EPiC Series in Computing

Volume 94, 2023, Pages 112–123

Proceedings of 24th International Conference on Logic for Programming, Artificial Intelligence and Reasoning

Guiding an Instantiation Prover with Graph Neural Networks

Karel Chvalovský[1], Konstantin Korovin[2], Jelle Piepenbrock[13], and Josef Urban[1]

[1] Czech Technical University in Prague
{karel.chvalovsky, jelle.piepenbrock, josef.urban}@cvut.cz
[2] University of Manchester, Manchester, U.K.
konstantin.korovin@manchester.ac.uk
[3] Radboud University Nijmegen

Abstract

In this work we extend an instantiation-based theorem prover iProver with machine learning (ML) guidance based on graph neural networks. For this we implement an interactive mode in iProver, which allows communication with an external agent via network sockets. The external (ML-based) agent guides the proof search by scoring generated clauses in the given clause loop. Our evaluation on a large set of Mizar problems shows that the ML guidance outperforms iProver's standard human-programmed priority queues, solving more than twice as many problems in the same time. To our knowledge, this is the first time the performance of a state-of-the-art instantiation-based system is doubled by ML guidance.

1 Introduction

In the recent years, machine learning (ML) and related AI methods are being increasingly combined with automated deduction. One of the most interesting tasks in this area is equipping fast state-of-the-art automated theorem provers (ATPs) with *efficient internal guidance* of their calculi based on learning from many previous proof-search decisions. This is challenging, because the fast ATPs typically generate and evaluate thousands to millions of inferences within seconds. While related AI/TP tasks such as learning-based *premise selection* [1], *tactical guidance* [5], and *neural conjecturing* [19] can use relatively slow and expensive ML methods that are called only rarely on a single problem, internal guidance requires efficient ML methods and their nontrivial integration with the fast ATPs.

In the last five years, several advances in internal guidance were made for connection-based [10, 15, 20] and resolution/superposition-based ATPs [8, 17, 7]. However, there has so far been only limited success in guiding instantiation-based systems, which are – thanks to the integration with powerful SAT solvers – today becoming very competitive [2].

In this work we therefore develop strong internal ML guidance for one of today's main instantiation-based theorem provers: iProver [11] (Section 2). In more detail, the ML guidance is based on naming-invariant graph neural networks (GNNs) (Section 3). To combine the two,

we develop an interactive mode in iProver, which allows communication with external agents via network sockets (Section 4). The GNN guidance is then implemented as an instance of such external agent – a GPU server (Section 5). Our evaluation on a large set of Mizar problems (Section 6) shows that the ML guidance doubles the performance of iProver's standard human-programmed priority queues (Section 7). To our knowledge, this is the first time a state-of-the-art instantiation-based system is considerably improved by neural ML guidance.

2 iProver

2.1 iProver and the Inst-Gen Calculus

iProver [11] is an automated theorem prover for quantified first-order logic. At the core of iProver is an instantiation calculus, Inst-Gen [4, 12], which can be combined with resolution and superposition calculi [3]. The Inst-Gen calculus is based on propositional reasoning to deal with propositional abstractions of first-order clauses and model-guided incremental instantiations using unification to generate new first-order instances. At a high level, the procedure works as follows.

Given a set of first-order clauses S, its propositional abstraction $S\bot$ is obtained by mapping all variables to a designated ground term $\bot$. A propositional solver is applied to $S\bot$ and it either proves that $S\bot$ is unsatisfiable and in this case the set of first-order clauses S is also unsatisfiable or shows that $S\bot$ is satisfiable and in this case returns a propositional model of the abstraction $S\bot$. This propositional model is analysed if it can be extended to a full first-order model. If it can not be extended then it is possible to show that there must be complimentary literals in the model that are unifiable. In this case the Inst-Gen calculus produces instances of relevant clauses with the most general unifier which resolves this conflict in the model and provide sufficient information to the propositional solver that this conflict will not occur in the future. This loop is repeated with more instances added until the unsatisfiability is witnessed by the propositional abstraction or a saturated set is obtained (possibly in the limit) in which case the original first-order formula is satisfiable. The Inst-Gen calculus is *refutationally complete*, which means that if the set of first-order clauses S is unsatisfiable then in a finite number of iterations, the propositional abstraction of derived instantiations $S'\bot$ will be unsatisfiable [4].

2.2 Guiding iProver

Thus, iProver leverages the power of propositional solvers for a) solving propositional abstractions, and b) guiding instantiations based on propositional models. Although this approach often works well in practice, one of the major bottlenecks is the number of generated first-order instances with only a few of them usually needed in the final proof.

In this work we therefore investigate how machine learning can be used to select iProver inferences that are most likely to be used in a proof. In particular, the propositional model typically leads to multiple Inst-Gen inferences that can be made to provide sufficient information to the propositional solver that will avoid particular conflicting assignment of unifiable literals in the propositional model. Figure 1 shows an overall scheme of the approach. The ML advice will be used to select the clauses for performing the Inst-Gen inferences (see Section 4 for details).

3 Name-Independent Graph Neural Network

To learn a machine learning heuristic for clause selection in the Inst-Gen calculus, we use a graph neural network [16]. Specifically, we use a PyTorch [14] implementation of the name-independent architecture developed originally for connection provers [13], but also used for the recent ENIGMA systems for the E automated theorem prover [6].

In this work, we will give a brief primer on the main ideas underlying this neural network architecture, so far as they are directly relevant to the current work. Given a first-order logic problem expressed as a set of clauses named *Clauses*, we parse the problem and create a (hyper)graph that represents the problem. In this representation, three different types of nodes are distinguished. These are the clause nodes $\mathcal{C}$, the symbol nodes $\mathcal{S}$ and the term nodes $\mathcal{T}$. The collection of all nodes is the set $\mathcal{N}$. Each clause node corresponds to a clause in the input file. Symbol nodes correspond to either function or predicate symbols.

The structure of the original first-order problem is reflected in the edges connecting these different types of nodes. This allows the network to see how the symbols are used without knowing their names, and therefore handle common ATP issues such as the typically quite unstable naming of Skolem symbols between problems. It is also important when used in ITP-based (*hammer*) scenarios, where new terminology is introduced frequently during the formalization (see Section 7.2). There are several different types of edges between the nodes. Clauses are connected to their literals, while the polarity of the literals is explicitly handled (so the network has a built-in notion of negation). Symbols are connected to the terms they are used in. For example, given a term $f(t_1, \ldots, t_k) \in \mathcal{T}$, with k subterms labelled as t_i, the nodes corresponding to the function symbol $f \in \mathcal{S}$ will be connected to t_i via the term node $f(t_1, \ldots, t_k)$, for $1 \leq i \leq k$.

This graph representation then contains the relationships between the various mathematical objects occurring in the problem. The basic idea of using this representation to learn a heuristic for clause selection is to let each node exchange messages with its neighbours, to update some state representation according to a learned transformation parameterized by weights that are learned by gradient descent on an error function. This message passing is iterated for a fixed amount of steps, after which the representations for clauses are used to predict whether the clause is useful or not. Initially, each node is represented by a vector of floating point numbers (embedding), which differs based on the type of node. We distinguish the following node types: conjecture clauses, non-conjecture clauses, function symbol nodes, predicate symbols, terms, literals and variables.

After some message-passing steps, the predicted probability of a clause being useful for the proof is computed by another learned transformation that takes the final representation of each clause node and combines this with the representation of the conjecture clauses. This predicted score for the clauses is then used to influence iProver's clause priority queues. The machine learning system is trained by minimizing, through *gradient descent*, the *binary cross-entropy* error function that measures how well the network can predict which clauses are useful for the proofs and which ones are not.

4 Interactive Mode

We have newly developed an Interactive Mode for iProver, which is in detail described in our repository.[1] In this mode, iProver communicates with an *external agent (EA)* (e.g., ML-based,

[1] https://gitlab.com/korovin/iprover/-/blob/master/README-interactive.md

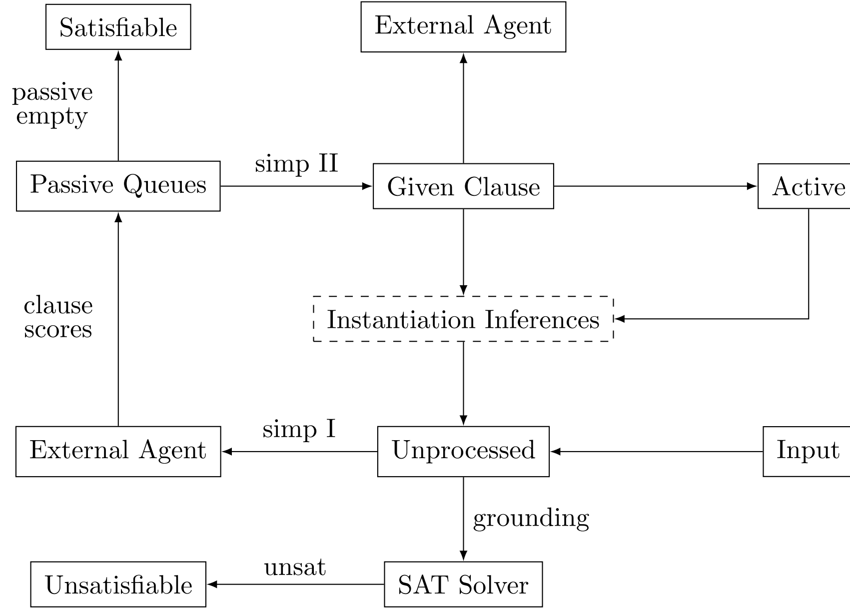

Figure 1: Interactive Given Clause Loop in iProver

that can be written in any language, e.g., Python) via TCP/IP sockets. The external agent can be used to provide proof search guidance by either assigning scores to clauses which are used for prioritising them for next inferences or explicitly selecting the given clause for the next inferences. The communication is bi-directional:

- iProver submits different messages to the agent, such as the given clauses, generated clauses, simplified clauses etc.
- The agent can guide the iProver search by different actions such as selection of the given clause, assigning scores for clauses in passive queues etc.

Figure 1 details the given clause loop extended with the external guidance. The Input clauses are first submitted to the Unprocessed set and simplified (simp I). Then, the EA evaluates the clauses and assigns scores that are used as priorities in *passive priority queues*. These are used to store Passive clauses, i.e., the clauses that are waiting to be involved in inferences. Each priority queue is based on a lexicographic combination of different clause features, such as: the number of literals, the number of variables, clause age, proof distance from the conjecture, etc. Priority queues are combined in a round-robin fashion with specified multipliers. The EA scores are treated as one of the clause features and can be used in a standalone priority queue based on just EA scores or combined in queues with other clause priority features.

The Given Clause is the clause selected from the priority queues for the next inferences with clauses stored in the Active set. The Given Clause is also submitted to the EA to be used as a part of the *context* for the next rounds of evaluations. After all inferences between the Given Clause and clauses in Active are performed, the Given Clause is moved to Active and all newly derived clauses are moved to the Unprocessed set. Groundings of clauses from the Unprocessed are also submitted to the SAT solver which is executed with some specified intervals. If the

SAT solver returns that the accumulated groundings are unsatisfiable, then the Input set of first-order clauses is also unsatisfiable and the problem is solved. If the Passive Queues are empty then due to the completeness of the procedure [4, 12], the Input set is satisfiable.

The problem of selecting the most suitable given clause is the "Holy Grail" of automated theorem proving: with perfect clause selection the proof can be directly reconstructed. In most cases, though the clause selection is far from perfect which results in explosion in the search space. One of our main contributions is to show that an External Agent based on GNN models can be trained to select clauses considerably better than human-based fine tuning of priorities.

5 GPU Server

We have developed[2] an external agent that uses a trained GNN to score clauses generated in the given clause loop. The main function is similar to E/ENIGMA, where a Python GPU server [7] is used to reduce the overhead of repeated model loading. However, in ENIGMA, the score requests are created as full graphs already in E, where the *context* consisting of the given clauses and conjectures is added to the generated clauses that are to be scored, and hence the server is *stateless*. The sole purpose of the server there is to receive the queries, evaluate them using a GPU, and send the results back. Here, we instead build a richer *stateful* server that keeps track of the given clauses and conjectures, and is itself capable of generating in various ways the context that is used to score the requests.

The agent is a Python server that contains three types of processes. The *main process* checks a network socket and distributes the incoming connections to the *state processes*. Each state process handles one client (iProver) connection. Whenever the state process receives a score request, it adds the context of the given clauses and conjectures and sends a request to a *GPU process*. To create contexts, the state process must keep all the generated clauses, given clauses, and conjectures in its memory. The GPU process then evaluates the clauses using a GNN preloaded on a GPU and sends the scores back to the state process that communicates them to the client iProver. While this architecture is more involved than the simple ENIGMA server, it is more flexible, allowing different parameterizations and experiments with the GNN guidance. We show in Section 7 that the overhead incurred by this more involved guiding architecture is reasonably low, and that its use results in very high *real time* improvements of iProver's performance.

6 Experimental Setting

6.1 Evaluation Problems

The evaluation is performed on a large benchmark of 57 880 problems[3] originating from the Mizar Mathematical Library (MML) [9] exported to first-order logic by MPTP [18]. The Mizar problems are split[4] (in a 90-5-5% ratio) into 3 subsets: (1) 52k problems for *training*, (2) 2896 problems for *development*, and (3) 2896 problems for final evaluation (*holdout*). We use this split for the evaluation done here. Since we are interested in internal ATP guidance and not premise selection, we use problems with premises limited to those used in the human written Mizar proofs (*bushy problems*). As an additional measure of the generalization, we also evaluate

[2] https://gitlab.ciirc.cvut.cz/chvalkar/iprover-gnn-server

[3] http://grid01.ciirc.cvut.cz/~mptp/1147/MPTP2/problems_small_consist.tar.gz

[4] http://grid01.ciirc.cvut.cz/~mptp/Mizar_eval_final_split

(Section 7.2) the trained system on 13 370 theorems in 242 articles that were added in a newer version of MML (1382) and thus never seen during the training. More than half of those problems contain new terminology.

6.2 Data: Classsic vs Dynamic

The training data are collected from previous successful runs. The standard approach (*classic data*), which was introduced by the ENIGMA systems, is to take all given clauses and consider those that ended up in the proof as useful (*positive clauses*) and those not used as useless (*negative clauses*). Hence, for each proof, we have one training example (a graph of clauses) containing the useful/useless clauses and the conjecture we want to prove. However, the size of such a training example corresponds to the *final* size of the set of all given clauses proposed in the proof search, not to the *intermediate* sizes of the *actual* score requests generated multiple times during the proof search. This means that in the classic setting, the distribution of the training graph sizes may be shifted quite far from the actual distribution encountered during the proof search. Such subtle shifts between the training and evaluation distributions have quite often a negative effect on the performance of more complex machine learning architectures.

To remedy that, we newly experiment here also with using the actual score requests (*dynamic data*) from the proof searches as the training data. However, such score requests contain not only the given clauses but also the passive clauses. It is not always clear how the passive clauses should be labeled, since they may or may not lead to an alternative proof. For simplicity, we use a common pragmatic approach and consider them as negatives. We also want to prevent the learning from focusing on too many minor alternative proofs of the same problem and neglecting problems that have fewer proofs. When we learn from multiple collections of proofs (produced in multiple iterations of the proving and learning over the training data), we sample these proofs so that each problem contributes the same number of training examples.

6.3 iProver Settings

We run iProver using its instantiation mode.[5] The score requests are performed in batches of size approximately 1000 to improve the performance. For details, see Appendix B. For our evaluations of the baseline non-guided iProver, we run it with its standard human-programmed priority queues for clause selection in the Inst-Gen calculus in three different modes. These modes are the non-interactive mode, the interactive mode without ML evaluation, and the interactive mode with ML evaluation that is ignored.

In the *non-interactive mode*, iProver does not communicate with the server. In the interactive mode without ML evaluation (*no eval*), the ML server returns zeros without any evaluation. The purpose of this mode is to measure the overhead caused by the communication protocol and processing requests. In the interactive mode with ignored ML evaluation (*ignore eval*), the server evaluates clauses but the scores are ignored by iProver. By running in this mode, we can investigate the overhead incurred by the ML calculations.

In the setting where the scores provided by the server are used for clause selection, we use two modes: *solo* and *coop*. In the solo mode, there is just one priority queue ordered by the scores provided by the server. In the coop mode, we combine server-provided scores with the human-programmed priority queues in an equal ratio, see Appendix B.

[5]Recent iProver can also use superposition and resolution and combine them with instantiation. However, in this work our focus is to establish if instantiation-based methods can be effectively guided by learning.

7 Results

As a starting reference point, we run iProver with the human-programmed priority queues for clause selection to collect the initial training examples. In this mode, without being slowed down by the server (non-interactive mode), iProver solves 502 theorems out of the 2896 theorems in the development set (Section 6.1) in 15 seconds. However, to extract the training data, we need to run iProver in the interactive mode. It solves 451 and 482 theorems in the development set in ignore eval and no eval modes, respectively. Therefore, there is some overhead from the communication and GNN calculations, but its impact is manageable.

Using the no eval mode, we get 9087 training proofs and also 482 proofs in the development set that are used as a validation set for finding the best performing model. We train three models, one with classic data and two with dynamic data (taking randomly 4 or 10 queries from the successful runs, respectively). Each model is run in either solo, or coop mode. Two best performing models on the development set were trained on dynamic data using 4 and 10 samples and run in the coop mode, see Table 1. We use these two models to obtain further training data for the next iteration. It is worth mentioning that models trained on dynamic data perform significantly better than models trained on classic data.

In the next iteration, we use all proofs found so far by iProver in the no eval mode together with the proofs found using the two best performing models. This yields 14 994 and 834 proved theorems on the training and development set, respectively. We again trained three models, from which the best performing was the model trained on the dynamic data using 4 samples run in the solo mode. The model using 10 dynamic samples run in the coop mode was reasonably complementary, and we also used it for the next iteration.[6]

In the last iteration, we trained on 18 452 theorems solved (using possibly multiple proofs available for each theorem) on the training set and used 1026 problems solved on the development set for evaluating the best performing models. In this iteration, we also trained dynamic models with an increased size of embeddings (from 16 to 32) and the number of layers (from 10 to 11). Increasing the size of embeddings leads to a better performance, see Table 1, but increasing solely the number of layers does not. However, the best performing models come from increasing both the size of embeddings and the number of layers.

7.1 Holdout Set Performance

iProver using guidance from our best performing model solves 1094 problems on the development set and 1093 on the holdout set. Moreover, it solves a very similar fraction of problems on the training set. Similar results hold also for other models, see Table 1.

The training procedure seems to be quite robust to overfitting on the training data and to generalize well. This could be due to several aspects: (i) different runs lead to different sets of given clauses, (ii) only a limited number of the dynamic samples is seen during the training per problem (4 or 10), and (iii) the contexts are randomly sampled from the available given clauses.

Interestingly, when we evaluate just the accuracy of the trained GNN model, its performance on the train, development and holdout sets slightly differs. The GNN has a balanced accuracy of 0.9503 on the training examples, 0.9397 on the examples from the development set that were used for selecting the best performing model, and 0.9406 on the examples from the holdout set. This difference is probably not large enough to cause significant differences in the ultimate ATP performance.

[6] It is likely that the model with 10 dynamic samples performed worse than the model trained on 4 dynamic samples per problem, because models in this iteration were trained for a shorter period of time.

Iter.	Solver (15 s)	Data	Model	Devel	Holdout	Train
	Ignore eval			451	455	
	No eval			482	475	9087
	Non-interactive			502	500	
0	Solo	classic	$d = 16$, $l = 10$	663	656	
0	Coop	classic	$d = 16$, $l = 10$	699	704	
0	Solo	dynamic (4)	$d = 16$, $l = 10$	714	723	
0	Coop	dynamic (4)	$d = 16$, $l = 10$	744	729	13 403
0	Solo	dynamic (10)	$d = 16$, $l = 10$	739	739	
0	Coop	dynamic (10)	$d = 16$, $l = 10$	760	759	13 534
1	Solo	dynamic (4)	$d = 16$, $l = 10$	951	945	16 964
1	Coop	dynamic (10)	$d = 16$, $l = 10$	834	835	14 953
2	Solo	classic	$d = 16$, $l = 10$	674	689	
2	Coop	classic	$d = 16$, $l = 10$	739	741	
2	Solo	dynamic (4)	$d = 16$, $l = 10$	1004	1017	
2	Solo	dynamic (4)	$d = 16$, $l = 11$	1003	987	
2	Solo	dynamic (4)	$d = 32$, $l = 10$	1028	1032	
2	Solo	dynamic (4)	$d = 32$, $l = 11$	1033	1032	
2	Coop	dynamic (4)	$d = 16$, $l = 10$	955	945	
2	Coop	dynamic (4)	$d = 16$, $l = 11$	945	942	
2	Coop	dynamic (4)	$d = 32$, $l = 10$	984	990	
2	Coop	dynamic (4)	$d = 32$, $l = 11$	988	983	
2	Solo	dynamic (10)	$d = 16$, $l = 10$	1018	1022	
2	Solo	dynamic (10)	$d = 16$, $l = 11$	922	901	
2	Solo	dynamic (10)	$d = 32$, $l = 11$	1068	1063	
2	Solo	dynamic (10)	$d = 32$, $l = 11$	**1094**	1093	
2	Coop	dynamic (10)	$d = 16$, $l = 10$	955	956	
2	Coop	dynamic (10)	$d = 16$, $l = 11$	897	883	
2	Coop	dynamic (10)	$d = 32$, $l = 10$	1018	1024	
2	Coop	dynamic (10)	$d = 32$, $l = 11$	1037	1034	

Table 1: Proving-learning iterations and their performance on the devel and holdout sets. Model parameters are the size of embeddings (d) and the number of layers (l).

7.2 Transfer to Newly Added Mizar Articles

We have also tested our trained system on the problems from a newer version of Mizar (1382) that has 242 new articles and 13 370 theorems in them; more than half of which contain new terminology. Whereas iProver in the non-interactive mode solves 1662 theorems, iProver guided by our best model (dynamic trained on 10 samples with $d = 32$ and $l = 11$) solves 3657 theorems. Hence the improvement is similar to the results on the dataset that we used for our training, see Section 6.1.

8 Conclusion

We have developed an efficient learning-based guidance for the Inst-Gen calculus and shown that the performance of the instantiation prover iProver is very substantially improved by doing the inferences recommended by a name-independent graph neural network. The amount of theorems proved on our holdout set by the ML-guided iProver is more than doubled compared to the amount of proofs the unguided solver can find. Additionally, we found that the model

also generalizes very well to a large number of problems added to the Mizar Mathematical Library much later than our initial data set. This indicates that we can reasonably expect the trained predictors to generalize to new problems coming in over time.

The interactive interface of iProver that we have developed here can be used for other purposes. The interface exposes some of the internal data of iProver, which may be further relevant to the clause prediction task. We have already seen that the dynamic setting improves over the classic ENIGMA setting. In the future, we could use the interface e.g. for giving the GNNs access to the SAT model which may serve as a semantic characterization of the search.

9 Acknowledgements

This work was partially supported by the European Regional Development Fund under the Czech project AI&Reasoning no. CZ.02.1.01/0.0/0.0/15_003/0000466 (KC, JP, JU), Amazon Research Awards (JP, JU), by the Czech MEYS under the ERC CZ project POSTMAN no. LL1902 (KC, JP), EPSRC grant EP/V000497/1 UK (KK), and the EU ICT-48 2020 project TAILOR no. 952215 (JU).

References

[1] Jesse Alama, Tom Heskes, Daniel Kühlwein, Evgeni Tsivtsivadze, and Josef Urban. Premise selection for mathematics by corpus analysis and kernel methods. *J. Autom. Reasoning*, 52(2):191–213, 2014.

[2] Martin Desharnais, Petar Vukmirovic, Jasmin Blanchette, and Makarius Wenzel. Seventeen provers under the hammer. In *ITP*, volume 237 of *LIPIcs*, pages 8:1–8:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.

[3] André Duarte and Konstantin Korovin. Implementing superposition in iProver (system description). In Nicolas Peltier and Viorica Sofronie-Stokkermans, editors, *Automated Reasoning - 10th International Joint Conference, IJCAR 2020, Paris, France, July 1-4, 2020, Proceedings, Part II*, volume 12167 of *Lecture Notes in Computer Science*, pages 388–397. Springer, 2020.

[4] Harald Ganzinger and Konstantin Korovin. New directions in instantiation-based theorem proving. In *18th IEEE Symposium on Logic in Computer Science (LICS 2003), 22-25 June 2003, Ottawa, Canada, Proceedings*, pages 55–64. IEEE Computer Society, 2003.

[5] Thibault Gauthier, Cezary Kaliszyk, Josef Urban, Ramana Kumar, and Michael Norrish. Tactictoe: Learning to prove with tactics. *J. Autom. Reason.*, 65(2):257–286, 2021.

[6] Zarathustra A Goertzel, Jan Jakubův, Cezary Kaliszyk, Miroslav Olšák, Jelle Piepenbrock, and Josef Urban. The isabelle enigma. In *13th International Conference on Interactive Theorem Proving*, 2022.

[7] Zarathustra Amadeus Goertzel, Karel Chvalovský, Jan Jakubuv, Miroslav Olsák, and Josef Urban. Fast and slow Enigmas and parental guidance. In *FroCoS*, volume 12941 of *Lecture Notes in Computer Science*, pages 173–191. Springer, 2021.

[8] Jan Jakubuv and Josef Urban. Hammering Mizar by learning clause guidance. In John Harrison, John O'Leary, and Andrew Tolmach, editors, *10th International Conference on Interactive Theorem Proving, ITP 2019, September 9-12, 2019, Portland, OR, USA*, volume 141 of *LIPIcs*, pages 34:1–34:8. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

[9] Cezary Kaliszyk and Josef Urban. MizAR 40 for Mizar 40. *J. Autom. Reasoning*, 55(3):245–256, 2015.

[10] Cezary Kaliszyk, Josef Urban, Henryk Michalewski, and Miroslav Olšák. Reinforcement learning of theorem proving. In *Advances in Neural Information Processing Systems 31: Annual Conference*

on Neural Information Processing Systems 2018, NeurIPS 2018, 3-8 December 2018, Montréal, Canada., pages 8836–8847, 2018.

[11] Konstantin Korovin. iProver - an instantiation-based theorem prover for first-order logic (system description). In Alessandro Armando, Peter Baumgartner, and Gilles Dowek, editors, *Automated Reasoning, 4th International Joint Conference, IJCAR 2008, Sydney, Australia, August 12-15, 2008, Proceedings*, volume 5195 of *Lecture Notes in Computer Science*, pages 292–298. Springer, 2008.

[12] Konstantin Korovin. Inst-Gen - A modular approach to instantiation-based automated reasoning. In Andrei Voronkov and Christoph Weidenbach, editors, *Programming Logics - Essays in Memory of Harald Ganzinger*, volume 7797 of *Lecture Notes in Computer Science*, pages 239–270. Springer, 2013.

[13] Miroslav Olšák, Cezary Kaliszyk, and Josef Urban. Property invariant embedding for automated reasoning. In *ECAI 2020*, pages 1395–1402. IOS Press, 2020.

[14] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

[15] Michael Rawson and Giles Reger. lazycop: Lazy paramodulation meets neurally guided search. In *TABLEAUX*, volume 12842 of *Lecture Notes in Computer Science*, pages 187–199. Springer, 2021.

[16] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80, 2009.

[17] Martin Suda. Improving enigma-style clause selection while learning from history. In *CADE*, volume 12699 of *Lecture Notes in Computer Science*, pages 543–561. Springer, 2021.

[18] Josef Urban. MPTP 0.2: Design, implementation, and initial experiments. *J. Autom. Reasoning*, 37(1-2):21–43, 2006.

[19] Josef Urban and Jan Jakubuv. First neural conjecturing datasets and experiments. In Christoph Benzmüller and Bruce R. Miller, editors, *Intelligent Computer Mathematics - 13th International Conference, CICM 2020, Bertinoro, Italy, July 26-31, 2020, Proceedings*, volume 12236 of *Lecture Notes in Computer Science*, pages 315–323. Springer, 2020.

[20] Zsolt Zombori, Josef Urban, and Miroslav Olsák. The role of entropy in guiding a connection prover. In *TABLEAUX*, volume 12842 of *Lecture Notes in Computer Science*, pages 218–235. Springer, 2021.

A Server Settings

We run the server with 48 state workers, 32 GPU workers, and the query max size is 2048. The context is created from a random sample of at most 512 given clauses and all negated conjectures (may be among given clauses). In no eval mode we run server with the `--zero_scores` option.

The evaluations were run on NVIDIA DGX-1 with dual 20-core Inter E5-2698 v4, 512 GB RAM, and 8 NVIDIA Tesla V100 GPU cards.

B iProver Settings

We compile iProver (branch `2022_sockets`) with `STATIC=true z3=false` in the debug mode (`let dbg_global_flag = true`), which is useful for extracting proofs for further iterations. For every problem, we run `tptp4X -t noint -u machine -N` to rename integers to constants.

We always run 16 iProvers in parallel on the same machine as we run the server and they communicate via the loopback network interface. The server overhead is relatively small (overall

load is usually 22–23) with a GPU utilization between 10–20% even with our biggest models ($d = 32$ and $l = 11$). Time limits (in real time) are always 15 seconds.

The detailed settings are as follows.

Non-interactive mode

```
iproveropt \
--interactive_mode false \
--inst_learning_loop_flag false \
--schedule none \
--preprocessing_flag false \
--instantiation_flag true \
--superposition_flag false \
--resolution_flag false \
--time_out_real 15 \
--inst_unprocessed_bound 1000
```

Ignore eval and no eval mode

```
iproveropt \
--interactive_mode true \
--external_ip_address "127.0.0.1" \
--external_port "12300" \
--inst_learning_loop_flag false \
--schedule none \
--preprocessing_flag false \
--instantiation_flag true \
--superposition_flag false \
--resolution_flag false \
--time_out_real 15 \
--inst_unprocessed_bound 1000
```

Solo mode

```
iproveropt \
--interactive_mode true \
--external_ip_address "127.0.0.1" \
--external_port "12300" \
--inst_learning_loop_flag false \
--schedule none \
--preprocessing_flag false \
--instantiation_flag true \
--superposition_flag false \
--resolution_flag false \
--inst_passive_queue_type priority_queues \
--inst_passive_queues_freq "[1]" \
--inst_passive_queues "[[+external_score]]" \
--time_out_real 15 \
--inst_unprocessed_bound 1000
```

Coop mode

```
iproveropt \
--interactive_mode true \
--external_ip_address "127.0.0.1" \
--external_port "12300" \
--inst_learning_loop_flag false \
--schedule none \
--preprocessing_flag false \
--instantiation_flag true \
--superposition_flag false \
--resolution_flag false \
--inst_passive_queue_type priority_queues \
--inst_passive_queues_freq "[27;25;2]" \
--inst_passive_queues "[[+external_score];[-conj_dist;+conj_symb;-num_var];
[+age;-num_symb]]" \
--time_out_real 15 \
--inst_unprocessed_bound 1000
```

C GNN Settings

We trained models with node, symbol, and clause embeddings of size 16 with 10 layers. In the last iteration, we also tried models with embeddings of size 32 and with 11 layers. We have not performed other hyperparameter searches. Layer normalization is used after every message passing step to ensure training stability.

The models were trained on various servers with NVIDIA GTX 1080 Ti, Tesla V100, and A40 GPUs. We trained each model for roughly two days (with a limit of 100 epochs). The only exception was iteration 1 where we trained models just for one day to get more proofs quickly. The Adam optimization algorithm with a learning rate 0.0005 was used for training.