A parser for the efficient induction of biological grammars

Bryant, CH and Fredouille, DC

A Parser for the Efficient Induction of Biological Grammars

Christopher H. Bryant* and Daniel Fredouille

The Robert Gordon University, Aberdeen, UK
Email: {df|chb}@comp.rgu.ac.uk
Phone: +44(0)1224 262747 Fax: +44 (0)1224 262727

Abstract. We are interested in using ILP to infer context-free grammars (CFGs) representing sets of biological sequences. ILP systems are well suited to this task in the sense that CFGs are often represented as logic programs using the Definite Clause Grammar formalism. The speed at which ILP systems can generate biological grammars has been shown to be a bottleneck. When learning a biological grammar, training sequences must be parsed repeatedly during the search and thus the ILP system speed is dependent upon the efficiency of the parser. We tackle this speed problem by providing ILP systems written in PROLOG with a parser for CFGs. We show empirically that this reduces the overall execution time by an amount larger than 45% of the time that ILP systems would normally spend parsing.
Keywords: Context-Free Grammar inference (CFG), Biological sequences.

1 Introduction

We are investigating how ILP systems can be improved for the task of acquiring Context-Free Grammars (CFG) from sets of biological sequences (DNA, RNA and proteins). We call such grammars *biological grammars*. Grammars have been used with success to model biological sequences (e.g. with String Variable Grammars [1]). However, the hand development of grammars is difficult and requires expensive human expertise. Thus, given the enormous volume of data arising from genome projects, there is a need to automate the acquisition of biological grammars from sets of biological sequences. ILP is a natural choice for this task since grammars are often represented by logic programs using the Definite Clause Grammar (DCG) formalism [2], and ILP systems infer logic programs. In [3], CPROGOL was used to infer biological grammars. In these experiments, although the visited portion of the search space was quite restricted, the inference took time of the order of days to complete. Such long training times imply that only the exploration of a very limited number of research avenues can be tried. This paper attempts to alleviate this bottleneck by taking a first step towards a more efficient method of acquiring biological grammars.

Rather than design yet another ILP system to meet the particular requirements of this application, we have developed a fast Context-Free Grammar

* Contact author.

(CFG) parser which can be used by standard ILP systems. During learning, this parser, rather than the PROLOG deduction procedure, is used whenever examples are parsed by inferred grammars. Consequently, this speeds-up the computation of the coverage of the examples, and hence the inference process.

We explain in Section 2 how we introduced the parser in a PROLOG engine: this is a necessary step for its use by a wide range of ILP systems. Section 3 then describes the incorporation of the parser into an ILP system. Datasets, programs and documentation for the experiments are available at `http://www.comp.rgu.ac.uk/staff/chb/research/data_sets/ilp05`.

2 Introducing the parser in a Prolog engine

Method Our motivation for introducing a CFG parser in a PROLOG interpreter (instead of directly in a given ILP system) is that the predicates associated to parsing can then be used by any ILP system implemented in PROLOG.

Many algorithms have been designed to parse sequences in CFGs. However, all but two of these algorithms are not adapted to either parsing of biological grammars or parsing during an inference process. The two exceptions are *depth-first top-down* parsing, in short DFTD, and *chart parsing*. We chose to integrate a DFTD parser, rather than a chart parser, into a PROLOG interpreter because we have shown elsewhere [4] that DFTD parsing is more efficient for the relatively simple CFGs that we plan to generate using ILP.

The technical solutions to integrate a parser into a PROLOG engine are limited. Extreme choices are implementing the parser: using the PROLOG language, as a stand alone program (and using system calls from PROLOG to access it), or by modifying directly the code of a PROLOG interpreter. These solutions are either generic and slow, or specific and fast. A compromise was found thanks to the YAP PROLOG implementation [5]. YAP provides some mechanisms to add user-defined built-in predicates implemented in C. More precisely, C functions can be compiled into an object file which can be introduced at running time into the YAP PROLOG interpreter. This enables dedicated functions of the object file to be considered as built-in predicates of YAP, without changing YAP as such. We therefore implemented the parser as a stand alone program in C, and added an interface layer to download the parser in YAP, providing predicates related to parsing.

Evaluation We compared parsing times of our parser with the YAP theorem prover parsing times, on grammars and sequences that were encountered during a biological grammar inference process. The particular process that we considered deals with neuro-peptide precursor (NPP) proteins [3]. Among the different experiments reported in [3], we selected the one that took the longer to complete. This experiment involved inferring on subsequences of the NPPs called *middle*.

From the 76 positive and 2910 random examples of *middle* used in [3], we generated grammars by running an inference process with ALEPH [6]. The `cost/3` ALEPH predicate used during this inference was modified such as to save

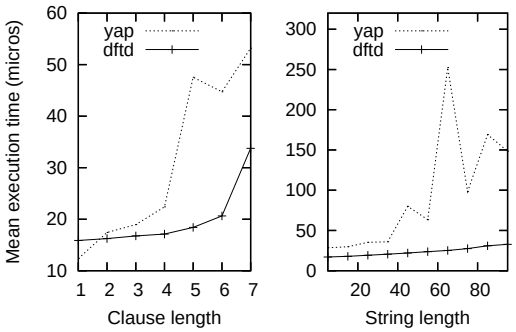

Fig. 1. Mean parsing time for a sequence (y-axis) as a function of the size of the inferred clause (x-axis, left plot), and the size of the parsed sequences (x-axis, right plot). For this last plot, a point with x-coordinate X corresponds to an average over sequences of lengths $X \pm 5$.

each encountered hypothesised clause (i.e., grammar rule) in a file. Running the inference process generated 699 889 grammar rules in DCG form.

We parsed all *middle* examples in all the grammars obtained by the previously described process. This parsing was done using on one hand the YAP theorem prover and on the other our DFTD parser called from YAP. Resulting mean parsing times per sequence are shown in Figure 1. They are plotted respectively as a function of the examples' lengths and the inferred clauses lengths. The total time for parsing all sequences in all grammars was 25h45 with YAP. With our DFTD parser introduced in YAP, this time was reduced to 11h30. Figure 1 shows that not only is our parser much faster on average, but also that it has a much more stable behaviour than YAP. Our parser therefore possesses some potential to speed-up inference. The next section presents how to use it during an ILP based biological grammar acquisition process.

3 Introducing the parser in ILP

The ALEPH ILP system [6] will be used during this section to illustrate our points. The method to call the parser during inference is explained in Subsection 3.1. The experimental evaluation is presented in Subsection 3.2.

3.1 Method

An inference process takes basically place in two stages: (1) initialisation of the parameters and data structures associated to inference, (2) inference. The use of our parser follows these two stages: it first has to be initialised using the background knowledge and examples, and then used during inference.

Initialisation The information that has to be given to the parser comprises the grammar and the examples. However, the particular grammar rules and examples which must be passed to the parser are not known in advance. Grammar rules which were known *a priori* will be available from the background knowledge but those that are to be learned do not, of course, exist before inference begins.

The set of examples that we need to parse can change during inference when, for example, we use the cover-set algorithm. The background knowledge and examples therefore only enable to realise a partial initialisation, which needs to be completed for each evaluated clause. This partial initialisation is done after ALEPH has stored the background knowledge and examples (i.e., after the `read_all/1` call). ALEPH predicates are then used to fill in the parser data structures: the examples and grammar elements can be obtained from the `example/3` and `determination/2` predicates of ALEPH.

Parsing during inference Two possibilities can be considered to use the parser during inference. Solution (1) consider calling the parser in a user defined evaluation function for the inferred clauses. Solution (2) consists in modifying the ILP system itself such as to call our parser instead of the PROLOG theorem prover for the coverage computation. Solution (1) can be seen as more generic since we do not modify the ILP system itself. However, it requires the ILP system to provide access to its evaluation function. Solution (2) has to be implemented in an ad-hoc manner in each ILP system, but does not need any requirement (except that the ILP system has to be implemented in PROLOG). We have implemented both solutions, the next paragraph describes briefly the first one.

In ALEPH, the `cost/3` predicate can be used to define the inferred clause evaluation function (using option `set(evalfn,user)`). A call `cost(C,[P,N,L],Res)` provides the inferred clause (`C`), the count of positive and negative examples covered (`P` and `N`), the inferred clause length (`L`). The user is then supposed to unify the `Res` argument with the desired cost for clause `C`.

We are not interested in changing the evaluation score of the clauses; rather we want to compute `P` and `N` using our parser. To do this, we told ALEPH at initialisation not to compute the `P` and `N` values itself (option `set(lazy_on_cost,true)`). At inference time, in the `cost/3` predicate, we use the `C` value to complete the grammar initialisation, and the `'$aleph_global'(atoms_left, atoms_left(Type,Left))` predicate to know the list of examples to parse. We then call our parser to obtain `P` and `N`, and compute the cost function value (e.g. `Res is N+L-P`, a classical cost value).

3.2 Evaluation

Method As in section 2, we use the *middle* dataset to test our approach. The following implementations of the ILP system were tested[1]: i-YAP: Inferring with ALEPH (unmodified, and therefore using YAP theorem prover); i-DFTD: Inferring with ALEPH, the parsing being done by our parser embedded in ALEPH code; i-DFTD-cost: Inferring with ALEPH, the parsing being done by our parser introduced in ALEPH `cost/3` predicate. For reasons that will become clear, we also considered the implementation i-YAP-cost: inferring with ALEPH using the YAP theorem prover, but called from the `cost/3` predicate. We observed the behaviour of the different algorithms with respect to the value of the `nodes`

[1] Each implementation infers the same set of rules if called with the same parameters.

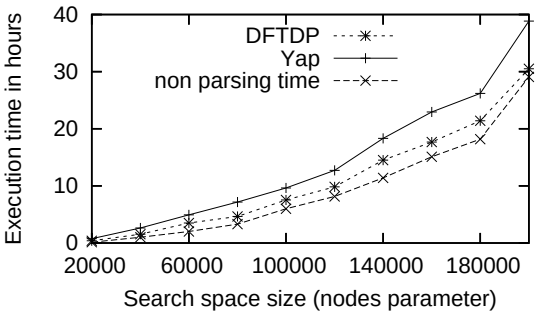

Fig. 2. Inference times (y-axis) as a function of the size of the explored search space (x-axis, the nodes parameter of ALEPH). The curves corresponds respectively to inference time for i-DFTD-cost and i-YAP-cost, the third curve is an estimate of the time spend constructing and searching the search space, i.e., the time not spent on parsing.

ALEPH parameter. This parameter corresponds to the limit on the number of evaluated clauses for each example generalisation. Inference processes with i-YAP, i-DFTD, and i-DFTD-cost implementations were started for different values of the `nodes` parameter.

Results If the `node` value was less than 40 000, the experiments ended correctly in a few hours. However, experiments with implementations i-YAP and i-DFTD for a `nodes` value larger than 40 000 had to be stopped after running more than 4 days. A deeper analysis showed that it is likely that over a value of 40 000 for the `nodes` setting, implementations i-YAP and i-DFTD needed more memory than what was available[2], which leads to an unpredictable behaviour for YAP. Implementation i-DFTD-cost does not need as much memory as implementations i-YAP and i-DFTD (it recomputes the coverage at each evaluation instead of storing it). Implementation i-DFTD-cost was therefore able to explore a larger portion of the search space. In practice we want, such as to obtain biologically pertinent grammars, to explore portions of the search spaces as large as possible[3]. To compare our parser with the YAP theorem prover in this case, we used implementation i-YAP-cost which calls YAP theorem prover for the parsing, but which does not require ALEPH to store the coverage. As for implementation i-DFTD-cost, implementation i-YAP-cost was able to explore larger search spaces.

The running times for implementations i-DFTD-cost and i-YAP-cost are plotted in Figure 2 as a function of the `nodes` parameter values. The figure shows an

[2] YAP was called with 5Mb of heap area, 100Mb of stack area, and 5Mb of Trail area. Complementary experiments shown that the mentionned problem was still present when using the maximum memory available on our computer (500Mb of stack area).

[3] Our aim is not to evaluate the grammar pertinence, and therefore no test set validation has been realised. Our claim for the exploration of large portions of the search space can however be justified by the number of randoms of the training set classified as positives with the rules obtained for the different `nodes` values: inference with a `nodes` value of 20 000 gave 1 573 such randoms (on a total of 2 910), showing the inadequacy of the inferred rules (the random are believed to contain a small proportion of *middles*). The accepted number of randoms drops to 617 for `nodes`=200 000, the obtained rules therefore having better chances to be biologically useful.

improvement of running times when using our parser instead of the YAP theorem prover. To evaluate this result, we have to take into account that the inference process has to construct the clauses to evaluate (in addition to parsing). Therefore, we obtained via some complementary experiments (not described here) an estimate of the time not spent parsing (see Figure 2). This estimate provides a base-line against which to compare our results, namely the absolute maximum possible speed-up that can result from accelerating parsing during inference. From this curve, we can see that we improved the parsing time by more than 45% to the optimum for all `nodes` parameter values, even reaching 85% for a `nodes` value of 200 000. This gives a gain between 18% (`nodes`=160 000) up to 47% (`nodes`=20 000) on total inference times, depending on the `nodes` value. This gain seems to stabilise around 20% for `nodes` values larger than 80 000. It is interesting to see that the deeper the search space visited, the greater the proportion of the time spent to construct this space instead of parsing examples. This suggests other possible approaches to improve inference time for large search spaces.

4 Conclusion

We have shown that ILP systems which are encoded in PROLOG can acquire biological CFGs more quickly by using our proposed parser instead of the PROLOG theorem prover. The gain in execution time is of more than 45% of the time that such ILP systems would normally spend for parsing.

Moreover, the implemented system shows that algorithms available in specific application domains can be integrated in a set of ILP engines by using only one implementation. This seems an interesting alternative to the design of ad hoc ILP systems for particular applications.

Acknowledgements This work is supported by EPSRC grant GR/S68682/01. The authors would like to thank Simon Topp (company GlaskoSmithKline) for his inputs about the NPP/middle data set.

References

1. Searls, D.B.: String variable grammar: A logic grammar formalism for the biological language of DNA. Journal of Logic Programming **12** (1993)
2. Pereira, F., Warren, D.H.D.: Definite clause grammars for language analysis - a survey of the formalism and a comparison with augmented transition networks. Artificial Intelligence **13** (1980) 231–278
3. Muggleton, S.H., Bryant, C.H., Srinivasan, A., Whittaker, A., Topp, S., Rawlings, C.: Are grammatical representations useful for learning from biological sequence data? - a case study. Journal of Computational Biology **5** (2001) 493–522
4. Fredouille, D., Bryant, C.H.: Speeding-up parsing of biological context-free grammars. In: Comb. Pat. Match.: 16th Symp. Volume 3537., Springer-Verlag (2005)
5. Damas, L., Santos Costa, V.: Yap. `http://www.ncc.up.pt/~vsc/Yap` (1989)
6. Srinivasan, A.: A Learning Engine for Proposing Hypotheses (Aleph). `http://web.comlab.ox.ac.uk/oucl/research/areas/machlearn/Aleph` (1993)