tree navigation without cycles. *Theory and Practice of Logic Programming*, 1(5):617–630, 2001.

[150] H. Y. Ou, F. B. Guo, and C. T. Zhang. Gs-finder: a program to find bacterial gene start sites with a self-training method. *The International Journal of Biochemistry and Cell Biology*, 36:535–544, 2004.

[151] A. Pallejà, E. Harrington, and P. Bork. Large gene overlaps in prokaryotic genomes: result of functional constraints or mispredictions? *BMC genomics*, 9(1):335, 2008.

[152] V. Pavlović, A. Garg, and S. Kasif. A Bayesian framework for combining gene predictions. *Bioinformatics*, 18(1):19–27, 2002.

[153] F. C. N. Pereira and D. H. D. Warren. Definite clause grammars for language analysis - survey of the formalism and a comparison with augmented transition networks. *Artificial Intelligence*, 13(3):231–278, 1980.

[154] M. Petit and H. Christiansen. Viterbi computation for a constrained hidden markov model. *Actes JFPC*, 2009.

[155] R. T. Pomerantz and M. O'Donnell. The replisome uses mRNA as a primer after colliding with RNA polymerase. *Nature*, 456:762–766, 2008.

[156] M. S. Poptsova and J. P. Gogarten. Using comparative genome analysis to identify problems in annotated microbial genomes. *Microbiology*, 156(7):1909–1917, 2010.

[157] *The Single UNIX Specification, version 2. Regular Expressions.* The Open Group, 1997.

[158] S. C. Potter, L. Clarke, V. Curwen, S. Keenan, E. Mongin, S. M. J. Searle, A. Stabenau, R. Storey, and M. Clamp. The ensembl analysis pipeline. *Genome Research*, 14(5):934–941, 2004.

[159] L. R. Rabiner. A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition. In *Proceedings of the IEEE*, volume 77, 1989.

[160] J. Raimundo and R. Rocha. Global Trie for Subterms. In S. Abreu and V. S. Costa, editors, *Proceedings of the 11th Colloquium on Implementation of Constraint and LOgic Programming Systems, CICLOPS'2011*, pages 34–48, Lexington, Kentucky, USA, July 2011.

[161] I. Ramakrishnan, P. Rao, K. Sagonas, T. Swift, and D. Warren. Efficient access mechanisms for tabled logic programs. *The Journal of Logic Programming*, 38(1):31–54, 1999.

[162] F. Riguzzi and T. Swift. The pita system: Tabling and answer subsumption for reasoning under uncertainty. In *Theory and Practice of Logic Programming, International Conference on Logic Programming Special*, volume 11, pages 433–449. Cambridge Univ Press, 2011.

[163] E. P. C. Rocha and A. Danchin. Essentiality, not expressiveness, drives gene-strand bias in bacteria. *Nature genetics*, 34:377–378, 2003.

[164] R. Rocha, F. Silva, and V. S. Costa. A tabling engine for the Yap Prolog system. In *Proceedings of the 2000 APPIA-GULP-PRODE Joint Conference on Declarative Programming (AGP'00)*, La Habana, Cuba, December 2000.

[165] I. Rogozin, K. Makarova, J. Murvai, E. Czabarka, Y. Wolf, R. Tatusov, L. Szekely, and E. Koonin. Connected gene neighborhoods in prokaryotic genomes. *Nucleic Acids Research*, 30(10):2212–2223, 2002.

[166] L. Rokach. Ensemble-based classifiers. *Artificial Intelligence Review*, 33(1-2):1–39, 2009.

[167] B. J. Ross. Probabilistic pattern matching and the evolution of stochastic regular expressions. *Late Breaking Papers at the 1999 Genetic and Evolutionary Computation Conference*, 13(3):285–300, 2000.

[168] D. Roth and W. Yih. Integer linear programming inference for conditional random fields. In *Proc. of the International Conference on Machine Learning*, pages 737–744, Bonn, Germany, August 2005.

[169] S. Roweis. Constraint hidden markov models. In *Proc. of the International Conference of Advances in Neural Information Processing System*, pages 782–788, Denver, USA, Dec. 1999.

[170] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*, volume 48 of *Prentice Hall Series In Artificial Intelligence*. Prentice Hall, 2003.

[171] S. P. Sadedin, B. Pope, and A. Oshlack. Bpipe : A tool for running and managing bioinformatics pipelines. *Bioinformatics*, 2012.

[172] S. L. Salzberg, M. Pertea, A. L. Delcher, M. J. Gardner, and H. Tettelin. Interpolated markov models for eukaryotic gene finding. *Genomics*, pages 1–8, Jun 1999.

[173] T. Sato. A statistical learning method for logic programs with distribution semantics. In L. Sterling, editor, *Proceedings of the 12th International Conference on Logic Programming*, pages 715–730, Cambridge, June 13–18 1995. MIT Press.

[174] T. Sato. A glimpse of symbolic-statistical modeling by PRISM. *J. Intell. Inf. Syst*, 31(2):161–176, 2008.

[175] T. Sato. Generative Modeling by PRISM. *Proceedings of the International Conference on Logic Programming*, LNCS 5649:24–35, 2009.

[176] T. Sato. Logic-based probabilistic modeling. In H. Ono, M. Kanazawa, and R. J. G. B. de Queiroz, editors, *WoLLIC*, volume 5514 of *Lecture Notes in Computer Science*, pages 61–71. Springer, 2009.

[177] T. Sato. A general MCMC method for bayesian inference in logic-based probabilistic modeling. In T. Walsh, editor, *IJCAI*, pages 1472–1477. IJCAI/AAAI, 2011.

[178] T. Sato. PRISM, PRogramming in Statistical Modeling, Link checked September 2012. Website; `http://sato-www.cs.titech.ac.jp/prism/`.

[179] T. Sato and Y. Kameya. Prism: A language for symbolic-statistical modeling. In *15th International Joint Conference on Artificial Intelligence (IJCAI97)*, pages 1330–1339, 1997.

[180] T. Sato and Y. Kameya. PRISM: A language for symbolic-statistical modeling. In *IJCAI*, pages 1330–1339, 1997.

[181] T. Sato and Y. Kameya. A viterbi-like algorithm and EM learning for statistical abduction. In *Proceedings of UAI2000 Workshop on Fusion of Domain Knowledge with Data for Decision Support*, June 16 2000.

[182] T. Sato and Y. Kameya. A dynamic programming approach to parameter learning of generative models with failure. In *Proceedings of ICML Workshop on Statistical Relational Learning and its Connection to the Other Fields (SRL-04)*, 2004.

[183] T. Sato and Y. Kameya. Learning through failure. In L. D. Raedt, T. Dietterich, L. Getoor, and S. H. Muggleton, editors, *Probabilistic, Logical and Relational Learning - Towards a Synthesis*, number 05051 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2006. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany.

[184] T. Sato and Y. Kameya. New advances in logic-based probabilistic modeling, 2007.

[185] T. Sato and Y. Kameya. New advances in logic-based probabilistic by PRISM. In *Probabilistic Inductive Logic Programming*, LNCS, pages 118–155. Springer, 2008.

[186] T. Sato, Y. Kameya, and K. Kurihara. Variational bayes via propositionalized probability computation in prism. *Annals of Mathematics and Artificial Intelligence*, 54(1-3):135–158, Nov. 2009.

[187] T. Sato, Y. Kameya, and N.-F. Zhou. Generative modeling with failure in prism. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI2005)*, pages 847–852, 2005.

[188] T. Sato and K. Kubota. Viterbi training in prism. In *Proceedings of the ICML-12 Workshop on Statistical Relational Learning*, 2012.

[189] T. Sato and P. Meyer. Tabling for infinite probability calculation. In A. Dovier and V. S. Costa, editors, *Technical Communications of the 28th International Conference on Logic Programming (ICLP'12)*, Leibniz International Proceedings in Informatics (LIPIcs), pages 348–358, Dagstuhl, Germany, 2012. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[190] T. Sato, N.-F. Zhou, Y. Kameya, and Y. Izumi. *PRISM User's Manual (Version 2.0)*, 2010.

[191] A. Schliep, W. Rungsarityotin, A. Schönhuth, and B. Georgi. The general Hidden Markov Model library: Analyzing systems with unobservable states. In *Proceedings of the Heinz-Billing-Price*, pages 121–136, 2004.

[192] S. P. Shah, G. P. McVicker, A. K. Mackworth, S. Rogic, and B. F. F. Ouellette. Genecomber: combining outputs of gene prediction programs for improved results. *BIOINFORMATICS*, 19(10):1296–1297, Jun 2003.

[193] E. Shapiro. The family of concurrent logic programming languages. *ACM Computing Surveys*, 21(3):412, Sept. 1989.

[194] T. Shibuya and I. Rigoutsos. Dictionary-driven prokaryotic gene finding. *Nucleic acids research*, 30(12):2710–2725, 2002.

[195] M. E. Skinner, A. V. Uzilov, L. D. Stein, C. J. Mungall, and I. H. Holmes. Jbrowse: A next-generation genome browser. *Genome Research*, 19(9):1630–1638, 2009.

[196] J. Sneyers, W. Meert, J. Vennekens, Y. Kameya, and T. Sato. CHR(PRISM)-based probabilistic logic learning. *TPLP*, 10(4-6):433–447, 2010.

[197] J. Sneyers, T. Schrijvers, and B. Demoen. Dijkstra's algorithm with Fibonacci heaps: An executable description in CHR. In M. Fink, H. Tompits, and S. Woltran, editors, *WLP*, volume 1843-06-02 of *INFSYS Research Report*, pages 182–191. Technische Universität Wien, Austria, 2006.

[198] J. Sneyers, P. V. Weert, T. Schrijvers, and L. D. Koninck. As time goes by: Constraint handling rules. *TPLP*, 10(1):1–47, 2010.

[199] Z. Somogyi and K. F. Sagonas. Tabling in mercury: Design and implementation. In P. V. Hentenryck, editor, *Practical Aspects of Declarative Languages, 8th International Symposium, PADL 2006, Charleston, SC, USA, January 9-10, 2006, Proceedings*, volume 3819 of *Lecture Notes in Computer Science*, pages 150–167. Springer, 2006.

[200] R. Sorek, V. Kunin, and P. Hugenholtz. CRISPR - a widespread system that provides acquired resistance against phages in bacteria and archaea. *Nat Rev Micro*, 6(3):181–186, Mar. 2008.

[201] L. Stein, C. Mungall, S. Shu, M. Caudy, M. Mangone, A. Day, E. Nickerson, J. Stajich, T. Harris, A. Arva, et al. The generic genome browser: a building block for a model organism system database. *Genome research*, 12(10):1599–1610, 2002.

[202] A. C. Stewart, B. Osborne, and T. D. Read. Diya: a bacterial annotation pipeline for any genomics lab. *Bioinformatics*, 25:962–963, 2009.

[203] T. Swift. Design patterns for tabled logic programming. In S. Abreu and D. Seipel, editors, *INAP*, volume 6547 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2009.

[204] T. Swift and D. S. Warren. *The XSB Programmer's Manual. Version 3.3*, June 2011.

[205] T. Swift and D. S. Warren. XSB: Extending prolog with tabled logic programming. *Theory and Practice of Logic Programming, Special issue on Prolog systems*, pages 157–187, 2012.

[206] T. Swift, D. S. Warren, et al. The xsb programmerÕs manual: vols. 1 and 2, 2009.

[207] Y. K. Taisuke Sato, Neng-Fa Zhou and Y. Izumi. PRISM user's manual (version 2.0.3), 2012.

[208] H. Tamaki and T. Sato. OLD resolution with tabulation. In E. Y. Shapiro, editor, *ICLP*, volume 225 of *Lecture Notes in Computer Science*, pages 84–98. Springer, 1986.

[209] M. Tech, N. Pfeifer, B. Morgenstern, and P. Meinicke. Tico: a tool for improving predictions of prokaryotic translation initiation sites. *Bioinformatics*, 21(17):3568–3569, 2005.

[210] J. D. Thompson, T. J. Gibson, and D. G. Higgins. Multiple sequence alignment using clustalw and clustalx. *Current Protocols in Bioinformatics*, 2002.

[211] K. Thompson. Regular Expression Search Algorithm. *Communications*, 11(6):419–422, 1968.

[212] J. L. Thorne, H. Kishino, and J. Felsenstein. An evolutionary model for maximum likelihood alignment of DNA sequences. *Journal or Molecular Evolution*, pages 114–124, 1991.

[213] E. Torarinsson and S. Lindgreen. War: Webserver for aligning structural rnas. *Nucleic Acids Research*, 36(suppl 2):W79–W84, 2008.

[214] K. Ueda. Guarded horn clauses. Technical Report TR-103, ICOT, Tokyo, 1985.

[215] K. Usdin. The biological effects of simple tandem repeats: lessons from the repeat expansion diseases. *Genome research*, 18(7):1011–9, July 2008.

[216] A. van Belkum, S. Scherer, L. van Alphen, and H. Verbrugh. Short-sequence DNA repeats in prokaryotic genomes. *Microbiology and molecular biology reviews : MMBR*, 62(2):275–93, June 1998.

[217] P. Van Hentenryck, V. Saraswat, and Y. Deville. Design, implementation, and evaluation of the constraint language cc(fd). *Constraint Programming*, 910:293–316, May 1995.

[218] A. J. Viterbi. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE Transactions on Information Theory*, 13:260–269, April 1967.

[219] D. Warren and A. Center. *An abstract Prolog instruction set*, volume 309. SRI International, 1983.

[220] J. Watson and F. Crick. The structure of dna. *Cold Spring Harbor*, pages 123–131, 1953.

[221] Y. Wolf, I. Rogozin, A. Kondrashov, and E. Koonin. Genome alignment, evolution of prokaryotic genome organization, and prediction of gene function using genomic context. *Genome research*, 11(3):356–372, 2001.

[222] C. K. Wong and A. K. Chandra. Bounds for the string editing problem. *J. ACM*, 23(1):13–16, 1976.

[223] Y. Xu and J. P. Gogarten, editors. *Computational Methods for Understanding Bacterial and Archaeal Genomes*, volume 7 of *ISBN 1-86094-982-1*. Imperial College Press, 2008.

[224] T. Yada, T. Takagi, Y. Totoki, Y. Sakari, and Y. Takeda. Digit: A novel gene finding program by combining gene-finders. *Pacific Symposium on Biocomputing*, pages 375–387, Nov 2003.

[225] Y. Zhang, P. V. Baranov, J. F. Atkins, and V. N. Gladyshev. Pyrrolysine and selenocysteine use dissimilar decoding strategies. *Journal of Biological Chemistry*, 280(21):20740–20751, 2005.

[226] N. Zhou. The language features and architecture of b-prolog. *Theory and Practice of Logic Programming*, 12(1):189–218, 2012.

[227] N. Zhou and C. Have. Efficient tabling of structured data with enhanced hash-consing. *Theory and Practice of Logic Programming*, 2012.

[228] N. Zhou, Y. Kameya, and T. Sato. Mode-directed tabling for dynamic programming, machine learning, and constraint solving. In *Proceedings of the International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 213–218, 2010.

[229] N. Zhou, T. Sato, and Y. Shen. Linear tabling strategies and optimizations. *Theory and Practice of Logic programming*, 8(01):81–109, 2008.

[230] N.-F. Zhou, Y.-D. Shen, L.-Y. Yuan, and J.-H. You. Implementation of a linear tabling mechanism. In E. Pontelli and V. S. Costa, editors, *PADL*, volume 1753 of *Lecture Notes in Computer Science*, pages 109–123. Springer, 2000.

[231] Y. Zhou, Y. Liang, C. Hu, L. Wang, and X. Shi. An artificial neural network method for combining gene prediction based on equitable weights. *Neurocomputing*, 71(4-6):538–543, Jan 2008.

[232] M. Zuker. Mfold web server for nucleic acid folding and hybridization prediction. *Nucleic Acids Research*, 31(13):3406–3415, July 2003.