An ILP refinement operator for biological grammar learning

Fredouille, DC, Bryant, CH, Jayawickreme, CK, Jupe, S and Topp, S

http://dx.doi.org/10.1007/978-3-540-73847-3_24

An ILP Refinement Operator for Biological Grammar Learning

Daniel C. Fredouille[1], Christopher H. Bryant[1]*,
Channa K. Jayawickreme[2], Steven Jupe[3], Simon Topp[4]

[1] School of Computing, The Robert Gordon University, Aberdeen, UK.
[2] Discovery Research Biology, GlaxoSmithKline, Durham, USA.
[3] Department of Bioinformatics, GlaxoSmithKline, Stevenage, UK.
[4] Department of Bioinformatics, GlaxoSmithKline, Harlow, UK.

Abstract. We are interested in using Inductive Logic Programming (ILP) to infer grammars representing sets of biological sequences. We call these biological grammars. ILP systems are well suited to this task in the sense that biological grammars have been represented as logic programs using the Definite Clause Grammar or the String Variable Grammar formalisms. However, the speed at which ILP systems can generate biological grammars has been shown to be a bottleneck. This paper presents a novel refinement operator implementation, specialised to infer biological grammars with ILP techniques. This implementation is shown to significantly speed-up inference times compared to the use of the classical refinement operator: time gains larger than 5-fold were observed in $\frac{4}{5}$ of the experiments, and the maximum observed gain is over 300-fold.

1 Introduction

A significant challenge in the analysis and interpretation of biological sequence data is the discovery of patterns common to sequences sharing a given biological function. The use of such patterns is twofold: (1) they can be used to annotate sequences of unknown function, providing molecular biologists with a likely function for such sequences; (2) they can help biologists to understand how functions are realised because they represent common points between sequences of similar functions.

Patterns in the form of grammars have been used with success to model biological sequences, we call these *biological grammars*. Many formalisms have been used for this task, including String Variable Grammars (SVG) [Sea93], Patscan patterns [DLO97], Prosite patterns [FPB+02], Basic Gene Grammars [LMR01] and Probabilistic Regular or Context-Free Grammars [BCD+04,SBH+94]. However, the hand development of grammars, using for example the formalisms of [Sea93] or of [LMR01], is difficult and requires expensive human expertise. Moreover, some patterns might be too subtle to be recognised by a human expert.

* Contact author, chb@comp.rgu.ac.uk.

Thus, given the enormous volume of data arising from genome projects, the acquisition of biological grammars from sets of biological sequences needs to be automated.

We propose to use Inductive Logic Programming (ILP) to infer biological grammars. The advantage of ILP for this purpose is twofold: first ILP infers logic programs, and logic programs have been shown to be useful for representing hand designed biological grammars (*e.g.*, [Sea93]); second, unlike most machine learning technique, ILP is able to bias inference to take expert knowledge into account. This is certainly an advantage in this application domain since, as biological sequences are not just sequences but represent molecules with physical and chemical properties, potential parts of the target grammar are often available as expert knowledge.

ILP however has an important drawback: inference speed. The usual approach to obtaining a more efficient inference process is to use language and search biases. The former allows the search space to be reduced, while the latter influences its exploration [LD94, sec. 1.3]. This approach have been used to infer grammars over proteins [MBS+01], the biases being integrated into mode declarations and pruning predicates. However, despite the efforts of Muggleton et al. [MBS+01], some inference processes took days to run while exploring a small fraction of the search space. Such long running times were also confirmed by Bryant & Fredouille [BF05]. This drawback has made it difficult to discover the true potential of ILP for biological grammar acquisition.

We propose to tackle this speed problem by hard-coding the languages and search biases of Muggleton et al. [MBS+01] in Muggleton's refinement operator [Mug95] (Section 2). Compared to classical techniques influencing refinements with respect to background knowledge (*e.g.*, mode declarations, typing, ... , see [Tau94] for an early review), our technique sacrifices the range of applications to the advantage of efficiency. We empirically show that this sacrifice is worthwhile since our refinement operator can lead to very significant speeds-up of biological grammar inference: gains in inference times larger than 5-fold were obtained in $\frac{4}{5}$ of the experiments, with the maximum observed gain being over 300-fold (Section 3).

Grammars and Biological Sequences Biological sequences are defined either over an alphabet of 4 letters (DNA or RNA sequences), or over an alphabet of 20 letters (protein sequences). Each letter of such sequences represents a chemical unit which is called a nucleic acid for DNA or RNA sequences, or an amino-acid for proteins.

A *context-free grammar* can be seen as a set of rules which represents sets of sequences. For biological grammars, these sequences are biological sequences. The rules of a context-free grammar can be represented using the logic formalism known as *Definite Clause Grammar* (DCG) [PW80]. In this formalism a sequence over a finite alphabet of letters is represented as a list, each element of the list corresponding to a letter of the sequence. Figure 1 gives an example of such a grammar.

```
target(A,B):-gap(A,C),al(C,D),bl(D,E),gap(E,B).
gap(A,A).                        al([a|X],X).
gap(A,[_|B]):-gap(A,B).          bl([b|X],X).
```

Each predicate in a DCG *clause takes as input (first argument) a list representing the sequence to analyse, and outputs (second argument) the part of the list remaining when removing a prefix that the predicate matches. For this* DCG*, a call* `target(Seq,[])` *will succeed if and only if* `Seq` *contains the sublist* `[a,b]` *(i.e., the subsequence 'a b'). Indeed, the* `gap/2` *predicate matches any sequence, and the* `al/2` *(resp.* `bl/2`*) predicate matches sequences starting with 'a' (resp. 'b'). The main rule (*`target/2`*) therefore accepts all sequences starting with any sequence, followed by 'a b', followed by any sequence.*

Fig. 1. A simple grammar represented as a DCG.

We denote the length of a sequence s by $|s|$. Biological sequences are of very variable length, protein sequences' lengths range from roughly 50 to many thousands of letters. Note the contrast with natural language, where sequences rarely exceed 500 letters. This implies that it is very unlikely that any grammar of reasonable size can characterise all parts of a set of biological sequences (even if these sequences share a common biological function). From this observation, the notion of gap – as introduced by Figure 1 – is very important to biological grammars: `gap/2` is the predicate which can be used to cover parts of the biological sequences uncharacterised by the rest of the grammar rules. A typical biological grammar therefore includes some well characterised parts separated by gaps.

Inference of Biological Grammars with ILP Different approaches to grammar learning with ILP have been considered, mainly [CP99,PC01,MBS+01]. These papers differ in two main points. The first application is natural language grammars [CP99,PC01] while the application in [MBS+01] is biological grammars. Second, the representation of grammars in [CP99,PC01] uses chart parsing tables, while [MBS+01] uses DCGs.

Our work takes its roots in the approach of Muggleton et al. [MBS+01] and uses the DCG formalism. To infer a rule like the `target/2` rule of Figure 1, the idea of [MBS+01] is to provide: (1) examples under the form `target(L,[]).` where `L` is a list of letters representing a biological sequence; and (2) DCG predicates as background knowledge (in Figure 1 this corresponds to providing the `gap/2`, `al/2` and `bl/2` definitions). The inference process aim is to combine them into new DCG rules optimising the evaluation function.

In this framework, the ILP system has to be prevented from inferring logic rules which do not represent DCGs. A main constraint is that the user can only provide background predicates respecting the DCG semantic, *i.e.*, they must have two arguments, take a sequence s as first argument, and return in the second argument a suffix of s. We also have to ensure that the inferred rule is a DCG, leading to the following constraints:

c0) the head of the rule contains two variables;

c1) the first variable of the head must be unified with the first variable in the first literal of the body;
c2) the second variable of all body literals but the last must be unified with the first variable of the following literal;
c3) the second variable of the head must be unified with the second variable of the last body literal;
c4) all couples of variables unspecified by points (c1-c3) must not be unified[5].

Part of these constraints can be enforced using mode declarations of the following form [MBS+01]: `modeh(1,target(+rl,-rl))` and `modeb(n,bk_predicate(+rl,-rl))`. In these declarations, `rl` is a predicate accepting lists of letters; `target` is the predicate to infer; `bk_predicate` is a background knowledge predicate and `n` is its ambiguity (*i.e.*, the maximum number of times a backtrack on this predicate can succeed). These declarations enforce c0-c1 but only partially c2-c4[6]: Muggleton et al. [MBS+01] had to use pruning predicates to enforce the remaining parts (Subsection 2.2 gives more details on this point).

2 A Refinement Operator for Biological Grammar Inference

This section is divided in two parts. Subsection 2.1 considers the notion of bottom clause and how this notion can be simplified for biological grammars. The bottom clause defines elements of the search space. Subsection 2.2 details the proposed refinement operator to explore this space.

2.1 Bottom Clause Construction

Bottom Clause for Biological Grammar Learning The notion of bottom clause was introduced by Muggleton in [Mug95]. Such a clause, denoted by $bot_c(e)$ is constructed from a positive example e and represents the most specific logic program, defined using the background knowledge and respecting the mode declarations, that covers e. The CProgol algorithm [Mug95] works by taking an example e_1 and constructing its bottom clause $bot_c(e_1)$. It then searches through the sets of clauses θ-subsuming $bot_c(e_1)$ to return the best one with respect to the evaluation function. Further clauses are inferred using the same strategy iteratively, but starting from the set of yet uncovered examples (uncovered by any of the already inferred clauses); this is the principle of the *cover set* algorithm.

An example of bottom clause for biological grammar learning is given in Figure 2. In this example the background knowledge predicates are limited (for the sake of explanation) to two physical properties of the letters (representing

[5] For example, a rule `target(A,B):-foo1(A,A),foo2(A,B).` is not a DCG rule but respects (c0-c3).

[6] For example, a rule `target(A,B):-foo1(A,C),foo2(A,C),gap(C,B).` respects the mode declarations but violates c2.

```
target(A0,A4) :-   gap(A0,A0), gap(A0,A1), ... , gap(A4,A4),
                   neg(A0,A1), neg(A3,A4), small(A0,A1), small(A1,A2).
```

Fig. 2. Bottom clause for a (fake) protein sequence `[d,p,i,e]` using background knowledge from three predicates: the `gap/2` predicate and two predicates related to physical properties of amino-acids (`neg/2` and `small/2`).

amino-acids). The `neg/2` (resp. `small/2`) predicate corresponds to negatively charged (resp. small) amino-acids. They can be defined by the sets of rules: $\{$`neg([`α`|X],X).` : $\alpha \in \{$`d,e`$\}\}$ and $\{$`small([`α`|X],X).` : $\alpha \in \{$`a,c,d,g,n,p,s,t`$\}\}$. Then there is the `gap/2` predicate which, as explained in Section 1, can be considered to be compulsory for biological grammar inference processes and allows parts of the sequences uncharacterised by the inferred grammar to be covered.

Calls to DCG predicates which succeed match a prefix of the input sequence and return the remaining suffix. Therefore, each variable of the bottom clause $bot_c(e)$ corresponds to a suffix of the sequence e. We emphasise this fact in Figure 2 by using notation $\mathtt{A}_i$ for the variable corresponding to the suffix of e starting at position i in e (positions are between letters of e, and 0 is the position before the first letter). Therefore, on Figure 2, the presence of a predicate `foo(`$\mathtt{A}_i$`,`$\mathtt{A}_j$`)` means that `foo/2` matches the subsequence of e between positions i and j. This particularity of bottom clauses constructed over DCGs enables the use of a simplified representation: we call it the *bottom automaton*.

From Bottom Clause to Bottom Automaton In the bottom automaton, denoted by $bot_a(e)$, positions are represented by states and transitions represent background predicates: a transition `foo` between states i and j meaning that predicate `foo/2` matches sequence e between positions i and j. Since they violate (c4), transitions such that $i = j$ are ignored (*i.e.*, transitions corresponding to predicates of the form `foo(X,X)` in the bottom clause). A bottom automaton is represented in Figure 3. On this figure, unlabelled transitions are those for the `gap/2` predicate. The initial (resp. final) state of this automaton corresponds to position 0 (resp. $|e|$) of e, and is represented with a short incoming (resp. outgoing) arrow.

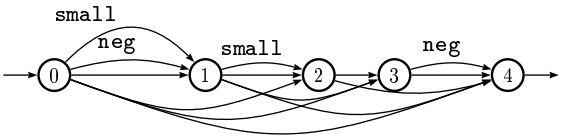

Fig. 3. The bottom automaton $bot_a(e)$ equivalent to the bottom clause $bot_c(e)$ of Figure 2.

By definition, the bottom automaton and the bottom clause are two equivalent representations of the same concept. The important things to see is that all DCG θ-subsuming the bottom clause and respecting constraints (c0) to (c4) correspond to a path from the initial state 0 to the final state $|e|$ in `bot_a(e)`. For example, the path: $0 \rightarrow$ `gap` $\rightarrow 1 \rightarrow$ `small` $\rightarrow 2 \rightarrow$ `gap` $\rightarrow 4$, which accepts sequence `gap small gap`, corresponds to rule:
`target(A,B):-gap(A,C),small(C,D),gap(D,B).` In addition, a path not ending in state $|e|$ corresponds to a rule respecting all constraints but (c3). (This fact will be useful later on.) For example, the path: $0 \rightarrow$ `neg` $\rightarrow 1 \rightarrow$ `small` $\rightarrow 2$ corresponds to rule: `target(A,_):-neg(A,B),small(B,_).`

Creating the Bottom Automaton Algorithm 1 shows how to create the bottom automaton. A similar procedure could be used to create a bottom clause specialised to biological grammar learning, but the formalism of the bottom automaton turned out to be easier to use. Optimisations of this algorithms use properties (p1) and (p2), linked to the `gap/2` predicate:

p1) All states of $bot_a(e)$ can be reached from state 0 (equivalently, $bot_c(e)$ contains $|e|+1$ different variables)[7].

p2) We have $\frac{(|e|)(|e|+1)}{2}$ transitions by symbol `gap` in the bottom automaton (equivalently, $\frac{(|e|+1)(|e|+2)}{2}$ `gap/2` predicates in the bottom clause, this number is larger than the number of transitions in the bottom automaton to count predicates of the form `foo(X,X)`).

Property (p1) allows us not to compute the set of positions of e that can be reached by the use of the background knowledge: we know that all positions are reached (hence the loop on line 2 of Algorithm 1). Property (p2) means that very large bottom clauses are considered during inference (*e.g.*, if $|e| = 200$, the bottom clause is of minimal size 20301). We can circumvent this problem; since it is known that `gap` transitions are present between each couple of states of the automaton, it is easier not to store them (line 3, condition `pred`$\neq$`gap`): instead, a particular treatment in the refinement operator can be used to introduce gaps when needed. The advantage is twofold: gain in memory, since this prevents the bottom clause size being quadratic in the sequence length; gain in execution time, since the bottom automaton can then be constructed faster.

The size of the resulting automaton is in $O(|e| \times |BK| \times max(ma))$, where $|BK|$ is the number of background knowledge predicates and $max(ma)$ is the maximum ambiguity encountered for a predicate different from `gap/2`. As $max(ma)$ can be considered much smaller than the sequence length[8], the automaton size can be considered linear in this length (compare with the $O(|e|^2)$ number of elements in a bottom clause storing gaps). The time complexity of Algorithm 1 is in $O(|e| \times |BK| \times (K_1 + max(ma) \times K_2))$, where K_1 is the cost of obtaining

[7] This is because the `gap/2` predicate can return all suffixes of its input sequence.

[8] In practice, only the gap predicate is so ambiguous that it can match in between all positions of the example sequence.

Algorithm 1 Construction of the bottom automaton without the `gap` transitions over a sequence e.

1: Create $|e|+1$ states, labelled 0 to $|e|$.
2: **for** i in $[0, |e|]$ **do**
3: **for all** background knowledge predicate `pred` (`pred`$\neq$`gap`) **do**
4: **let** ma **be** the ambiguity degree of `pred` as stated by mode declarations
5: **let** S **be** the list of $\texttt{A}_j$ obtained by backtracking up to ma times on $\texttt{pred(A}_i\texttt{,A}_j\texttt{)}$
6: **for all** $\texttt{A}_j$ in S **do**
7: Add a transition between states i and j labelled by `pred`

the list S (which depends on the background predicates implementation), and where K_2 is the cost of inserting a transition in the automaton (line 7); this cost is, in our implementation, in $O(log(|BK|))$.

2.2 Refinement Operator

Using the bottom automaton, we propose a refinement operator adapted to biological grammar learning. Our operator can be seen as a specialisation of the classical ILP refinement operator introduced by Muggleton [Mug95]. This is the same specialisation that Muggleton et al. [MBS+01] achieved using pruning predicates. We therefore start by describing [MBS+01] pruning, and then explain how we integrate this pruning into the refinement operator.

Removing Non DCG Rules using Pruning Muggleton et al. [MBS+01] pruned all rules not respecting constraints (c2) or (c4). Rules with two following gaps in the body were also pruned since two following gaps are equivalent to a single gap. Rules violating (c0) and (c1) do not need to be pruned: they are not present in the space thanks to mode declarations. Finally, rules only violating (c3) were not pruned: they were refined to enable all DCG rules of the search space to be reached. The rules returned by inference processes do however respect (c3): indeed, the mode declarations allow them to be present in the search space but not to be returned by the inference process.

In practice this corresponds to refining rules of the form:

$\texttt{target(A,\_) :- foo}_0\texttt{(A,B), ... , foo}_m\texttt{(X,\_).}$ (r0)

into rules of the following forms:

$\texttt{target(A,\_) :- foo}_0\texttt{(A,B), ... , foo}_m\texttt{(X,Y), foo}_{m+1}\texttt{(Y,\_).}$ (r1)
$\texttt{target(A,Z) :- foo}_0\texttt{(A,B), ... , foo}_m\texttt{(X,Y), foo}_{m+1}\texttt{(Y,Z).}$ (r2)

Rules (r0) and (r1) violates (c3) while rule (r2) is an inferable DCG rule. Using this strategy, the search space was reduced to a tree containing all DCG rules θ-subsuming $bot_c(e)$ (*i.e.*, the operator is optimal for the DCG rules), the rules respecting (c0) to (c4) being the leaves of the tree and internal nodes being rules violating only (c3). Such a tree, corresponding to the bottom clause of Figure 2, is given in Figure 4.

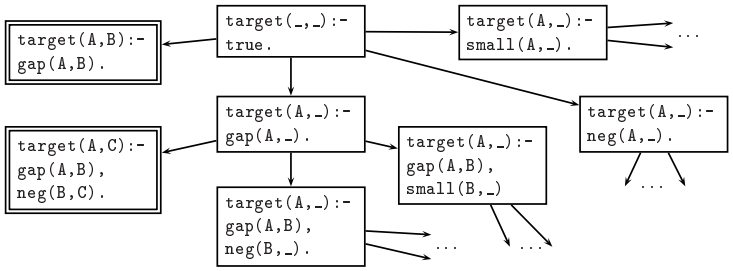

Fig. 4. First levels of the search space associated with the pruning proposed by [MBS+01] and the bottom clause of Figure 2. Refinements are represented by arrows. Double boxes correspond to inferable rules, while single boxes are internal nodes of the search space.

The Proposed Refinement Operator By integrating the constraints in the refinement operator, we generate only rules that are correct for the application, instead of generating rules that need to be pruned. Our refinement operator is described by Algorithm 2.

Algorithm 2 Refinement operator for grammar inference in the space defined by $bot_a(e)$.

let `tgt(A,_):- foo`$_0$`(A,B), ... ,foo`$_m$`(X,_)` **be** the rule to refine
Compute the set of reachable states in $bot_a(e)$ by `foo`$_0$`, ... ,foo`$_m$
$M \leftarrow \{0\}$ (The marked states, starting with the initial state)
for $v \in [0, m]$ **do**
  if `foo`$_v$`=gap` **then** $M \leftarrow \{i \in \mathbb{N} : min(M) < i \leq |e|\}$
  else $M' \leftarrow \emptyset$
    for transitions $i \xrightarrow{\texttt{foo}_v} j$ in $bot_a(e)$ with $i \in M$ **do** $M' \leftarrow M' \cup \{j\}$
    $M \leftarrow M'$
Compute predicates that can be added at the end of the refined rule
if `foo`$_m$ $=$ `gap` **then** $P_2 \leftarrow P_1 \leftarrow \emptyset$ **else** $P_1 \leftarrow P_2 \leftarrow \{$`gap`$\}$
for $i \in M$ **do**
  for transitions $i \xrightarrow{\texttt{pred}} j$ in $bot_a(e)$ **do**
    $P_1 \leftarrow P_1 \cup \{$`pred`$\}$
    if $j = |e|$ **then** $P_2 \leftarrow P_2 \cup \{$`pred`$\}$
Compute the set of possible refinements
$R \leftarrow \emptyset$ (The set of refinements)
for `pred` $\in P_1$ **do** $R \leftarrow R \cup \{$`tgt(A,_):-foo`$_0$`(A,B), ... ,foo`$_m$`(X,Y),pred(Y,_)`$\}$
for `pred` $\in P_2$ **do** $R \leftarrow R \cup \{$`tgt(A,Z):-foo`$_0$`(A,B), ... ,foo`$_m$`(X,Y),pred(Y,Z)`$\}$

Consider rule (r0), to refine it the algorithm has to add a predicate at its end, but also has to ensure that the obtained rule is in the search space defined by $bot_a(e)$. As legal predicates correspond to paths in the bottom automaton, the problem can be reduced to path searching. Refinements into rules of the (r1)

form have to correspond to paths starting from state 0, but which can end in any state i of the automata. This means that the rule can cover the first i letters of e and that further refinement is needed to cover the remaining letters. For rules of the form (r2) the paths have, in addition, to end in state $|e|$: this ensures that the rule is able to cover e.

Therefore, the first step of Algorithm 2 is to find all states of the bottom automaton that can be reached, starting from state 0, by the sequence of predicates $\texttt{foo}_0 \ \ldots \ \texttt{foo}_m$. This can be done using dynamic programming to mark the states that are reachable in the bottom automaton by following transition $\texttt{foo}_0$ from state 0, then $\texttt{foo}_1$ from the marked states, ..., up to $\texttt{foo}_m$. This work is done by lines 2-8 of Algorithm 2 (among these, line 5 takes into account that gap rules are not stored in the bottom automaton: gaps match any sequence so, when it is encountered, the updated marked states are all positions larger than the current minimal marked state). We can then deduce, from the set of marked states, the possible predicates to use for the (r1) refinements: *i.e.*, all predicates present on outgoing transitions of the marked states (lines 9-14 of Algorithm 2); and for (r2) refinements, *i.e.*, those that also enable to reach state $|e|$. Finally, we construct the set of refined rules from the original rule and these sets of predicates (lines 15-18). Algorithm 2 avoids returning redundant rules by working with sets instead of lists, and prevents rules which contain 2 consecutive gaps, which is meaningless (condition line 10).

3 Experimental Evaluation

In this section we report our empirical investigation of the time gain obtained by using our refinement operator.

3.1 Experimental Method

The experiments concern inference of grammars over protein sequences. We consider two aspects of the problem: inference of a grammar from positive and random examples as proposed by Muggleton [Mug97], and inference from positive and negative examples. The implementation of the bottom automaton algorithm and the refinement operator, as well as the public part of the datasets, can be found at `http://www.comp.rgu.ac.uk/staff/chb/research/data_sets/ilp06/refine_op`.

Positive and Random Dataset The dataset for positive only learning was provided by experiments of Muggleton et al. [MBS+01]. Among the different experiments reported in [MBS+01] we selected the one that took the longest to complete because in this case the efficiency of the grammar acquisition was a bottleneck. This experiment involved inferring on subsequences, called *middle*, of Neuropeptide Precursors Proteins (NPPs). The examples comprise 76 positive and 2 910 random *middle* sequences. The length of these sequences vary from 5 letters to 95. We denote this dataset by PosRand. This dataset is in the public domain.

Positive and Negative Dataset The data set for discriminative learning consists of two sets of sequences representing two qualitatively distinct classes, Gi/o and Gs/q, of a protein family known as the G-protein coupled receptors (GPCRs) [PPL02]. Data allowing the classification of these proteins into the two sets is proprietary. The Gi/o and Gs/q datasets contain 43 and 94 sequences respectively. GPCRs have a characteristic 7 membrane-spanning regions and thus have regions outside the cell, within the cell membrane and inside the cell.

In this paper we present results for one of the parts inside the cell, called intracellular loop #2, and for this inference process, the Gs/q sequences were used as positive examples while the Gi/o sequences were used as negative examples. The lengths of these sequences vary from 12 to 46 letters. We denote this dataset by PosNeg.

Inference Processes and Parameters All experiments have been running on a SunBlade2500 under SunOS 5.8. We used the Aleph [Sri93] implementation of Muggleton's refinement operator [Mug95] to test our ideas, instead of the original CProgol implementation. This choice was made because Aleph is much easier to modify than CProgol: it gives the user a large number of options including defining a user refinement operator and preventing the default bottom clause construction. We denote inference with Aleph, using Muggleton's operator [Mug95], by ref-ℵ, and inference with Aleph, using our biological grammar dedicated bottom clause construction and refinement operator, by ref-g.

The principle of the experiments is to explore the search space up to a given depth with both systems and observe execution times, knowing that the inferred rules from both systems are the same (see Appendix A) because the explored search spaces are the same. We considered maximum exploration depths in the search space (corresponding to Aleph parameter `clauselength`) of 4, 5 and 6 for the PosRand dataset, and of 5 and 6 for the PosNeg dataset. Inference with `clauselength` less than 5 on PosNeg is not interesting because, for biological reasons on this dataset, rules are required to start and end with the gap predicate: the head and the two gaps already count for 3, so a value of 4 corresponds to using background knowledge rules one at a time, hence the starting value of 5.

The evaluation function used for PosRand dataset was Muggleton's evaluation function for positive only learning [Mug97]. A different evaluation function was needed for the PosNeg dataset because it does not contain randoms. The Gi/o and Gs/q subsets of the PosNeg dataset contain a very different number of sequences while having the same importance to the biologists. Therefore, to avoid biasing the inference toward one class, we decided to use an evaluation function which weights the examples of each class by the inverse of the number of instances of the class available. The evaluation function used is the accuracy over the weighted examples, *i.e.*, $\text{acc} = \frac{1}{2} * (\frac{p}{P} + \frac{n}{N})$, where P (resp. N) is the size of the positive (resp. negative) training set size, and p (resp. n) is the number of positive (resp. negative) training examples covered (resp. rejected) by the rule.

After preliminary experiments, it became clear that inference times were strongly influenced by the `minacc` setting of Aleph. This parameter is a thresh-

Table 1. Inference times (seconds) on the PosRand and PosNeg datasets. (*): Experiments where the `nodes` limit of Aleph was exceeded. (+): Experiments stopped after running more than the indicated time.

		PosRand			PosNeg	
	`min`	`clauselength`			`clauselength`	
Algo	`acc`	4	5	6	5	6
REF-ℵ	0.1	1 213	*17 530	*+324 000	956	* 69 329
REF-G		151	1 897	38 899	1 087	52 299
Gain		8.0	9.24	>8.3	0.9	1.3
REF-ℵ	0.5	1 311	*+334 000	*+413 280	834	* 60 827
REF-G		131	1 944	55 878	265	12 098
Gain		10.0	>171.8	>7.4	3.1	5.0
REF-ℵ	0.9	1 802	*+511 220	*+511 200	1232	* 120 030
REF-G		156	1 601	63 253	186	5 541
Gain		11.6	> 319.0	>8.1	6.6	21.7

old on the minimal precision of inferable rules with respect to the training examples[9]. We therefore considered inferences with different `minacc` values (0.1, 0.5 and 0.9) to obtain an idea of the gain in different inference situations.

3.2 Results of the Experiments

The running times are listed in Table 1. Very different speed-ups were obtained depending on the data, the `minacc` and the `clauselength` parameters. Speed-up factors over 5 were obtained in $\frac{4}{5}$ of the experiments, the best speed-up obtained being over 300-fold (PosRand, `minacc`= 0.9, `clauselength`= 5). It is possible that even greater speed-ups can be achieved since many experiments on PosRand using the default refinement operator had to be stopped after running more than 90h, while the corresponding experiments using the hand-made refinement finished by themselves (always in less than a day). Moreover, many REF-ℵ experiments reached the limit on the maximum number of nodes to be explored by the algorithm on at least some of the algorithm cycles (the `nodes` parameter of Aleph was set to 500 000). Inference with REF-G never reached this limit. This implies two things: (1) the potential gains, given an unlimited value for `nodes` are larger than those shown in Table 1 (cells with the * symbol); (2) given that the search is exhaustive, the results of the REF-G experiments guarantee that the obtained clauses are the best possible up to the given `clauselength`.

The inference times of Table 1 clearly show the advantages of our refinement operator, both in the PosRand and the PosNeg experiments. In practice only one inference process was slower when using our refinement operator (Table 1, PosNeg, `minacc`= 0.1, `clauselength`= 5). This possible loss is however small

[9] Precision is defined here as the number of accepted positives over the sum of the number of accepted positives and accepted negatives.

compared to the potential gains observed for all other parameters. The best gains were obtained with larger values for the `minacc` setting. This is good news because maximising the precision is usually desirable.

3.3 Interpretation of Experiments

Even if the results obtained are very satisfying, being able to explain their variation can help us understand how to improve them. A potential explanation of this variation is that ALEPH is using different optimisations to speed-up the search. We discuss how two of these optimisations can explain the variation of the results of Table 1.

The gain of 0.9 observed with PosNeg, `clauselength`=5 and `minacc`=0.1 We will refer to the first of these optimisations as optimisation A. Optimisation A, which is only available to REF-$\aleph$, is that ALEPH knows that its default refinement operator is working by specialising rules. It can use this information to prevent, when evaluating the performance of a rule, the parsing of examples which were rejected by its father rule. This optimisation does not take place if the user provides his own refinement since ALEPH does not know if the refinement operator works by specialisation or generalisation (or both). The gain of 0.9 for POSNEG with `minacc`=0.1 and `clauselength`=5 could be the extra parsing time needed outweighing the optimisations brought by the refinement operator.

The Improvement in Gain when Augmenting `minacc` Another optimisation of ALEPH, Optimisation B, uses the `minacc` value to prevent the parsing of some examples when the evaluated clauses are at maximum depth. This is done using the formulae $\texttt{minacc} = \frac{p}{p+n}$ (where p and n are respectively the number of positives and negatives/randoms accepted by the rule), ALEPH computes the maximum number of negatives/randoms that the rule can accept once the number of positives covered is known (*i.e.*, $\frac{(1-\texttt{minacc})p}{\texttt{minacc}}$). It then stops parsing if this number is reached and if the clause is of size `clauselength` (smaller rules have to be evaluated as they could be refined). This optimisation is available both for REF-$\aleph$ and REF-G.

Conjecture To summarize, thanks to optimisation B, the higher the `minacc` value, the smaller is the number of examples to parse. Now, there is an effect of optimisation B on optimisation A: when `minacc` is high, optimisation B is very efficient and optimisation A cannot reduce the number of examples to parse much more. We therefore make the following conjecture to explain the increasing gains with respect to `minacc`: when increasing `minacc`, optimisation A has less and less effect, and the gain offered by REF-G over REF-$\aleph$ is more and more visible. This suggests that:

- the true gain of our optimisation is closer to that observed when `minacc`=0.9;

– both higher and more stable gains with respect to `minacc` could be obtained by making Optimisation A available for inference with REF-G.

4 The Quality of the Resulting Grammars

This paper has focused on the speed at which ILP systems can generate biological grammars. Elsewhere [BFW+06] we have published results concerning the quality of the resulting grammars. We have applied our refinement operator implementation to a hard protein function inference task: the prediction of the coupling preference of GPCR proteins [BFW+06]. The time needed to execute the experiments reported in [BFW+06] was approximately two months. It would have taken much longer to obtain the same results if we had used the default refinement operator (REF-ℵ). We estimate that it would have taken 10 months given that Table 1 suggests a five fold time gain for similar tasks (POSNEG, `minacc`=0.5).

While this does illustrate why our refinement operator implementation (REF-G) is important for hard protein function inference tasks, further work is need to establish whether, given the same amount of run-time, REF-G results in significant improvements in the quality of the resulting grammars in comparison to REF-ℵ.

5 Other Applications of the Bottom Automaton Formalism

This paper has shown how our bottom automaton formalism can be used to implement one particular refinement operator, namely the one introduced by [Mug95]. However it could be used to implement other refinement operators; the formalism itself does not place constraints on the exploration strategy.

Moreover, we believe that the strategy used to create the bottom automaton could be usefully reused for problems involving complex examples (e.g., trees, graphs,...) which involve trying to find rules matching substructures of those examples (e.g., subgraphs or subtrees). Indeed simplifications similar to those proposed in Algorithms 1 and 2 (i.e. those linked to the gap/2 predicate) could also be considered in such problems.

Conclusion

We have integrated the biases of biological grammar inference into a dedicated ILP refinement operator. We have shown that, by using this operator, inference running times can decrease very significantly compared to the previously used technique using pruning predicates: time gains larger than 5-fold where obtained in most experiments, and the best observed gain is over 300-fold.

Acknowledgements This work is funded by the UK EPSRC grant GR/S68682. The authors would like to thank Ashwin Srinivasan of IBM, India for his very informative insights about the ALEPH ILP system, and for his fast updates of the system. The authors would like to acknowledge the contributions made by the Systems Research, Transgenics & Gene Cloning, and Gene Expression & Protein Biochemistry groups at GlaxoSmithKline to the proprietary GPCR classification data used in this work.

References

[BCD+04] A. Bateman, L. Coin, R. Durbin, R. D. Finn, V. Hollich, S. Griffiths-Jones, A. Khanna, M. Marshall, S. Moxon, E. L. L. Sonnhammer, D. J. Studholme, C. Yeats, and S. R. Eddy. The Pfam protein families database. *Nucleic Acids Research*, 32:D138–D141, 2004.

[BF05] C.H. Bryant and D. Fredouille. A parser for the efficient induction of biological grammars. In S. Kramer and B. Pfahringer, editors, *15^{th} International Conference on ILP: late-breaking paper track*. University of Bonn, 2005. `http://wwwbib.informatik.tu-muenchen.de/infberichte/2005/TUM-I0510.idx`.

[BFW+06] C.H. Bryant, D. Fredouille, A. Wilson, Channa K. Jayawickreme, S. Jupe, and S. Topp. Pertinent background knowledge for learning protein grammars. In J. Furnkranz, T. Scheffer, and M. Spiliopoulou, editors, *Proceedings of the 17^{th} European Conference on Machine Learning*, number 4212 in Lecture Notes in Artificial Intelligence, pages 54–65. Springer-Verlag, Berlin, 2006.

[CP99] J. Cussens and S. Pulman. Experiments in inductive chart parsing. In James Cussens, editor, *LLL'99*, pages 72–83, Bled, Slovenia, June 1999.

[DLO97] M. Dsouza, N. Larsen, and R. Overbeek. Searching for patterns in genomic data. *Trends in Genetics*, 13(12):497–498, 1997.

[FPB+02] L. Falquet, M. Pagni, P. Bucher, N. Hulo, C. J. Sigrist, K. Hofmann, and A. Bairoch. Protein data bank. *Nucleic Acid Research*, 30:235–238, 2002.

[LD94] N. Lavrač and S. Džeroski. *Inductive Logic Programming: Techniques and Applications*. Ellis Hortwood, New York, 1994.

[LMR01] S.-W. Leung, C. Mellish, and D. Robertson. Basic Gene Grammars and DNA-ChartParser for language processing of *Escherichia coli* promoter DNA sequences. *Bioinformatics*, 17(3):226–236, 2001.

[MBS+01] S. H. Muggleton, C. H. Bryant, A. Srinivasan, A. Whittaker, S. Topp, and C. Rawlings. Are grammatical representations useful for learning from biological sequence data? – a case study. *Journal of Computational Biology*, 5(8):493–522, 2001.

[Mug95] S. H. Muggleton. Inverse entailment and Progol. *New Generation Computing, Special issue on Inductive Logic Programming*, 13(3-4):245–286, 1995.

[Mug97] S. H. Muggleton. Learning from positive data. In *Proceedings of the Sixth International Workshop on Inductive Logic Programming*, volume 1314 of *Lecture Notes in Computer Science*, pages 358–376. Springer-Verlag, 1997.

[PC01] S. Pulman and J. Cussens. Grammar learning using inductive logic programming. In *Oxford University Working Papers in Linguistics, Philology and Phonetics*, volume 6, pages 31–45. Oxford University, 2001.

[PPL02] K.L. Pierce, R.T. Premont, and R.J. Lefkowitz. Seven-transmembrane receptors. *Nat Rev Mol Cell Biol*, 3(9)(6):39–50, 2002.

[PW80] F. Pereira and D. H. D. Warren. Definite clause grammars for language analysis – a survey of the formalism and a comparison with augmented transition networks. *Artificial Intelligence*, 13(3):231–278, May 1980.

[SBH+94] Y. Sakakibara, M. Brown, R. Hughey, I. Saira Mian, K. Sjölander, R. C. Underwood, and D. Haussler. Stochastic context-free grammars for tRNA modeling. *Nucleic Acids Research*, 22:5112–5120, 1994.

[Sea93] D. B. Searls. String variable grammar: A logic grammar formalism for the biological language of DNA. *Journal of logic Programming*, 12, 1993.

[Sri93] A. Srinivasan. A Learning Engine for Proposing Hypotheses (Aleph). `http://web.comlab.ox.ac.uk/oucl/research/areas/machlearn/Aleph`, 1993.

[Tau94] B. Tausend. Representing biases for inductive logic programming. In F. Bergadano and L. De Raedt, editors, *ECML-94, European Conference on Machine Learning*, volume 784 of *LNCS*, pages 427–430. Springer, 1994.

A Appendix: the Equality of Rules Inferred by REF-G and REF-ℵ

In the REF-G implementation, ALEPH is prevented from constructing the bottom clauses, therefore the inferred rules are *not* checked for consistency with the mode declarations. This means that rules violating (c3) (*i.e.*, of the form `target(A,_) :- foo`$_0$`(A`$_0$`,A`$_1$`), ... ,foo`$_m$`(A`$_m$`,_)`) can be inferred when using REF-G but not when using REF-ℵ. For the sake of comparison between REF-G and REF-ℵ, we added in REF-G an ALEPH `false/0` predicate rejecting clauses violating (c3). (Like the `prune/1` predicate, the `false/0` predicate can be used to prevent the inference of some rules; however, unlike rules rejected by `prune/1`, rules rejected by `false/0` are refined.) After this modification of the REF-G code, the rules inferred by both systems were the same.

When using the REF-G operator in other frameworks, adding a `false/0` predicate is not needed. Indeed, since `gap/2` predicates are allowed in rules, we can systematically transform a rule violating (c3), *i.e.*, of the form: `target(A,Z) :- foo`$_0$`(A,B), ... ,foo`$_m$`(X,Y)`, where Z and Y are free variables, into a rule: `target(A,Z) :- foo`$_0$`(A,B), ... ,foo`$_m$`(X,Y),gap(Y,Z)`.
These two rules cover the same examples. Therefore, without adding a `false/0` predicate, the algorithm explores for free, for all rules ending with a gap, one step deeper in the search space. The only drawback is that a small syntactic correction must be applied to the inferred rules.