Graph2Tac: Online Representation Learning of Formal Math Concepts

Lasse Blaauwbroek [*1] **Miroslav Olšák** [*2†] **Jason Rute** [*3] **Fidel Ivan Schaposnik Massolo** [1] **Jelle Piepenbrock** [4 5] **Vasily Pestun** [1 3‡]

Abstract

In proof assistants, the physical proximity between two formal mathematical concepts is a strong predictor of their mutual relevance. Furthermore, lemmas with close proximity regularly exhibit similar proof structures. We show that this *locality* property can be exploited through online learning techniques to obtain solving agents that far surpass offline learners when asked to prove theorems in an unseen mathematical setting. We extensively benchmark two such online solvers implemented in the Tactician platform for the Coq proof assistant: First, Tactician's online k-nearest neighbor solver, which can learn from recent proofs, shows a $1.72\times$ improvement in theorems proved over an offline equivalent. Second, we introduce a graph neural network, Graph2Tac, with a novel approach to build hierarchical representations for new definitions. Graph2Tac's online definition task realizes a $1.5\times$ improvement in theorems solved over an offline baseline. The k-NN and Graph2Tac solvers rely on orthogonal online data, making them highly complementary. Their combination improves $1.27\times$ over their individual performances. Both solvers outperform all other general-purpose provers for Coq, including CoqHammer, Proverbot9001, and a transformer baseline by at least $1.48\times$ and are available for practical use by end-users.

1. Introduction

Formal libraries of mathematics are structured hierarchically. Definitions build on top of each other and lemmas are proven with the help of other lemmas. On a high level, entire formalization projects concerning different mathematical topics may also depend on one another. The result is an enormous web of formal mathematical knowledge. In its center, one finds the most foundational parts of mathematics such as logic and arithmetic. Towards the outskirts of the web, the mathematics becomes increasingly specialized, as knowledge is built towards specific goals such as proving the four-color theorem (Gonthier, 2008). As the web of formal mathematics matures, its core stabilizes into static knowledge while the outskirts are ever-changing, as new insights emerge, connections are made and new theorems are developed.

A good mathematician, whether human or artificial, should possess a deep and intuitive understanding of the static, foundational core of mathematics. This is gained through training and practice (in the case of an AI, on a large, offline corpus). However, a mathematician must also have the capability to quickly gain a working understanding of the concepts, lemmas, and proving techniques in a new mathematical domain. This is a learned skill. An experienced mathematician can grok a new math concept from its definition and perhaps by looking at a few examples. For an AI, this amounts to "online representation learning": It must have a procedure to quickly build new representations for concepts, without having to retrain from scratch.

In this work, we explore online representation learning for the Coq proof assistant (The Coq Development Team, 2023) through the Tactician platform (Blaauwbroek, 2023; Blaauwbroek et al., 2020b). This platform provides a large-scale dataset and machine-to-machine interaction protocols that give an external proving agent (solver) online access to the full internal formal knowledge base loaded into Coq. We evaluate the capabilities of two online solvers, Tactician's built-in k-nearest neighbor (k-NN) solver (Blaauwbroek et al., 2020a; Zhang et al., 2021) and a novel model called Graph2Tac against a number of baselines.

Figure 1 summarizes the the paradigm of both online solvers. During offline training, the model develops a deep understanding of the mathematics found in a static Coq dataset. It also learns to represent new information in an online manner. During evaluation, Graph2Tac incorporates information

[*]Equal contribution [1]IHES [2]University of Cambridge [3]IBM Research [4]Radboud University [5]Czech Technical University in Prague †Work performed at IHES ‡Work performed while at IHES and IBM Research until June 2022. Correspondence to: Lasse Blaauwbroek <lasse@blaauwbroek.eu>, Jason Rute <jason.rute@ibm.com>.

Proceedings of the 41st International Conference on Machine Learning, Vienna, Austria. PMLR 235, 2024. Copyright 2024 by the author(s).

from new definitions and theorems, and the online k-NN incorporates information from new proofs. When the user invokes Tactician's `Suggest` command, the model uses the current state to suggest a next proof step, called a *tactic* in Coq. Each tactic is made up of an instruction followed by arguments. When the user invokes Tactician's `synth` tactic, a solver searches for a full proof of the current theorem, calling the model at each step of the search.

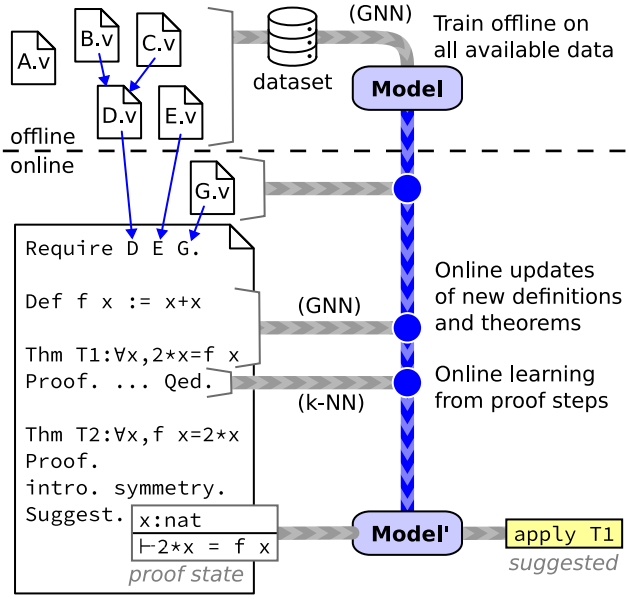

Figure 1. Our two paradigms of online learning: the online k-NN learns from recent proofs and the GNN (Graph2Tac) from recent definitions.

The two online solvers we discuss in Section 3, k-NN and Graph2Tac, operate on entirely different kinds of online data and as a result have orthogonal capabilities (discussed in Section 3.4). The k-NN solver can learn on-the-fly from tactic scripts written by users. As a result, it can make highly relevant suggestions when the proof of the current lemma bears similarities to the surrounding proofs.

Graph2Tac, shown in Figure 2, is a novel graph neural network (GNN) that, in addition to standard offline learning capabilities, contains a novel *definition embedding neural network* and *task*. Our goal is to provide representations for definitions (including theorems) both seen during training and inference. For definitions seen during training, the model learns a table of definition embeddings (center of the figure). These are used for representing definitions in proof states, where the graph of each proof state is used by the proof step neural network to suggest a tactic for that proof state. They are also used for representing lemmas and other possible tactic arguments, and are incorporated into argument prediction. For new definitions, not seen during training, the definition embedding neural network computes an embedding from the definition graph. In order to make sure these calculated embeddings agree with the learned embeddings, we co-train with a definition embedding task (upper right of the figure) which keeps them aligned. Each calculated embedding is hierarchical, depending not only on the graph of this definition, but the representations of component definitions.

Contributions We demonstrate the crucial importance of incorporating online information into models. Comparing an offline and online version of the k-NN solver demonstrates a proving performance improvement from 15.0% to 25.8% (1.72x). Section 3.2 introduces a new definition training task for GNNs that improves performance from 17.4% to 26.1% (1.5x). Our Graph2Tac model is based on an encoding of Coq's mathematical knowledge as a single interconnected mono-graph (Section 2) that enables calculating hierarchical representations of definitions. Due to integration with the Tactician platform, Graph2Tac is the first neural network model conveniently available to Coq end-users. Graph2Tac and k-NN are highly complementary, together proving 33.2% of the test set.

We compare the online models to a variety of other general-purpose solvers for Coq, including a transformer baseline, CoqHammer, Proverbot9001, ASTactic, TacTok and Passport (Section 3.3). Both online solvers prove at least 1.48x more theorems than any of the comparisons, making them the strongest by some margin. An extensive analysis of our experiments is in Section 4 and several appendices. Our code is publicly available, and every step of our work is fully reproducible (Appendix A).

Background and Related work This work fits into a growing body of work on machine learning and theorem proving. For a full account, see Appendix C. While the online capabilities of existing models are fairly limited, one common source of online information is argument selection (also called premise selection or lemma selection), where the model selects useful lemmas or definitions arguments for either the current theorem, proof state, or tactic. In argument selection, the proof state and lemmas are encoded via similar methods and the lemmas most related to the proof states are selected. Proof states and lemmas can be represented by hand-crafted features (Gauthier et al., 2017), S-expressions (Bansal et al., 2019; Sanchez-Stern et al., 2020), trees (Yang & Deng, 2019), pretty printed text (Yang et al., 2023), or graphs (Paliwal et al., 2020). They are then encoded with appropriate machine learning models. Transformer-language based approaches (Han et al., 2021) even forgo argument selection, synthesizing lemma arguments directly from memory of the training data. While convenient, this removes the ability to predict online arguments.

Tactician (Blaauwbroek et al., 2020b) uses a different online approach, relying on online classifiers (including the afore-

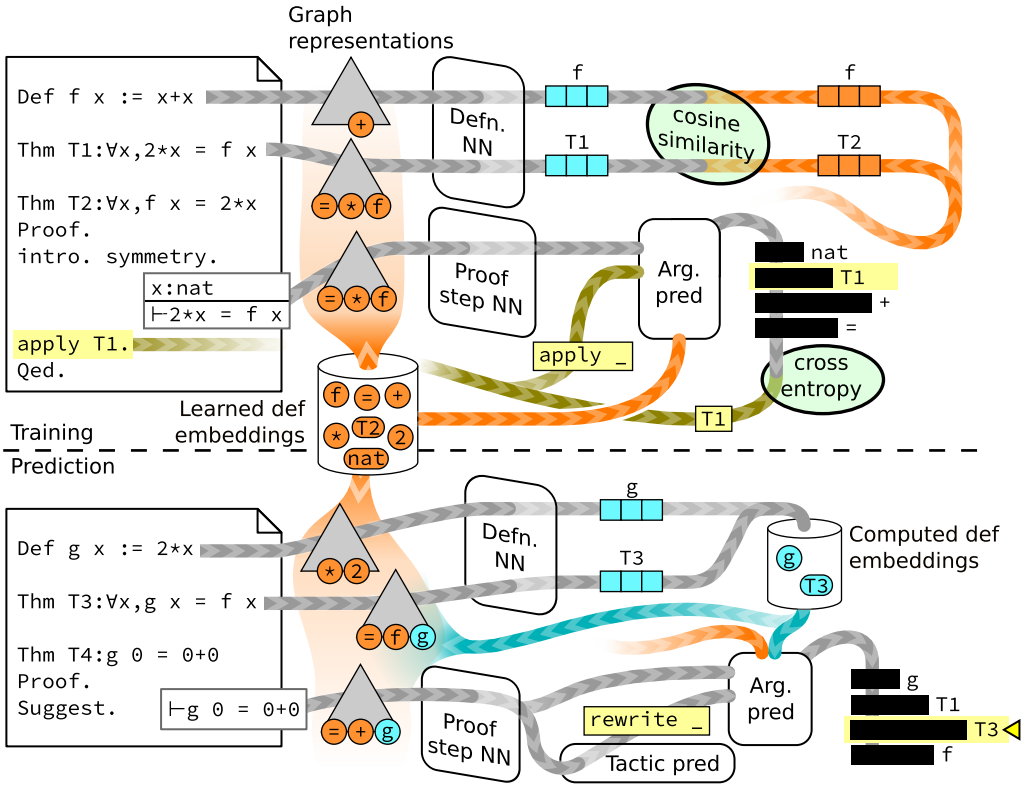

Figure 2. Our novel definition training task. At the center is a table of learned definition embeddings (orange). During training (above the line), we train both a proof step network to predict tactics (including arguments) and a definition network to calculate definition embeddings (blue) similar to those in the table. During inference (below the line), the definition network is used to calculated hierarchical embeddings (blue) for new definitions that may in turn be used to represent proof states and predict tactic arguments.

mentioned online k-NN) to select tactic commands from available proof data.

However, none of these methods incorporate online definition information, in neither the representations of proof states nor lemmas. While some models may use the definition name either as a token or a string (Sanchez-Stern et al., 2023), they do not "look inside" the definition. This is the main novelty of Graph2Tac's hierarchical definition representation.

2. Graph-based Interaction with Coq

In this work, we use a new graph-based data format to communicate between solvers and Coq. In this format, all Coq definitions and proof states are stored in one large interconnected graph (Blaauwbroek, 2024; 2023). Figure 3 shows a subset of this large graph that represents the proof state and environment at the start of the `Suggest` command in Figure 1. This novel graph encoding provides the following advantages: (1) The graph is designed to faithfully encode all kernel objects in Coq's environment in a reversible manner (2) References to global definitions, such as + in Figure 3, are explicitly referenced by edges to the definition node, avoiding any ambiguity with name resolution. (3) Local variables are connected to their binders, *e.g.* $\forall$, λ, via edges. This eliminates the need for local variable names. (4) By using a novel graph hashing algorithm (Blaauwbroek et al., 2024), equal terms are shared across parts of the graph leading to more compact representations.

This mono-graph can contain hundreds of millions of nodes and is too large for a graph neural network. See Appendix D for an illustration. Instead, our models hierarchically process subgraphs representing proof states and definitions. Each proof state and definition has a root node in the large graph. To obtain a smaller graph, we calculate the forward closure from the root node, stopping when a definition node is encountered. Theorems are special cases of definitions where the root node references both the theorem statement and its proof term. We omit the proof term from the theorem's definition graph to reduce the graph size.[1] In Figure 3 each subgraphs received a different color. Each color corresponds to either T_1, $=$, f, $+$, $\mathbb{N}$, or the current proof state of theorem T_2, which is still under construction. Notice that

[1]This is justified by the principle of *proof irrelevance*: To use a theorem one does not need to know its proof.

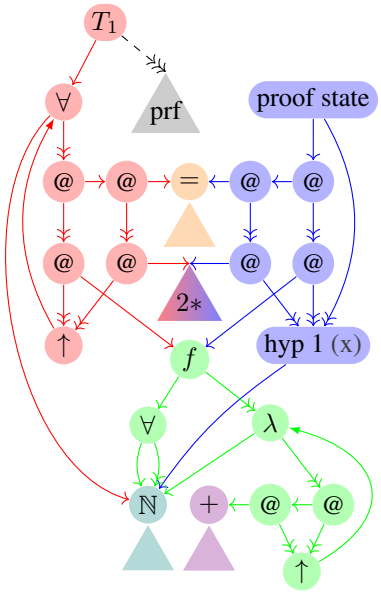

Figure 3. A graph-based representation of the example given in Figure 1. Each color represents a unique definition or proofstate subgraph. These colored subgraphs are used as inputs to our neural networks. Green corresponds to the function f, red to the theorem $T1$, and blue to the proof state from the example. Each node is either a definition node ($T1$, f, $\mathbb{N}$, $=$, $+$) or a node representing the part of a term. For example, @ is function application, and $\uparrow$ is a variable whose outgoing edge points to its binder. Edges are labeled, but not shown here. Triangles represent portions of the graph which we didn't expand for clarity. Some nodes are shared by multiple colored regions such as the subgraph for $2*$. Definition nodes (but not the rest of the definition) are also shared with all colored regions they touch. The proof terms for $T1$ is excluded from its colored region. See Blaauwbroek (2024) for full details.

T_1 and the proof state share the subterm associated with $2*$ (not drawn in full detail).

Mutually recursive definitions, such as inductive datatypes, have multiple root nodes that reference each other. Such *definition clusters* are treated as a single unit. The subgraphs extracted from the mono-graph are topologically ordered according to their dependencies, so that they can be processed in an appropriate order by the neural network.

To train the model, a large dataset of proofs is extracted where each proof state has an associated tactic. A tactic such as `rewrite plus_comm in H` is decomposed into a *base tactic* `rewrite _ in _`, and arguments `plus_comm` and `H`. While arguments may be arbitrary Coq terms and even other kinds of objects, our prediction model only predicts local hypotheses or global definitions. See Appendix D for more information on the dataset.

The Tactician `synth` tactic and `Suggest` command can communicate via this graph format with a Python client running a machine learning model. Tactician sends the entire mono-graph of global definitions along with the current proof state. The client returns a list of suggested tactics and scores. This integration makes our solver usable in practice and allows us to perform a massively parallel benchmark of our model on any Coq Opam package.

3. Description of Solvers

Here, we describe the solvers that are part of the comparison in this paper, including the online k-NN in Section 3.1, our novel Graph2Tac architecture 3.2 and a number of baseline comparisons in Section 3.3.

3.1. Online k-Nearest Neighbor

The fastest practical solver currently available in the Tactician framework is an online k-nearest neighbor (k-NN) model (Blaauwbroek et al., 2020a). It builds a database of proof states and associated tactics and extracts hand-crafted features from those proof states (Zhang et al., 2021). When the model is queried, it looks up proof states with the most similar features to the current proof state and returns the corresponding tactics, ordered by similarity. It does not require any training. The online k-NN uses a simple search over the 1000 most recent tactic proof steps in the global context. We also compare to an offline version that searches over all currently loaded proofs that are part of our training data. The offline k-NN uses locality sensitive hashing forests (LSHF) to speed up the search, similar to the k-NN solver found in Tactician[2]. See Appendix L for additional k-NN variations and ablations.

3.2. Graph Neural Network and Definition Task

Graph2Tac primarily consists of two parts, a *definition task* and a *prediction task*, shown in Figure 4. The input to each of these is a directed graph with labeled nodes and edges representing either a definition for the definition task or a proof state for the prediction task. We additionally associate metadata with each graph. Definition graphs have *root nodes* (node `f` in the figure), one for each definition defined in that graph. Inductive data types define multiple mutually dependent concepts in the same graph, *e.g.* `bool`, `true`, `false`, and may therefore have multiple roots. A proof state graph comes with *local context nodes*, one for each hypothesis. Both types of graphs contain *definition nodes*, which are leaf nodes that reference definitions in the global context. With each definition node, we store an index to the definition embedding table described in the next paragraph. For the prediction task, in addition to the proof state graph, the input includes the indices of all definitions

[2]We do not limit the offline k-NN to 1000 examples because the obvious "recency" selection criterion does not apply.

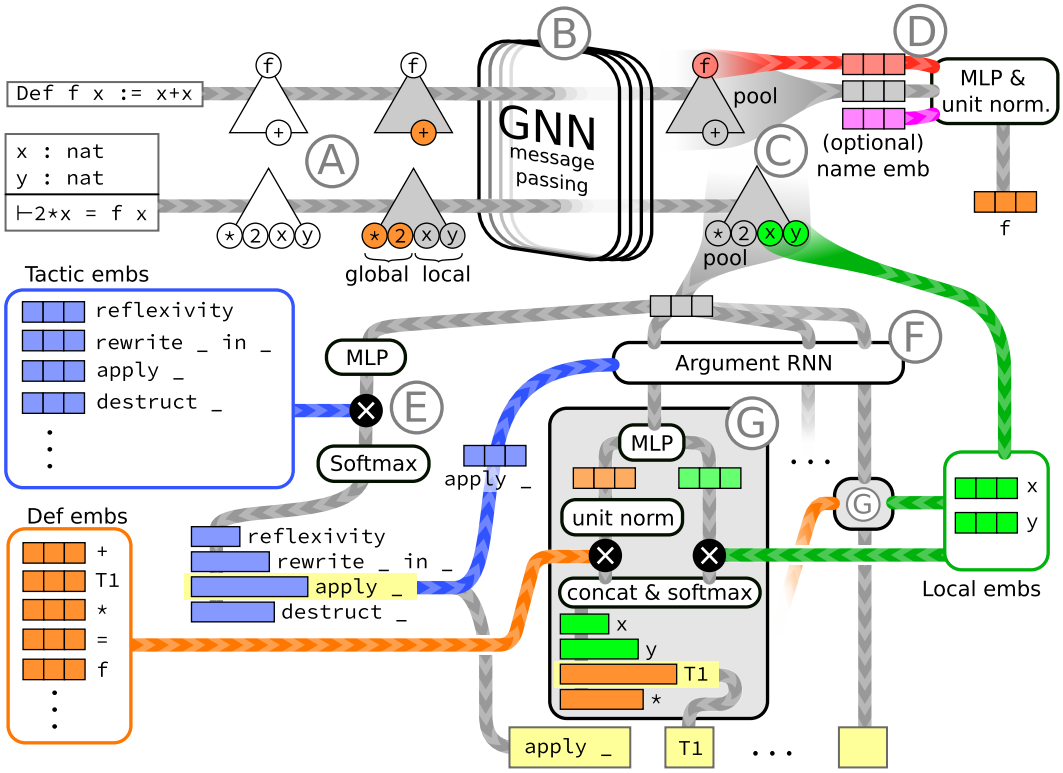

Figure 4. Detailed Graph2Tac model architecture, composed of a definition task (the top path going from the definition of f to its embedding) and prediction task (the path going from the proof state $2 * x = fx$ to the predicted tactic `apply T1`). The edge and node embedding tables are excluded from the image.

and tactics currently available in Coq's global context.

There are four learned embedding tables: edge labels and node labels (not shown in Figure 4), base tactics that occur in the training dataset, and definitions that occur in the training dataset. All embeddings have dimension equal to the model's hidden dimension, $h_{\text{dim}} = 128$. The definition and node embeddings are constrained to be unit-normalized. During inference, the definition table will be dynamically updated with new definitions as discussed later.

We transform the input graph, be it a proof state or definition graph, using the following approach (Figure 4, step A). Each graph is pruned to 1024 nodes. For effective message passing, we duplicate all edges to go in both directions and also add self-edges. The edges are assigned edge embeddings from the corresponding embeddings table. This table has $2E + 1$ entries, accounting for the E original edge labels, the E reverse edge labels, and the self-edge label. Non-definition nodes are assigned embeddings from the node embeddings table based on their node label. Definition nodes are instead assigned embeddings from the definition embeddings table, except for the root nodes in the definition task. Those are masked with null embeddings, as the goal of the definition task is to predict the corresponding definition embeddings.

The transformed graphs are put into a message-passing GNN (Figure 4, step B). The GNN consists of 8 hops where the tth hop transforms a node embedding x_n^t according to the following two steps:

$$\hat{x}_n^{t+1} = \text{ReLU}\left(\frac{1}{\deg(n)} \sum_{m,e\colon m \xrightarrow{e} n} \text{Dense}_{\theta_t}(e, x_m^t)\right)$$

$$x_n^{t+1} = \text{Layernorm}\left(x_t + \text{Dropout}_{0.1}(\text{MLP}_{\psi_t}(\hat{x}_n^{t+1}))\right)$$

The first step is a graph convolution layer where each node embedding is updated according to incoming edges from neighboring nodes. Here, $\deg(n)$ is the number of incoming edges $m \xrightarrow{e} n$ with target n and edge embedding e. The dense layer has output dimension h_{dim}. The next step consists of a 2-layer MLP (Dense, ReLU, Dense) with an inner hidden dimension of $2h_{\text{dim}}$, then Dropout, Residual, and Layernorm. The weights for each hop are separate, but the definition and prediction tasks both use the same GNN backbone, sharing the same weights.

The output of the GNN is a graph with the same structure as the input, but with updated node embeddings. Both the

definition and prediction tasks use mean pooling (Figure 4, step C) to obtain a single vector embedding for the graph. For the definition task, the pooled embedding is concatenated with each of the embeddings of the root nodes for the definition, then fed into a two-layer MLP, and finally unit normalized (step D). Optionally, along with each root node embedding, we additionally concatenate a name embedding for the definition, using a bidirectional LSTM (not-shown) to embed the fully qualified Coq identifier string, *e.g.* "Coq.Init.Logic.and". For the prediction task, the pooled embedding is fed into a two-layer MLP (step E). The output is multiplied by the tactic embedding table using inner product and put through a softmax layer to obtain the base tactic probabilities.

To predict the arguments for a tactic, we use a simple two-layer RNN (Figure 4, step F). The initial hidden state of the RNN is the embedding for the chosen tactic in the tactic embedding table. (During training the ground truth tactic is provided as input. The decoding method used during inference is described below.) The number of steps of the RNN corresponds to the number of arguments required by the given tactic.

Each argument is predicted as follows (Figure 4, step G). The RNN output is fed into a pair of two-layer MLPs (Dense, ReLU, Dense), resulting in a pair of query embeddings, one for global argument prediction and one for local argument prediction. The global argument prediction query is unit normalized. For global arguments, we take the inner product of the global argument query with each definition in the embedding table resulting in a logit for each definition. Since the inner product is bounded (as both vectors in the inner product are unit normalized), we scale the global logits by a learned temperature parameter. For each local argument, we use the GNN output embedding for that local node in the graph and take the inner product with the local argument query. Concatenating the local and global logits and performing softmax we get a distribution over all possible arguments.

Since many of the calculations used in our model take place on variable-size lists of objects, *e.g.* local arguments, our implementations rely heavily on ragged tensors in TensorFlow (Abadi et al., 2015) and graph tensors in TF-GNN (Ferludin et al., 2022).

We train on batches of 512 definitions and 512 proof states. The loss for the definition task is cosine similarity.[3] The loss for the training task is the cross-entropy for the full tactic sequence. For example, for the tactic `apply T1` the loss is $-\log P(\texttt{apply _}) - \log P(\texttt{T1} \mid \texttt{apply _})$ using the probabilities coming from the model. The combined loss of the two tasks is $\mathcal{L} = 1000\,\mathcal{L}_{def} + \mathcal{L}_{tactic}$.

[3] If a definition graph contains multiple definitions, the loss is divided by $\sqrt{n}$ where n is the number of entry points.

During inference, the tactic embedding table is masked for tactics which both occur in the training data and are available in the Coq's current state. Similarly, the definition embedding table is masked for all definitions available in the current global context. However, for new definitions not seen during training, we first calculate an embedding using the definition task. If there are multiple new definitions, we compute embeddings from each definition graph individually, updating the embeddings in a topologically sorted order so that those for dependencies are computed before those for latter definitions which depend on those dependencies.

At inference time, the output of the tactic task is a list of tactic suggestions, where each sequence starts with a base tactic, *e.g.* `apply _` and then contains the arguments for that tactic, if they are required. We use beam search decoding with a beam width of 256 to generate 256 tactic suggestions for each proof state.

We train three models: G2T-Named uses the name embedding in step D, whereas G2T-Anon does not. G2T-NoDef is trained without a definition task. Each of these is run with three configurations on the definition model during inference: The Recalc configuration calculates embeddings for *all* definitions, Update only calculates embeddings for *new* definitions—*i.e.* those not seen during training, and Frozen uses random unit normalized embeddings in place of the definition model. G2T-NoDef is only used as G2T-NoDef-Frozen. G2T-Anon-Update is the primary model.

3.3. Baselines for Comparison

We formally compare the Graph2Tac and k-NN models with three baselines outlined below. Note that some obvious baselines, such as SMTCoq (Armand et al., 2011) and Itauto (Besson, 2021) were excluded because they are not general purpose solvers. Furthermore, the online random forest described by Zhang et al. (2021) is excluded because it does not scale to large formalizations.

`firstorder auto with *` As a baseline, we use Coq's built-in `firstorder` reasoning tactic combined with the programmable proof search tactic `auto`, with all available hint databases.

CoqHammer (Czajka & Kaliszyk, 2018) translates theories expressed in Coq to a first-order language understood by the external automated theorem provers Vampire, CVC4, Z3 and Eprover. Once an external proof is found, the premises required for the proof are extracted, and the proof is reconstructed inside of Coq through the `sauto` family of higher-order solving tactics (Czajka, 2020).

Proof State Text-based Transformer We implement a

decoder-only transformer baseline that operates on the textual representations of the proof states, and predicts a textual representation of the tactic. We use the GPT2 implementation available via the Transformers Python library (Wolf et al., 2020). The embedding size is set to 768 and it has 12 layers, as in one of the models described in that paper. Our approach is similar to that of Han et al. (2021); Jiang et al. (2022); Yang et al. (2023), except our transformer models are trained from scratch only on Coq proof data.

Informal Comparisons Appendix I contains informal comparisons to highly relevant neural solvers such as Proverbot9001 (Sanchez-Stern et al., 2020), ASTactic (Yang & Deng, 2019) and their derivatives. Due to technical differences, a direct comparison was not achievable. Nonetheless, we are confident that Graph2Tac and k-NN significantly outperform these solvers for two reasons. First, CoqHammer is known to outperform these neural solvers. Given that Graph2Tac and the k-NN significantly outperform CoqHammer, we can conclude that they also surpass the other neural solvers. Second, Appendix I contains informal experiments that directly compare to the k-NN solver.

3.4. Capabilities of Graph2Tac vs k-NN vs Transformer

The Graph2Tac, online k-NN, and transformer solvers are all based on the Tactician platform. Nevertheless, their input and output capabilities are quite different. The online k-NN model sees tactics and their arguments as a black box. It can only predict the *exact* tactic expressions available in its database. Its power comes from speed and the ability to amend its database with new proof scripts. Graph2Tac uses a fixed set of tactics and cannot learn from new proof scripts. However, it can choose lemmas and hypotheses as arguments for these tactics. Due to its definition embedding task, it can predict lemmas not available in the training set. Finally, the transformer has no online learning capabilities. However, unlike the k-NN and Graph2Tac, it can generate tactics with arbitrary terms as arguments (but it must "hallucinate" references to new lemmas).

4. Evaluation

Experimental Setup To evaluate the performance of the solvers described above, we randomly choose 2000 theorems from the Coq packages in our testing set. The solvers are given a time limit of 10 minutes per theorem to synthesize a proof. To look at package-specific results, we sample up to 500 test theorems per package with a time limit of 5 minutes. The search procedure utilized by the solvers based on the Tactician framework is a modification of Dijkstra's algorithm that performs iterative deepening in order to relieve memory pressure and has an optimization that avoids backtracking between independent subgoals. For more information, see Appendix G.

During evaluation, each solver is limited by the operating system to one CPU with two hyperthreads. All processes, including Coq and any external processes such as neural networks and ATP's, must share this CPU. An exception is made for the Transformer-GPU solver, which is permitted to utilize a dedicated GPU.

We explore two artificially aggregated solvers, that simulate running n solvers concurrently while still adhering to the one-CPU computational budget. The number of theorems "solved" by the aggregate solvers in time t is the number solved by any of the components in time t/n. "G2T-Anon-Update + k-NN-online" is intended to simulate a solver capable of learning both from new definitions in the global context and new tactic proof scripts. "CoqHammer combined" is an aggregate of all four ATP backends and the reconstruction tactic called `best`. This simulates the intent of CoqHammer's authors to run many backends in parallel, while maintaining a fair comparison with other solvers.

Results The left plot of Figure 5 shows the fraction of test theorems solved over time for various solvers. One can see both the progress over time and an indication of the startup cost of each solver. The right plot of Figure 5 replaces time with the number of calls to the underlying model, giving an indication of the usefulness of that model's predictions in search irrespective of model speed.[4]. This and other plots remove the results from the *hott* package because HoTT replaces Coq's built-in standard library, upon which CoqHammer depends. The *tlc* package is also removed because G2T-Anon-Update was able to exploit an inconsistency as explained in Appendix Q.

Among the k-NN solvers, the online k-NN solves many more theorems (25.8% vs 15.0%) showcasing the importance of incorporating of recent online proof data. (Further k-NN ablations are found in Appendix L.)

Among the G2T solvers, the online solvers using the definition task to compute embeddings for new definitions outperform the G2T-NoDef-Frozen baseline (26.1% vs 17.4%) which does not compute embeddings for new definitions. Note our baseline G2T-NoDef-Frozen performs similarly to the transformer baselines in terms of model calls (and even better in per-time performance). This suggests that our baseline is at least comparable to the common transformer baselines in other papers. The addition of names in G2T-Named-Update fares slightly worse than the main G2T solver G2T-Anon-Update. (Further G2T ablations are found in Appendix N.)

CoqHammer combined fairs better than the transformer solver, even the variant using a GPU for model inference.

[4]Pass rate is also impacted by the number of tactic suggestions returned in a model call. Too few tactics may lead to the search terminating early. Our faster models also return more suggestions.

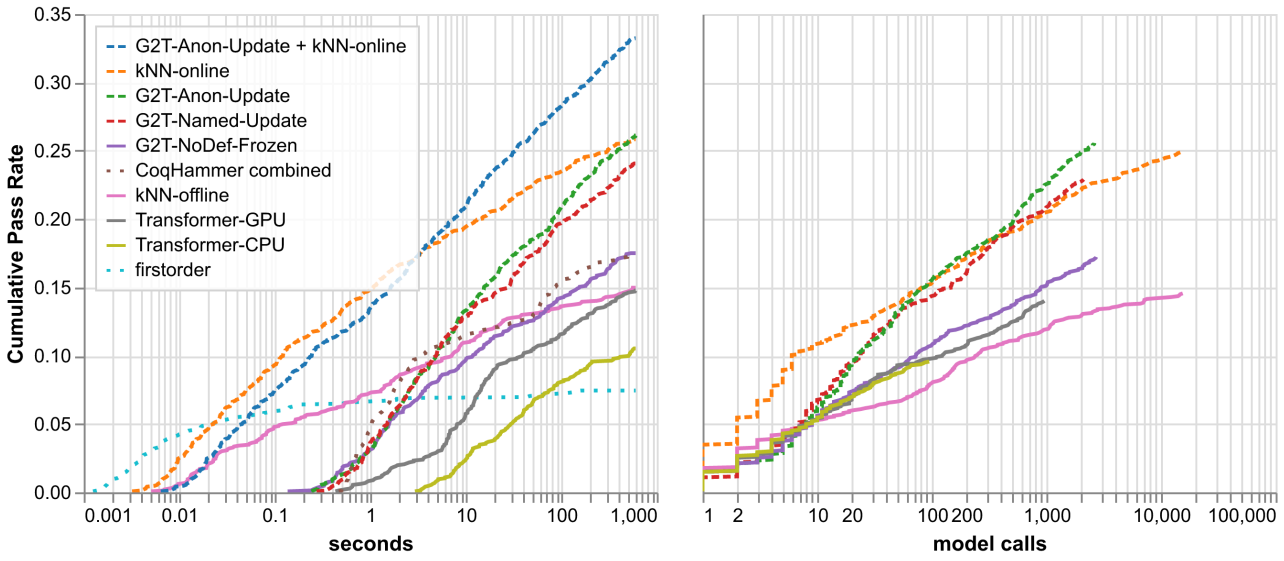

Figure 5. Fraction of theorems proved over time and per number of calls to the model. Online solvers are dashed, offline solvers are solid, and symbolic solvers are dotted.

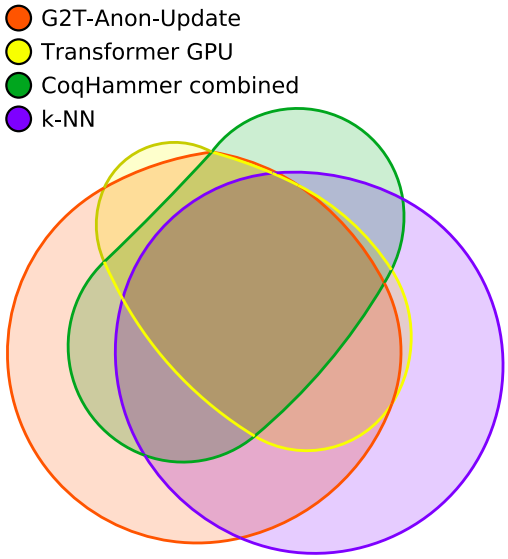

Figure 6. Venn diagram of the theorems proved.

See Appendix J for a detailed breakdown of the CoqHammer results.

The k-NN solvers benefit from being fast, hence both k-NN solvers excel at lower time limits. In Appendix I we suggest the online k-NN is at least as powerful as existing neural Coq solvers in the literature. However, for higher time limits, the G2T-Anon-Update model starts to overtake, suggesting that if larger amounts of search are required, there is value in more sophisticated neural models. It is similar relative to model calls.

Moreover, the online k-NN and G2T approaches are quite complementary as the combined solver G2T-Anon-Update + k-NN is much better than either approach alone. We also see this in the Venn Diagram in Figure 6.

To investigate the effect of the online setting, Figure 7 breaks down per-time pass rate by the number of new dependencies not seen during training, including transitive dependencies. The online models significantly outperform the offline models when a theorem has at least 10 new dependencies. See Appendix P for a more in-depth analysis.

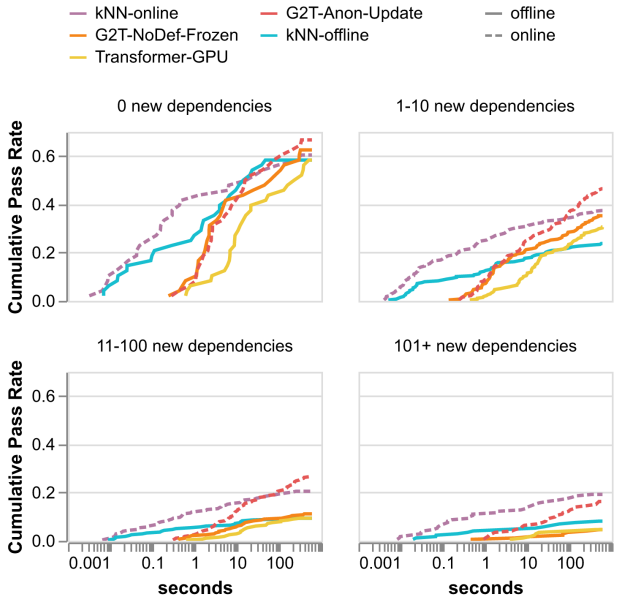

Figure 7. Cumulative pass rate by number of new dependencies.

5. Discussion and Future Work

Our definition task improved a neural theorem prover from 17.4% to 26.1% in the difficult setting of proving theorems in never-before-seen packages. This, in addition to the success of the online k-NN approach, shows the importance of online learning in this setting. We leave as future work how to unify the G2T and online k-NN approaches shown in Figure 1. Ideally, such a model should also account for new tactics, as well as learn from how new definitions and tactics are used in new theorems. One avenue is exploring if our model can be fine-tuned in real time.

To improve the definition task, given that G2T-Anon-Update outperformed G2T-Named-Update, we wonder if adding the names makes the definition task too easy for the model. There may also be alternatives to our definition task, using ideas in self-supervised or contrastive learning, or using social-network-size graph models to process the entire graph of interconnected definitions at once.

Theorem proving and programming share a lot of similarities and concerns, so it is useful to explore how this work relates to code generation. We leave open how our methodology could apply to text-based models. Retrieval augmented transformers are a possible approach (Yang et al., 2023), but may not scale to the full definition hierarchy.

Impact Statement

In a world increasingly dependent on technology, the need for verifiably safe and secure hardware and algorithms is becoming increasingly urgent. This is even more so as a larger amount of computer code is written by AI agents. While interactive theorem provers, such as Coq, have been repeatedly used to verify critical systems, the process is still too labor-intensive for widespread adoption. Machine learning research, such as ours, which improves the state of the art in automated theorem proving for ITPs has the potential to revolutionize the development of safe and secure code.

Acknowledgments

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, Amazon Research Awards, EU ICT-48 2020 project TAILOR no. 952215, and by the Czech MEYS under the ERC CZ project POSTMAN no. LL1902. Lasse Blaauwbroek acknowledges travel support from the European Union's Horizon 2020 research and innovation programme under grant agreement No 951847.

We would also like to thank Alex Sanchez-Stern and Emily First for providing the data used in Appendix I.

References

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems. 2015. URL https://www.tensorflow.org/. Software available from tensorflow.org.

Michaël Armand, Germain Faure, Benjamin Grégoire, Chantal Keller, Laurent Théry, and Benjamin Werner. A modular integration of SAT/SMT solvers to coq through proof witnesses. In Jean-Pierre Jouannaud and Zhong Shao (eds.), *Certified Programs and Proofs - First International Conference, CPP 2011, Kenting, Taiwan, December 7-9, 2011. Proceedings*, volume 7086 of *Lecture Notes in Computer Science*, pp. 135–150. Springer, 2011. doi: 10.1007/978-3-642-25379-9_12. URL https://doi.org/10.1007/978-3-642-25379-9_12.

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, 2019. URL http://proceedings.mlr.press/v97/bansal19a.html.

Frédéric Besson. Itauto: An extensible intuitionistic SAT solver. In Liron Cohen and Cezary Kaliszyk (eds.), *12th International Conference on Interactive Theorem Proving, ITP 2021, June 29 to July 1, 2021, Rome, Italy (Virtual Conference)*, volume 193 of *LIPIcs*, pp. 9:1–9:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi: 10.4230/LIPIcs.ITP.2021.9. URL https://doi.org/10.4230/LIPIcs.ITP.2021.9.

Lasse Blaauwbroek. Tactician's web of large-scale formal knowledge, December 2023. URL https://doi.org/10.5281/zenodo.10028721.

Lasse Blaauwbroek. The Tactician's web of large-scale formal knowledge. *arXiv preprint*, January 2024. doi: 10.48550/arXiv.2401.02950. URL https://doi.org/10.48550/arXiv.2401.02950.

Lasse Blaauwbroek, Josef Urban, and Herman Geuvers. Tactic learning and proving for the coq proof assistant. In Elvira Albert and Laura Kovács (eds.), *LPAR 2020: 23rd International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Alicante, Spain, May 22-27, 2020*, volume 73 of *EPiC Series in Computing*, pp. 138–150. EasyChair, 2020a. doi: 10.29007/wg1q. URL https://doi.org/10.29007/wg1q.

Lasse Blaauwbroek, Josef Urban, and Herman Geuvers. The tactician - A seamless, interactive tactic learner and prover for coq. In Christoph Benzmüller and Bruce R. Miller (eds.), *Intelligent Computer Mathematics - 13th International Conference, CICM 2020, Bertinoro, Italy, July 26-31, 2020, Proceedings*, volume 12236 of *Lecture Notes in Computer Science*, pp. 271–277. Springer, 2020b. doi: 10.1007/978-3-030-53518-6_17. URL https://doi.org/10.1007/978-3-030-53518-6_17.

Lasse Blaauwbroek, Miroslav Olšák, and Herman Geuvers. Hashing modulo context-sensitive α-equivalence. *arXiv preprint*, January 2024. doi: 10.48550/arXiv.2401.02948. URL https://doi.org/10.48550/arXiv.2401.02948.

Jasmin Christian Blanchette, David Greenaway, Cezary Kaliszyk, Daniel Kühlwein, and Josef Urban. A learning-based fact selector for isabelle/hol. *J. Autom. Reason.*, 57(3):219–244, 2016a. doi: 10.1007/s10817-016-9362-8. URL https://doi.org/10.1007/s10817-016-9362-8.

Jasmin Christian Blanchette, Cezary Kaliszyk, Lawrence C. Paulson, and Josef Urban. Hammering towards QED. *J. Formaliz. Reason.*, 9(1):101–148, 2016b. doi: 10.6092/issn.1972-5787/4593. URL https://doi.org/10.6092/issn.1972-5787/4593.

Karel Chvalovský, Jan Jakubuv, Martin Suda, and Josef Urban. ENIGMA-NG: efficient neural and gradient-boosted inference guidance for E. In Pascal Fontaine (ed.), *Automated Deduction - CADE 27 - 27th International Conference on Automated Deduction, Natal, Brazil, August 27-30, 2019, Proceedings*, volume 11716 of *Lecture Notes in Computer Science*, pp. 197–215. Springer, 2019. doi: 10.1007/978-3-030-29436-6_12. URL https://doi.org/10.1007/978-3-030-29436-6_12.

Lukasz Czajka. Practical proof search for coq by type inhabitation. In Nicolas Peltier and Viorica Sofronie-Stokkermans (eds.), *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*, pp. 28–57. Springer, 2020. doi: 10.1007/978-3-030-51054-1_3. URL https://doi.org/10.1007/978-3-030-51054-1_3.

Lukasz Czajka and Cezary Kaliszyk. Hammer for coq: Automation for dependent type theory. *J. Autom. Reason.*, 61(1-4):423–453, 2018. doi: 10.1007/s10817-018-9458-4. URL https://doi.org/10.1007/s10817-018-9458-4.

Oleksandr Ferludin, Arno Eigenwillig, Martin Blais, Dustin Zelle, Jan Pfeifer, Alvaro Sanchez-Gonzalez, Sibon Li, Sami Abu-El-Haija, Peter Battaglia, Neslihan Bulut, Jonathan Halcrow, Filipe Miguel Gonçalves de Almeida, Silvio Lattanzi, André Linhares, Brandon Mayer, Vahab Mirrokni, John Palowitch, Mihir Paradkar, Jennifer She, Anton Tsitsulin, Kevin Villela, Lisa Wang, David Wong, and Bryan Perozzi. TF-GNN: graph neural networks in tensorflow. *CoRR*, abs/2207.03522, 2022. URL http://arxiv.org/abs/2207.03522.

Emily First and Yuriy Brun. Diversity-driven automated formal verification. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, pp. 749–761, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392211. doi: 10.1145/3510003.3510138. URL https://doi.org/10.1145/3510003.3510138.

Emily First, Yuriy Brun, and Arjun Guha. Tactok: Semantics-aware proof synthesis. *Proc. ACM Program. Lang.*, 4(OOPSLA), nov 2020. doi: 10.1145/3428299. URL https://doi.org/10.1145/3428299.

Thibault Gauthier, Cezary Kaliszyk, and Josef Urban. Tactictoe: Learning to reason with hol4 tactics. In Thomas Eiter and David Sands (eds.), *LPAR-21. 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, volume 46 of *EPiC Series in Computing*, pp. 125–143. EasyChair, 2017. doi: 10.29007/ntlb. URL https://easychair.org/publications/paper/WsM.

Zarathustra Amadeus Goertzel, Jan Jakubuv, Cezary Kaliszyk, Miroslav Olsák, Jelle Piepenbrock, and Josef Urban. The isabelle ENIGMA. In June Andronick and Leonardo de Moura (eds.), *13th International Conference on Interactive Theorem Proving, ITP 2022, August 7-10, 2022, Haifa, Israel*, volume 237 of *LIPIcs*, pp. 16:1–16:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. doi: 10.4230/LIPIcs.ITP.2022.16. URL https://doi.org/10.4230/LIPIcs.ITP.2022.16.

Georges Gonthier. The four colour theorem: Engineering of a formal proof. In Deepak Kapur (ed.), *Computer Mathematics*, pp. 333–333, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-87827-8.

Thomas Gransden, Neil Walkinshaw, and Rajeev Raman. Sepia: Search for proofs using inferred automata. In Amy P. Felty and Aart Middeldorp (eds.), *Automated Deduction - CADE-25*, pp. 246–255, Cham, 2015. Springer International Publishing. ISBN 978-3-319-21401-6.

Jesse Michael Han, Jason Rute, Yuhuai Wu, Edward Ayers, and Stanislas Polu. Proof artifact co-training for theorem proving with language models. In *International Conference on Learning Representations*, 2021.

Y. F. Hu. Efficient and high quality force-directed graph drawing. *The Mathematica Journal*, 10:37–71, 2005. URL http://www.mathematica-journal.com/issue/v10i1/contents/graph_draw/graph_draw.pdf.

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.

Jan Jakubuv and Josef Urban. ENIGMA: efficient learning-based inference guiding machine. In Herman Geuvers, Matthew England, Osman Hasan, Florian Rabe, and Olaf

Teschke (eds.), *Intelligent Computer Mathematics - 10th International Conference, CICM 2017, Edinburgh, UK, July 17-21, 2017, Proceedings*, volume 10383 of *Lecture Notes in Computer Science*, pp. 292–302. Springer, 2017. doi: 10.1007/978-3-319-62075-6_20. URL https://doi.org/10.1007/978-3-319-62075-6_20.

Jan Jakubuv, Karel Chvalovský, Miroslav Olsák, Bartosz Piotrowski, Martin Suda, and Josef Urban. ENIGMA anonymous: Symbol-independent inference guiding machine (system description). In Nicolas Peltier and Viorica Sofronie-Stokkermans (eds.), *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*, pp. 448–463. Springer, 2020. doi: 10.1007/978-3-030-51054-1_29. URL https://doi.org/10.1007/978-3-030-51054-1_29.

Albert Qiaochu Jiang, Wenda Li, Szymon Tworkowski, Konrad Czechowski, Tomasz Odrzygózdz, Piotr Milos, Yuhuai Wu, and Mateja Jamnik. Thor: Wielding hammers to integrate language models and automated theorem provers. In *NeurIPS*, 2022.

Cezary Kaliszyk and Josef Urban. Mizar 40 for mizar 40. *Journal of Automated Reasoning*, 55(3):245–256, 2015.

Cezary Kaliszyk, Lionel Mamane, and Josef Urban. Machine learning of coq proof guidance: First experiments. In Temur Kutsia and Andrei Voronkov (eds.), *6th International Symposium on Symbolic Computation in Software Science, SCSS 2014, Gammarth, La Marsa, Tunisia, December 7-8, 2014*, volume 30 of *EPiC Series in Computing*, pp. 27–34. EasyChair, 2014. doi: 10.29007/lmmg. URL https://doi.org/10.29007/lmmg.

Ekaterina Komendantskaya, Jonathan Heras, and Gudmund Grov. Machine learning in proof general. In *Proceedings 10th International Workshop On User Interfaces for Theorem Provers (UITP 2012)*, Electronic Proceedings in Theoretical Computer Science, pp. 15–41, Australia, 2013. Open Publishing Association. doi: 10.4204/EPTCS.118.2.

Daniel Kühlwein, Jasmin Christian Blanchette, Cezary Kaliszyk, and Josef Urban. Mash: Machine learning for sledgehammer. In Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie (eds.), *Interactive Theorem Proving - 4th International Conference, ITP 2013, Rennes, France, July 22-26, 2013. Proceedings*, volume 7998 of *Lecture Notes in Computer Science*, pp. 35–50. Springer, 2013. doi: 10.1007/978-3-642-39634-2_6. URL https://doi.org/10.1007/978-3-642-39634-2_6.

Guillaume Lample, Timothée Lacroix, Marie-Anne Lachaux, Aurélien Rodriguez, Amaury Hayat, Thibaut Lavril, Gabriel Ebner, and Xavier Martinet. Hypertree proof search for neural theorem proving. In *NeurIPS*, 2022.

Maciej Mikuła, Szymon Antoniak, Szymon Tworkowski, Albert Qiaochu Jiang, Jin Peng Zhou, Christian Szegedy, Łukasz Kuciński, Piotr Miłoś, and Yuhuai Wu. Magnushammer: A transformer-based approach to premise selection. *arXiv preprint arXiv:2303.04488*, 2023.

Aditya Paliwal, Sarah M. Loos, Markus N. Rabe, Kshitij Bansal, and Christian Szegedy. Graph representations for higher-order logic and theorem proving. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*, pp. 2967–2974. AAAI Press, 2020. doi: 10.1609/AAAI.V34I03.5689. URL https://doi.org/10.1609/aaai.v34i03.5689.

Bartosz Piotrowski, Ramon Fernández Mir, and Edward Ayers. Machine-learned premise selection for lean. In Revantha Ramanayake and Josef Urban (eds.), *Automated Reasoning with Analytic Tableaux and Related Methods*, pp. 175–186, Cham, 2023. Springer Nature Switzerland. ISBN 978-3-031-43513-3.

Stanislas Polu and Ilya Sutskever. Generative language modeling for automated theorem proving. *arXiv preprint arXiv:2009.03393*, 2020.

Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. 2019.

Jason Rute, Miroslav Olšák, Lasse Blaauwbroek, Fidel Ivan Schaposnik Massolo, Jelle Piepenbrock, and Vasily Pestun. Artifact for: Graph2Tac: Learning Hierarchical Representations of Math Concepts in Theorem Proving, December 2023. URL https://doi.org/10.5281/zenodo.10410473.

Alex Sanchez-Stern, Yousef Alhessi, Lawrence Saul, and Sorin Lerner. Generating correctness proofs with neural networks. In *Proceedings of the 4th ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, MAPL 2020, pp. 1–10, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450379960. doi: 10.1145/3394450.3397466. URL https://doi.org/10.1145/3394450.3397466.

Alex Sanchez-Stern, Emily First, Timothy Zhou, Zhanna Kaufman, Yuriy Brun, and Talia Ringer. Passport: Improving automated formal verification using identifiers. *ACM Trans. Program. Lang. Syst.*, 45(2), jun 2023. ISSN 0164-0925. doi: 10.1145/3593374. URL https://doi.org/10.1145/3593374.

The Coq Development Team. The Coq Proof Assistant, July 2023. URL https://doi.org/10.5281/zenodo.8161141.

Josef Urban. Translating mizar for first order theorem provers. In Andrea Asperti, Bruno Buchberger, and James H. Davenport (eds.), *Mathematical Knowledge Management, Second International Conference, MKM 2003, Bertinoro, Italy, February 16-18, 2003, Proceedings*, volume 2594 of *Lecture Notes in Computer Science*, pp. 203–215. Springer, 2003. doi: 10.1007/3-540-36469-2_16. URL https://doi.org/10.1007/3-540-36469-2_16.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pp. 38–45, Online, October 2020. Association for Computational Linguistics. URL https://www.aclweb.org/anthology/2020.emnlp-demos.6.

Kaiyu Yang and Jia Deng. Learning to prove theorems via interacting with proof assistants. In *International Conference on Machine Learning (ICML)*, 2019.

Kaiyu Yang, Aidan Swope, Alex Gu, Rahul Chalamala, Peiyang Song, Shixing Yu, Saad Godil, Ryan Prenger, and Anima Anandkumar. LeanDojo: Theorem proving with retrieval-augmented language models. *arXiv preprint arXiv:2306.15626*, 2023.

Liao Zhang, Lasse Blaauwbroek, Bartosz Piotrowski, Prokop Černỳ, Cezary Kaliszyk, and Josef Urban. Online machine learning techniques for coq: A comparison. In Fairouz Kamareddine and Claudio Sacerdoti Coen (eds.), *Intelligent Computer Mathematics*, pp. 67–83, Cham, 2021. Springer International Publishing. ISBN 978-3-030-81097-9.

A. Reproducibility

This paper carefully describes our methods for building, training, and testing our models and solvers. Our solvers are intended for use by both researchers and Coq end-users. For end-users, we provide optimized, pre-trained models that can integrate directly with Coq. Installation and usage instructions can be found on Tactician's website[5]. The models run on a typical Linux or x86 Mac machine and do not require a GPU. The model can also be hosted on an external server (for example, with a GPU), and used by a locally running Tactician instance, via a TCP connection.

In this paper, we use a freely available dataset of Coq Data (Blaauwbroek, 2023). Information on the construction of the dataset can be found in Blaauwbroek (2024) and Blaauwbroek et al. (2024). The dataset is freely available under the original licenses of its constituent Coq projects. Most of these licenses are open source. However, some licenses place restrictions on commercial usage. All our models are trained from scratch, without the use of proprietary pre-training data.

To enable the full reproduction of the experiments in this paper, all code, datasets, benchmark parameters, benchmark results, and analysis scripts are available in a separate artifact (Rute et al., 2023). It includes, among others, the following elements:

- Archived versions of all code repositories used in the project. Among others, they include the following:
  - Graph2Tac: Code to train and evaluate the GNN.[6].
  - Text2Tac: Code to train and evaluate the Transformer.[7]
  - The main Tactician repository.[8]
  - The code repository for Tactician's API used to integrate external agents into Coq.[9] This also includes the PyTactician Python library that can be used to extract data from the dataset and interface with Coq.[10]

[5]Installation instructions: https://coq-tactician.github.io/api/graph2tac

[6]Github: https://github.com/IBM/graph2tac
PyPI: https://pypi.org/project/graph2tac

[7]Github: https://github.com/JellePiepenbrock/text2tac
PyPi: https://pypi.org/project/text2tac

[8]https://github.com/coq-tactician/coq-tactician

[9]https://github.com/coq-tactician/coq-tactician-api

[10]PyPi: https://pypi.org/project/pytactician
Docs: https://coq-tactician.github.io/api/pytactician-pdoc

  - The benchmarking system used to evaluate models.[11] This system allows massively parallel evaluations of agents on all Coq projects packaged in the Opam package manager.

- Trained models for Graph2Tac and Text2Tac. They correspond exactly with the models used to obtain the results in this paper. We also include the scripts and parameters used to train these models.
- Benchmark scripts that were used to invoke the benchmarking system for each agent evaluated in this paper. The scripts include the precise parameters used to invoke the models, as well as a reference to the precise Git commit of all involved code.
- The raw data for the benchmarks used in this paper. It includes which theorems were tested on, which were proved, and how long the solver took in terms of seconds, model calls, and tactic executions. This is all in hopes of facilitating future comparison and collaboration.
- Analysis notebooks that distill all the raw data into the curated data as presented in this paper. They will allow the interested reader to verify our results and perform further analysis.

The artifact should enable anyone to replicate each stage of our results, including re-training the models, re-running all benchmarks, and re-doing our analysis. Although reproducing the entire pipeline would require significant work, we expect that our benchmarking data can be re-used easily for comparison against future solvers.

Users can also train and evaluate their own agents with the code we provided and the open dataset. While we trained the graph models for three weeks from scratch on two A100s, in Appendix M we show that a model trained in two days achieves similar results. Users may also train models on a different set of Coq packages, including new or private Coq developments, via the data extraction tools provided with the dataset and with our training code.

B. Threats to Validity

Like other results in machine learning, there are many subtleties that we may not have accounted for in this research, but here we list the important points to look out for.

In theorem proving, it is a known challenge to compare to other comparable solvers. We are unable to provide a direct comparison in our benchmarking system to most existing Coq solvers, and the informal comparison in Appendix I is inconclusive due to methodological differences. Greater challenges exist for comparing to solvers in HOL-Light, Isabelle, and Lean. More standardized benchmarks are needed.

The comparisons we did perform also run the risk of not being an apples-to-apples comparison. The G2T and transformer models use different setups. The former uses TensorFlow and custom-written beam search, while the latter uses standard machine learning APIs for PyTorch which may not have been as optimized for our setting. Further, our baseline transformer solver is only an approximation of similar work. The transformer models of Polu & Sutskever (2020); Han et al. (2021); Lample et al. (2022); Jiang et al. (2022); Yang et al. (2023) are larger, pre-trained, and use more resources. It may be that emergent capabilities arise from scaling which are not seen in our smaller models. However, one reason we chose to avoid pretraining (besides parity with our graph model) is the strong risk that our test data will be leaked into the pretraining data. (See the discussion on this topic in Han et al. (2021).) Also, we were specifically exploring models that could be run with the resources available to a typical Coq user. Similarly, our choice to run CoqHammer on a single thread with default settings may not showcase its strongest capabilities. See the discussion in Appendix J.

The evaluation in Section 4 and the further ablation in Appendix N of the G2T family of models relies on comparison with the G2T-X-Frozen models using random embeddings. There may be alternative ways to perform this baseline. Passport (Sanchez-Stern et al., 2023) uses definition embeddings from names. HOList (Bansal et al., 2019) and ProverBot9001 (Sanchez-Stern et al., 2020) use a special token for unknown definitions.

Our choice to split on package was essential to testing our online setting, and comparable to other Coq solvers, but it also means our results are more variable. A different split would likely give different pass rates. Even our decision to exclude *hott* and *tlc* from our results heavily influences our final results. This is different from papers in other ITPs which split randomly by theorem instead of package. Similarly, we only train one of each model instead of multiple instances, and our models were difficult to reliably train. This also increases the variability of our results.

The data we use decomposes tactics into smaller component tactics. This is in contrast to models like PACT (Han et al., 2021) which not only do not decompose tactics but also train on combinations of tactics. It is possible our decomposition leads to more steps needed for a proof, and this may especially punish slower models like the transformer which can generate more complicated tactic combinations. Also, our G2T model uses a fairly simple representation of tactics. There are 100s of base tactics (we excluded any that occurred fewer than 6 times) and one Coq tactic, *e.g.* `apply`,

[11] Github: https://github.com/coq-tactician/benchmark-system

may be represented as many different base tactics in our model (*e.g.* one for each possible number of arguments). Also, we only allow tactic arguments which are single definitions or local context hypotheses. The G2T model has no way to predict term arguments, which are common in Coq proofs.

During benchmarking, it was common to encounter Coq errors. In the end, any error we counted as a failed proof attempt. Some systematic errors we fixed with patches to Coq, but different solvers led to different errors, and some were easier to fix than others. Further, an error in a test theorem could even cause an error in a later test theorem in the same file. CoqHammer had a larger number of errors.

C. Extended Related Work

We take some extra space in this appendix to touch upon more related work.

In recent years, there have been machine learning approaches for various interactive theorem proving systems. In the Coq ecosystem specifically, a series of articles (First et al., 2020; First & Brun, 2022; Sanchez-Stern et al., 2023) was based on the dataset provided by CoqGym (Yang & Deng, 2019), which contains many Coq packages to benchmark machine learning systems. Other projects in similar directions were GamePad (Huang et al., 2019), an interaction system based on the coqtop REPL, and ProverBot9001 (Sanchez-Stern et al., 2020). Another line of work is based on the Tactician system (Blaauwbroek et al., 2020b), for which implementations of k-nearest neighbors and random forest algorithms were built (Zhang et al., 2021). Early systems for Coq were Proof General (Komendantskaya et al., 2013) and SEPIA (Gransden et al., 2015) and the experiments by Kaliszyk et al. (2014). A system that is often used for comparisons is CoqHammer (Czajka & Kaliszyk, 2018).

For other interactive theorem proving systems, there has also been a lot of work done in recent years. Machine learning guidance was used to prove problems from the Mizar ITP system (Urban, 2003; Kaliszyk & Urban, 2015). TacticToe (Gauthier et al., 2017) is an ITP machine learning system for HOL4. For HOLight, there is the HOList (Bansal et al., 2019) system. In Isabelle, the Sledgehammer system was also extended with machine learning (Kühlwein et al., 2013; Blanchette et al., 2016a). For the recently developed Lean system, there are, for example, LeanDojo (Yang et al., 2023), integrated decision trees (Piotrowski et al., 2023), and work using Language Models (LM) (Han et al., 2021; Lample et al., 2022). There have also been LM-based approaches for other systems, for example Magnushammer (Mikuła et al., 2023) for Isabelle and the GPT-f system (Polu & Sutskever, 2020) for MetaMath.

Our work is novel compared to the previous related work: we use a graph-based dataset extracted with Coq kernel knowledge which allows us to develop a graph neural network that learns the meaning of definitions by exploiting the hierarchical structure of the data. This architecture tackles the problem of learning online, making use of new definitions in new projects.

In interactive theorem proving settings, the term 'hammer' is often used to indicate built-in automation (of various kinds, depending on the interactive proving system used). There has been recent work on learning for which proof states to invoke other automation in the Thor project (Jiang et al., 2022).

Aside from the machine learning guidance on the ITP level, there has also been work on improving the performance of first-order automated theorem provers (and SMT solvers), that form the foundation for the 'hammer' approaches, on ITP-derived problems. Work on the 'hammer' approach in various settings was surveyed in Blanchette et al. (2016b). The ENIGMA system (Jakubuv & Urban, 2017) was originally used on first-order translated versions of problems from the Mizar system, there were several improvements on the Mizar version (Chvalovský et al., 2019; Jakubuv et al., 2020), including a graph-neural-network-based version. A similar system was also trained and tested on Isabelle problems (Goertzel et al., 2022).

D. The Universe of Mathematical Concepts

The dataset we utilize is extracted from 120 different Coq packages from the Opam package manager. These packages were selected by a SAT solver as the largest mutually consistent set of packages available for Coq v8.11. Their topics vary wildly, including analysis, compiler and programming language formalization, separation logic, homotopy type theory, and much more. The graph extracted from these formalizations consists of over 250 million nodes, which encode 520k definitions, of which 266k are theorems, and 4.6M proof state transformations[12].

We divide packages into training and test where no test package depends on a training package. To do so, we induced a random topological order on the Coq packages, with regard to the dependency graph. The resulting list was then split such that the average percentage of theorems and of proof states in the training split is close to 90% (in our case, it is 91.3% of all theorems and 88.8% of all proof states).

Figure 8 shows a rendering of a small section of the dataset. In particular, only the most basic of mathematical concepts that are part of the "Prelude" in Coq's standard library, are rendered. The full graph contained in the dataset would be

[12]Roughly half of the definitions are derived from each other through Coq's module and section mechanism.

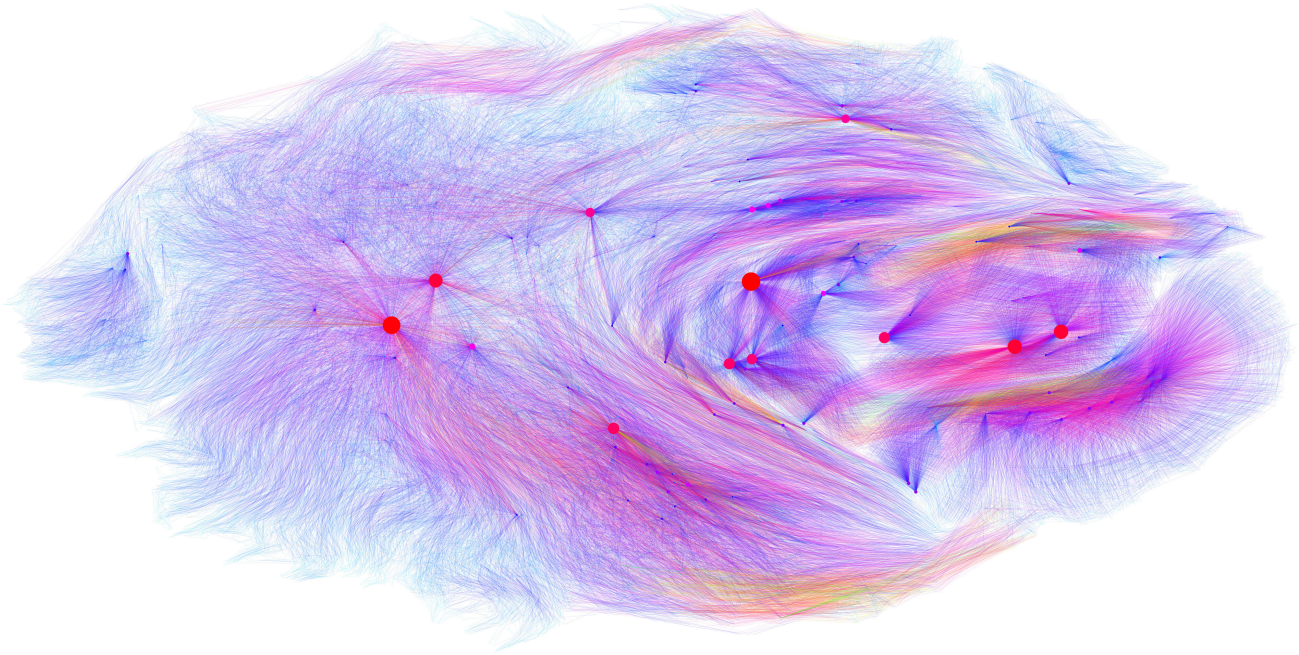

Figure 8. A rendering of a small section of the known mathematical universe as formalized in the Coq proof assistant.

over 3000 times larger. The graph includes all definitions, lemmas, and proof terms in the Prelude, as well as a representation of all intermediate proof states that were generated during the proving process and the partial proof terms generated by tactics to transition between proof states. Together, they form a single, interconnected graph. The graph is fully de-duplicated, such that every sub-term is represented only once (but may be referenced many times).

The size and color of a node in the rendering is dependent on how many times it is referenced. As nodes get more popular, they increase in size. Examples of large nodes are the universes Prop and Type, the inductive definition for Leibnitz equality and the inductive definitions for true and false. Not all popular nodes are definitions, however. Other popular mathematical entities include hypotheses that posit the existence of natural numbers and booleans, and even some anonymous subterms that happen to occur very often.

The placement of each node is determined by the *sfdp* force-directed graph drawing engine of the Graphviz visualization suite (Hu, 2005). As a result, related nodes will be rendered close together. This is particularly apparent for inductive definitions and their constructors, where it often appears as if the constructors are "planets" that orbit around a corresponding inductive that acts as their "sun". The color of each edge is determined by its length as determined by the rendering engine.

E. Further Details on the G2T Architecture

Here we give further details about the G2T architecture described in Section 3.2.

Graph Inputs for Definitions and Proof States The inputs to the definition and prediction task are directed graphs. In Figure 4, the input to the definition task is a graph corresponding to the definition $f\,x := x + x$. The graph of f can be seen in Figure 3 as the eight node subgraph corresponds to all the green edges and all the nodes touching the green edges. This includes the root node f, the global definition nodes for dependencies $\mathbb{N}$ and $+$, and a number of other non-definition nodes for binders ($\forall$ and λ), for function application (@), and for the variable x shown as $\uparrow$. The edge labels come with labels that are not shown. Variables do not need names since they are connected to their binders via labeled edges. Even though we include the nodes $\mathbb{N}$ and $+$, we do not include the nodes in their respective definition graphs. This keeps the size of our input graph manageably small.

Similarly, in Figure 4, the proof state input to the prediction network corresponds to a proof state with two local hypotheses $x : \mathbb{N}, y : \mathbb{N}$ and the goal $\vdash 2 * x = f\,x$. This is almost the subgraph shown with blue edges in Figure 3 except it contains an extra local hypothesis y. Like variables, we do not store the names of local hypotheses in the graph. The triangle $2 * x$ represents the subgraph of nodes shared between the proof state and T_1. (To compress our graph, common terms are shared.) The nodes in that triangle are

also part of the proof state graph.

In step A of Figure 4, we assign embeddings as follows:

- Root nodes like f get a null vector of all zeros.
- Definition nodes like 2, $*$, $+$, and $\mathbb{N}$ are assigned embeddings from the definition embedding table.
- Other nodes, are assigned embeddings from the node embedding table based only on the node type. These include nodes for binders, function application, variables, and local constants.
- Edges are assigned embeddings according to their edge types (after adding in extra edges as described in Section 3.2).

While Step A treats local context nodes the same as any other non-definition node, they become important in Step G, where we use the GNN embeddings for the local context nodes to select arguments.

RNN for Local and Global Argument Prediction The RNN described in Step F of Figure 4 is a simple two layer RNN described mathematically as follows. Let t be the embedding of the base tactic (`apply _` in Figure 4). Let p be the pooled embedding of the proof state, that is the mean of all the node embeddings in the output graph of the GNN. The goal is to compute an embedding x_i for each argument position i in the base tactic. (For example, `rewrite _ in _` has two argument positions, `reflexivity` has zero positions, and `apply _` has one position. Hence in the example in Figure 4, there is only one RNN output.) The computation is as follows:

$$\begin{aligned} x_i^{(0)} &:= p \\ h_i^{(n)} &:= t && \text{for } n \in \{0,1\} \\ x_i^{(n+1)}, h_{i+1}^{(n)} &:= \mathrm{ReLU}(\mathrm{Dense}(x_i^{(n)}, h_i^{(n)})) \end{aligned}$$

The two inputs and two outputs to the dense layer are concatenated. The RNN output corresponding to the ith argument position is $x_i^{(2)}$. This is used as the input to Step G in Figure 4.

The Loss for Definition and Prediction task Given a definition d with index i_d and graph g_d, let $\mathrm{DefTask}(g_d)$ be the computed embedding for the definition task, and $\mathrm{DefEmb}(i_d)$ be the embedding in the definition embedding table. The definition loss is the cosine similarity loss

$$\begin{aligned} \mathcal{L}_{\text{def}} &:= 1 - \frac{\mathrm{DefTask}(g_d) \cdot \mathrm{DefEmb}(i_d)}{|\mathrm{DefTask}(g_d)|\ |\mathrm{DefEmb}(i_d)|} \\ &= 1 - \mathrm{DefTask}(g_d) \cdot \mathrm{DefEmb}(i_d) \end{aligned}$$

where $\cdot$ is inner product. The denominator is not needed since both $\mathrm{DefTask}(g_d)$ and $\mathrm{DefEmb}(i_d)$ are unit normalized.

The prediction task loss is calculated as follows. Let $P_{\text{tactic}}(T \mid S)$ be the probability the model assigns to a base tactic T given a proof state S. Let $P_{\text{arg}_i}(A \mid S, T)$ be the probability the model assigns to an argument A (either local or global) given a proof state S and the base tactic T. (Notice, even though we use an RNN, the probability of the ith argument only depends on the tactic and proof state.) Then if the ground truth tactic sequence is the base tactic T with arguments $A_0, A_1, ..., A_{n-1}$, the loss is calculated as the cross entropy

$$\begin{aligned} \mathcal{L}_{\text{tactic}} := &- \log P_{\text{tactic}}(T \mid S) - \log P_{\text{arg}_0}(A_0 \mid S, T) \\ &- \log P_{\text{arg}_1}(A_1 \mid S, T) - ... \\ &- \log P_{\text{arg}_n}(A_{n-1} \mid S, T) \end{aligned}$$

Adding New Definitions to the Current State During inference, every time we get to a new theorem, we query Coq (via Tactician) for any new definitions added to the global context. This gives us a sequence of definitions $d_0, ..., d_{n-1}$. For each definition d, we first check if that definition was seen during training (using a hash based on the graph representation). If so, we align the definition index given by Coq to the one we already have. If not, we use $\mathrm{DefTask}(g_d)$ to calculate the definition embedding, where g_d is the graph of definition d and DefTask is our definition task neural network. All the embedding tables are in the neural network, including the definition embedding table. We may assume, by the order definitions are returned from Coq, that any prerequisite definitions needed to calculated $\mathrm{DefTask}(g_d)$ have already been added to the definition embedding table. After calculating $\mathrm{DefTask}(g_d)$, we manually update the definition embedding table to include the newly calculated embedding for d.

As an implementation detail, the most expensive part of changing the embedding table is recompiling the network, and this is only needed when we change the size of the table. To reduce this during inference, we allocate a larger definition embedding table than needed, similar to a resizable array.

F. Training Details

Graph2Tac graph models were trained on 8 CPUs (with 128 GB RAM) and 2 NVIDIA A100 GPU cards for about 20 days. Optimization used the Adam optimizer with a learning rate of 0.0003 and L2 regularization set to 1.0e-05. Training was restarted every 24 hours from the current checkpoint, but we did not save optimizer states between restarts. We noticed periodic instability in training and occasionally had to restart training from a checkpoint.

During the design of the model, some hyperparameters, such as the coefficient 1000 for the loss, were tuned manually on a small subset of the training data using other training packages for validation.

The transformer was a GPT2-style model (Radford et al., 2019) from the Huggingface Transformers library. The model was trained to predict the tactics based on the proof states. To determine whether the model was sufficiently trained, we monitored the validation accuracy and stopped training when the validation performance stagnated.

The GPU evaluations and the training of the transformer were performed on machines with two 20-core Inter E5-2698v4, a working memory of 512 GB RAM, and 8 NVIDIA Tesla V100 GPU cards.

G. Search Procedure Details

The G2T, transformer, and k-NN *models* only predict single tactics. In order to turn our models into a *solver*, which can find a whole proof, we combine them with the search procedure of Tactician (Blaauwbroek et al., 2020a), namely Dijkstra's algorithm[13] to explore the tree of possible proofs, but modified to use iterative deepening instead of a queue. For G2T and transformer, each possible tactic is scored by the log probability predicted by the corresponding model. The length of a path (partial proof) is the sum of the negative log probabilities of its steps. With iterative deepening, each iteration i only explores paths whose distance is below an upper bound D^i_{max} (depth first where the tactics are ordered by score). If an iteration exhausts all valid paths without finding a proof, the search starts from the beginning with the new upper bound

$$D^{i+1}_{\text{max}} = D^{i+1}_{\text{max}} + 1 + D^i_{extra}$$

where D^i_{extra} is the negative log probability needed to increase the left-most path by one tactic. Our iterative deepening search uses constant memory at the expense of only a constant multiplicative increase in time.

A Coq tactic can produce multiple independent subgoals. Each step in our search procedure predicts a tactic for the first goal of the stack and applies the resulting tactic to that goal only. If these goals are fully independent (have no shared E-variables), then our DFS takes this into account by not backtracking into a previously solved subgoal earlier in the proof. See Tactician (Blaauwbroek et al., 2020a) for more details.

[13] Dijkstra's algorithm is a special case of best-first search algorithms, and the "best-first search" of Polu & Sutskever (2020) and following works (Han et al., 2021; Jiang et al., 2022) is equivalent to vanilla Dijkstra's algorithm when the search nodes are ordered by the cumulative log probability of the tactic predictions.

H. All Results

Table 1 shows the results for the 10-minute evaluation where packages *hott* and *tlc* are excluded.

Figure 9 shows the cumulative pass rates for all packages and all the models discussed in Section 4. We see that neither the G2T-Anon-Update nor the online k-NN always performs better. In many cases G2T-Anon-Update either overtakes the online k-NN or appears as if it will overtake the online k-NN if given enough time. As discussed in Appendix Q, the *tlc* package is special in that G2T-Anon-Update was able to prove a significant number of theorems using an inconsistent axiom.

solver	pass rate
CoqHammer combined	0.174
G2T-Anon-Update	0.261
G2T-Anon-Update + kNN-online	0.332
G2T-Named-Update	0.241
G2T-NoDef-Frozen	0.175
Transformer-CPU	0.105
Transformer-GPU	0.148
firstorder	0.075
kNN-offline	0.150
kNN-online	0.258

Table 1. Pass rates for 10-minute time limit.

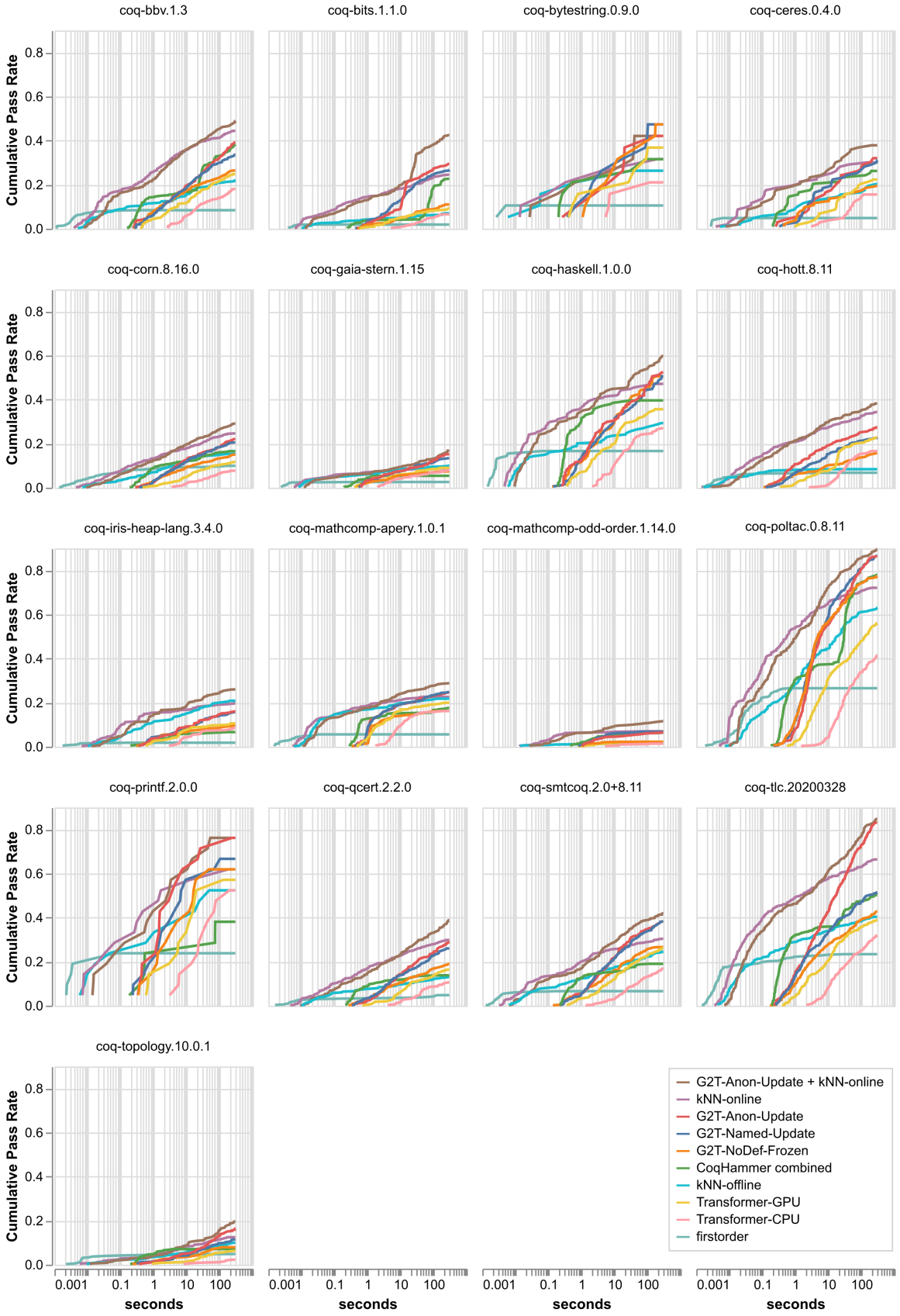

Figure 9. Cumulative pass rates for all packages.

I. Informal Comparison on CoqGym Test Packages

Besides highlighting our novel definition task, our work is the first work we are aware of comparing k-NN, transformer, and graph-based solvers for an interactive theorem prover. Perhaps surprisingly, the online k-NN performs much better than expected, especially since most recent papers use powerful neural models like transformers.

The primary benchmark for Coq is the CoqGym benchmark, consisting of 28 test projects. Unfortunately, of those 28, only seven are included in our dataset (our data is harvested from Opam packages, while the CoqGym data is retrieved from Github). Of those seven projects, all but one (*poltac*) are included in our training set, making it impossible to do a fair evaluation with any of our trained models. Nonetheless, we can compare our trained models on *poltac* and our untrained models, such as the online k-NN model, on five of the remainder.[14]

In Tables 2 and 3, we compare our work to two other families of benchmarks on the CoqGym test projects. The first family are benchmarks of Yang & Deng (2019), First et al. (2020), First & Brun (2022), and Sanchez-Stern et al. (2023). These include the neural solver ASTactic and its extensions TacTok, Diva, TacTok+Passport, as well as an independent benchmark of CoqHammer. The second is a benchmark of the neural solver Proverbot9001 (Sanchez-Stern et al., 2020). While the original ProverBot9001 results were for CompCert, we use data from the ProverBot9001 benchmark found at `https://proverbot9001.ucsd.edu/compare/`. It is a comparison to CoqGym using approximately similar training and test packages, but run in the ProverBot9001 system. For our family of benchmarks, we include both the trained solvers reported in our results section (only tested on *poltac*), as well as a full 10 minute online k-NN and CoqHammer benchmark run on all theorems in all six projects.

It should be emphasized that these comparisons are informal and non-conclusive due to a number of methodological differences:

- The ASTactic/TacTok/Diva and ProverBot9001 benchmarks train on the CoqGym training suite, or in the case of ProverBot9001 an approximation of the CoqGym training suite, while our G2T models train on our training suite containing different Coq projects.
- Coq versions used by the benchmarks vary between version 8.10 and 8.12.
- Each benchmark may use different versions of test packages, which includes different (in some cases significantly different) number of test theorems.
- The ASTactic/TacTok/Diva family of benchmarks were run for 10 minutes, although TacTok and Diva run multiple solvers in parallel for a total run time, respectively, of 2×10 and 64×10 minutes. Some of our benchmarks were run for 5 minutes and some for 10. ProverBot9001, on the other hand, does not use a time limit, instead stopping the search based on search depth. Nonetheless, the ProverBot9001 results were specifically designed by the creators to compare to the ASTactic/TacTok family of benchmarks.
- There are differences in hardware. ASTactic, for example, uses two cores, where we restrict ourselves to one core. Since ProverBot9001 does not use time limits, hardware does not matter.

Nonetheless, the previous CoqGym benchmarks have a similar flavor to ours. All results test on entire packages not seen during training. The benchmarks were all run without a GPU (except our transformer baseline). None of the systems use large-scale pre-training on internet data, so there is little risk of data leakage.

The results in Table 2 and Table 3 suggest that the online k-NN model, despite being one of the simplest models we compare, is one of the strongest. In particular, it outperforms the other previously published neural models, only doing worse than ProverBot9001 on one project.

Transitively, this suggests our best G2T models are also state of the art for Coq, and we indeed see this on *poltac* in Table 3.

Finally, note that our CoqHammer combined aggregate solver performs similarly to the previous benchmark of CoqHammer, suggesting our aggregate is a good approximation (see Section J). Nonetheless, our CoqHammer benchmarks and those in CoqGym likely use different versions of CoqHammer and likely rely on different combinations of ATP backends.

In conclusion, more work is needed to compare the methods in this paper to other methods in the literature. The strong showing of the online k-NN baseline, which until now has not been properly compared to other systems, suggests that many of the approaches in the literature, especially the common transformer approaches, may be surprisingly weak, especially in the online setting of new files or packages not seen during training.

[14] We excluded *goedel* as it has completely changed between the CoqGym benchmarks and ours. It has been abandoned and merged into *hydra-battles*. Our package of *goedel* only contains 17% of the 606 theorems found in the older *goedel* package in CoqGym.

	time (min)	pass rate (solved/total) buchberger	coquelicot	hoare-tut	huffman	zorns-lemma	total
ASTactic	10	0.097 (70/725)	0.065 (95/1467)	0.056 (1/18)	0.080 (25/314)	0.067 (10/149)	0.105 (319/3036)
CoqHammer	10	0.229 (166/725)	0.186 (273/1467)	**0.333** (6/18)	**0.236** (74/314)	0.121 (18/149)	0.272 (826/3036)
Diva	64×10	0.204 (148/725)	0.128 (188/1467)	0.278 (5/18)	0.178 (56/314)	0.121 (18/149)	0.193 (585/3036)
TacTok	2×10	0.105 (76/725)	0.068 (100/1467)	0.278 (5/18)	0.089 (28/314)	0.081 (12/149)	0.110 (333/3036)
TacTok+Passport	2×10	0.126 (91/725)	0.082 (120/1467)	0.222 (4/18)	0.096 (30/314)	0.081 (12/149)	0.129 (392/3036)
ProverBot9001	NA	0.237 (178/750)	0.107 (187/1743)	0.280 (7/25)	0.225 (71/316)	0.115 (18/156)	0.194 (639/3299)
CH combined	10	**0.284** (213/750)	0.148 (258/1743)	0.320 (8/25)	0.207 (65/314)	0.100 (26/259)	0.242 (822/3400)
kNN-online	10	0.264 (198/750)	**0.234** (408/1743)	0.320 (8/25)	0.169 (53/314)	**0.282** (73/259)	**0.284** (967/3400)

Table 2. Comparison of results across three families of benchmarks. Note each family has a different number of theorems. The total column includes the *poltac* results in Table 3. The Diva and TacTok solvers run multiple solvers in parallel, each for 10 minutes.

	time (min)	pass rate (solved/total) poltac
ASTactic	10	0.325 (118/363)
CoqHammer	10	0.796 (289/363)
Diva	64×10	0.468 (170/363)
TacTok	2×10	0.309 (112/363)
TacTok+Passport	2×10	0.372 (135/363)
ProverBot9001	NA	0.576 (178/309)
CoqHammer combined	10	0.816 (252/309)
G2T-Anon-Update	5	0.864 (267/309)
G2T-Named-Update	5	**0.867** (268/309)
G2T-NoDef-Frozen	5	0.773 (239/309)
Transformer (GPU)	5	0.560 (173/309)
kNN-online	10	0.735 (227/309)

Table 3. Comparison of results on *poltac*, which is in both our test set and the CoqGym test set, across three families of benchmarks.

J. Analysis of CoqHammer Benchmarks

A core design principle of CoqHammer is the ability to translate a higher-order theory into a first-order theory and export it to a varied set of automatic theorem provers. Data provided by the original CoqHammer publication (Czajka & Kaliszyk, 2018) suggests that the various ATP's are highly complementary. Indeed, this is confirmed in a publication on Tactician (Blaauwbroek et al., 2020a) which compares an early version of its online k-NN solver with CoqHammer on Coq's standard library. The online k-NN solver is able to beat any of CoqHammer's individual ATP backends by some margin. But when combined, the ATP's achieve a higher pass-rate than the online k-NN solver.

To obtain a fair comparison, we benchmarked all of CoqHammer's ATP backends individually. We then calculate a time-scaled combination of the ATP's in order to simulate the parallel execution of all ATP's on a single CPU. Surprisingly, our results are quite different from what is reported in previous publications.[15] As can be seen from the Venn diagram in Figure 11, the ATP's are not nearly as complimentary on our testing set as one would expect. The Vampire backend is able to prove most of the theorems obtained by other ATP's. Only the `sauto` and `best` tactics remain highly complementary to Vampire. Figure 10 shows that a combination of `best` and Vampire produces the same results as an aggregation of all solvers, but faster. As such, unless one truly has CPU-threads to spare for parallel solving, using many ATP's is not effective.

As a result, CoqHammer is no longer able to beat the online k-NN and G2T solver on our testing set. We speculate that the root cause of this discrepancy is the use of a more complex test set, on which the higher-order to first-order translation of CoqHammer is far less effective. Presumably, during the development of CoqHammer, the translation was tuned particularly well on Coq's standard library. This translation appears to generalize to other packages less well than one would hope.

[15] Note that we are unable to reproduce the benchmark results on Coq's standard library because CoqHammer depends on the same standard library. This creates a circularity that cannot easily be broken.

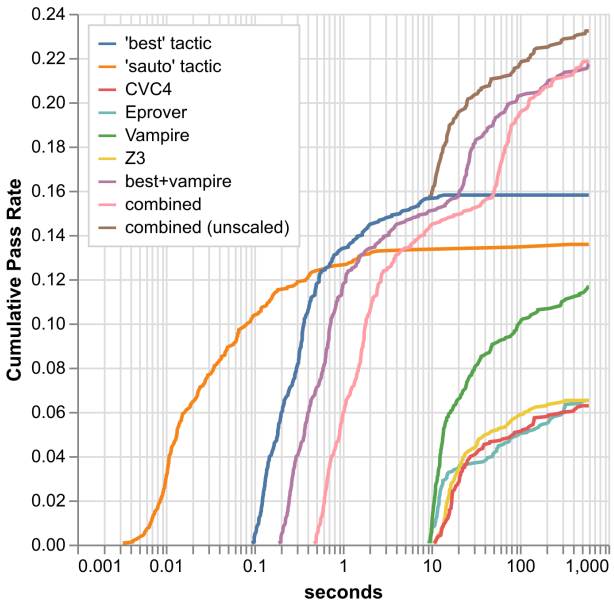

Figure 10. Pass rates for individual solvers of CoqHammer and various aggregates. (Includes all test packages except *hott*.)

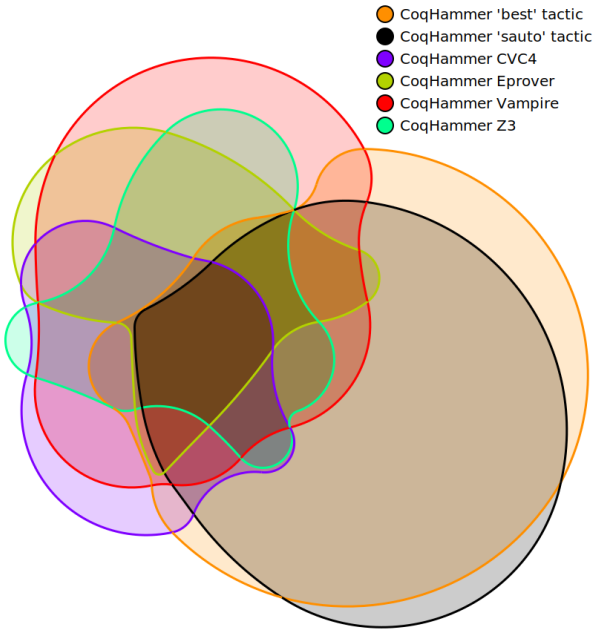

Figure 11. Venn diagram showing complementarity between CoqHammer solvers. (Includes all test packages except *hott*.)

K. The Relative Speed of the Models

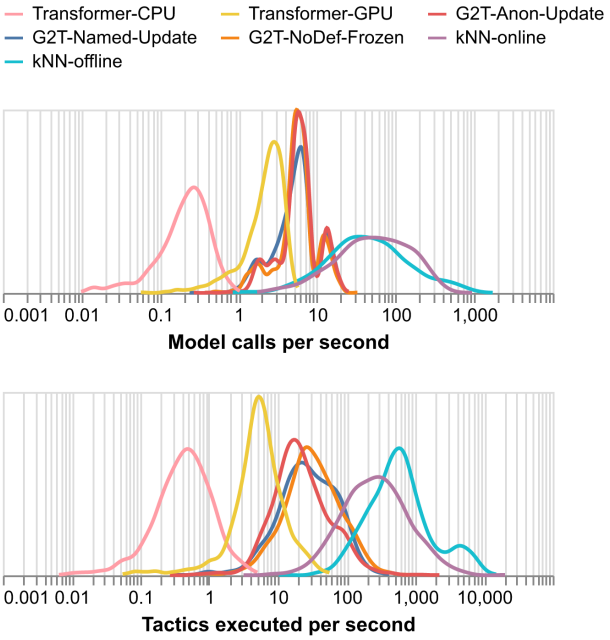

Figure 12. Density plots of the speed of the various models, in tactics executed per second and model calls per second.

Figure 12 shows the speed of the models for the attempted proofs in the test set. Notice the various families of models can differ by an order of magnitude in speed, with k-NN models being the fastest, and transformers being the slowest.

L. Ablation of k-NN Solvers Seeing Recent Proofs

In this section, we explore variations of the k-NN setup. The online and offline k-NN solvers in our main results have multiple differences from each that determine which examples in the global context the k-NN can search over. The online k-NN can search over only the most recent 1000 tactics, but can see proofs up to and including those in the current file. The offline k-NN is not restricted to 1000 tactics, but can see all the tactics that were available at training time, and uses locality sensitive hashing forests (LSHF) to account for the larger search space.

To break down the contribution of each difference, in Table L, we specify six solvers that vary on two dimensions. The first dimension is how online the model is. The most online models search over proofs up to and including the current file. The next level excludes the current file. The final level excludes all but the training data.

The second dimension is whether the model only sees the 1000 most recent tactics. For proofs in different files, the order of the theorems is someone arbitrary, and this likely

Solver	Global context proof data		LSHF	Notes
	files	recent		
kNN-recent-online	imports + current file	1000		"online k-NN" in results
kNN-LSHF-online	imports + current file	all	✓	k-NN in Tactician
kNN-recent-allButFile	imports	1000		
kNN-LSHF-allButFile	imports	all	✓	
kNN-recent-offline	imports in training data	1000		
kNN-LSHF-offline	imports in training data	all	✓	"offline k-NN" in results

Table 4. k-NN models in the ablation.

affects the results. If we are not restricting to 1000 tactics, then we use LSHF to speed up the search.

The solver kNN-LSHF-online is the default k-NN solver in Tactician (Blaauwbroek et al., 2020b). The solvers kNN-recent-online and kNN-LSHF-online are the solvers in the results section.

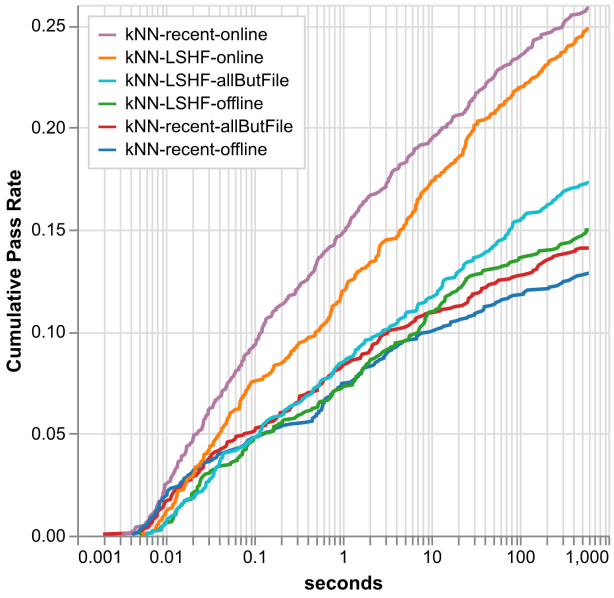

Figure 13. Pass rates for the different k-NN solver variations.

Figure 13 gives the performance of all six solvers. There is a significant performance advantage for the full online solvers, which can see previous proofs in the same file, and even a bit more advantage when the model is restricted to *only* the most recent 1000 tactics. This suggests that our online k-NN (kNN-recent-online), gets much of its power from copying nearby tactics of proofs in the same file.

There is also some smaller advantage when going from a model that can see only the training data to one that can see all but the most recent file. This again shows the importance of recent online information.

The worst models are the ones restricted to the most "recent" 1000 tactics, but where they cannot see the current file. This is likely an artifact of the arbitrary way that proofs are ordered outside the current file.

Overall these results suggest the online k-NN approach may be slightly brittle in that it depends a lot on what recent proof data is available in the global context. Nonetheless, given the speed and simplicity of the k-NN solvers, and how even the offline k-NN models have non-negligible performance, we think they are good baseline models for theorem proving.

M. Effect of Training Time on G2T Solvers

The G2T-Anon model was trained for 247 epochs over 20 days. In this section we show that we can achieve comparable results with only 27 epochs in 2 days. In table 5 and Figure 15, we see that a model trained for only 27 epochs (the earliest checkpoint we saved) does as well, if not better, than a model trained for the full 247 epochs, despite the fact that the latter model has better validation accuracy. (Moreover, we found this behavior was consistent across packages.) This is important for the following reasons: (1) Faster model training allows for more rapid development and experimentation. (2) During training, after about 100 epochs we regularly encounter instabilities requiring us to restart training from a previous checkpoint. (3) Shorter training time likely means shorter fine-tuning, hopefully making it possible for end users to fine-tune our model on new Coq projects, even with limited GPU availability.

Training Time			Validation		Test
epochs	GPU-hours	days	loss	accuracy	pass rate
27	91.5	2	8.65	0.497	**0.267**
247	907.1	20	**8.25**	**0.567**	0.261

Table 5. Statistics for two G2T-Anon checkpoints. Training times do not account for the four instances we restarted training from an earlier checkpoint. Validation accuracy is the proportion of instances where the generated base tactic plus arguments (using greedy decoding) match the ground truth from the held-out validation set. The pass rate uses the G2T-Anon-Update configuration.

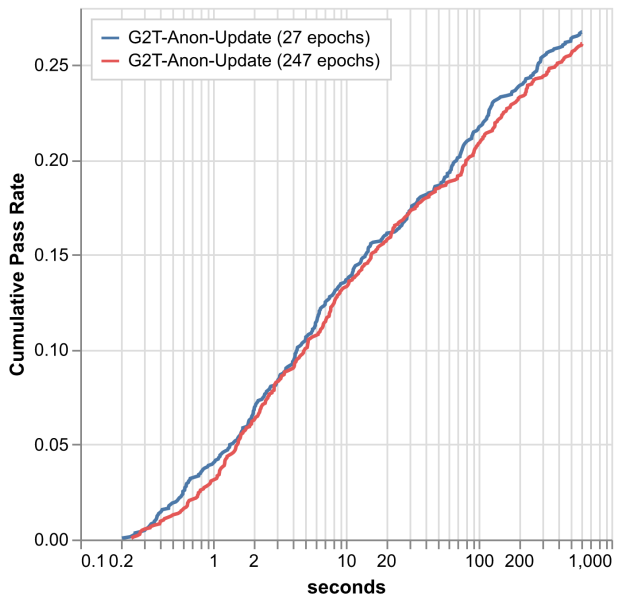

Figure 14. Pass rates for different checkpoints of G2T-Anon.

N. Ablation of G2T Solvers and Symmetry Breaking with Names

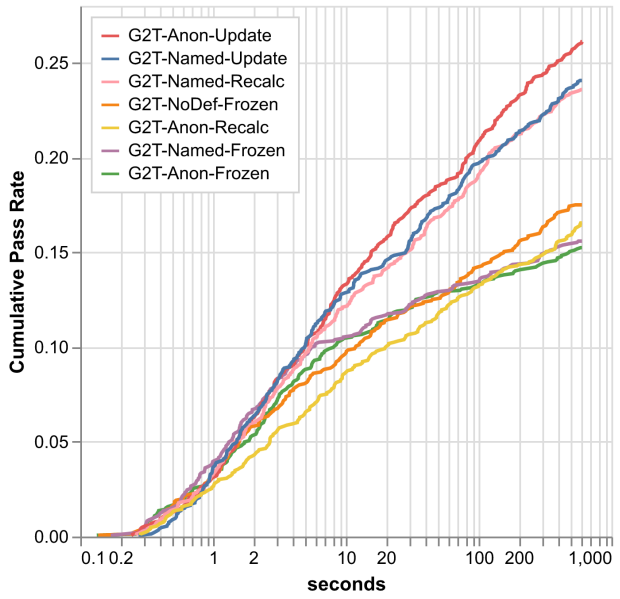

Figure 15. Pass rates for the different G2T solver variations.

Figure 15 shows all seven G2T variations mentioned in Subsection 3.2. In particular, we see that using the frozen version of a model where the definition task is not used to compute new definition embeddings hurts model performance. This suggests that our definition network is not just a helpful auxiliary task during training, but an essential component of our online system.

Also, note that the models with and without name embeddings have different behaviors when we recalculate all definition embeddings including those seen during training. G2T-Named-Recalc shows the same performance as G2T-Named-Update. Recalculating all embeddings would simplify the solver interface since one does not have to keep track of which definitions were part of training.

However, this is not true of G2T-Anon-Recalc. One possible reason for this discrepancy is that names break symmetries when calculating definition embeddings from scratch. Isomorphic definitions have isomorphic graphs and hence receive the same embedding by our definition model (if name embeddings are not also used). One example of this is the embeddings for the Boolean values `true` and `false`. Further, this symmetry can propagate. If `true` and `false` are given the same embedding, then so are the Boolean operations `andb` and `orb` since the graph of the former is isomorphic to the latter with `true` and `false` swapped.

O. Additional Combinations with the Online k-NN

In our main results, we remark on how the online k-NN solver and the G2T-Anon-Update solver are particularly complimentary. We demonstrated this by plotting the combined time-scaled aggregate in Figure 5. This is additionally supported by the Venn diagram in Figure 6 which showed the G2T-Anon-Update model and the online k-NN to be most disjoint.

In Figure 16 we further show that of all the solvers, the online k-NN pairs best with the G2T-Anon-Update and G2T-Named-Update models, which both use our novel definition embeddings. This again supports the complementary of the two online approaches, as represented in Figure 1.

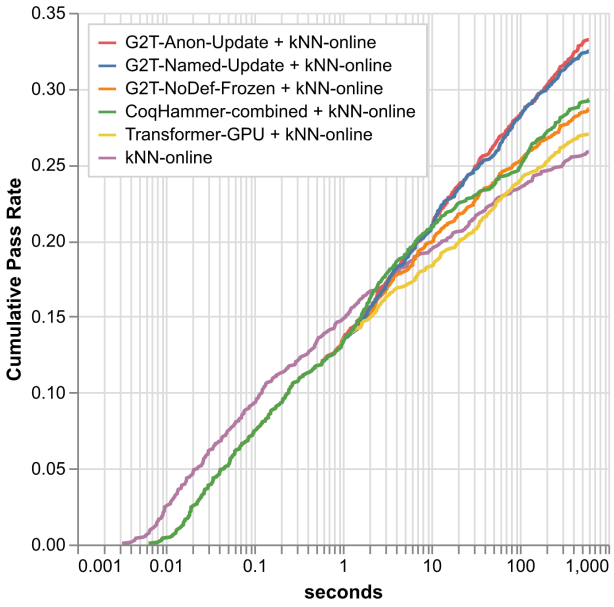

Figure 16. Pass rates of various solvers combined with the k-NN solver. All combined solvers scale time so that time is evenly divided among the two component solvers. The uncombined k-NN is provided for reference.

P. Impact of the Online Setting

An important distinction of our paper is our use of entire packages for testing, instead of splitting theorems into test and train randomly as many other papers do.

To investigate this, we measure, for each of our test theorems, the number of new definition dependencies not seen during training required for that theorem (both in its theorem statement and proof). This includes transitive dependencies.

Figure 17 shows the number of solved theorems as a function of new dependencies. Figure 18 shows the per-time pass rate, conditioned on the number of dependencies. Figure 19 is the same, but conditioned on the number of model calls.[16] In general, having more new dependencies is more challenging for all the solvers.

For 0 new dependencies, we see less difference between all the solvers, and our best solvers solve over 60% of theorems. But when we move to a higher number of new dependencies, the offline models, including the transformer models, the offline k-NN, and G2T-NoDef-Update, underperform the online models, the online k-NN, G2T-Anon-Update, and G2T-Named-Update. This is yet more justification that the main benefit of our definition task and the online k-NN is their usefulness in online settings.

[16]Note that due to discrepancies in how we collect the dependency data, not all test theorems are included in the figures.

Nonetheless, we do notice a small improvement in the G2T-Anon-Update solver over the G2T-NoDef-Frozen solver in the 0 new dependency setting. This suggests that the definition training task may show some usefulness on its own as an auxiliary task during training even when not used at inference time.

The online k-NN solver starts to outperform the G2T models when dependencies get into the hundreds. It is unclear why this happens. In particular, there are many correlations between the number of new dependencies and the package. Larger Coq developments have a larger number of new dependencies, but they may support some solvers over others for other orthogonal reasons.

Q. Discovery of Inconsistent Axioms

After benchmarking our solvers we discovered that three of our test packages contained inconsistent axioms. These axioms are used for development or special logical manipulation but are not intended to be used blindly in theorems, as they can be used to prove false results.

As shown in Table 6, some of our solvers were able to discover these axioms and use them. The most significant case was `skip_axiom` in the *tlc* package. G2T-Anon-Update managed to solve 192 of the 500 test theorems (38.4%) for the 5-minute benchmark using this axiom. Figure 20 shows the results with all test theorems and when restricting to theorems for which no model used `skip_axiom`.

It is impressive that our model was able to discover the importance of this axiom, but it also makes the evaluation of our results difficult. For example, on *tlc*, the name-less G2T model significantly outperforms the one with names, but mostly because it was able to exploit this axiom.

The axioms are as follows. In *tlc*:

```
Axiom skip_axiom : False.
```

In *hott*:

```
Inductive No_Empty_for_admitted := .
Axiom No_Empty_admitted
  : No_Empty_for_admitted.
```

In *haskell*:

```
Definition undefined {a : Type} : a.
Admitted.
```

In *quickchick* (used for training):

```
Axiom ignore_generator_proofs : False.
```

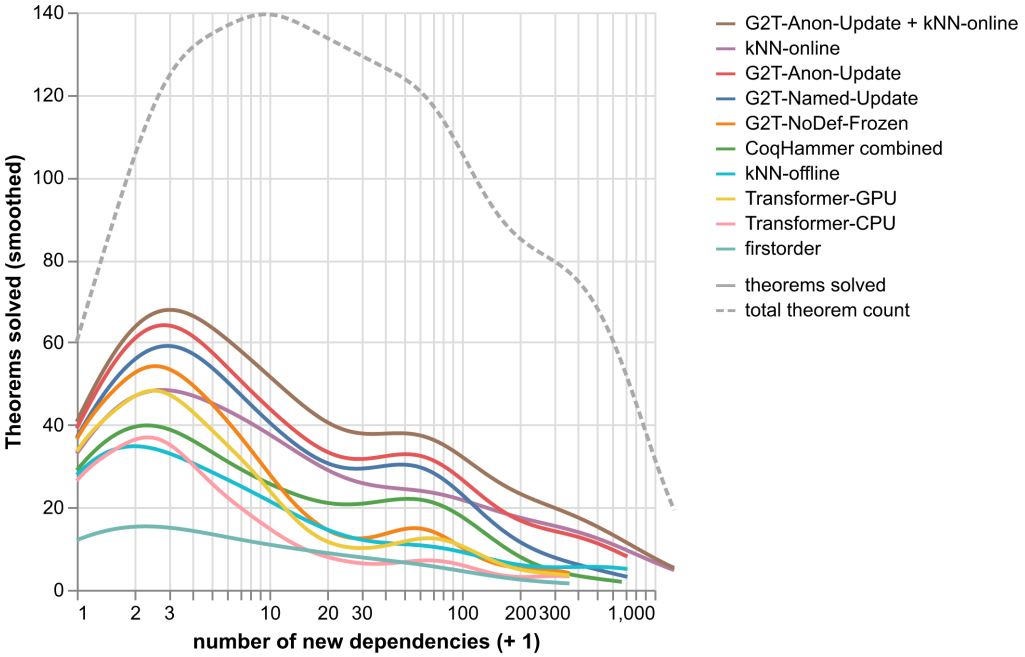

Figure 17. Theorems solved as a function of new dependencies. (Smoothed to account for small numbers in each category.) The dotted line shows the total number of test tactics (also smoothed) as some numbers of dependencies are more common.

package	axiom	solver	count	percent
haskell	`undefined`	G2T-Anon-Update	3	1.7%
haskell	`undefined`	G2T-Named-Update	8	4.6%
haskell	`undefined`	G2T-NoDef-Frozen	13	7.5%
hott	`No_Empty_admitted`	G2T-Anon-Update	9	1.8%
hott	`No_Empty_admitted`	kNN-online	2	0.4%
tlc	`skip_axiom`	G2T-Anon-Update	192	38.4%
tlc	`skip_axiom`	G2T-Named-Update	5	1.0%
tlc	`skip_axiom`	G2T-NoDef-Frozen	25	5.0%

Table 6. The number of times the solvers use inconsistent axioms to prove a test theorem in the 5 minute benchmark.

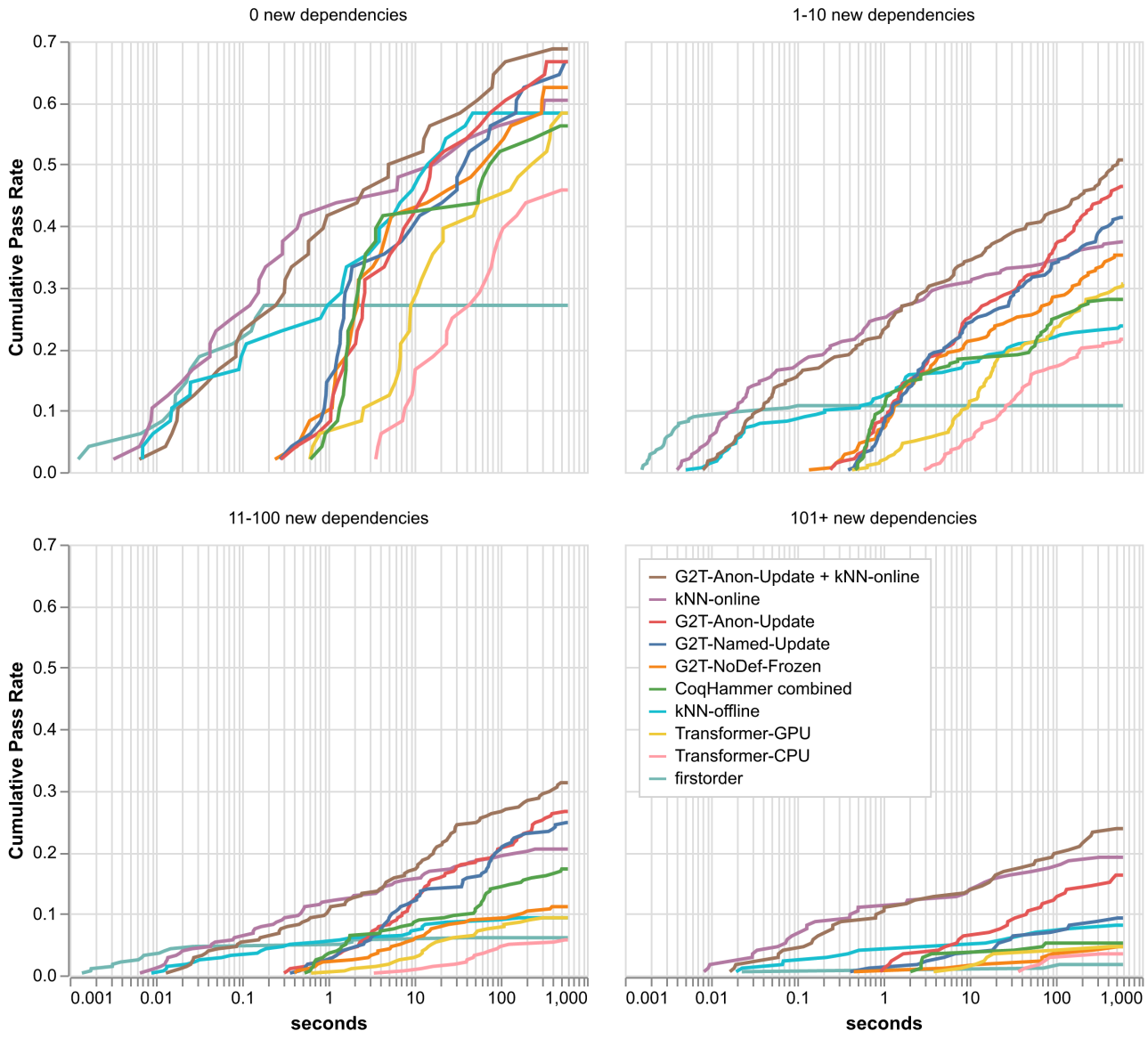

Figure 18. Cumulative pass rates as a function of time, split into four dependency categories.

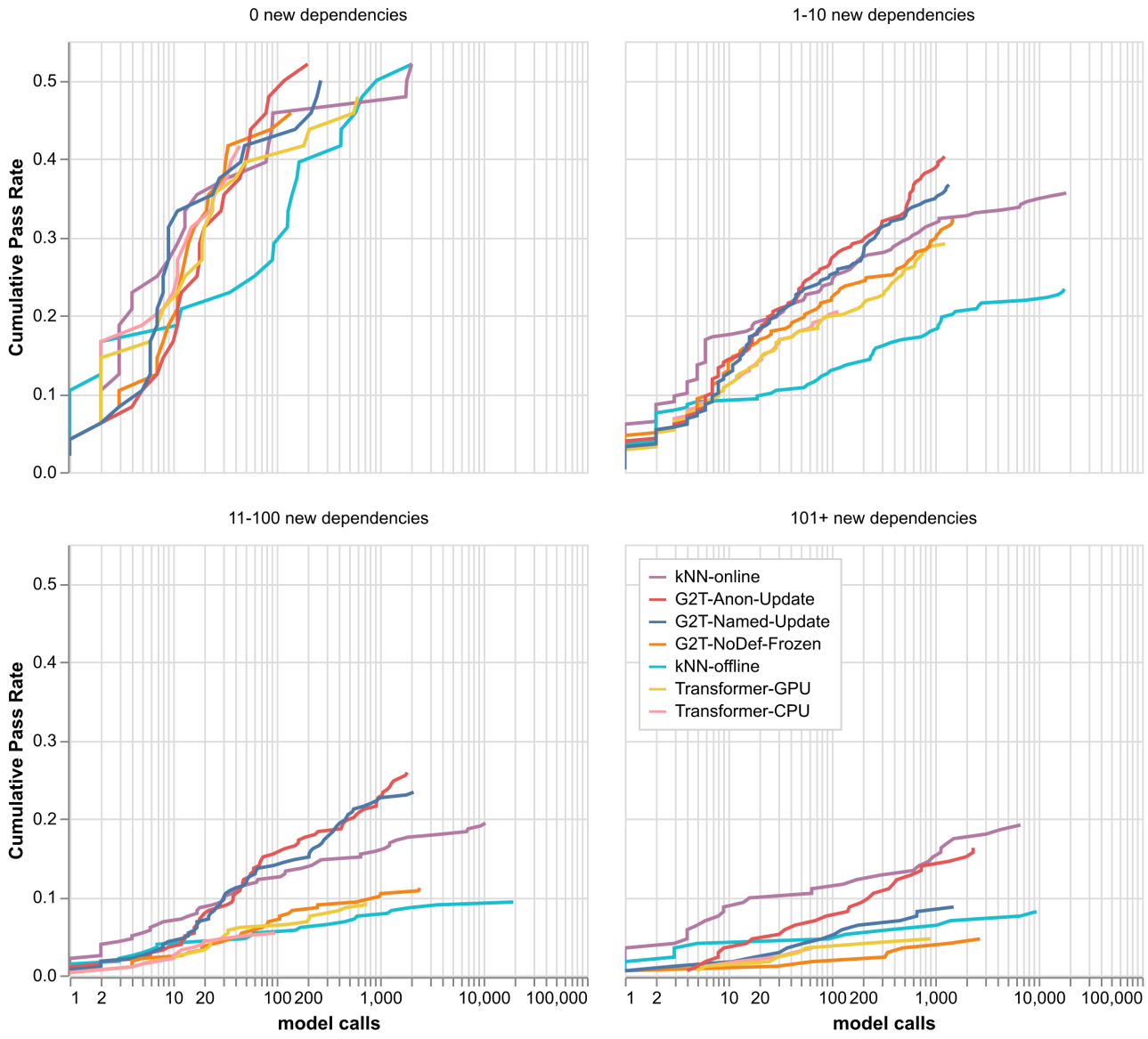

Figure 19. Cumulative pass rates as a function of model calls, split into four dependency categories.

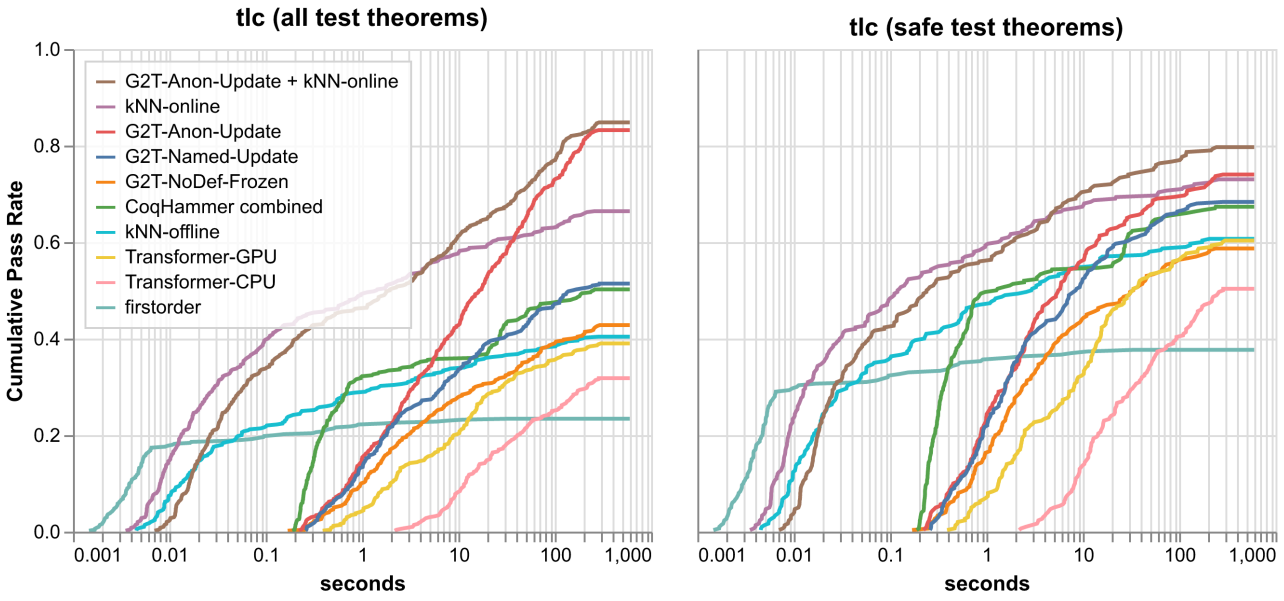

Figure 20. Pass rates on the *tlc* test theorems. The left plot shows the results on 500 theorems theorems. The right plot restricts to test theorems not solved using an inconsistent axiom by any of the benchmarked solvers.

R. Examples of Theorems Solved

Here we provide illustrative, cherry-picked examples of how different solvers are able to exploit different types of information.

Using a New Definition in a Theorem In this example from *odd-order*, the theorem `nsKM` is in a Coq section and depends on both the section variable `M` and a new definition `K`, defined to be the product of the normal groups of `M`.

```
Variables M P0 : {group gT}.
Let K := M`_\F%G.

Let nsKM : K <| M.
Proof. exact: gFnormal. Qed.
```

G2T-Anon-Update was the only model to find a proof, finding the same proof (on its first attempt) as the original using the lemma `gFnormal`.

```
Lemma gFnormal gT (G : {group gT})
  : F gT G <| G.
```

One possibility is that G2T-Anon-Update is using the hierarchical nature of our definition task, as one has to go back a few steps in the definition hierarchy to verify that `K` fits the form `F gT G` used in `gFnormal`.

Using a New Lemma in a Proof In this next example from *bits*, the lemma `toZp_inj` uses the immediately preceding new lemma `toZpK`, which was not seen during training.

```
Lemma toZpK n : cancel (@toZp n) (@fromZp n).
Proof. case E: (n == 0).

Lemma toZp_inj n : injective (@toZp n).
Proof. apply (can_inj (@toZpK _)). Qed.
```

G2T-Anon-Update and G2T-Named-Update can both solve this theorem with a similar proof: `apply : can_inj. exact : toZpK.`, something neither the offline models or nor the online k-NN was capable of.

However, the the online k-NN can solve a similar example `tiV` from *bits*, because `tiV` is the third in a list of facts, all using the same proof.

```
Let cycW : cyclic W.
Proof. by case: ctiW. Qed.

Let oddW : odd #|W|.
Proof. by case: ctiW. Qed.

Let tiV : normedTI V G W.
Proof. by case: ctiW. Qed.
```

The online k-NN can retrive that tactic `by case: ctiW.` since it is in its database of recent proofs. In this case, neither G2T-Anon-Update nor G2T-Named-Update could solve this, nor any of the offline solvers. (However, CoqHammer's `best` tactic was successful.)

Using New Tactics The online k-NN can also find and use new tactics, something none of the other solvers can do.

The package *hott* is very different from the training data, often using custom tactics, like the tactic `make_equiv` used in the proof of `equiv_sigma_assoc'`.

```
Definition equiv_sigma_assoc'
  `(P : A -> Type)
  (Q : forall a : A, P a -> Type)
  : {a : A & {p : P a & Q a p}}
    <~> {ap : sig P & Q ap.1 ap.2}.
Proof.
  make_equiv.
Defined.
```

It was easy for the online k-NN to find this proof as most theorems in this file used the same tactic, while none of the other solvers could solve this.

Predicting Tactics with Terms While the transformer model rarely solved a proof that the wasn't also solvable by the G2T models, there is an interesting example from *hott*. The proof of `meet_lb_3_assoc_m` relies on constructing the term `y ⊔ z`, something that, of all the machine-learned solvers in this paper, only the transformer is capable of.

```
Lemma meet_lb_3_assoc_m x y z
  : x ⊔ (y ⊔ z) ≤ y.
Proof.
transitivity (y ⊔ z).
- apply meet_lb_r.
- apply meet_lb_l.
Qed.
```

(This theorem was also proved by the firstorder tactic.)

Using Name Information Last, in *tlc*, consider the theorem `LibListZ.length_map`.

```
Lemma length_map :
  forall A B (l:list A) (f:A->B),
  length (map f l) = length l.
Proof using.
  intros. unfold length. rewrite~ length_map.
Qed.
```

Note, the proof of this theorem calls `LibList.length_map`:

```
Lemma length_map : forall f l,
  length (map f l) = length l.
```

While it looks like these two theorems says the exact same thing, the first theorem uses `LibListZ.length` which gives an integer length while the second uses `LibList.length` which gives a natural number length. Because G2T-Named-Update can see the name of the definitions, it had an advantage over G2T-Anon-Update in this case and was able to solve `LibListZ.length_map` with a similar proof, while G2T-Anon-Update did not.

S. Patterns in the Definition Embeddings

Our three models—G2T-Anon, G2T-Named, and G2T-NoDef—take different approaches in training the learned definition embeddings. In Figures 21, 22, and 23 we show UMAP representations of the learned definition embeddings.

Overall we see that G2T-Anon and G2T-Named have a more intricate structure. Since G2T-Anon does not have access to names, the embeddings are more focused on structure, including the top-level structure of what type of definition it is.

It is not clear what the major clusters of the G2T-Named model are, although we can determine it is not just related to easy-to-identify features of the name or the definition, but instead seems to be a combination of multiple factors.

As for the G2T-NoDef, since there is no definition task, the only signal used to train the embedding is how the definition is used, either as a node in a proof state or as an argument to a tactic. The center seems to be almost exclusively made up of non-theorem definitions. Further, we have noticed at least some of the radial clusters are associated with particular tactics. For example, we found a cluster almost exclusively made up of definitions used as an argument to `rewrite _`.

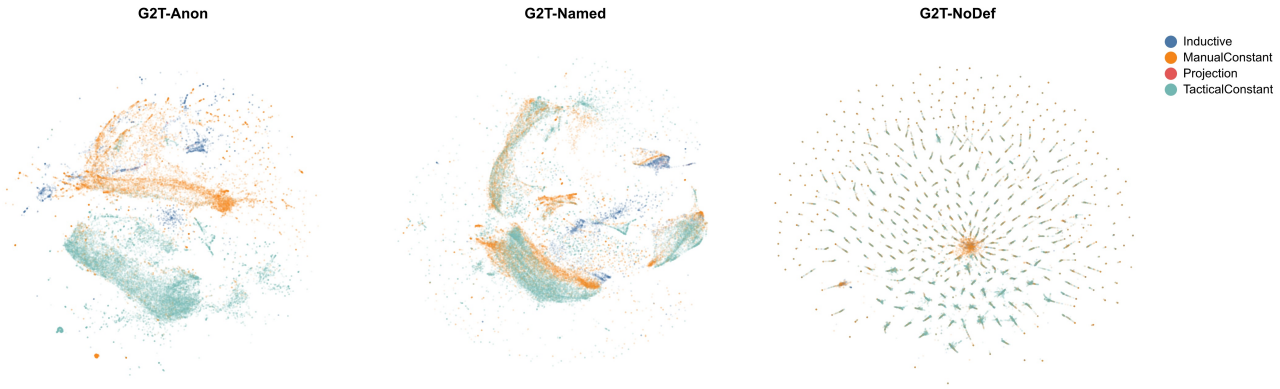

Figure 21. UMAP representations of learned definition embeddings colored by definition type. The G2T-Anon embeddings are clustered between tactical constants (mostly theorems with tactical proofs), manual constants (definitions defined via terms), and inductively defined definitions. (Projections in record types are relatively rare.) Manual and tactical constants are less separated in G2T-Named, but inductive definitions still do not mix. Manual constants make up a significant portion of the center cluster of the G2T-NoDef embeddings.

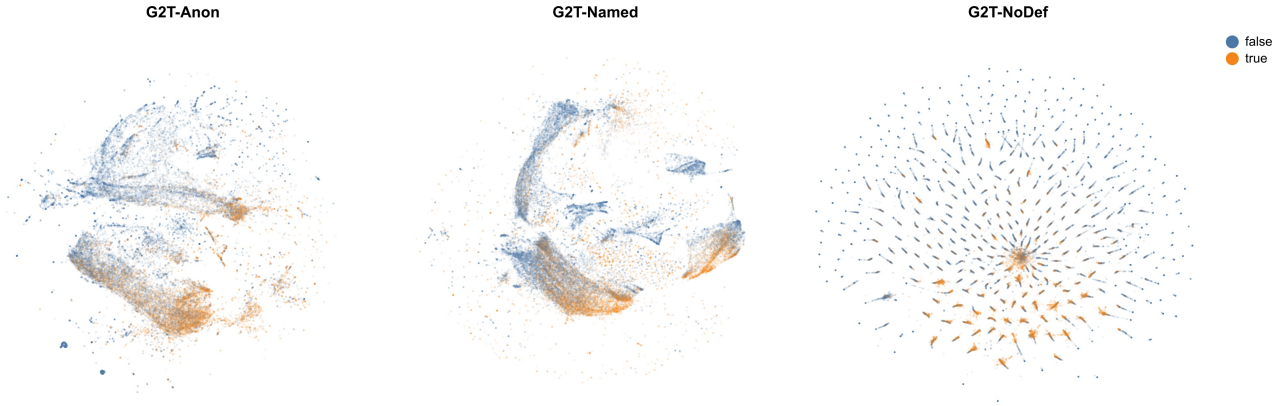

Figure 22. UMAP representations of learned definition embeddings colored by whether the definition was used as an argument of a tactic. All the model definition embeddings take this dimension into account. This makes sense since argument selection is an important part of tactic prediction.

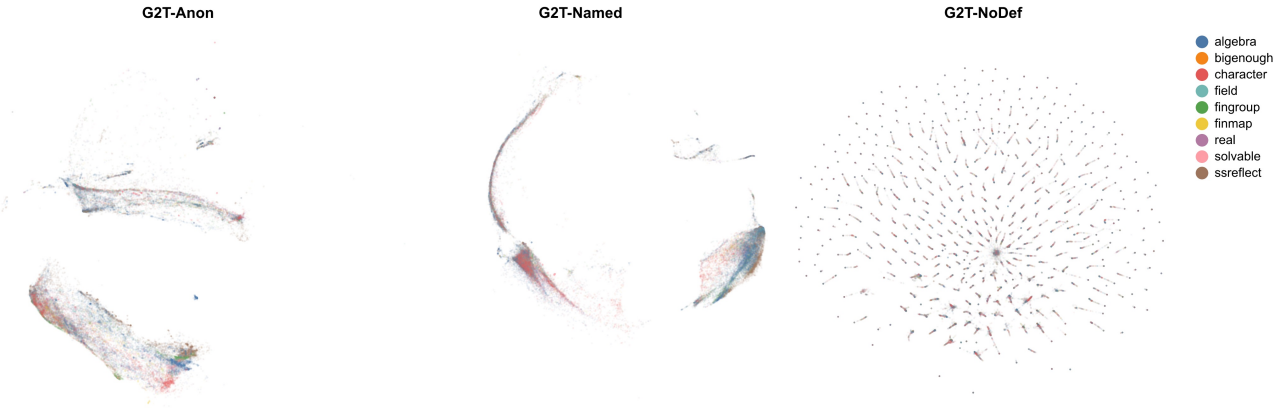

Figure 23. UMAP representations of learned definition embeddings restricted to definitions whose qualified names start with `mathcomp` colored by the second part of the name after `mathcomp`, *e.g.* `mathcomp.`**`algebra`**`.matrix.Matrix`. The G2T-Named model clusters more by these categories, whereas in the G2T-Anon they show as smoothed-out bands.