REFACTOR: LEARNING TO EXTRACT THEOREMS FROM PROOFS

Jin Peng Zhou, Yuhuai Wu *
University of Toronto, Vector Institute
jinpeng.zhou@mail.utoronto.ca, ywu@cs.toronto.edu

Qiyang Li
University of California, Berkeley
qcli@berkeley.edu

Roger Grosse
University of Toronto, Vector Institute
rgrosse@cs.toronto.edu

ABSTRACT

Human mathematicians are often good at recognizing modular and reusable theorems that make complex mathematical results within reach. In this paper, we propose a novel method called theoREm-from-prooF extrACTOR (REFACTOR) for training neural networks to mimic this ability in formal mathematical theorem proving. We show on a set of unseen proofs, REFACTOR is able to extract 19.6% of the theorems that humans would use to write the proofs. When applying the model to the existing Metamath library, REFACTOR extracted 16 new theorems which are used frequently in the Metamath library, with an average usage of 733.5 times. With newly extracted theorems, we show that the existing proofs in the MetaMath database can be refactored to shorten the proof lengths. Lastly, we demonstrate that the prover trained on the new-theorem refactored dataset is able to prove more test theorems.

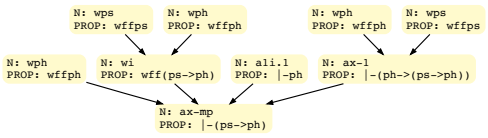
1 INTRODUCTION

In the history of calculus, one remarkable early achievement was made by Archimedes in the 3rd century BC, who established a proof for the area of a parabolic segment to be $4/3$ that of a certain inscribed triangle. In the proof he gave, he made use of a technique called the *method of exhaustion*, a precursor to modern calculus. However, as this was a strategy rather than a theorem, applying it to new problems required one to grasp and generalize the pattern, as only a handful of brilliant mathematicians were able to do. It wasn't until millennia later that calculus finally became a powerful and broadly applicable tool, once these reasoning patterns were crystallized into modular concepts such as limits and integrals.

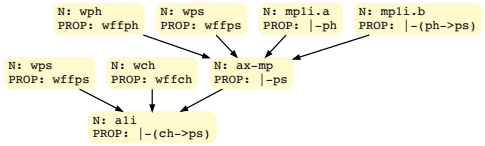
A question arises – can we train a neural network to mimic human's ability to extract modular components that are useful? In this paper, we focus on a specific instance of the problem in the context of theorem proving, where the goal is to train a neural network model that can discover reusable theorems from a set of mathematical proofs. Specifically, we work under formal systems where each mathematical proof is represented by a tree called *proof tree*. Moreover, one can extract some connected component of the proof tree that constitutes a proof of a standalone theorem. Under this framework, we can reduce the problem to training a model that solves a binary classification problem where it determines whether each node in the proof tree belongs to the connected component that the model tries to predict.

To this end, we propose a method called theoREm-from-prooF extrACTOR (REFACTOR) for mimicking human's ability to extract theorems from proofs. Specifically, we propose to reverse the

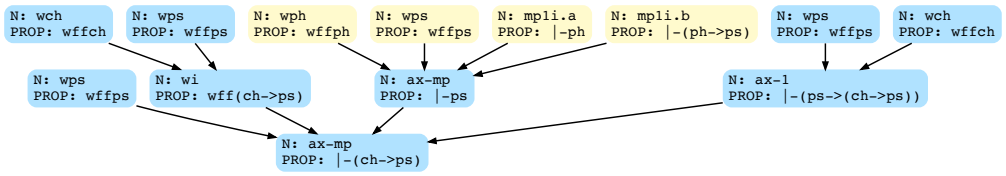
*Equal Contribution



(a) The proof tree of `a1i`.



(b) The proof tree of `mp1i`.



(c) The proof tree of `mp1i` with theorem `a1i`'s proof expanded (colored in blue).

Figure 1: In (a) and (b), we show proof tree visualizations of the theorem `a1i` and `mp1i`. Each node contains two pieces of information: `N` refers to the the name associated with the node, and `PROP` refers to the proved proposition that is obtained by applying all theorem applications above that node. In (c), we also show the expanded proof tree of `mp1i` with `a1i`'s proof being expanded and colored in blue, namely, the set of nodes $\mathcal{V}_{target}$ that are the targets for our proposed learning task.

process of human theorem extraction to create machine learning datasets. Given a human proof T, we take a theorem s that is used by the proof. We then use the proof of theorem s, T_s, to re-write T as T' such that T' no longer contains the application of theorem s, and replace it by using the proof T_s. We call this re-writing process the *expansion* of proof T using s. The expanded proof T' becomes the input to our model, and the model's task is to identify a connected component of T', T_s, which corresponds to the theorem s that humans would use in T.

Our experimental result establishes the first proof of concept using neural network models to extract theorems from proofs. Our best REFACTOR model is able to extract exactly the same theorem, without even seeing instances of it in the training set, as human's ground truth about 19.6% of time. We also observe that REFACTOR's performance improves when we increase the model size. It shows promising results that further scaling up the model size might allow it to mimic human much better in extracting reusable theorems than our reported results.

Interestingly, when REFACTOR's prediction does not match the ground truth human theorems, the prediction can also be a new theorem that is not in the existing library of proofs. We developed an algorithm to verify whether the predicted component constituent a valid proof of a theorem, and we found REFACTOR extracted 1907 valid, new theorems. We also apply REFACTOR to proofs from the existing Metamath library, from which REFACTOR extracts another 16 novel theorems. Remarkably, those 16 proofs are used very frequently in the Metamath library, with an average usage of 733.5 times. Furthermore, with newly extracted theorems, we show that the human theorem library can be refactored, and hence the proof length are shortened. The extracted theorem reduces approximately 400k nodes in total. Lastly, we demonstrate that training a prover on the refactored dataset leads to better proof success rates in proving new test theorems.

2 Method

2.1 Sub-component of a Proof Tree as a Theorem

We provide some background of Metamath in Appendix B.1. One key idea is that a mathematical proof can be represented as a proof tree. Interestingly, one can also identify some components of the proof tree as an embedded proof for another theorem. To start with, given a node in a proof tree, one can treat the entire subtree above that node as a proof of the node (more precisely, the proposition contained in the node, i.e., `PROP`). For example, in the proof of `a1i` in Figure 1 (a), the subtree

above the node `ax-1` are two hypotheses `wffph` and `wffps`, and they constitute a proof of the proposition `|- (ph-> (ps->ph))` contained in the node `ax-1`.

In addition to the entire subtree above a node, one may identify some connected component of the tree as a valid theorem. For example, in Figure 1 (c), we show that the proof of the theorem `mp1i` contains an embedded proof of the theorem `a1i`. The embedded proof is colored in blue, and there is a one-to-one correspondence between these blue nodes and the nodes in the proof of `a1i` shown in Figure 1 (a). One can hence refactor the proof with an invocation of the theorem `a1i`, resulting in a much smaller tree shown in Figure 1 (b).

In general, there are certain criteria a component needs to satisfy to be identified as a valid proof of a theorem. In Appendix A.2, we develop such an algorithm in more detail that performs the verification. We will use that to verify the prediction given by a neural network model.

To conclude, in this section, we establish the equivalence between theorem extraction from a proof as to the extraction of a sub-component from a proof tree. This allows us to formalize the problem as a node-level prediction problem on graphs as we introduce next.

2.2 Problem Formulation

The model is given a proof tree $\mathcal{G}$ with a set of nodes $\mathcal{V}$, edges $\mathcal{E}$, and node features x_v which correspond to the name `N` and the proposition `PROP` associated with each node. The task of the model is to output a subset of nodes $\mathcal{V}_{\text{target}} \subset \mathcal{V}$ that correspond to an embedded proof of a useful theorem. We cast the problem as a node-level binary classification problem that predicts whether each node belongs to $\mathcal{V}_{\text{target}}$. Without loss of generality, we let all nodes in $\mathcal{V}_{\text{target}}$ to have labels of 1 and the rest 0.

We use a graph neural network parametrized by θ to take the graph and node feature as input, and outputs a scalar $\hat{P}_v$ between 0 and 1 for each node $v \in \mathcal{V}$, representing the probability belonging to $\mathcal{V}_{\text{target}}$. Our objective is a binary cross entropy loss between the node level probabilities and the ground truth target for a graph. Because the number of nodes usually vary significantly across proofs, instead of treating each node equally, we also normalize the loss by the number of nodes in the graph[1]:

$$\mathcal{L}(G, \theta) = -\frac{1}{|\mathcal{V}|} \sum_{v \in \mathcal{V}_{\text{target}}} \log P(\hat{P}_v = 1 | \mathcal{G}, \theta) \tag{1}$$

$$-\frac{1}{|\mathcal{V}|} \sum_{v \notin \mathcal{V}_{\text{target}}} \log P(\hat{P}_v = 0 | \mathcal{G}, \theta) \tag{2}$$

We then seek the best parameters by minimizing the loss over all proof trees:

$$\arg\min_{\theta} \sum_{G} \mathcal{L}(G, \theta). \tag{3}$$

2.3 REFACTOR: Theorem-from-Proof Extractor

With the problem formulated, we now describe how to generate training data points of proof trees $\mathcal{G}$ with suitable targets $\mathcal{V}_{\text{target}}$ defined. Even though we specialize our discussion in the context of Metamath, the same technique can be applied to most other formal systems for creating datasets of theorem extraction.

We think of the theorem as a function whose arguments are a set of hypotheses and the output is a conclusion, as mentioned in Appendix B.1. Instead of calling the theorem by its name, we intentionally duplicate the body of its proof tree, and manually replace their nominal arguments with the arguments we wish to pass in context. There are three key steps: 1. identifying the proof tree associated to the theorem (e.g., `a1i` in Figure 1 (a)), substituting nominal arguments with the ones in the proof context (e.g., substituting leaf nodes `wffph`, `wffps` and `|-ph` in Figure 1 (a) with nodes

[1] In our preliminary experiments we found the normalized loss gave better performance than treating each node equally.

Table 1: Node level and proof level accuracy of REFACTOR with various model sizes.

K, d, Number of Trainable Parameters	Training Node Accuracy	Training Proof Accuracy	Test Node Accuracy	Test Proof Accuracy
5, 64, 80k	89.4%	5.1%	77.4%	2.3%
5, 128, 222k	91.3%	9.9%	78.6%	3.0%
5, 256, 731k	93.7%	17.3%	80.1%	4.4%
10, 256, 1206k	97.5%	37.5%	84.3%	13.3%
10, 512, 4535k	97.9%	42.7%	85.6%	19.6%

Table 2: Theorem usage and their contribution to refactoring

	# Theorem Used	Total Usage	Average Usage	Max Usage	Average Number of Nodes Saved	Total Number of Nodes Saved
Expanded	670	147640	77.4	60705	196.7	375126
Original	14	11736	733.5	8594	2025.8	32413
Total	684	159376	82.9	60705	211.9	407539

`wffps`, `wffch` and `|-ps` in Figure 1 (b) respectively[2]), and finally copy and replace it to where the expanded node is located (e.g, replace `a1i` node in Figure 1 (b) with the substituted `a1i` to arrive at Figure 1 (c)). We present a more formal and detailed exposition of the algorithm in Appendix A.1.

3 Experiments

3.1 Q1 - How many human-defined theorems does the model extract?

We provide details on dataset preprocessing and model architectures in Appendix C.1 and C.2 respectively. On the theorem extraction dataset obtained from Appendix C.1, REFACTOR was able to correctly classify 85.6% (Node Accuracy) of the nodes. For 19.6% (Proof Accuracy) of the proofs, REFACTOR was able to correctly classify all of the nodes and fully recover the theorem that the human use. We also show that our approach scales well with the model size (Table 1). As we increase the model by around 50x from 80k to 4M, both node and proof accuracy improve. In particular, the proof accuracy goes up significantly from 2.3% to 19.6%. This shows promise that the accuracy can be further improved by using a larger model with a larger dataset. Additional analysis on what makes model perform well is provided in Appendix C.3.

3.2 Q2 - Can REFACTOR extract new useful theorems?

In this section, we investigate whether REFACTOR can extract new useful theorems. We used the best model (i.e., the largest model) in Table 1 for the results analyzed in this section. We explored two ways of extracting new theorems. We first investigated the incorrect predictions of REFACTOR on the theorem extraction dataset. When the prediction differs from the ground truth, it can correspond to a valid proof. We also applied REFACTOR on the human proofs of nodes less than 5000 from the library `set.mm`. In both cases, we used the algorithm developed in details in Appendix A.2 to verify whether a prediction leads to a valid theorem.

We extracted in total 1923 new theorems: 1907 from the expanded dataset, 16 from `set.mm`. We then computed the number of usages in `set.mm` for each newly extracted theorem, reported in Table 2. The average number of usages is 83 times, showing nontrivial usefulness of these theorems. Notably, the theorems extracted on `set.mm` are even more frequently used – 733.5 times on average. We think that because the human library is highly optimized, it is harder to extract new theorems from existing proofs. But a successful extraction is likely to be of better quality as the proof tree input represents a true human proof rather than a synthetically expanded proof. We provide results for model predictions where they do not constitute valid theorems in Appendix C.4.

3.3 Q3 - Can we improve a theorem library using the extracted theorems?

We evaluated the reusability of the extract theorems by measuring the compression in the size of the library that these new theorems would allow. Intuitively, when the new theorems are broadly reusable, we would expect the proofs in the library could be shortened by using the new theorems

[2]Note that these three nodes in Figure 1 (b) are parents, namely, arguments to `a1i` node in Figure 1 (b).

as part of the proofs. In this paper, we consider a specific re-writing procedure, which alternates between 1) matching the extracted theorems against the proofs in the library and 2) replacing the matched proportion of the proofs with the application of the new theorems.

With the 16 new extracted theorems from original dataset, the new library obtained from refactoring was indeed smaller (See Table 2). These new theorems on average saved 2025.8 nodes which is an order of magnitude more than those from the expanded dataset (196.7 nodes). Nevertheless, this shows that extracted theorems from both expanded and human datasets are frequently used in refactoring the theorem library. In total, we were able to refactor 14092 out of 27220 theorems in the MetaMath database. We demonstrate the usefulness of these refactored theorems in theorem proving in Appendix C.6.

4 Conclusion

In this paper, we study the problem of extracting useful theorems from mathematical proofs in the Metamath framework. As proofs are represented as *proof trees* in formal systems, we formalize theorem extraction as a node-level binary classification problem on proof trees. We propose one way to create datasets for the problem and additionally develop an algorithm to verify the validity of the prediction. We demonstrate that our best graph neural network model was able to extract unseen human theorem 19.6% of the time. When the model's prediction did not match the human theorem ground truth, we can additionally extract 1907 theorems from the dataset. We further applied the model on the existing Metamath library and found it was able to extract 16 new theorems, each was used 733.5 times on average in the entire Metamath database. After theorem refactoring, those 16 new theorems saved 32413 proof nodes of the entire dataset. Finally, by training the refactored proofs, we show a prover achieved better proof success rate on test theorems.

Our work represents the first proof-of-concept of theorem extraction using neural network models. We see there are various ways to improve the existing model, such as scaling up the model size, or using more powerful architectures such as transformers to autoregressively predict the target, all of which are left to future works. Lastly, we would like to note that our methodology is not only generic for formal mathematical theorem extraction, but also has the potential to be applied to other applications, such as code refactoring.

References

Jacob Andreas, Marcus Rohrbach, Trevor Darrell, and Dan Klein. Neural module networks. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 39–48, 2015.

Kshitij Bansal, Sarah M. Loos, Markus N. Rabe, Christian Szegedy, and Stewart Wilcox. Holist: An environment for machine learning of higher order logic theorem proving. In Kamalika Chaudhuri and Ruslan Salakhutdinov (eds.), *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, volume 97 of *Proceedings of Machine Learning Research*, pp. 454–463. PMLR, 2019a. URL http://proceedings.mlr.press/v97/bansal19a.html.

Kshitij Bansal, Christian Szegedy, Markus N. Rabe, Sarah M. Loos, and Viktor Toman. Learning to Reason in Large Theories without Imitation. *arXiv preprint arXiv:1905.10501*, 2019b.

Michael Chang, Abhishek Gupta, Sergey Levine, and Thomas L. Griffiths. Automatically composing representation transformations as a means for generalization. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019. URL https://openreview.net/forum?id=B1ffQnRcKX.

Eyal Dechter, Jonathan Malmaud, Ryan P. Adams, and Joshua B. Tenenbaum. Bootstrap learning via modular concept discovery. In Francesca Rossi (ed.), *IJCAI 2013, Proceedings of the 23rd International Joint Conference on Artificial Intelligence, Beijing, China, August 3-9, 2013*, pp. 1302–1309. IJCAI/AAAI, 2013. URL http://www.aaai.org/ocs/index.php/IJCAI/IJCAI13/paper/view/6890.

Kevin Ellis, Lucas Morales, Mathias Sablé-Meyer, Armando Solar-Lezama, and Josh Tenenbaum. Learning libraries of subroutines for neurally-guided bayesian program induction. In Samy Bengio,

Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett (eds.), *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, pp. 7816–7826, 2018. URL https://proceedings.neurips.cc/paper/2018/hash/7aa685b3b1dc1d6780bf36f7340078c9-Abstract.html.

Kevin Ellis, Catherine Wong, Maxwell I. Nye, Mathias Sablé-Meyer, Luc Cary, Lucas Morales, Luke B. Hewitt, Armando Solar-Lezama, and Joshua B. Tenenbaum. Dreamcoder: Growing generalizable, interpretable knowledge with wake-sleep bayesian program learning. *CoRR*, abs/2006.08381, 2020. URL https://arxiv.org/abs/2006.08381.

Alexander L. Gaunt, Marc Brockschmidt, Nate Kushman, and Daniel Tarlow. Differentiable programs with neural libraries. In *ICML*, 2017.

Thibault Gauthier, Cezary Kaliszyk, Josef Urban, Ramana Kumar, and Michael Norrish. Learning to prove with tactics. *CoRR*, abs/1804.00596, 2018. URL http://arxiv.org/abs/1804.00596.

M. Gori, G. Monfardini, and F. Scarselli. A New Model for Learning in Graph Domains. In *Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005.*, volume 2, pp. 729–734 vol. 2, 2005.

Christopher Hahn, Frederik Schmitt, Jens U. Kreber, Markus N. Rabe, and Bernd Finkbeiner. Transformers Generalize to the Semantics of Logics. *arXiv preprint arXiv:2003.04218*, 2020.

William L Hamilton, Rex Ying, and Jure Leskovec. Inductive representation learning on large graphs. *arXiv preprint arXiv:1706.02216*, 2017.

Daniel Huang, Prafulla Dhariwal, Dawn Song, and Ilya Sutskever. GamePad: A learning environment for theorem proving. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019. URL https://openreview.net/forum?id=r1xwKoR9Y7.

Drew A Hudson and Christopher D Manning. Compositional attention networks for machine reasoning. In *International Conference on Learning Representations (ICLR)*, 2018.

Cezary Kaliszyk and Josef Urban. Learning-assisted theorem proving with millions of lemmas. *J. Symb. Comput.*, 69:109–128, 2015. doi: 10.1016/j.jsc.2014.09.032. URL https://doi.org/10.1016/j.jsc.2014.09.032.

Cezary Kaliszyk, Josef Urban, and Jirí Vyskocil. Lemmatization for stronger reasoning in large theories. In Carsten Lutz and Silvio Ranise (eds.), *Frontiers of Combining Systems - 10th International Symposium, FroCoS 2015, Wroclaw, Poland, September 21-24, 2015. Proceedings*, volume 9322 of *Lecture Notes in Computer Science*, pp. 341–356. Springer, 2015. doi: 10.1007/978-3-319-24246-0_21. URL https://doi.org/10.1007/978-3-319-24246-0_21.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In Yoshua Bengio and Yann LeCun (eds.), *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015. URL http://arxiv.org/abs/1412.6980.

Guillaume Lample and François Charton. Deep learning for symbolic mathematics. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020. URL https://openreview.net/forum?id=Ske31kBtPr.

Wenda Li, Lei Yu, Yuhuai Wu, and Lawrence C. Paulson. Isarstep: a benchmark for high-level mathematical reasoning. In *International Conference on Learning Representations*, 2021. URL https://openreview.net/forum?id=Pzj6fzU6wkj.

Jiayuan Mao, Chuang Gan, Pushmeet Kohli, Joshua B. Tenenbaum, and Jiajun Wu. The neuro-symbolic concept learner: Interpreting scenes, words, and sentences from natural supervision. In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=rJgMlhRctm.

Bartosz Piotrowski and Josef Urban. Guiding Inferences in Connection Tableau by Recurrent Neural Networks. In Christoph Benzmüller and Bruce Miller (eds.), *Intelligent Computer Mathematics*, pp. 309–314, Cham, 2020. Springer International Publishing. ISBN 978-3-030-53518-6.

Stanislas Polu and Ilya Sutskever. Generative language modeling for automated theorem proving. *CoRR*, abs/2009.03393, 2020. URL https://arxiv.org/abs/2009.03393.

Markus N Rabe, Dennis Lee, Kshitij Bansal, and Christian Szegedy. Mathematical reasoning via self-supervised skip-tree training. *arXiv preprint arXiv:2006.04757*, 2020.

Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The Graph Neural Network Model. *IEEE Trans. Neural Networks*, 20(1):61–80, 2009. doi: 10.1109/TNN.2008.2005605. https://doi.org/10.1109/TNN.2008.2005605.

Josef Urban and Jan Jakubův. First Neural Conjecturing Datasets and Experiments. In Christoph Benzmüller and Bruce Miller (eds.), *Intelligent Computer Mathematics*, pp. 315–323, Cham, 2020. Springer International Publishing. ISBN 978-3-030-53518-6.

Qingxiang Wang, Chad Brown, Cezary Kaliszyk, and Josef Urban. Exploration of neural machine translation in autoformalization of mathematics in mizar. *Proceedings of ACM SIGPLAN International Conference on Certified Programs and Proofs*, 2020.

Daniel Whalen. Holophrasm: a neural automated theorem prover for higher-order logic, 2016.

Yuhuai Wu, Honghua Dong, Roger B. Grosse, and Jimmy Ba. The scattering compositional learner: Discovering objects, attributes, relationships in analogical reasoning. *CoRR*, abs/2007.04212, 2020. URL https://arxiv.org/abs/2007.04212.

Yuhuai Wu, Albert Jiang, Jimmy Ba, and Roger Grosse. INT: An Inequality Benchmark for Evaluating Generalization in Theorem Proving. In *International Conference on Learning Representations*, 2021. URL https://openreview.net/forum?id=O6LPudowNQm.

Kaiyu Yang and Jia Deng. Learning to Prove Theorems via Interacting with Proof Assistants. In *Proceedings of International Conference on Machine Learning (ICML)*, 2019.

A FURTHER EXPLANATIONS OF THE ALGORITHMS

A.1 THEOREM EXPANSION

We discuss our theorem expansion algorithm in this section. It is worth noting that the existing proofs from the Metamath library cannot be directly used because it does not contain meaningful targets. However, the human proofs can instead give us hints as to how to construct such data points. To illustrate, in Figure 1 (b), the proof of `mp1i` invokes a theorem application with `a1i`, which is a theorem that human considered useful and stored in the library. Our idea is to reverse the process of theorem extraction, by expanding the proof of `a1i` in the proof of `mp1i` to obtain a synthetic proof shown in 1 (c). In this expanded proof of `mp1i`, one can see the proof of `a1i` is embedded as a component colored in blue, hence creating a suitable target for theorem extraction.

An overview of the algorithm can be found in Algorithm 1. The algorithm takes input of two proof trees where the first proof tree uses the theorem that the second proof tree shows as one of the steps.

We explain our algorithm with the example from Figure 1. Specifically, proof tree T corresponds to Figure 1 (b) and proof tree T_s corresponds to Figure 1 (a). The theorem we want to expand is `a1i` and we first obtain all its arguments using `GetArguments` function. We treat each theorem as a function and its arguments are the hypothesis of the theorem used to compute the conclusion. Consequently, the nominal arguments are `wph`, `wps` and `a1i.1`. Next, we obtain contextual arguments, which are those specific hypotheses used in the context of the proof. Each hypothesis are represented by the entire subtree above each parent of c. Concretely, the contextual arguments of the `a1i` node in (b) are `wps`, `wch` and [`wph`, `wps`, `mp1i.a`, `mp1i.b`, `ax-mp`]. Here, we use square bracket to enclose a subtree that has more than one node, which is treated holistically as the third contextual argument. Note that we can clearly see a one-to-one correspondence between the nominal arguments and the contextual arguments: (`wph`→`wps`, `wps`→`wch` and `a1i.1`→[`wph`, `wps`, `mp1i.a`, `mp1i.b`, `ax-mp`]). We then simply replace all nodes in the proof tree of `a1i` using this mapping. This gives us [`wps`, `wch`, `wps`, `wi`, `wph`, `wps`, `mp1i.a`, `mp1i.b`, `ax-mp`, `wps`, `wch`, `ax-1`, `ax-mp`]. We generate its proof tree representation with `GetProof` function. Finally we replace the subtree above `a1i` with the new proof tree which in this case happens to be the entire proof of `mp1i` and this leads to the final expanded proof in Figure 1 (c).

Lastly, note that there are many options for theorem expansion. Firstly, one single proof can contain multiple theorems, and each theorem can be expanded either simultaneously or one by one. In addition, one can even recursively expand theorems by expanding the theorem inside of an expanded proof. For simplicity, in this work, we only expand one theorem at a time, and for every theorem in a proof. Hence, for a proof that contains M total number of theorem applications, we create M data points for learning theorem extraction. We leave investigations of more sophisticated expansion schemes to future work.

Algorithm 1 Theorem Expansion Algorithm Pseudocode

procedure EXPANSION
  Input: proof tree T that uses theorem s at node c.
  Input: proof tree of theorem s: T_s.
  nominalArguments = GetArguments(T_s)
  contextualArguments = [GetSubtree(p) for p in GetParents(c)]
  allNodeNames = GetAllNodeNames(T_s)
  f : nominalArguments $\rightarrow$ contextualArguments.
  $f(i^{th}$ element of nominalArguments) $\triangleq$ i^{th} element of contextualArguments
  for each name $N \in$ allNodeNames **do**
    if $N \in$ nominalArguments **then**
      replace N with $f(N)$
  replacedProof = GetProof(allNodeNames)
  replace entire subtree above node c with replacedProof
  return T

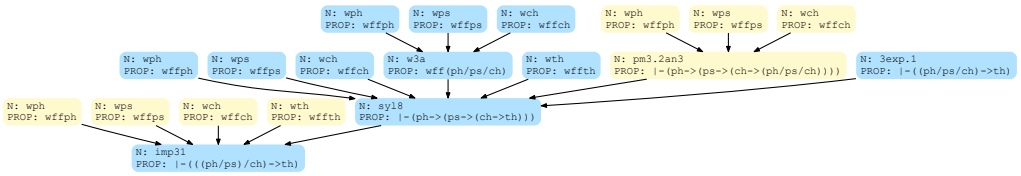
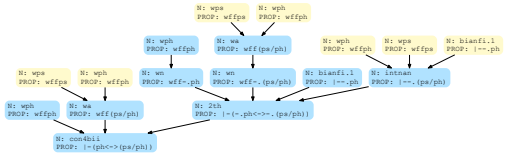
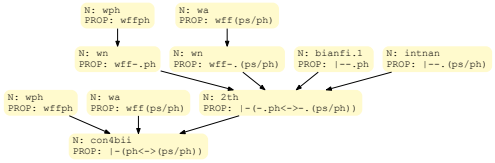


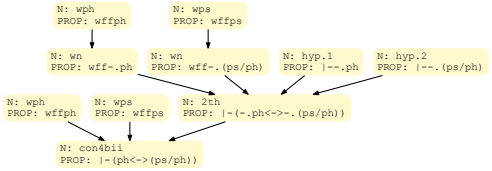
Figure 2: A proof tree prediction where nodes with output probability greater than 0.5 have been colored blue. This proof tree does not satisfy the constraint to be a valid theorem because only one of the parent nodes of the root are predicted to be in $\mathcal{V}_{target}$.



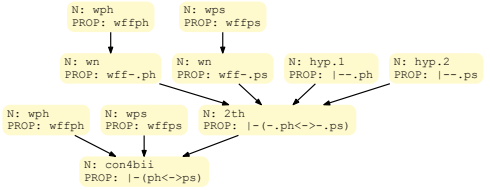
(a) A prediction made by REFACTOR with $\hat{\mathcal{V}}_{target}$ in blue.



(b) $\hat{\mathcal{V}}_{target}$ extracted from (a).



(c) $\hat{\mathcal{V}}_{target}$ extracted as in (b) with leaf node name and proposition replaced.



(d) A valid proof tree extracted and verified.

Figure 3: Visualization of theorem verification algorithm.

A.2 Theorem Verification

In this section, we present our algorithm to determine whether a predicted component made by REFACTOR constitutes a valid theorem. On a high level, our algorithm checks two necessary conditions and performs standardization before feeding the node names of extracted component to a verifier which we describe next.

We describe how we can verify Metamath proofs represented by a conclusion and a list of node names such as the ones seen in the previous section. This can be easily achieved by calling `GetProof` from Algorithm 1 on the list of nodes names which follow a Reverse Polish Notation (RPN), and the function call returns a proof tree labelled with propositions (i.e., `PROP`) . We then compare between the proposition given in the bottom node (conclusion) to the given conclusion specified by the theorem. The proof is verified if and only if the two conclusions are the same. We refer to this simple procedure as Metamath verifier.

For the theorem verification algorithm, We first take all node prediction with value greater than 0.5 as the set of extraction nodes, which we represent as $\hat{\mathcal{V}}_{target}$ (see Figure 3 (a) and (b)). We first check if $\hat{\mathcal{V}}_{target}$ forms a connected component i.e. a tree structure, as disjoint set of nodes cannot be a valid new theorem. Secondly, one necessary constraint for a valid extracted theorem is that for each node in $\hat{\mathcal{V}}_{target}$, either none or all of its parent nodes need to be present in $\hat{\mathcal{V}}_{target}$. If only some but not all parents are present, this corresponds to a step of theorem application with an incorrect number of arguments. We illustrate one example that violates this constraint in Figure 2. As seen in this example, only one parent of the root node is in $\hat{\mathcal{V}}_{target}$ and similarly one parent node of `syl8` is not in $\hat{\mathcal{V}}_{target}$. Because of these missing arguments, this will not be a valid new theorem. We note that although the extraction algorithm can be implemented in a way such that it "auto-completes" the

arguments by adding additional necessary nodes into the set of extracted nodes, we choose not to do so in order to make sure the submodule is entirely identified by REFACTOR.

Once the extracted nodes pass these checks, we perform a so-called standardization. Here we once again leverage functions defined in Algorithm 1. Specifically, we replace all node names of leaf nodes with a pre-defined set of node names allowed in Metamath such as `wph`, `wps`. This can be achieved by first obtaining arguments of the extracted component via `GetArguments` and replacing these arguments in a fashion similar to Algorithm 1 except this time the nominal arguments are from the extracted component and contextual arguments will be the pre-defined arguments from Metamath convention. As seen in Figure 3 (c), we replace all leaf node names `wa` with `wps`.

After standardization, we simply feed all the node names of the extracted component into the verifier we have described to determine whether it is a valid theorem. For example, node names in (c) [`wph`, `wps`, `wph`, `wn`, `wps`, `wn`, `hyp.1`, `hyp.2`, `2th`, `con4bii`] are fed into the verifier and we arrive at Figure 3 (d).

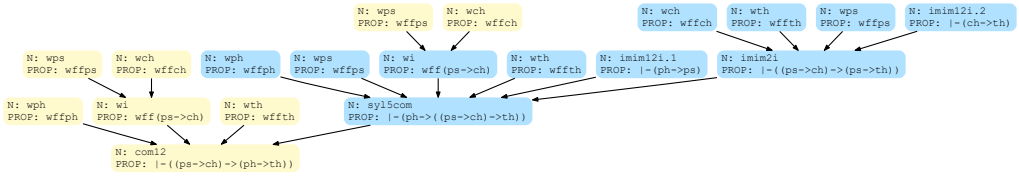


Figure 4: An example prediction that fails to be extracted as a new theorem due to no valid substitution plan in standardization. Specifically, the blue node `wi` cannot be substituted to a basic argument allowed in Metamath while still keeping the proof tree valid.

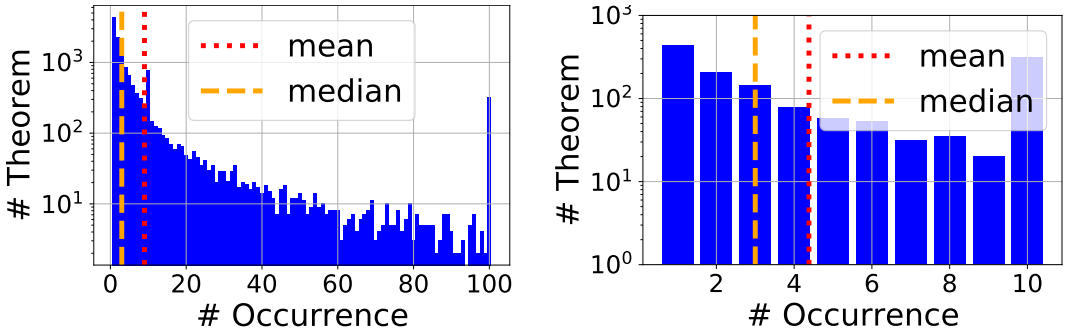
Intuitively, this standardization process can be thought of as an reverse process of the steps performed in proof expansion algorithm. Instead of replacing simple and basic nominal arguments with complex contextual ones, we use pre-defined simple contextual arguments from Metamath to replace the complex nodes in the extracted proof tree. We note that verifying a proof after standardization is not always possible. Consider an example in Figure 4 where the two parent nodes of blue node `wi` are not included in $\hat{\mathcal{V}}_{target}$ but in fact included in $\mathcal{V}_{target}$. Because of this, we need to replace `wi` with a basic argument in Metamath such as `wta`. However, with this replacement, the arguments of `syl5com` will no longer be valid because it needs an expression with two `wff` variables in the node we substituted. Therefore, there will be no valid substitution and this proof tree prediction cannot be extracted as a new theorem. We discard the extracted components that cannot be verified after standardization and only consider the ones that can be verified as new theorems.

B Background

B.1 Metamath and Proof Representation

In this section, we describe how one represents proof in the Metamath theorem proving environment. We would like to first note that even though the discussion here specializes in the Metamath environment, most of the other formal systems (Isabelle/HOL, HOL Light, Coq, Lean) have very similar representations. The fundamental idea is to think of a theorem as a function, and the proof tree essentially represents an abstract syntax tree of a series of function applications that lead to the intended conclusion.

Proof of a theorem in the Metamath environment is represented as a tree. For example, the proof of the theorem `a1i` is shown in Figure 1 (a). Each node of the tree is associated with a *name* (labeled as `N`), which can refer to a premise of the theorem, an axiom, or a proved theorem from the existing theorem database. Given such a tree, one can then traversing the tree from the top to bottom, and iteratively prove a true proposition (labeled as `PROP`) for each node by making a step of *theorem application*. The top-level nodes usually represent the premises of the theorem, and the resulting proposition in the bottom node matches the conclusion of the theorem. In such a way, the theorem is proved.



(a) Entire dataset statistics

(b) Test dataset statistics

Figure 5: Number of theorems vs number of occurrences of our dataset. Both (a) and (b) show noticeable occurrence imbalance with (b) being less due to our further subsampling of a maximum 10 occurrence.

We now define one step of theorem application. When a node is connected by a set of parent nodes, it represents a step of theorem application. In particular, one can think of a theorem as a function that maps a set of hypothesis to a conclusion. Indeed, a node in the tree exactly represents such function mapping, that is to map the set of propositions of the parent nodes, to a new conclusion specified by the theorem. Formally, given a node c whose associated name refers to a theorem T, we denote its parent nodes as $\mathcal{P}_c$. We can then prove a new proposition by applying the theorem T, to all propositions proved by nodes in $\mathcal{P}_c$.

The proof of the theorem `a1i` in Figure 1 (a) consists of 3 theorem applications. The top-level nodes are the hypotheses of the theorem. Most of the hypotheses state some expression is a well-formed formula so that the expression can be used to form a syntactically correct sentence. The more interesting hypothesis is `a1i.1`, that states `|-ph`, meaning `ph` is assumed to be true. In the bottom node, the theorem invokes the theorem `ax-mp`, that takes in four propositions as hypotheses, and return the conclusion `|-(ps->ph)`. In plain language, the theorem is a proof of the fact that if `ph` is true, then `(ps->ph)` is also true.

B.2 Graph Neural Networks

Graph Neural Networks (GNN) is a powerful class of architectures that is effective for representation learning over data with known graph structures (Gori et al., 2005; Scarselli et al., 2009). The input of a GNN is typically an augmented graph $G = (V, E)$ where each node $v \in V$ is augmented with a feature vector $h_v^{(0)}$. The GNN then maps these feature vectors $\{h_v\}_{v \in V}$ to a set of embedding vectors $\{h_v^{(K)}\}_{v \in V}$ through iterative applications of a neighbourhood aggregation function. In particular,

$$h_v^{(k)} = \mathrm{A}\Big(h_v^{(k-1)}, \sum_{u \in \mathcal{N}(v)} h_u^{(k-1)}; W^{(k)}\Big), \tag{4}$$

where $\mathcal{N}(v) = \{u | u \in V \wedge (v, u) \in E\}$, $\{W^{(k)}\}$ are weights of the GNN. The resulting node embedding vectors $\{h_v^{(K)}\}$ is able to incorporate the information of all its k-hop neighbours.

C Supplementary Experiment Results

C.1 Dataset and Pre-processing

We applied REFACTOR to create datasets from the main and largest library of Metamath, `set.mm`. In order to fairly compare prover performance reported from Whalen (2016), we used their version of `set.mm`, which contains 27220 theorems. We also filtered out all expanded proofs with more than 1000 nodes or contain nodes features of character length longer than 512. This gave rise to 257264 data points for training theorem extraction before theorem maximum occurrence capping, which we describe next.

Table 3: Node level and proof level accuracy of REFACTOR with different input configurations. **No edge**: all the edges in the graph are removed; **Leaves→Root**: only keep the edges are in the same direction of the paths that go from leaves to their parents; **Leaves←Root**: same as Leaves→Root except all the edges are all reversed; **Leaves↔Root**: the original graph with bidirectional edges. **Node Features**: whether or not the node features are fed as input to the model. All the experiments are run with $K = 10$ and $d = 256$.

	Training Node Accuracy	Training Proof Accuracy	Test Node Accuracy	Test Proof Accuracy
No edge + Node Features	86.8%	0.1%	74.9%	0.1%
Leaves→Root + Node Features	87.1%	0.5%	75.2%	0.1%
Leaves←Root + Node Features	96.6%	6.0%	88.1%	3.5%
Leaves↔Root	86.3%	0%	74.2%	0%
Leaves↔Root + Node Features (**REFACTOR**)	97.5%	37.5%	84.3%	13.3%

We noted that the distribution of theorem usage in `set.mm` is highly imbalanced. To prevent the model from learning to only extract a few numbers of common theorems due to their pervasiveness, we employed a subsampling of the data with respect to theorem occurrence to balance the dataset. Specifically, in the training set, for those theorems that occur more than 100 times as extraction targets, we subsampled 100 data points per theorem. In Figure 5 (a), we plot a histogram of theorem occurrence versus the number of theorems. As seen in the figure, the distribution roughly follows a power-law distribution with 4000 theorems only used once in `set.mm`, and a substantial number of theorems that occur beyond 100 times. For the validation and test set, as we wanted to evaluate the model on a diverse set of extraction targets, we capped the maximum number of occurrences as 10 using subsampling. The occurrence histogram of the test dataset is shown in Figure 5 (b) and the total number of expanded proofs in our dataset after capping theorem maximum occurrence is 124294.

To evaluate the model's generalization ability, we performed a target-wise split on the dataset. That is, we split the dataset in a way that the prediction targets, namely, the theorems to be extracted, are different for the train, valid and test set. By doing so, we discouraged simple memorization of common theorems and extracting them from unseen proofs.

C.2 Model Architecture and Training Protocol

In this section, we describe our neural network architecture parameters and other training details. We used a character-level tokenization for the node feature, which is a concatenation of texts in the fields `N` and `PROP` (see Figure 1). For each node, we first embedded all the characters with an embedding matrix, followed by two fully connected layers. We then averaged over all embeddings to obtain a vector representation of a node. We used these vector representations as the initial node embeddings to a graph neural network. We used K GraphSage convolution Hamilton et al. (2017) layers with size d and two more fully connected layers with sigmoid activation at the end to output the scalar probability. The size of the character embedding was set to 128 and the number of hidden neurons in all the fully connected layers was set to 64. Both K and d are hyperparameters.

For all of our model training, we used a learning rate of *1e-4* with Adam optimizer Kingma & Ba (2015). All methods were implemented in Pytorch[3] and Pytorch Geometric library [4]. We ran all experiments on one NVIDIA Quadro RTX 6000, with 4-core CPUs.

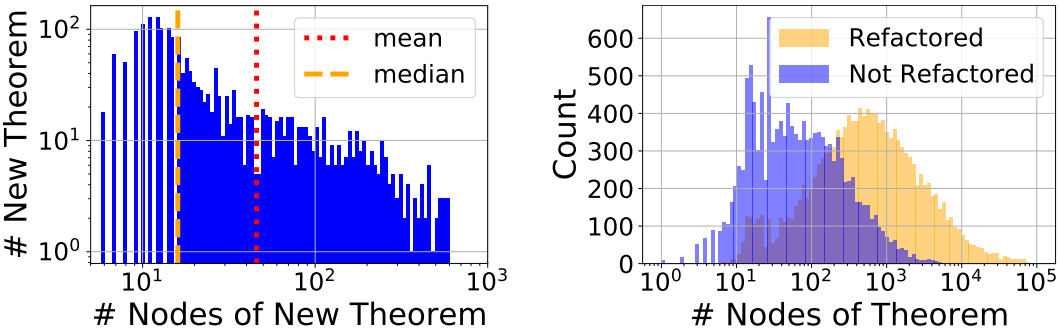
C.3 Q1 - How many human-defined theorems does the model extract?

To understand what mechanism in the GNN made the theorem extraction possible, we re-trained the model, but with different configurations compared to the original training procedure. In particular, we examined the case where all the edges are removed (No edge) as well as two types of uni-directional connections: 1) only edges that go from leaves to root are included (Leaves→Root) and 2) only edges that go from root to leaves are included (Leaves←Root). In addition, we were curious to see whether the graph structure alone is sufficient for theorem prediction when no node features are provided.

For all the experiments, we used a model with $K = 10$ and $d = 256$. We summarize the results of these data configurations in Table 3 and report node level and proof level accuracy on training and

[3]https://pytorch.org/

[4]https://pytorch-geometric.readthedocs.io/en/latest/



(a) (b)

Figure 6: (a) Distribution of number of nodes in new theorems extracted. The model mostly extracts short theorems but is also capable of extracting theorems that have hundreds of nodes. (b) Distribution of number of nodes of refactorable and not refactorable proofs. Refactorable proofs are generally longer than those that are not.

Table 4: An analysis of incorrect predictions on the theorem extraction dataset. We observe there are still substantial amount of predictions that lead to valid theorems.

Dataset	Total	Not Tree & Invalid	Tree & Invalid	Tree & Valid
Training	64349	13368	47521	3460
Validation	4766	1175	3238	353
Test	4822	1206	3348	328
`set.mm`	22017	8182	13470	365

test set. It can be seen that both edge connection and input node feature information is crucial in this task as both (No edge + Node Features) and (Leaves↔Root) achieved minimum proof level accuracy. Interestingly, the direction of edge led to a drastically different performance. Leaves→Root + Node Features performs poorly in proof level accuracy whereas Leaves←Root + Node Features achieved comparable performance with bidirectional edges (Leaves↔Root + Node Features).

This phenomenon can be explained by recognizing the fact that there are many identical hypothesis nodes in a proof due to MetaMath's low-level nature. For example, there are three identical leaf nodes `wps` in Figure 1 (c). If the edges only point from hypothesis to conclusion, the message for two identical hypothesis leaves will always be the same due to no incoming messages. Hence, it is theoretically impossible to make correct predictions on the proof level. On the other hand, the opposite direction of edges does not suffer from this limitation as there is only one root in the proof tree. Empirically, this configuration is able to achieve decent performance, but still far behind the performance of the model with bi-directional edges.

C.4 Q2 - Can REFACTOR extract new useful theorems?

The number of valid theorems from the incorrect predictions on the theorem extraction dataset, and the predictions on `set.mm` are listed under *Tree & Valid* in Table 4. We observe that there were a non-trivial amount of predictions that led to valid theorems. Remarkably, we see REFACTOR was able to extract valid theorems in the real human proofs (`set.mm`), despite the fact that human proof distribution may be very different from the training distribution. Adding up all extracted theorems from both approaches, we arrived at 4204 new theorems. We notice that among them, some new theorems were duplicates of each other due to standardization and we kept one copy of each by removing all other duplicates. We also removed 302 theorems extracted on `set.mm` that corresponded to the entire proof tree. In the end, we were left with 1923 unique new theorems with 1907 and 16 from the expanded and original dataset respectively. We showed examples of extracted new theorems in the Appendix C.5. We also plot the distribution of number of proof nodes of the extracted theorems in Figure 6 (a). We can see the newly extracted theorems are of various sizes, spanning almost two orders of magnitudes.

Table 5: Proof success rate comparison.

Setting	1 min	5 min
Holophrasm Whalen (2016)	-	14.3%
Holophrasm (ours)	11.5%	15.1%
REFACTOR	**13.1**%	**15.5**%

We additionally performed a more detailed analysis on the predictions, by classifying them into three categories. The first category is denoted by *Non-Tree & Invalid* where the prediction is a disconnected set of nodes and hence it is impossible to form a new theorem. In the second category *Tree & Invalid*, the prediction is a connected component and hence forming a sub-tree, but it still does not satisfy other conditions outlined in our algorithm description to be a valid proof of a theorem. The last category *Tree & Valid* corresponds to a prediction that leads to an extraction of new theorem previously not defined by humans. We present the number of predictions for each category in Table 4. Surprisingly, we noticed the model predicted a substantial amount of disconnected components. We hypothesize this may be because our current model makes independent node-level predictions. We believe an autoregressive model has a great potential to improve on this problem, and we leave it to future work.

C.5 Extracted theorems

In Figure 7, we show the top 10 most frequently used new theorems in refactoring. Among them, two are extracted from the original `set.mm` and the rest are extracted from the expanded dataset. It is worth noting that although these theorems generally have fewer than 10 nodes each, they in total contribute to more than 78% of total number of nodes saved in refactoring, suggesting the pervasiveness and reusability of these extracted theorems in `set.mm`.

C.6 Q3 - Are newly extracted theorems useful for theorem proving?

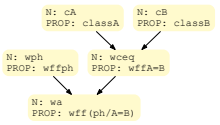
We further demonstrated the usefulness of our new theorems with an off-the-shelf neural network theorem prover, Holophrasm Whalen (2016). We trained two Holophrasm provers, one with the original dataset, and the other with the dataset augmented with the refactored proofs.

We evaluated the proof success rate in Table 5. We used the default values for all hyperparameters of the prover, and we evaluated proof success rates on a hold-out suit of test theorems. We report the results with the time limit of each proof search set to 1 and 5 minutes. Compared to the reported result in Whalen (2016) under a 5-minute limit, our re-implementation was able able to obtain a slightly higher success rate (15.1%). It can be seen that by training on the refactored dataset, the prover's proof success rate improved under both 1 and 5 min limits, demonstrating the usefulness of REFACTOR in theorem proving.

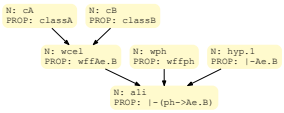
D Related Work

Lemma Extraction Our work is mostly related to the work of Kaliszyk & Urban (2015); Kaliszyk et al. (2015). The authors propose to do lemma extraction on the synthetic proofs generated by Automated Theorem Provers (ATP) on the HOL Light and Flyspeck libraries. They showed the lemma extracted from the synthetic proofs further improves the ATP performances for premise selection. However, their proposed lemma selection methods require human-defined metrics and feature engineering, whereas we propose a novel way to create datasets for training a neural network model to do lemma/theorem selection. Unfortunately, as the Metamath theorem prover is not equipped with ATP automation to generate synthetic proofs, we could not easily compare our method to these past works. We leave more thorough comparisons on the other formal systems to future work.

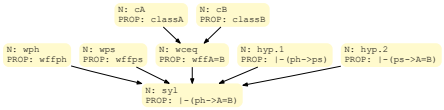
Discovering Reusable Structures Our work also is related to a broad question of discovering reusable structures and sub-routine learning. One line of the work that is notable to mention is the EC-style learning algorithms Dechter et al. (2013); Ellis et al. (2018; 2020). These works focus on program synthesis while trying to discover a library of subroutines. As a subroutine in



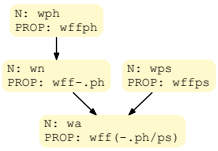
(a) Used 60705 times, from expanded dataset



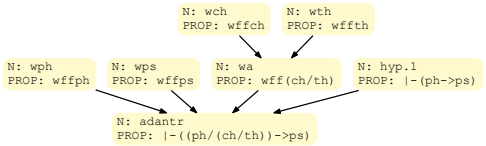
(b) Used 11375 times, from expanded dataset



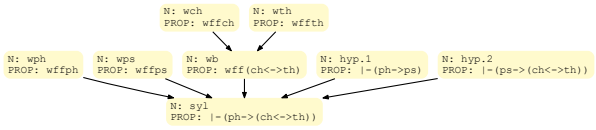
(c) Used 11125 times, from expanded dataset



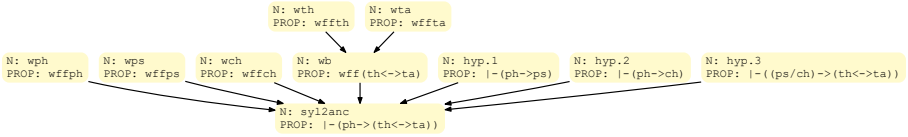
(d) Used 8594 times, from `set.mm`



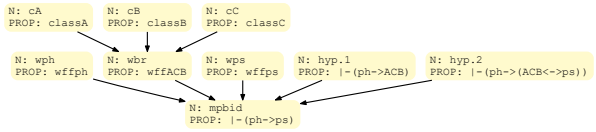
(e) Used 7241 times, from expanded dataset



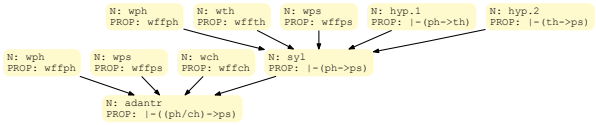
(f) Used 4693 times, from expanded dataset



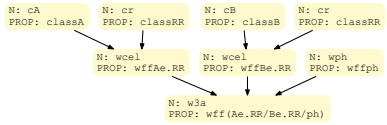
(g) Used 4437 times, from expanded dataset



(h) Used 3428 times, from expanded dataset



(i) Used 3376 times, from expanded dataset



(j) Used 2933 times, from `set.mm`

Figure 7: Top 10 most frequently used theorems in refactoring.

programming serves a very similar role as a theorem for theorem proving, their work is of great relevance to us. However they approach the problem from a different angle: they formalize subroutine learning as a compression problem, by finding the best subroutine that compresses the explored solution space. However, these works have not yet been shown to be scalable to realistic program synthesis tasks or theorem proving. We, on the other hand, make use of human data to create suitable targets for subroutine learning and demonstrate the results on realistic formal theorem proving. Another related line of work build inductive biases to induce modular neural networks that can act as subrountines Andreas et al. (2015); Gaunt et al. (2017); Hudson & Manning (2018); Mao

et al. (2019); Chang et al. (2019); Wu et al. (2020). These works usually require domain knowledge of sub-routines for building neural architectures hence not suitable for our application.

Machine Learning for Theorem Proving Interactive theorem provers have recently received enormous attention from the machine learning community as a testbed for theorem proving using deep learning methods (Bansal et al., 2019a;b; Gauthier et al., 2018; Huang et al., 2019; Yang & Deng, 2019; Wu et al., 2021; Li et al., 2021; Polu & Sutskever, 2020). Previous works demonstrated that transformers can be used to solve symbolic mathematics problems (Lample & Charton, 2020), capture the underlying semantics of logical problems relevant to verification (Hahn et al., 2020), and also generate mathematical conjectures (Urban & Jakubův, 2020). Rabe et al. (2020) showed that self-supervised training alone can give rise to mathematical reasoning. Li et al. (2021) used language models to synthesize high-level intermediate propositions from a local context. Piotrowski & Urban (2020) used RNNs to solve first-order logic in ATPs. Wang et al. (2020) used machine translation to convert synthetically generated natural language descriptions of proofs into formalized proofs.