Ordering Subgoals in a Backward Chaining Prover

Gergő Csaba Kertész[1], Gergely Papp[2], Péter Szeredi[1], Dániel Varga[2], and Zsolt Zombori[2,3]

[1] Budapest University of Technology and Economics, Budapest
[2] Alfréd Rényi Institute of Mathematics, Budapest
[3] Eötvös Loránd University, Budapest

Introduction Many automated theorem provers are based on *backward chaining*: reasoning starts from a goal statement which we aim to prove and each inference step reduces one of the goals to a (possibly empty) set of new subgoals. We thus maintain a set of open goals that need to be proven and the proof is complete once the open goal set becomes empty. For each goal, there can be several valid inferences, resulting in different successor goal sets and selecting the right inference constitutes the core problem of such theorem provers which has been thoroughly studied in the past half century.

There is, however, another decision to make during proof search, which has been largely underappreciated in the theorem proving community: this is the order in which we select goals from the open goal set. When goals do not share variables, their ordering is irrelevant as their proofs do not influence each other. However, variables establish connections between goals and a certain proof of one goal can result in variable instantiations that make it impossible to prove another goal.

The importance of subgoal ordering is recognised in the SETHEO [7] system. The authors give heuristic justifications for starting with goals that are less likely to succeed and use various manually crafted features to approximate this ordering. [2] use this heuristic ordering and show that when one proof attempt of one first goal fails, it is sometimes better to proceed with another goal before trying alternative attempts of the same goal.

In this paper, we look at the leanCoP [8] connection tableau calculus and explore how goal ordering influences theorem proving performance. leanCoP is a first order theorem prover which translates the problem into clausal normal form and builds a proof tree whose nodes are goals, using two kinds of inference steps: *extension* steps add children to a leaf node while *reduction* steps close leaf nodes. In leanCoP both steps are applied on the leftmost open node by default.

Goal Ordering Database When an extension step adds children to a leaf node, their order is determined and fixed by the order of literals in the input clauses. We start by building a database that reflects the effect of permuting the newly added goals. We introduce two modifictions to leanCoP: 1) we keep track of the number of inference steps for each goal and 2) after each extension, we attempt to finish the proof using **all** possible permutations of the new goals. Suppose our input clause set is C, the current goal is G, we select clause $(\neg G, H)$ where $H = H_1, H_2, \ldots, H_n$ to extend G and permutation $\sigma(H)$ yields a complete proof of G in I steps. Then we add the following tuple to our database:

$$< C, G, H, \sigma, \sigma(H), I >$$

When all permutations of a given (G, H) pair yield the same inference count, then the corresponding tuples are omitted. To avoid infinite branches, we impose a time limit T on the proof of each subgoal after depth D.

Heuristic Goal Ordering Our database aims to provide experimental evidence for the sensibility of goal ordering. We demonstrate its benefit by constructing a heuristic goal ordering strategy upon manual inspection of the data.

We run data extraction on a small set of 131 problems (referred to as the *training set*) from the Mizar40 [4] dataset, extracted from the Mizar Mathematical Library [1]. We employ a time limit of 10 sec for each subgoal. Upon observing the output, we indentify the following simple heuristics:

1. Negative literals should be tried before positive ones.
2. Equality predicates should be proven before other predicates.
3. Equality predicates with variables on both sides should be proven after any other predicates.

We evaluate this heuristic ordering using original leanCoP (without data extraction) and compare it with three baselines: **original**, **random** ordering and **reverse** ordering. Besides the small training set that we used for constructing the ordering, we evaluate on the larger *M2k* [3] benchmark introduced in [5], which consists of 2003 problems from Mizar40.

Table 1: Performance on the training and M2k datasets using various goal orderings: **heuristic**, **original**, **random**, **reverse**. We enforce a 1 sec time limit per problem. *Succ* is the number of problems proven and *Inf* is the average number of inferences in successful proofs.

	heuristic		**original**		**reverse**		**random**	
	Succ	**Inf**	**Succ**	**Inf**	**Succ**	**Inf**	**Succ**	**Inf**
training	**53**	4900	46	2504	47	1173	46	2553
M2k	**824**	2938	712	3103	783	3041	707	2388

Table 1 shows that our simple ordering brings a significant improvement of 16%, in the number of problems proven, on both datasets. The performance of original leanCoP is at par with random ordering, as expected. However, it is rather surprising that simply reversing the goals improves performance. These results demonstrate that ordering does make a big impact on the performance of the system.

The number of inferences required by the various orderings is harder to compare as they are averaged on different sets of problems. However, we can compare two orderings on the set of problems that are solved by both. Heuristic ordering requires less inferences than original, reverse ordering and random ordering in 60%, 65%, 52% (training set) and 60%, 69%, 65% (M2k) of these problems, respectively.

Learning Guided Goal Ordering Our simple heuristic ordering is the result of a superficial human glance over the extracted data. It is certainly not optimal and certainly not universal for all datasets. It was merely meant to demonstrate that there are useful patterns to be extracted from goal ordering statistics. The next logical step is to use machine learning and use a trained model for goal ordering. We experiment with various machine learning frameworks available in Python and expose the trained models to the Prolog implementation of leanCoP via the *pyswip* [9] package. We have implemented the full pipeline of data generation, model training and evaluation and very recently started running learning assisted experiments.

Predicting a permutation of input elements is not a typical machine learning task and we considered three approaches of modeling it:

1. Build a model that gets a sequence of goals and returns a score. This is what our database provides directly, however, model evaluation involves running the model on all permutations of the input and selecting the one that maximizes the output.
2. Build a model that takes a single goal and returns a score. The goals are then evaluated separately and ordered based on the output. This approach is faster to evaluate then the previous one, however, it is not guaranteed that there exists a proper goal scoring function that is consistent with the scores assigned to the permutations in the dataset.
3. Build a sequence to sequence model that returns the optimal permutation of the input sequence. This is the easiest to evaluate, however, it requires the most sophisticated functionality from the model.

So far, we have experimented with the first approach, i.e., training a model that turns the input sequence into a single score. For embedding of goals, we used the features introduced in [6]. These features do not take the current goal into account, only the structure of the clause that the goal is resolved with, hence, we can precompute the clause orderings before proof search, resulting in a constant computational overhead.

Our experimental results are preliminary, but in the current state, we find that it is rather easy to obtain great (above 90%) accuracy on the training data with shallow (2-3 layer) neural networks. However, it is much harder to see any improvement in terms of problems solved. We are currently working to find the best training architecture and parameters.

Conclusion and Future Work Our work explores the effect of changing the order in which goals are proven in a backward chaining theorem prover. We modify the leanCoP connection tableau calculus to extract statistics about different orderings and then use a manually designed heuristic ordering to demonstrate the potential of the extracted data in designing ordering guidance. We believe that even stronger guidance is achievable using machine learning, which is the focus of our ongoing work.

Acknowledgments This work was supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.3-VEKOP-16-2017-00002), the Hungarian National Excellence Grant 2018-1.2.1-NKP-00008 and by the Hungarian Ministry of Innovation and Technology NRDI Office within the framework of the Artificial Intelligence National Laboratory Program.

References

[1] Adam Grabowski, Artur Kornilowicz, and Adam Naumowicz. Mizar in a nutshell. *J. Formalized Reasoning*, 3(2):153–245, 2010.

[2] Ortrun Ibens and Reinhold Letz. Subgoal alternation in model elimination. In Didier Galmiche, editor, *Automated Reasoning with Analytic Tableaux and Related Methods*, pages 201–215, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.

[3] Cezary Kaliszyk and Josef Urban. M2K dataset.

[4] Cezary Kaliszyk and Josef Urban. Mizar40 dataset.

[5] Cezary Kaliszyk, Josef Urban, Henryk Michalewski, and Miroslav Olšák. Reinforcement learning of theorem proving. In *NeurIPS*, pages 8836–8847, 2018.

[6] Cezary Kaliszyk, Josef Urban, and Jiří Vyskočil. Efficient semantic features for automated reasoning over large theories. In Qiang Yang and Michael Wooldridge, editors, *Proc. of the 24th International Joint Conference on Artificial Intelligence (IJCAI'15)*, pages 3084–3090. AAAI Press, 2015.

[7] Reinhold Letz, Johann Schumann, Stefan Bayerl, and Wolfgang Bibel. SETHEO: A high-performance theorem prover. *J. Autom. Reason.*, 8(2):183–212, 1992.

[8] Jens Otten and Wolfgang Bibel. leanCoP: lean connection-based theorem proving. *J. Symb. Comput.*, 36:139–161, 2003.

[9] Yüce Tekol and contributors. PySwip v0.2.10, 2020.