DIRECTED MULTI-RELATIONAL GCNS FOR PREMISE SELECTION IN LEAN

Anonymous authors
Paper under double-blind review

ABSTRACT

Premise selection is a central bottleneck for interactive theorem proving over large formal libraries. Semantic similarity can miss syntactic roles that govern premise applicability, while Lean elaboration can give similar expressions different tree representations through implicit arguments and typeclass evidence. We present R-DGCN, which learns from expression graphs designed to reduce this variation while retaining role distinctions. Retrieval preprocessing reduces recurring expression-tree variation. Typed edges encode constructor-field roles, and paired reverse edges expose subterms to their surrounding context. A relation-specific graph convolutional encoder combines mean and attention pooling to capture global structure and emphasize informative nodes in shared query–theorem embeddings. Precomputed theorem embeddings support retrieval, followed by directed collapse-match (CM) reranking that refines learned similarity using explicit structural coverage. Retrieval comparisons show complementary benefits from learned and structural scores, with recall gains most consistent at larger retrieval cutoffs. Precomputed theorem embeddings enable low online retrieval latency.

1 INTRODUCTION

Premise selection and theorem retrieval are recurring tasks in automated and interactive theorem proving: given a proof state or goal, a system retrieves a small set of potentially useful lemmas from a large theorem library under latency-sensitive constraints (Yang et al., 2023; Kühlwein et al., 2012; Bansal et al., 2019). Retrieval is a bottleneck for data-driven theorem proving and mathematical formalization (Yang et al., 2024; Li et al., 2024b). The same task occurs in proof assistants such as Lean (De Moura et al., 2015b; Moura & Ullrich, 2021), Coq (Barras et al., 1999), Isabelle (Paulson, 1994), HOL Light (Harrison, 1996), and Metamath (Megill & Wheeler, 2019). The expanding Lean libraries MATHLIB (mathlib Community, 2020) and Optlib (Li et al., 2025a;b; 2024a) increase the scale of this retrieval setting. Prior work addresses this retrieval bottleneck from several directions including semantic search tools, learned premise ranking, structure-aware matching, and retrieval-augmented proving. Lean search tools such as Lean State Search (Tao et al., 2025), Loogle! (Breitner, 2024), Moogle (Morph Labs, 2023), LeanExplore (Asher, 2025), LeanSearch (Gao et al., 2024), and LeanFinder (Lu et al., 2025) retrieve theorems through semantic, informal, or intent-aware representations of proof states, theorem statements, or mathematical expressions. Beyond semantic search tools, learning-based premise selection encodes conjectures and premises into a shared representation space for scalable ranking or approximate nearest-neighbor retrieval (Irving et al., 2016; Wang et al., 2017; Bansal et al., 2019), supported by benchmarks and environments such as HolStep (Kaliszyk et al., 2017), higher-order logic graph representations (Paliwal et al., 2020), and datasets for premise recommendation and mathematical formalization (Bauer et al., 2023). Retrieval-augmented proving systems further connect premise selection with downstream proving pipelines. ReProver augments LLM-guided proving with retrieved premises, while LeanHammer integrates premise selection with translation to external automated theorem provers and proof reconstruction (Yang et al., 2023; Zhu et al., 2025). Structure-aware approaches such as Tree-based Search (Wang et al., 2025) use explicit matching over Lean expression trees. Role-aware graph representations further distinguish constructor field positions such as functions, arguments, binder types, and bodies.

Lean elaboration makes implicit arguments, universe parameters, and coercions explicit in kernel terms (De Moura et al., 2015a; Moura & Ullrich, 2021), while typeclass resolution synthesizes instance arguments (Selsam et al., 2020; Baanen, 2022). Lean exposes these expressions as structured

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

terms with applications and binders through its metaprogramming interface (De Moura et al., 2015b; Ebner et al., 2017). The inserted arguments, instance evidence, and wrapper nodes can introduce recurring variation in expression trees. At the same time, structural ranking cues are role-sensitive: the same subterm can carry different information depending on whether it appears as an operator, operand, binder type, hypothesis, or goal body. Retrieval representations should therefore reduce recurring elaboration-induced variation while preserving syntactic roles.

We present R-DGCN, a Lean-specific hybrid retrieval system that combines role-aware expression graphs with learned similarity and explicit structural matching. Query and theorem expressions undergo deterministic canonicalization of guarded unused binders, typeclass wrappers, and small-literal arithmetic, followed by common-subexpression elimination (CSE) following Tree-based Search (Wang et al., 2025). The resulting expressions are represented as directed graphs whose typed edges distinguish constructor-field roles, with paired reverse relations carrying parent context to subterms. A relational GCN applies relation-specific message passing and combines mean and attention pooling to produce query and theorem embeddings. Contrastive training on theorem-use pairs aligns these embeddings for cosine retrieval. Theorem embeddings are precomputed offline and each online query is encoded once and scored against the theorem matrix. State-visible declarations are retained before ranking and semantic-shortlist construction. Building on the CM formulation of Tree-based Search, we define weighted directional coverage of a theorem tree by the query, using label-dependent weights and leaf-matching rules that depend on node type. The final ranking combines this structural coverage with learned similarity on the shortlist. An optional identifier-overlap signal supplies lexical cues for human-facing search. The central design combines Lean-specific preprocessing and a constructor-field relation schema with dense cosine scoring over precomputed theorem embeddings and weighted structural reranking. Controlled comparisons evaluate the encoder and the complementarity of the learned and structural scores. Our main contributions are as follows.

(i) Lean-specific retrieval preprocessing. We design a deterministic pipeline combining unused binder simplification, typeclass wrapper stripping and bounded small-literal arithmetic normalization for graph construction. These steps reduce variation in elaborated binder, instance evidence, and operator forms, giving the graph encoder a common syntactic representation for comparing queries and theorem statements. The source kernel expressions remain unchanged for Lean checking.

(ii) Role-aware directed representation. We define a relation schema from Lean constructor fields and pair each role with a reverse edge. The relational GCN encodes these graphs while retaining operator and binder–body distinctions. Paired reverse relations let subterms incorporate their context, so the learned representation reflects both local structure and its role within the expression.

(iii) Hybrid retrieval with structural coverage. We design a new weighted CM score with label-dependent weights and node-type-specific leaf-matching rules, and combine its directional coverage with learned similarity for shortlist reranking. Label weights emphasize constants and literals, while the leaf rules accommodate variation in variable and instance subtrees. Directional coverage measures how much of a candidate theorem's structure is represented in the query, complementing global embedding similarity with explicit structural correspondence.

2 PRELIMINARIES

2.1 LEAN EXPRESSIONS AND RETRIEVAL REPRESENTATION

Lean represents theorem statements and proof goals as kernel-checked typed expressions produced by elaboration (De Moura et al., 2015b; Moura & Ullrich, 2021). We view an expression e as an AST $T(e)$ whose nodes correspond to core constructors such as applications, binders, constants, literals, and variable references. Retrieval over Lean expressions must handle elaboration-induced surface variation, such as implicit arguments and typeclass evidence, together with the structural variation introduced by binders and variable references.

Premise retrieval setting. The base proof-state record is captured immediately before a theorem-using tactic step and contains the target and local context. Reverting the local hypotheses gives a goal-side expression that is then processed by the same canonicalization pipeline used for theorem statements. Both benchmark ranking and online search use this query representation.

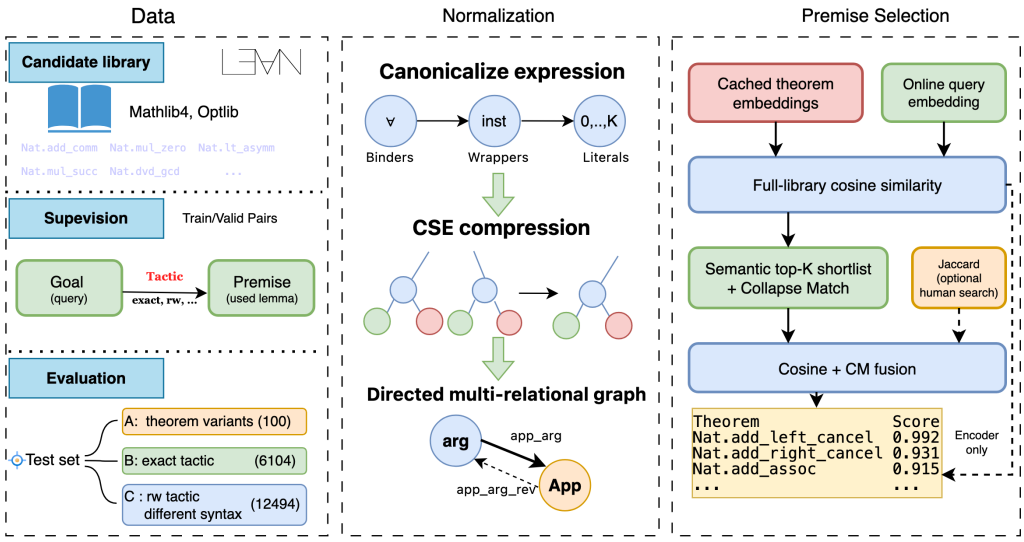

Figure 1: R-DGCN retrieval with offline theorem embeddings and online query encoding.

Given such a query expression e_q and a fixed theorem library $\mathcal{D} = \{e_1, \ldots, e_N\}$, we use $e_c \in \mathcal{D}$ for a theorem premise and write $\tilde{e}_q$ and $\tilde{e}_c$ when referring explicitly to their canonicalized forms. Their canonicalized syntax trees are denoted by $T_q = T(\tilde{e}_q)$ and $T_c = T(\tilde{e}_c)$. Premise retrieval ranks theorem entries by a scoring function $s(e_q, e_c)$ so that useful premises appear in the top-k results. We evaluate rankings against theorem-use supervision extracted from Lean proof traces.

2.2 Directed Multi-Relational Graph Formulation

Graph structure and directionality. For an expression e with syntax tree $T(e)$, we write its directed multi-relational graph as $G(e) = (V, E, \mathcal{R})$, where V is the node set, $\mathcal{R}$ is a finite set of relation types, and each typed edge $(u, v, r) \in E$ connects two nodes with relation $r \in \mathcal{R}$. The relation type records the structural role of the edge. For a query–theorem pair, we abbreviate canonicalized role-aware graphs as $G_q = G(\tilde{e}_q)$ and $G_c = G(\tilde{e}_c)$. Forward edges run from child nodes to parent nodes for bottom-up composition. For every forward relation r, we also introduce a distinct reverse relation r^{rev}, yielding $\mathcal{R}^{\pm} = \mathcal{R} \cup \{r^{\text{rev}} : r \in \mathcal{R}\}$. Each reverse relation has its own encoder parameters to propagate context from larger expressions back to their subterms.

Node labels and graph embeddings. Each node v has a normalized discrete label $\ell(v)$ recording its constructor category and payload when applicable. A vocabulary map ϕ initializes node features by embedding lookup, $\mathbf{x}_v = \mathbf{E}[\phi(\ell(v))]$. A graph encoder f_θ maintains a hidden state $\mathbf{h}_v^{(\ell)}$ for node v at layer ℓ and maps $G(e)$ to an expression embedding $\mathbf{z}_e = f_\theta(G(e))$. We write $\hat{\mathbf{z}}_e$ for its L_2-normalized form, which serves as the learned similarity interface for premise retrieval. The Lean-specific relation inventory and labels are described in Section 3.2.

3 Canonicalization and Role-Aware Representation

3.1 Canonicalization of Lean Expressions

Lean elaboration can produce varied expression trees for retrieval. We therefore apply a deterministic, retrieval-oriented structural abstraction before graph construction, $\tilde{e} = \mathcal{N}_{\text{arith}}(\mathcal{N}_{\text{wrap}}(\mathcal{N}_{\text{bind}}(e)))$, where $\mathcal{N}_{\text{bind}}$, $\mathcal{N}_{\text{wrap}}$, and $\mathcal{N}_{\text{arith}}$ are fixed rewrite families. The same rules are applied to training, evaluation, and theorem statements. They define the syntactic representation used for graph encoding and structural ranking. The source kernel expressions remain unchanged and are retained for Lean checking and evaluation. Appendix B lists the fixed rewrite rules and their structural effects.

Binder simplification. In the retrieval representation, we simplify an unused binder by $(\forall x : A,\ b) \longmapsto b$ when the body b does not use x and the binder type A satisfies a fixed syntactic guard. Removing the binder also lowers de Bruijn indices that refer to enclosing binders, preserving the scope of variable references. Appendix B specifies the guard and index adjustment.

Typeclass wrapper canonicalization. Lean elaborates overloaded notation through typeclass interfaces. The wrapper step strips selected Nat-level instance-evidence subtrees when the head, arity, type arguments, and name-suffix pattern match. For example, `LT.lt Nat Nat.instLTNat` $x\ y$ becomes `Nat.lt` $x\ y$. This represents a matching application by its operator and operands for retrieval. Appendix B.2 gives the complete trigger list and rewrite forms.

Small-literal arithmetic normalization. After wrapper canonicalization, natural-number additions with small literal offsets are rewritten as `Nat.add` $t\ k \longmapsto$ `Nat.succ`$^k(t)$ for $1 \le k \le K_{\text{lit}} = 10$, where k is a natural number literal and `Nat.succ`^{k} denotes k successor applications. For example, `Nat.add` $x\ 2$ becomes `Nat.succ`$(\texttt{Nat.succ}(x))$. This gives small offsets a common syntactic form for structural comparison and larger literals and non-matching forms remain unchanged.

3.2 Role-Aware Directed Multi-Relational Graph Construction

After preprocessing, we apply common-subexpression elimination (CSE), following Tree-based Search (Wang et al., 2025), to queries and theorem statements. Repeated non-constant subexpressions are replaced by variable placeholders within the serialized syntax tree, with each occurrence stored as a separate leaf. CM uses this compressed tree, and graph construction assigns typed edges to its constructor-field links using the formulation in Section 2.2. The resulting graphs are materialized offline for theorem entries and constructed with the same schema for online queries.

Syntax-role mapping via constructor fields. To represent how a subexpression participates in its parent, we derive edge types directly from Lean constructor fields. If a child node u occupies field f of a parent node v with constructor τ, we add the typed edge $(u, v, r_{\tau,f})$. The relation identifies the syntactic operation through τ and the child's role through f, making both available to message passing. Native binary `App` structure is retained: each application has one `app_fn` edge and one `app_arg` edge. `Lam` and `ForallE` distinguish binder types from bodies, while `LetE` separates type, value, and body components. This schema lets the encoder learn role-specific transformations when composing subterms, so a symbol's contribution can depend on whether it appears as an operator, argument, binder type, or body.

Paired reverse relations. The graph supports both the composition of subterms and the propagation of their enclosing context. Forward edges run from child to parent, allowing a constructor to aggregate information from its fields. For every forward relation r, we add a reverse relation r^{rev}. An argument-to-application edge $(u \xrightarrow{\texttt{app_arg}} v)$ is thus paired with $(v \xrightarrow{\texttt{app_arg_rev}} u)$. The encoder assigns separate transformations $\mathbf{W}_r^{(\ell)}$ and $\mathbf{W}_{r^{\text{rev}}}^{(\ell)}$ to the two directions, allowing bottom-up composition and top-down contextualization to learn different functions. Across successive layers, information can also pass between fields through their shared parent: function-side features can reach an argument along `app_fn` followed by `app_arg_rev`. A subterm can therefore be encoded using both its local structure and the context in which it is used, with field identity retained along each step.

Structure-aware label abstraction and vocabulary control. Node labels supply symbol content, while edge relations specify its structural use. We apply category-specific label normalization to share embeddings across recurring expression patterns and control the vocabulary size. Internal nodes retain their constructor categories, and constants retain a bounded namespace suffix. Typeclass-instance constants are grouped into arithmetic, ordering, divisibility, or other instances. Natural number literals use fine-grained buckets up to the preprocessing bound and a shared overflow bucket and bound variables use de Bruijn-depth buckets (0, 1, 2, and 3+), while free and meta variables are de-identified. These choices expose constructor identity, coarse literal magnitude, and binding depth to the encoder while reducing dependence on expression-specific names. A single vocabulary built from the canonicalized theorem corpus maps normalized labels to node embeddings, with `<UNK>` handling unseen labels. Applying the same map to queries and theorems provides a common node-feature space in which relation-specific message passing combines symbol content with syntactic context.

4 R-DGCN Retrieval and Fusion Ranking

R-DGCN instantiates a relational GCN (Schlichtkrull et al., 2018) on the role-aware Lean graphs defined above. Retrieval scores the theorem embedding matrix, filters each query to state-visible declarations, and optionally applies CM reranking to the resulting semantic shortlist. Figure 2 illustrates the encoder and scoring workflow.

4.1 R-DGCN Encoder and Similarity Scoring

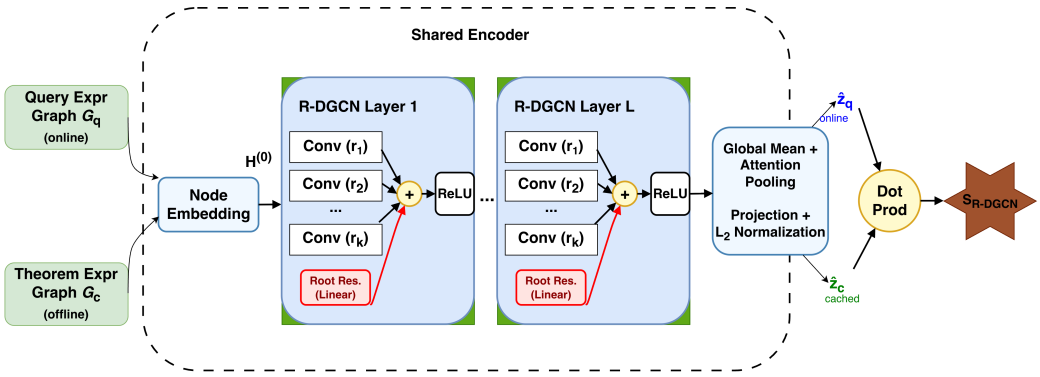

Figure 2: Shared R-DGCN encoder for offline theorem encoding and online query encoding.

Relation-specific message passing and node updates. Given a canonicalized expression graph $G(\tilde{e}) = (V, E, \mathcal{R})$ with paired relations $\mathcal{R}^{\pm}$, each node $v \in V$ is initialized by a label embedding $\mathbf{h}_v^{(0)} = \mathbf{x}_v \in \mathbb{R}^d$. For an L-layer encoder, messages at layer $\ell \in \{0, \ldots, L-1\}$ are aggregated separately for each relation type. Let $\mathcal{N}_r(v)$ be the set of neighbors u with a directed edge of type r from u to v, and let $c_{v,r} = \max(1, |\mathcal{N}_r(v)|)$ denote the relation-specific normalization. We compute

$$\mathbf{m}_v^{(\ell)} = \sum_{r \in \mathcal{R}^{\pm}} \sum_{u \in \mathcal{N}_r(v)} \frac{1}{c_{v,r}} \mathbf{W}_r^{(\ell)} \mathbf{h}_u^{(\ell)}, \tag{1}$$

where $\mathbf{W}_r^{(\ell)} \in \mathbb{R}^{d_{\ell+1} \times d_\ell}$ is a learnable transformation for relation r at layer ℓ. The directed multi-relational convolutions are applied without relation-level self-loops, so Eq. (1) aggregates only typed neighbor information. To retain each node's own features, we add an explicit root residual at every layer: $\tilde{\mathbf{h}}_v^{(\ell+1)} = \mathbf{m}_v^{(\ell)} + \mathbf{W}_{\text{self}}^{(\ell)} \mathbf{h}_v^{(\ell)}$. The result is then passed through BatchNorm and ReLU.

Pooling and similarity scoring. For premise selection, the final-layer node states $\{\mathbf{h}_v^{(L)}\}_{v \in V}$ are summarized by two complementary pooling operators. Mean pooling captures the global expression shape: $\mathbf{z}_{\text{mean}} = \frac{1}{|V|} \sum_{v \in V} \mathbf{h}_v^{(L)}$. To focus on structural anchors such as constants, predicates, and literals, we also compute attention weights. A small feed-forward scorer produces $s_v = \mathbf{w}_2^\top \text{ReLU}(\mathbf{W}_1 \mathbf{h}_v^{(L)})$, which is normalized as $a_v = \exp(s_v) / \sum_{u \in V} \exp(s_u)$. The attention readout is $\mathbf{z}_{\text{att}} = \sum_{v \in V} a_v \mathbf{h}_v^{(L)}$. We concatenate $[\mathbf{z}_{\text{mean}}; \mathbf{z}_{\text{att}}]$ and apply a two-layer projection with a ReLU nonlinearity. The output is L_2-normalized to obtain the final expression embedding $\hat{\mathbf{z}}_e$.

For a query expression e_q and a theorem expression e_c, the graph-embedding similarity is the cosine score $S_{\text{R-DGCN}}(e_q, e_c) = \hat{\mathbf{z}}_{e_q}^\top \hat{\mathbf{z}}_{e_c}$. Theorem embeddings are precomputed with the same preprocessing and checkpoint and stacked in frozen order as $\hat{\mathbf{Z}}_{\mathcal{D}} \in \mathbb{R}^{218415 \times 64}$. Only the query is encoded online. Similarity scores are

$$\mathbf{r} = \hat{\mathbf{Z}}_{\mathcal{D}} \hat{\mathbf{z}}_{e_q} \in \mathbb{R}^{|\mathcal{D}|}. \tag{2}$$

Encoder-only retrieval ranks by cosine, breaking ties by theorem order. Fusion uses $S_{\text{emb}} = (S_{\text{R-DGCN}} + 1)/2 \in [0, 1]$.

Training objective. For a batch of queries with normalized embeddings $\hat{\mathbf{q}}_i$ and candidate premises with embeddings $\hat{\mathbf{p}}_j$, define $s_{ij} = \hat{\mathbf{q}}_i^\top \hat{\mathbf{p}}_j / \tau$, with $\tau = 0.07$. Let $\mathcal{P}_i$ be the set of candidate columns whose premise identifiers are known positives for query i, and let $\mathcal{N}_i$ be the remaining sampled columns. The multi-positive InfoNCE objective is $\mathcal{L} = -B^{-1} \sum_{i=1}^{B} \log \frac{\sum_{j \in \mathcal{P}_i} \exp(s_{ij})}{\sum_{j \in \mathcal{P}_i \cup \mathcal{N}_i} \exp(s_{ij})}$. Each $\mathcal{N}_i$ contains up to 32 premise-unique negatives from the query's node-count bucket, selected by the fixed WL-guided hard/easy policy (Appendix D.6).

4.2 Retrieval and Fusion Ranking

Lean-aware token overlap. Qualified Lean declaration identifiers often encode useful library-level information through namespaces and conventional lemma suffixes. When a human or search interface explicitly supplies such an identifier-like query, we tokenize it by splitting namespace segments, underscores, CamelCase fragments, and common lemma-shape tokens such as comm, assoc, left/right, and inj/surj. Let $\tau(\cdot)$ denote the resulting token set, and let ι_q and ι_c denote the supplied query identifier and theorem declaration identifier. The identifier-overlap score is $S_{\mathrm{Jac}}(\iota_q, \iota_c) = \frac{|\tau(\iota_q) \cap \tau(\iota_c)|}{|\tau(\iota_q) \cup \tau(\iota_c)|}$. This Jaccard signal captures namespace, API-family, domain, object-type, and operation cues. In identifier-assisted human search, it provides an auxiliary lexical feature for bounded reranking alongside the structural scores.

Directed collapse-match. Building on the CM formulation of Wang et al. (2025), we define weighted directional coverage of the theorem tree by the query tree, using label-dependent node weights on canonicalized syntax trees. Variable and instance leaves contribute their weight for any nonempty query subtree. Other leaves contribute their weight when the same label occurs in that subtree. This explicit structural signal complements learned R-DGCN similarity. Let T_q and T_c denote the query and theorem trees, and let $\mathrm{Match}(T_q, T_c)$ be the maximum covered theorem weight under the admissible query-side collapses. With $W(T_c) = \sum_{v \in T_c} w(\ell(v))$ denoting the total theorem-tree weight, we define

$$S_{\mathrm{CM}}(T_q, T_c) = \frac{\mathrm{Match}(T_q, T_c)}{W(T_c)}. \tag{3}$$

Appendix C.4 gives the label comparisons, leaf rules, weights, recursion, and memoization. Within $\mathcal{C}^K_{\mathrm{emb}}(e_q)$, the fusion score combines the embedding score with directed collapse-match using coefficients α and γ: $S_{\mathrm{fuse}}(e_q, e_c) = \alpha\, S_{\mathrm{emb}}(e_q, e_c) + \gamma\, S_{\mathrm{CM}}(T_q, T_c)$, with $(\alpha, \gamma) = (0.7, 0.2)$. When a user explicitly supplies an identifier for human search, the optional identifier-assisted mode augments this score with coefficient β: $S(e_q, e_c) = S_{\mathrm{fuse}}(e_q, e_c) + \beta\, S_{\mathrm{Jac}}(\iota_q, \iota_c)$, with $\beta = 0.1$.

5 Numerical Experiments

5.1 Data and Evaluation Setup

The theorem library contains $|\mathcal{D}| = 218{,}415$ valid mathematical theorems extracted from Lean4's Mathlib4 v4.27.0-rc1. Test-A contains 100 queries and Test-B contains 6,104 queries, derived from Wang et al. (2025) and updated to align with the Mathlib4 version used in this work. Test-C contains 12,494 `rw` proof steps. Evaluation query records are deduplicated by query declaration name. Each evaluation unit is one retained query record with its recorded premise label; the declaration name identifies the query's parent declaration. Training and validation use a 90/10 record-level split of the selected supervision pairs. Appendices A and A.8 give the data details and metric definitions. The ranking tables measure retrieval of the recorded premises from state-visible candidates. All R-DGCN benchmark results and controlled comparisons reported in Tables 1–5 are name-free: query declaration identifiers are not exposed to the ranker, and $\beta = 0$. Query declaration names are retained as metadata for evaluation deduplication and parent-declaration exclusion.

Tool-level reference comparison. Table 1 compares R-DGCN with existing Lean search tools across the three query sets, examining retrieval coverage and the positions of relevant premises under the reported tool configurations. These are tool-level reference results, and LeanPremise follows its native target-eligibility and hit criteria (Appendix A.7). The Tree-based Search Test-B results are

Table 1: Tool-level reference comparison on Test-A/B/C (%).

Test Set	Method	Recall			Precision			F1			nDCG			MRR
		@1	@5	@10	@1	@5	@10	@1	@5	@10	@1	@5	@10	
Test-A	Lean Search	2.0	4.0	4.0	2.0	0.8	0.4	2.0	1.3	0.7	2.0	3.1	3.1	2.9
	Moogle	6.0	10.0	10.0	6.0	2.0	1.0	6.0	3.3	1.8	6.0	8.3	8.3	7.8
	Lean Explore	16.0	37.0	50.0	16.0	7.4	5.0	16.0	12.3	9.1	16.0	26.7	30.9	26.5
	Lean State Search	12.0	18.0	24.0	12.0	3.6	2.4	12.0	6.0	4.4	12.0	14.9	16.8	15.1
	LeanPremise	3.0	7.0	10.0	3.0	1.4	1.0	3.0	2.3	1.8	3.0	5.3	6.2	5.8
	Tree-based Search	82.0	84.0	86.0	82.0	16.8	8.6	82.0	28.0	15.6	82.0	83.1	83.7	83.3
	R-DGCN system	**94.0**	**99.0**	**99.0**	**94.0**	**19.8**	**9.9**	**94.0**	**33.0**	**18.0**	**94.0**	**96.6**	**96.6**	**95.9**
Test-B	Lean Search	14.6	25.2	28.0	14.6	5.0	2.8	14.6	8.4	5.1	14.6	20.4	21.3	19.2
	Moogle	6.4	13.4	16.3	6.4	2.7	1.6	6.4	4.5	3.0	6.4	10.1	11.1	9.6
	Lean Explore	9.2	17.2	20.8	9.2	3.4	2.1	9.2	5.7	3.8	9.2	13.4	14.5	13.0
	Lean State Search	8.7	17.9	22.0	8.7	3.6	2.2	8.7	6.0	4.0	8.7	13.5	14.8	13.0
	LeanPremise	6.1	13.2	17.3	6.1	2.6	1.7	6.1	4.4	3.1	6.1	9.8	11.1	9.9
	Tree-based Search	25.8	28.0	28.7	25.8	5.6	2.9	25.8	9.4	5.2	25.8	27.0	27.2	26.9
	R-DGCN system	**29.5**	**41.8**	**64.4**	**29.5**	**8.4**	**6.4**	**29.5**	**13.9**	**11.7**	**29.5**	**35.9**	**43.4**	**37.9**
Test-C	Lean Search	7.2	18.2	24.1	7.2	3.6	2.4	7.2	6.1	4.4	7.2	12.9	14.8	12.5
	Moogle	4.6	12.0	15.5	4.6	2.4	1.6	4.6	4.0	2.8	4.6	8.4	9.5	8.0
	Lean Explore	6.4	15.7	21.0	6.4	3.1	2.1	6.4	5.2	3.8	6.4	11.2	12.9	11.1
	Lean State Search	7.6	18.3	24.7	7.6	3.7	2.5	7.6	6.1	4.5	7.6	13.1	15.2	13.1
	LeanPremise	2.6	6.3	8.6	2.6	1.3	0.9	2.6	2.1	1.6	2.6	4.5	5.2	4.6
	Tree-based Search	18.6	26.5	31.4	18.6	5.3	3.1	18.6	8.8	5.7	18.6	22.7	24.3	23.2
	R-DGCN system	**19.0**	**34.2**	**42.1**	**19.0**	**6.8**	**4.2**	**19.0**	**11.4**	**7.6**	**19.0**	**27.0**	**29.5**	**26.9**

evaluated on the current 6,104-query manifest with Mathlib4 v4.27.0-rc1. Recall@k, Precision@k, F1@k, and nDCG@k at $k \in \{1, 5, 10\}$, together with MRR, summarize these aspects. Section 5.3 examines individual design choices through encoder comparisons and system ablations.

R-DGCN records higher recall and MRR across the three query sets in this tool-level reference comparison under each tool's stated evaluation rules (Table 1). On Test-A, it reaches 94% R@1 and 99% R@5, compared with 82% and 84% for Tree-based Search, placing the recorded premise first for most queries. The largest R@10 gain appears on Test-B, where recall increases from 28.7% to 64.4%, a difference of 35.7 percentage points. On Test-C, R@10 rises from 31.4% to 42.1%, extending the retrieval advantage to the rewrite-oriented query set. Higher MRR on both sets also indicates improved placement of recorded premises. The larger recall gains at @10 on Test-B and Test-C are consistent with combining learned relevance and explicit structural coverage. Constructor-field relations distinguish operator, argument, binder-type, and body positions, while reverse edges supply the surrounding expression context. Training on theorem-use pairs lets the encoder associate these role-sensitive patterns with premise relevance. CM then measures how much of a candidate theorem's weighted structure is covered by the query, adding an explicit matching signal to the learned similarity. These two views provide complementary ranking cues when a theorem shares structure with part of a larger goal, as can occur in rewrite queries. The controlled comparisons below examine the encoder and fusion scores on Test-B.

5.2 Retrieval Efficiency Results

Retrieval latency. Table 2 examines the cost of returning a ranked premise list during proof search with precomputed theorem embeddings. For Test-B, R-DGCN computes exact dense cosine scores against the precomputed theorem-embedding matrix, and the complete system reranks the semantic top-2,000 with CM. R-DGCN-only returns the cosine ranking without CM. The same R-DGCN candidate protocol is used in the retrieval-quality and latency evaluations. Tree-based Search and both R-DGCN variants are measured on the same hardware with the same per-query timer boundaries, from query processing to the final ranked premise list. Appendix A.6 specifies the timing configuration.

The R-DGCN system reduces median latency from 79.44 to 0.39 seconds relative to Tree-based Search, a roughly 200× speedup. P95 latency falls from 268.44 to 2.80 seconds, showing that the speed advantage extends to slower queries. R-DGCN-only returns a ranking in 19 milliseconds at the median and 23 milliseconds at P95, with little variation between these percentiles.

Table 2: Test-B retrieval latency (seconds/query). P50–P95 are percentiles and $\Delta = \mathrm{P95} - \mathrm{P50}$.

Method	P50	P75	P90	P95	Mean	Δ
Tree-based Search	79.44	102.29	206.76	268.44	91.54	189.00
R-DGCN-only	**0.019**	**0.020**	**0.021**	**0.023**	**0.020**	**0.0047**
R-DGCN system	**0.39**	**1.00**	**1.96**	**2.80**	**0.79**	**2.40**

The pipeline amortizes theorem encoding across queries by storing theorem embeddings offline. Each online query is encoded once, and library-wide similarity scores are computed against the cached embedding matrix. This organization places the broad candidate search in dense vector scoring and reserves tree matching for the semantic top-2,000. CM therefore adds explicit structural coverage on a fixed-size shortlist, while the encoder handles retrieval across the library. The complete system maintains a mean latency of 0.79 seconds with this reranking stage, combining structural refinement with subsecond average retrieval time.

5.3 Ablation Studies

The Test-B ablations examine the choice of expression representation and the contributions of score fusion, graph structure, and preprocessing.

Encoder and shortlist-scoring comparisons. Table 3 examines whether encoding Lean's constructor-field relationships improves premise ranking relative to a relation-agnostic tree encoder and textual or sequential encoders. The trained Transformer tests sequence encoding as an alternative to graph message passing. The fusion rows test whether adding CM to trained encoder scores improves ranking. The structural fingerprint separately rescores the fixed R-DGCN top-2,000 shortlist. Appendix D.8 gives the candidate and scoring settings for each representation.

Table 3: Encoder and shortlist-scoring comparisons on Test-B (%).

Encoder / representation	R@1	R@5	R@10	MRR
Standalone encoder comparisons				
Relation-agnostic tree encoder	8.09	26.75	27.33	17.47
Frozen sentence encoder (Lean strings)	10.53	17.87	21.79	14.36
Capacity-matched Transformer	13.02	19.68	23.75	17.32
R-DGCN-only	**20.95**	**28.05**	**47.58**	**26.92**
Shortlist rescoring and fusion				
Capacity-matched Transformer + CM	17.15	25.85	30.10	21.80
Structural fingerprint	14.12	21.02	42.04	19.47
Structural fingerprint + CM	13.02	19.45	41.24	18.19
R-DGCN system	**29.51**	**41.84**	**64.40**	**37.91**

R-DGCN-only leads the standalone encoders on all four metrics, with R@1 of 20.95% compared with 13.02% for the trained Transformer. Its relation-specific message passing directly aggregates operator–argument and binder–body structure, providing a structural bias for premise ranking. CM complements learned similarity with explicit weighted tree coverage, raising R-DGCN's R@10 from 47.58% to 64.40% and the Transformer's from 23.75% to 30.10%. These gains indicate that structural coverage refines rankings from different learned representations. The fingerprint rows separately assess handcrafted structural rescoring on the fixed R-DGCN top-2,000 shortlist.

System and encoder ablation. Table 4 examines three design choices. The component comparison tests the benefit of combining learned similarity and weighted directional coverage and CM reranking shares the complete system's embedding shortlist. The structural controls compare the complete relation schema with two coarser encodings: directed untyped GCN merges constructor-field roles, whereas role-only undirected GCN merges forward and reverse relation channels. The preprocessing comparison tests the full pipeline against binder-only, wrapper-only, and arithmetic-only variants. All preprocessing variants use the same CSE pass and encoder-only cosine ranking. Appendix D.8 defines the structural controls and their training settings.

Table 4: System, encoder, and preprocessing ablation on Test-B (%).

System condition	R@1	R@5	R@10	MRR
Component deletion				
R-DGCN system	**29.51**	**41.84**	**64.40**	**37.91**
R-DGCN-only	20.95	28.05	47.58	26.92
CM reranking	23.53	40.24	42.69	32.07
Structural encoder controls				
R-DGCN-only	**20.95**	**28.05**	**47.58**	**26.92**
Directed untyped GCN	15.65	22.20	42.99	21.68
Role-only undirected GCN	2.24	23.38	27.70	9.40
Preprocessing training ablations				
R-DGCN-only (full preprocessing)	**20.95**	**28.05**	**47.58**	**26.92**
Raw + binder simplification	11.16	18.43	25.80	16.15
Raw + wrapper stripping	11.99	19.53	26.79	17.05
Raw + arithmetic normalization	12.45	19.89	27.34	17.54

The fused score achieves 64.40% R@10, exceeding R-DGCN-only by 16.82 percentage points and CM reranking by 21.71 points. The two scores have complementary ranking strengths: CM reranking places more recorded premises in the top one and top five, while R-DGCN-only retrieves more within the top ten. Their combination improves all four metrics over either score alone, supporting fusion of learned similarity with weighted directional coverage on the shared shortlist. The complete role-and-direction configuration leads both tested structural controls on all four metrics. Its R@10 is 47.58%, versus 42.99% for directed untyped GCN and 27.70% for role-only undirected GCN. For preprocessing, the full pipeline reaches 47.58% R@10, compared with 25.80% for binder simplification and 26.79% for wrapper stripping. Its advantage across all four metrics supports combining the preprocessing rules when learning from Lean expressions.

Variation across seed checkpoints. To examine whether fusion gains persist across training runs, Table 5 compares five matched encoder-only and fused runs on Test-B. These include seed 42, used for the component comparisons in Table 4.

Table 5: Test-B results across five seeds (mean $\pm$ standard deviation, %).

Method	R@1	R@5	R@10	MRR
R-DGCN-only	19.57 ± 1.11	26.57 ± 1.14	46.78 ± 1.02	25.58 ± 1.07
R-DGCN system	28.19 ± 4.27	38.79 ± 4.14	60.73 ± 3.72	35.96 ± 4.06

Fusion raises mean R@10 from 46.78% to 60.73%, a gain of 13.95 percentage points, and improves all four metrics in every matched run. The benefit of combining learned similarity with structural coverage therefore recurs across separately trained encoders. Appendix D.10 gives the per-seed results.

6 Conclusion

We introduce a Lean premise retrieval system built around a role-sensitive structural representation and a scoring pipeline that combines Lean-specific canonicalization and common-subexpression compression, directed multi-relational graphs preserving constructor-field roles, an R-DGCN encoder that maps queries and theorems into a shared embedding space, and embedding search with directed CM reranking. The tool-level results report R-DGCN retrieval coverage across the three benchmarks, while the controlled Test-B comparisons show gains from score fusion. In the latency benchmark, the complete system uses precomputed theorem embeddings and achieves a median latency of 0.39 seconds, with lower median and tail retrieval times than Tree-based Search. The ablations support the complementary use of learned similarity and explicit structural coverage for premise ranking.

AI USE STATEMENT

Generative AI tools were used to assist with language editing, LaTeX and document formatting, and the preparation of scientific figures. They were not used to formulate the research hypotheses, develop the proposed methodology, derive mathematical claims or proofs, design the experiments, or generate or clean the experimental data. All AI-assisted text, code, and figures were reviewed and, where applicable, tested or independently verified by the authors, who take full responsibility for the final content and claims.

REFERENCES

Justin Asher. Leanexplore: A search engine for lean 4 declarations. *arXiv preprint arXiv:2506.11085*, 2025.

Anne Baanen. Use and abuse of instance parameters in the lean mathematical library. *arXiv preprint arXiv:2202.01629*, 2022.

Kshitij Bansal, Sarah Loos, Markus Rabe, Christian Szegedy, and Stewart Wilcox. Holist: An environment for machine learning of higher order logic theorem proving. In *International Conference on Machine Learning*, pp. 454–463. PMLR, 2019.

Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Yann Coscoy, David Delahaye, Daniel de Rauglaudre, Jean-Christophe Filliâtre, Eduardo Giménez, Hugo Herbelin, et al. The coq proof assistant reference manual. *INRIA, version*, 6(11):17–21, 1999.

Andrej Bauer, Matej Petković, and Ljupco Todorovski. Mlfmf: data sets for machine learning for mathematical formalization. *Advances in Neural Information Processing Systems*, 36:50730–50741, 2023.

Joachim Breitner. Loogle! `https://loogle.lean-lang.org/`, 2024.

Leonardo De Moura, Jeremy Avigad, Soonho Kong, and Cody Roux. Elaboration in dependent type theory. *arXiv preprint arXiv:1505.04324*, 2015a. doi: 10.48550/arXiv.1505.04324. URL `https://arxiv.org/abs/1505.04324`.

Leonardo De Moura, Soonho Kong, Jeremy Avigad, Floris Van Doorn, and Jakob von Raumer. The lean theorem prover (system description). In *International Conference on Automated Deduction*, pp. 378–388. Springer, 2015b.

Gabriel Ebner, Sebastian Ullrich, Jared Roesch, Jeremy Avigad, and Leonardo de Moura. A metaprogramming framework for formal verification. *Proceedings of the ACM on Programming Languages*, 1(ICFP):1–29, 2017. doi: 10.1145/3110278. URL `https://doi.org/10.1145/3110278`.

Guoxiong Gao, Haocheng Ju, Jiedong Jiang, Zihan Qin, and Bin Dong. A semantic search engine for mathlib4. *arXiv preprint arXiv:2403.13310*, 2024.

John Harrison. Hol light: A tutorial introduction. In *International Conference on Formal Methods in Computer-Aided Design*, pp. 265–269. Springer, 1996.

Geoffrey Irving, Christian Szegedy, Alexander A Alemi, Niklas Eén, François Chollet, and Josef Urban. Deepmath-deep sequence models for premise selection. *Advances in neural information processing systems*, 29, 2016.

Cezary Kaliszyk, François Chollet, and Christian Szegedy. Holstep: A machine learning dataset for higher-order logic theorem proving. *arXiv preprint arXiv:1703.00426*, 2017.

Daniel Kühlwein, Twan van Laarhoven, Evgeni Tsivtsivadze, Josef Urban, and Tom Heskes. Overview and evaluation of premise selection techniques for large theory mathematics. In *International Joint Conference on Automated Reasoning*, pp. 378–392. Springer, 2012.

Chenyi Li, Ziyu Wang, Wanyi He, Yuxuan Wu, Shengyang Xu, and Zaiwen Wen. Formalization of complexity analysis of the first-order algorithms for convex optimization. *arXiv preprint arXiv:2403.11437*, 2024a.

Chenyi Li, Zichen Wang, Yifan Bai, Yunxi Duan, Yuqing Gao, Pengfei Hao, and Zaiwen Wen. Formalization of algorithms for optimization with block structures. *arXiv preprint arXiv:2503.18806*, 2025a.

Chenyi Li, Shengyang Xu, Chumin Sun, Li Zhou, and Zaiwen Wen. Formalization of optimality conditions for smooth constrained optimization problems. *arXiv preprint arXiv:2503.18821*, 2025b.

Zhaoyu Li, Jialiang Sun, Logan Murphy, Qidong Su, Zenan Li, Xian Zhang, Kaiyu Yang, and Xujie Si. A survey on deep learning for theorem proving. *arXiv preprint arXiv:2404.09939*, 2024b.

Jialin Lu, Kye Emond, Kaiyu Yang, Swarat Chaudhuri, Weiran Sun, and Wuyang Chen. Lean finder: Semantic search for mathlib that understands user intents. *arXiv preprint arXiv:2510.15940*, 2025.

The mathlib Community. The lean mathematical library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*. Association for Computing Machinery, 2020.

Norman Megill and David A Wheeler. *Metamath: a computer language for mathematical proofs*. Lulu. com, 2019.

Morph Labs. Moogle: Semantic search over mathlib4. `https://www.moogle.ai/`, 2023.

Leonardo de Moura and Sebastian Ullrich. The lean 4 theorem prover and programming language. In *International Conference on Automated Deduction*, pp. 625–635. Springer, 2021.

Aditya Paliwal, Sarah Loos, Markus Rabe, Kshitij Bansal, and Christian Szegedy. Graph representations for higher-order logic and theorem proving. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pp. 2967–2974, 2020.

Lawrence C Paulson. *Isabelle: A generic theorem prover*. Springer, 1994.

Michael Schlichtkrull, Thomas N Kipf, Peter Bloem, Rianne Van Den Berg, Ivan Titov, and Max Welling. Modeling relational data with graph convolutional networks. In *European semantic web conference*, pp. 593–607. Springer, 2018.

Daniel Selsam, Sebastian Ullrich, and Leonardo de Moura. Tabled typeclass resolution. *arXiv preprint arXiv:2001.04301*, 2020. doi: 10.48550/arXiv.2001.04301. URL `https://arxiv.org/abs/2001.04301`.

Yicheng Tao, Haotian Liu, Shanwen Wang, and Hongteng Xu. Assisting mathematical formalization with a learning-based premise retriever. *arXiv preprint arXiv:2501.13959*, 2025.

Mingzhe Wang, Yihe Tang, Jian Wang, and Jia Deng. Premise selection for theorem proving by deep graph embedding. *Advances in neural information processing systems*, 30, 2017.

Zichen Wang, Anjie Dong, and Zaiwen Wen. Tree-based premise selection for lean4. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025.

Kaiyu Yang, Aidan Swope, Alex Gu, Rahul Chalamala, Peiyang Song, Shixing Yu, Saad Godil, Ryan J Prenger, and Animashree Anandkumar. Leandojo: Theorem proving with retrieval-augmented language models. *Advances in Neural Information Processing Systems*, 36:21573–21612, 2023.

Kaiyu Yang, Gabriel Poesia, Jingxuan He, Wenda Li, Kristin Lauter, Swarat Chaudhuri, and Dawn Song. Formal mathematical reasoning: A new frontier in ai. *arXiv preprint arXiv:2412.16075*, 2024.

Thomas Zhu, Joshua Clune, Jeremy Avigad, Albert Qiaochu Jiang, and Sean Welleck. Premise selection for a lean hammer. *arXiv preprint arXiv:2506.07477*, 2025.

A Data and Split Details

A.1 Theorem library and database schema

Controlled runs use one fixed Mathlib4 v4.27.0-rc1 library of $N = 218{,}415$ kernel-checked theorems, identified by fully qualified names and serialized for retrieval.

Theorem entries are stored in a PostgreSQL table `mathlib_filtered`. Table 6 summarizes its schema. Each row corresponds to one theorem entry and contains both raw and preprocessed representations. In particular, `expr_json` stores the elaborated Lean kernel expression, while `expr_cse_json` stores the canonicalized expression after common subexpression elimination (CSE). Structural statistics are stored in dedicated columns: `node_count`, `tree_depth`, and `simp_node_count`. The `hetero_graph_json` column stores the materialized role-aware directed multi-relational graph used for offline theorem encoding.

Table 6: Schema of the `mathlib_filtered` table used for premise retrieval.

Column name	Data type	Description
`id`	`integer`	Unique identifier of the theorem entry.
`name`	`varchar`	Fully qualified Lean constant name.
`statement_str`	`text`	Pretty-printed statement (inspection/debugging only).
`expr_json`	`jsonb`	Serialized elaborated Lean expression.
`expr_cse_json`	`jsonb`	Canonicalized expression after CSE (offline).
`node_count`	`integer`	Node count after canonicalization and CSE.
`tree_depth`	`integer`	Max depth of the canonicalized expression tree.
`simp_node_count`	`integer`	Node count after lightweight syntactic simplifications.
`hetero_graph_json`	`jsonb`	Serializable role-aware directed multi-relational graph.

A.2 Offline preprocessing and cached artifacts

Theorem expressions are processed offline by the canonicalization–CSE–graph pipeline (Sections 3.1–3.2). Node statistics are stored in dedicated columns of `mathlib_filtered`, while the materialized graph is stored in `hetero_graph_json`. Offline theorem encoding loads the graph payload without repeating symbolic graph construction; online retrieval reuses the resulting embedding matrix.

Serializable heterograph format. Each entry in `hetero_graph_json` is a JSON object of the form

$$\texttt{hetero_graph_json} = \{\ \texttt{"x"} : [x_0, \ldots, x_{n-1}],\ \texttt{"edges"} : \{\texttt{rel} \mapsto [[s, t], \ldots]\}\ \},$$

where `x` is a list of integer node labels and `edges` is a dictionary mapping each relation name from a fixed, predefined relation inventory to a list of directed edges.

Concretely, the list `x` has length n and stores, for each node in the canonicalized expression graph, the integer index of its normalized node label in the shared vocabulary $\mathcal{V}$. Serialization maps out-of-vocabulary or malformed labels to a dedicated `<UNK>` index, so stored node indices are valid.

For each relation `rel`, the value `edges[rel]` is a list of pairs `[src, dst]`, where `src` and `dst` are integer indices into `x`. Edges are directed and typed: each relation corresponds to a constructor-field role in the Lean expression AST (e.g., `app_fn`, `app_arg`, `forall_body`), and every relation is paired with an explicit reverse relation to enable direction-sensitive message passing. If a relation is not present in the in-memory graph, it is still included in the JSON payload with an empty edge list, ensuring a fixed and predictable schema.

The following schematic canonicalized expression tree illustrates the storage format. After canonicalization and CSE, the serialized expression remains a rooted syntax tree whose internal nodes are constructors and whose leaves are constants, literals, or variables, including CSE placeholders. Each occurrence has its own node; repeated placeholders do not point to a shared subtree. CM operates on this tree, while the GNN graph adds typed forward and reverse edges between its node occurrences.

```
ForallE
  binderType → Const(...)
  body → App
    app_fn → Const(...)
    app_arg → App
        app_fn → Const(...)
        app_arg → Var
```

Applications use Lean's native left-associated binary constructor: a multi-argument application is represented as nested `App` nodes, with one `app_fn` and one `app_arg` field at each node. This view assigns each child its constructor-field role, such as `binderType`, `body`, `app_fn`, or `app_arg`. These roles are preserved explicitly in the graph representation.

Node list (x). After node-label normalization, every node in the canonicalized tree is assigned a discrete vocabulary index. The ordered node list $\mathtt{x} = [x_0, x_1, \ldots, x_{n-1}]$ stores these indices, where each x_i corresponds to the normalized label of the i-th node with a constructor type plus a normalized payload. Out-of-vocabulary labels are mapped to a dedicated `<UNK>` index.

Typed directed edges. Typed edges derived from constructor–field roles are stored explicitly in the `edges` dictionary. Each key corresponds to one relation in the fixed relation inventory used by the graph encoder, and the associated value is a list of directed source–target index pairs `[src, dst]`, where `src` and `dst` index into `x`. Forward relations encode child-to-parent links (e.g., `forall_binderType`, `forall_body`, `app_fn`, `app_arg`), and each forward relation is paired with an explicit reverse relation (e.g., `forall_body_rev`, `app_arg_rev`) to support direction-sensitive propagation. A schematic JSON payload stored in the database thus has the form:

```
{
  "x": [x0, x1, ..., x(n-1)],
  "edges": {
    "forall_binderType": [[s, t], ...],
    "forall_body": [[s, t], ...],
    "app_fn": [[s, t], ...],
    "app_arg": [[s, t], ...],
    "app_arg_rev": [[t, s], ...],
    "forall_body_rev": [[t, s], ...],
    ...
  }
}
```

All relations in the inventory appear as keys in the dictionary, even if the corresponding edge list is empty. This design enforces a fixed and predictable schema for graph serialization and simplifies reconstruction of role-aware heterographs at load time.

Robustness and minimal payloads. Serialization enforces the conditions required by the strict deserializer used at retrieval time. Empty node lists are replaced with a single `<UNK>` node, edges with out-of-range endpoints are removed, and malformed graphs fall back to a minimal payload containing one node and empty edge lists for all relations.

A.3 Training supervision

The training supervision is extracted from Lean4's Mathlib4 version v4.27.0-rc1, focusing on theorem proofs containing explicit theorem-linked tactic evidence. In particular, we mine supervision from `rw`-style tactic steps, where the proof state immediately before tactic execution can be paired with the theorem used by that step.

Each raw supervision unit is a source row from the tactic trace and an applied theorem. The row's parent declaration is recorded as `declName`, its source row identifier is the query record identifier, and its serialized goal/query expression is stored in `mainGoalTypeJson`. For exact-based records, the explicit-argument types extracted from the elaborated item are stored separately as evaluation metadata and are not appended to the query expression. The query-side expression is canonicalized

and passed to graph construction. The applied theorem name `appFnName` is the positive premise identifier; its expression is looked up separately in `mathlib_filtered.expr_cse_json`. Thus the query expression is a trace-derived query artifact, whereas the premise expression is a theorem-library artifact.

We exclude trivial `rfl`-style steps and malformed or unavailable theorem references. The remaining valid positive pairs are stored as the main supervision set `train_premise_pos`. We also construct an augmented set `train_perturb_pos` by applying lightweight syntactic perturbations to the query side while retaining the same premise label.

The final R-DGCN checkpoint and the capacity-matched role-sequence Transformer use 44,000 supervision pairs: 40,000 original pairs from `train_premise_pos` and 4,000 syntactically perturbed pairs from `train_perturb_pos`. Perturbation is enabled with a mixing cap $r = 0.10$; the selected perturbations constitute approximately 9.09% of this pool. The mixed pool is split into 39,600 training and 4,400 validation records, as detailed in Appendix D.3.

A.4 Evaluation datasets

We evaluate two Tree-based Search-derived Lean4 premise-selection test sets, Test-A and Test-B (Wang et al., 2025), together with Test-C, a test set constructed by our extraction pipeline. An evaluation unit is one retained query record, identified by the fully qualified name of its parent declaration (the query declaration name). Deduplication uses this query-side name. The applied premise identifier is the record's label and is not part of the deduplication key. Each retained evaluation record has one recorded premise.

Test-A contains 100 deduplicated query records and serves as a diagnostic dataset. Lean 4.27.0-rc1 elaborates 99 statements; one record has an unmatched closing delimiter and is retained as an explicit parse failure in the fixed denominator. **Test-B** contains $6{,}104$ queries and is used as the primary benchmark for quantitative evaluation. Both datasets are derived from Wang et al. (2025). We updated Test-B for Lean/Mathlib v4.27.0-rc1 by removing unavailable legacy theorems. Test-B retains the Tree-based Search extraction metadata: the instrumented `exact` step is elaborated and the inferred types of its explicit arguments are recorded as an evaluation-side field. These types are used only to check the recalled premise; they are not appended to the benchmark query or supplied to the encoder. The ranked query is formed from the target and local context captured before tactic execution, after reverting the locals.

Test-C is constructed by our own extraction pipeline from Lean tactic traces and contains a broader, more heterogeneous query distribution. In contrast to Test-B, which consists exclusively of theorem commands, Test-C focuses on rewrite-style proof steps (e.g., `rw`-style rewriting) and includes queries arising from multiple command categories.

Specifically, Test-C contains $12{,}494$ queries spanning `theorem`, `instance`, `abbrev`, `example`, and `structure` commands. It contains a wider range of expression sizes, binder structures, and typeclass-related artifacts than the theorem-only Test-B setting, with queries from multiple command categories and proof-state forms.

Table 7 reports the distribution of `commandSyntaxKind` in Test-C, highlighting the dominance of rewrite-related queries originating from `theorem` and `instance` commands, while also including less frequent but structurally distinct forms such as `abbrev` and `example`.

The reported Test-A, Test-B, and Test-C sizes count retained query records after deduplication by query declaration name.

Table 7: Distribution of `commandSyntaxKind` in Test-C (ours).

`commandSyntaxKind`	Count
`theorem`	6,247
`instance`	5,404
`abbrev`	806
`example`	29
`structure`	8

Failure accounting. Input and execution failures remain in the fixed denominator, including the single Test-A delimiter failure. External reference tools use their reported configurations.

A.5 Precomputed embedding matrix and consistency checks

`IndexBuilder` generates and loads the theorem-embedding sidecar. The sidecar aligns each theorem's identifier, qualified name, canonicalized+CSE expression, and CM tree with its 64-dimensional embedding. The complete 218,415-row float32 matrix occupies approximately 56 MB. The matrix rows and identifier array follow the same frozen theorem order. At initialization, rows are L_2-normalized and the matrix is loaded for exact dense cosine scoring. The checkpoint, preprocessing configuration, and theorem order identify the embedding-cache version used for evaluation.

A separate two-query, three-theorem consistency check compares direct online encoding with sidecar lookup. It preserves the complete order and tie-break, with maximum absolute errors of 5.96×10^{-8} for embeddings and cosine scores and 2.09×10^{-8} for the fused score. A direct three-row probe of the full sidecar against the evaluated checkpoint gives a maximum embedding deviation of 1.04×10^{-7}.

A.6 Latency measurement details

The R-DGCN latency benchmark uses the semantic retrieval protocol in Appendix C.2: cosine scores are computed against the precomputed theorem-embedding matrix, filtered to state-visible non-parent declarations, and sorted before the semantic top-2,000 are selected for CM reranking. R-DGCN-only returns the filtered cosine ranking without CM.

All methods in Table 2 are measured on the same NVIDIA A10 (23 GB) and Intel Xeon Platinum 8369B host (16 logical CPUs). For each method, timing starts with per-query processing and ends when the final ranked premise list is produced; one-time initialization is excluded. The R-DGCN dense-scoring benchmark covers all 6,104 Test-B queries in frozen order across three sequential shards, with zero failed rows. Each shard loads the sidecar once; offline embedding computation, sidecar loading, and initial device transfer are outside the per-query timings. Query processing, candidate construction, encoding or lookup, cosine scoring, CM reranking where applicable, and final sorting are included in the online timings. Under this configuration, R-DGCN-only has a median latency of 0.019 seconds, a P95 of 0.023 seconds, and a mean of 0.020 seconds. The complete system has a median of 0.39 seconds, a P95 of 2.80 seconds, and a mean of 0.792945 seconds, including CM reranking.

A.7 LeanPremise target eligibility

For the LeanPremise rows in Table 1, Test-B and Test-C omit targets listed in LeanHammer's official blacklist, and a semantically consistent theorem name is counted as a hit. These rows follow LeanPremise's native target eligibility and hit criterion and are tool-level references with a different eligible query population.

A.8 Metric definitions

Encoder comparisons report Recall@k. Each query q is an evaluation unit as defined in Appendix A.4. Let $\mathcal{D}_q^{\mathrm{vis}}$ be the theorem declarations visible in the proof state after excluding the parent declaration. The ranked list $\mathcal{L}_q$ is obtained by sorting the cosine or fused scores restricted to $\mathcal{D}_q^{\mathrm{vis}}$; visibility filtering therefore precedes every reported rank. Let r_q^{pos} be the rank of the recorded theorem-use label in that list, or infinity when absent. Failed inputs or runs contribute zero under the fixed denominator. Define $h_{q,k}^{\mathrm{pos}} = \mathbf{1}[r_q^{\mathrm{pos}} \leq k]$.

For N queries, metrics are macro-averaged; table entries are these averages multiplied by 100:

$$\text{Recall@}k = \frac{1}{N}\sum_{q=1}^{N} h_{q,k}^{\text{pos}}, \qquad \text{Precision@}k = \frac{1}{N}\sum_{q=1}^{N} \frac{h_{q,k}^{\text{pos}}}{k},$$

$$\text{F1@}k = \frac{1}{N}\sum_{q=1}^{N} \frac{2p_{q,k}\,\text{rec}_{q,k}}{p_{q,k}+\text{rec}_{q,k}}, \qquad \text{nDCG@}k = \frac{1}{N}\sum_{q:1\le r_q^{\text{pos}}\le k} \frac{1}{\log_2(r_q^{\text{pos}}+1)},$$

$$\text{MRR} = \frac{1}{N}\sum_{q:r_q^{\text{pos}}<\infty} \frac{1}{r_q^{\text{pos}}},$$

where $p_{q,k} = h_{q,k}^{\text{pos}}/k$ and $\text{rec}_{q,k} = h_{q,k}^{\text{pos}}$ are the per-query precision and recall, respectively; the F1 term is defined as zero when both are zero. With one recorded premise per query, Precision@k and F1@k are deterministic functions of Recall@k; they are included for compatibility with the tool-level table. The ideal DCG is one.

B Canonicalization Details

This appendix provides implementation-level details of the canonicalization pipeline summarized in Section 3.1. The pipeline constructs the syntactic representation used by the graph encoder and CM. Source expressions remain available separately for Lean checking and evaluation.

B.1 Canonicalization pipeline and fixed parameters

Canonicalization is implemented as a fixed, recursive rewrite sequence that is applied uniformly to all expressions appearing in training supervision, evaluation queries, and the theorem library. Given an input expression e, the pipeline produces a canonicalized expression $\tilde{e}$ via

$$\tilde{e} = \mathcal{N}_{\text{arith}}\big(\mathcal{N}_{\text{wrap}}(\mathcal{N}_{\text{bind}}(e))\big),$$

where binder simplification, typeclass wrapper stripping, and arithmetic normalization are applied in this order. After canonicalization, a standard common subexpression elimination (CSE) pass is applied before downstream processing.

Canonicalization parameters are set independently of training and evaluation data; the relation schema is illustrated in Figure 3. The raw kernel expression is retained alongside the transformed representation.

Binder rule and index handling. After recursively processing the type A and body b of `ForallE`, the guard accepts exactly the outer constructors $\{\texttt{BVar}, \texttt{FVar}, \texttt{MVar}, \texttt{Sort}, \texttt{Const}\}$ for A. The binder is removed when its variable does not occur in b. The occurrence test starts at depth $d = 0$ and detects $\texttt{BVar}(d)$; it increments d only when entering the body of `Lam`, `ForallE`, or `LetE`. Binder types and let values are visited at the current depth. In the experimental preprocessing, deleting an unused binder applies the standard capture-avoiding de Bruijn shift: while traversing the body, an index j is unchanged when it refers to a binder inside the current traversal cutoff, and is replaced by $j-1$ when it refers to an outer binder. The occurrence test and this shift are applied recursively through applications, binders, let-expressions, metadata, and projections. The pass repeats to a fixed point. For example, $\forall x : \mathbb{N},\ \forall y : \mathbb{N},\ x = x$ becomes $\forall x : \mathbb{N},\ x = x$, because the inner $\texttt{BVar}(1)$ references to x are shifted to $\texttt{BVar}(0)$ when the unused y binder is removed.

CSE preprocessing. After the fixed rewrite sequence and binder-name conversion, CSE identifies repeated non-constant subexpressions by structural keys and replaces them with variable placeholder leaves. Repeated occurrences are stored as separate tree nodes, with no shared-subtree pointers. Graph construction and CM both consume this compressed tree; no DAG-to-tree expansion is required. Constant nodes are never replaced. For example, f `Nat Nat` remains two `Const(Nat)` leaves under the two argument fields. The same pass is used for queries and theorem statements, while the raw `expr_json` is stored separately.

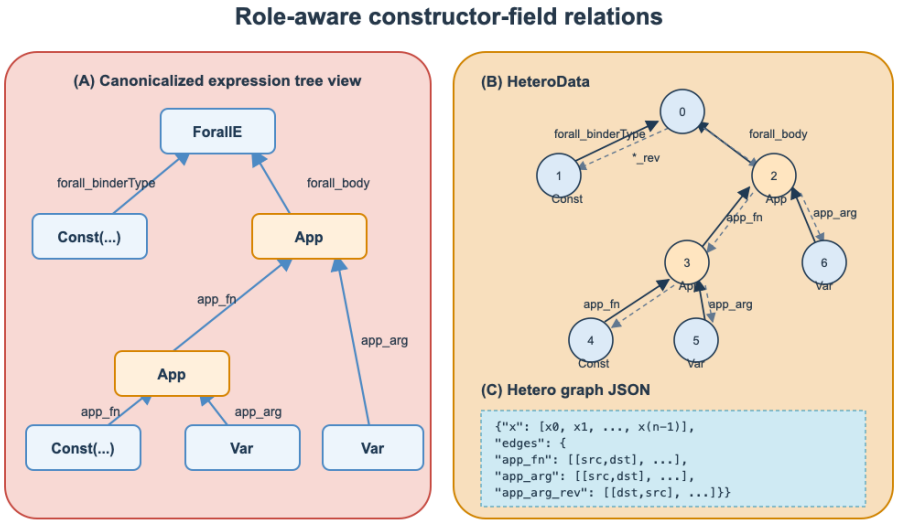

Figure 3: Role-aware directed multi-relational graph construction. The two-argument application uses two nested binary `App` nodes, each with one `app_fn` and one `app_arg` field. Each constructor field induces a distinct typed edge (child→parent), and each forward relation is paired with an explicit reverse relation to enable direction-sensitive propagation without collapsing syntactic roles.

Table 8: Constructor-field relations used in our role-aware heterograph.

Constructor	Forward relations (child→parent)
`App`	`app_fn`, `app_arg`
`Lam`	`lam_binderType`, `lam_body`
`ForallE`	`forall_binderType`, `forall_body`
`LetE`	`let_type`, `let_value`, `let_body`
`MData`	`mdata_expr`
`Proj`	`proj_struct`

Interaction with role-aware graph construction. After canonicalization and CSE, expressions use the fixed constructor-field inventory in Table 8.

The table is the complete relation inventory: canonicalization may change graph shape, but the inventory and its forward/reverse pairing remain fixed.

B.2 Typeclass wrapper stripping

Lean elaboration frequently introduces explicit typeclass evidence arguments, which inflate expression trees with instance subterms and obscure the operator–operand skeleton used by retrieval. Wrapper stripping removes selected instance-evidence subtrees from the retrieval representation using fixed syntactic patterns.

Trigger. The reported implementation uses four explicit Nat-level patterns: `LT.lt` with `instLTNat`, `LE.le` with `instLENat`, `HAdd.hAdd` with `instAddNat`, and `Dvd.dvd` with `instDvd`; the separate Nat `OfNat.ofNat` pattern strips a literal wrapper. A rewrite is triggered only when the type arguments and arity match and a recursive instance-subtree search finds the listed suffix. Unsupported or ambiguous applications are left unchanged.

Rewrite rule. When the trigger matches, we drop the explicit instance-evidence subtree and rewrite the application into a canonical Nat operator form. Table 9 lists representative Nat-level rewrites for retrieval.

Table 9: Examples of Nat-level wrapper stripping for retrieval.

Elaborated form	Canonical form
`LT.lt Nat Nat.instLTNat x y`	`Nat.lt x y`
`LE.le Nat Nat.instLENat x y`	`Nat.le x y`
`HAdd.hAdd Nat Nat Nat Nat.instAddNat x y`	`Nat.add x y`
`OfNat.ofNat Nat (Lit(natVal k)) Nat.instOfNat`	`Lit(natVal k)`

This procedure covers Nat-level order, arithmetic, divisibility, and literal coercions; other wrappers are left unchanged.

B.3 Arithmetic normalization for small literals

Arithmetic expressions involving small constant offsets often admit multiple syntactic encodings in Lean, leading to fragmented tree structures for otherwise similar goals. To reduce this variance, we normalize a restricted class of small-literal arithmetic patterns.

For natural numbers, additions by small constants ($1 \leq k \leq K_{\mathrm{lit}}$) are rewritten into a uniform successor-chain representation, with the fixed implementation value $K_{\mathrm{lit}} = 10$. Larger constants and non-matching arithmetic forms are left unchanged.

By restricting normalization to small offsets, this step captures common local shifts that frequently appear in rewrite-heavy proof states, while avoiding uncontrolled expansion of large numerals or distortion of overall expression shape.

C Online Theorem Retrieval and Fusion Implementation

This section records implementation details of the online retrieval pipeline in Section 4.2: a frozen embedding matrix supplies cosine scores, and $\mathcal{C}^{K}_{\mathrm{emb}}(e_q)$ denotes the semantic shortlist used for CM reranking.

C.1 Inputs, caching, and version consistency

Theorem graphs are encoded offline, while the query graph is built from the current proof-state goal and local context. The embedding matrix, theorem identifiers, and CM trees are aligned by frozen theorem order and use the same preprocessing configuration and checkpoint as the query encoder. The online query path does not use a recorded `exact` application or append its inferred explicit-argument types. For every proof-state query, the candidate pool is first restricted to declarations visible in that state and excludes the parent declaration. All Recall rankings and top-K shortlist construction use this filtered pool.

C.2 Embedding scoring and semantic shortlist construction

At initialization, the complete theorem-embedding matrix is loaded and row-normalized. Each online query is canonicalized, converted to a role graph, encoded, and normalized. Eq. (2) evaluates its cosine similarity against all theorem rows; the resulting rows are then masked to the query's visible, non-parent candidate pool before sorting. This dense scoring plus state-visible filtering specifies the R-DGCN encoder path; reference-representation settings are given in Appendix D.8. For a batch of B queries, the same computation is $\mathbf{R} = \hat{\mathbf{Q}}\hat{\mathbf{Z}}_{\mathcal{D}}^{\top} \in \mathbb{R}^{B\times|\mathcal{D}|}$; the encoder evaluation uses $B = 64$. Theorem embeddings are reused across queries.

The filtered scores are ordered in descending order, with frozen matrix row order as the tie-break. The encoder-only mode returns the first k entries for Recall evaluation. The fused mode selects the first $K = 2000$ entries from the same filtered ranking and looks up their aligned CM trees. Ties at the top-K boundary are resolved by the same row order, so candidate membership and final ranking are deterministic.

C.3 CM RERANKING AND FUSION

For each entry of $\mathcal{C}^K_{\mathrm{emb}}(e_q)$, we reuse its cosine score and map it to $S_{\mathrm{emb}} = (S_{\text{R-DGCN}} + 1)/2$. CM compares the query tree with the cached theorem tree. The weighted score $0.7S_{\mathrm{emb}} + 0.2S_{\mathrm{CM}}$ determines the final order, with frozen theorem order breaking ties. Benchmark runs use $\beta = 0$ and omit identifier-overlap scoring. The optional identifier-assisted human-search mode adds $0.1S_{\mathrm{Jac}}$ on the same semantic shortlist when a user explicitly supplies an identifier. Only query encoding, dense scoring, top-K selection, CM, and final ordering are performed per query.

The CM reranking control in Table 4 uses the same state-visible embedding shortlist as the complete system. After the filtered cosine ranking selects the first 2,000 entries, CM alone scores and reranks that fixed shortlist; the cosine scores are not included in the CM reranking order. Thus the CM reranking and fused rows share candidate membership, so their difference isolates the effect of combining embedding and CM scores on the same shortlist.

C.4 DIRECTED CM: IMPLEMENTATION-LEVEL DEFINITION

This subsection specifies the directed CM feature used as the structural term in our reranking score (Eq. (3)). CM is a bounded, role-sensitive feature: it allows *query-side* collapse to tolerate elaboration-induced surface variation, and it rewards theorem entries whose canonicalized operator/binder skeleton is covered by the query.

Directional coverage and normalization. We compute a directional coverage score $\mathrm{CM}(T_q \Rightarrow T_c) \in [0, 1]$ that measures how much of the theorem tree T_c can be covered by the query tree T_q under query-side collapse. The score is normalized by the theorem's total weight:

$$S_{\mathrm{CM}}(T_q, T_c) = \frac{\mathrm{raw}(T_q \Rightarrow T_c)}{W(T_c)}, \qquad W(T_c) = \sum_{v \in T_c} w(\ell(v)).$$

Node weights and leaf matching. We use a fixed label-dependent weight function $w(\cdot)$: $w(\texttt{App}) = 0.35$, $w(\texttt{Lam}) = w(\texttt{ForallE}) = w(\texttt{LetE}) = 0.85$, $w(\texttt{Const:}\star) = 1.25$, $w(\texttt{Lit:}\star) = 1.10$, $w(\texttt{BVar:}\star) = w(\texttt{FVar}) = w(\texttt{MVar}) = 0.70$, and default weight 1. Let $\mathcal{L}_c$ be the set of leaves in T_c, and let $\mathcal{V}_c \subseteq \mathcal{L}_c$ contain the variable leaves (`BVar`, `FVar`, and `MVar`) and leaves labeled `Const:inst`, which represent typeclass evidence. A leaf in $\mathcal{V}_c$ contributes its weight for any nonempty query subtree. A leaf in $\mathcal{L}_c \setminus \mathcal{V}_c$ contributes its weight if its label occurs in the query subtree, and zero otherwise.

Complete recursion, leaves, and ordered child coverage. Let $\mathrm{Match}(u, v)$ be the best matched weight for covering the theorem subtree rooted at $v \in T_c$ using the query subtree rooted at $u \in T_q$. Let $\mathrm{Labels}(u)$ be the set of labels occurring in the query subtree rooted at u. If either subtree is absent, $\mathrm{Match}(u, v) = 0$. For nonempty subtrees,

$$\mathrm{Match}(u, v) = \begin{cases} w(\ell(v)), & v \in \mathcal{V}_c, \\ w(\ell(v))\,\mathbf{1}[\ell(v) \in \mathrm{Labels}(u)], & v \in \mathcal{L}_c \setminus \mathcal{V}_c, \\ \max\left\{ \max\limits_{u' \in \mathrm{child}(u)} \mathrm{Match}(u', v), A(u, v) \right\}, & v \in T_c \setminus \mathcal{L}_c, \end{cases}$$

where $A(u, v) = 0$ when $\mathrm{sim}(\ell(u), \ell(v)) = 0$, and otherwise

$$A(u, v) = \mathrm{sim}(\ell(u), \ell(v))\, w(\ell(v)) + C(u, v).$$

For non-leaf nodes, $\mathrm{sim} = 1$ for exact constructor labels and 0 otherwise. For leaves in $\mathcal{L}_c \setminus \mathcal{V}_c$, label comparison is exact; leaves in $\mathcal{V}_c$ require no label comparison.

For $\mathrm{child}(u) = (u_1, \ldots, u_m)$ and $\mathrm{child}(v) = (v_1, \ldots, v_k)$, $C(u, v) = 0$ when either child list is empty. Otherwise it is the ordered-subsequence dynamic program $C(u, v) = \max(0, D_{m,k})$, with $D_{0,0} = 0$, $D_{a,0} = 0$, $D_{a,b} = -\infty$ for $b > a$, and, for $a \geq 1$ and $b \geq 1$,

$$D_{a,b} = \max(D_{a-1,b},\ D_{a-1,b-1} + \mathrm{Match}(u_a, v_b)).$$

The first term skips a query child; the second aligns it to the next theorem child, so every theorem child must be covered in order.

If either input tree is absent the score is zero; otherwise normalization uses the theorem's total node-occurrence weight. Per-node label sets and weights and pairwise matches are memoized.

C.5 Fusion computation schedule

Embedding scores first determine $\mathcal{C}^K_{\mathrm{emb}}(e_q)$. The embedding scores of those entries are then combined with their CM scores. Encoder-only retrieval directly returns the cosine top-k without invoking CM. The CM reranking control reuses $\mathcal{C}^K_{\mathrm{emb}}(e_q)$ and invokes CM without an embedding score.

D Model and Training Implementation Notes

This section records implementation-level choices that affect reproducibility of the graph encoder and the contrastive training procedure, including graph materialization conventions, vocabulary construction, supervision organization, batching, negatives, and optimization. High-level motivations are described in the main text; here we specify fixed procedures and numerical settings.

D.1 Relation inventory and graph materialization

Relation coverage and directionality. The constructor-field inventory and its child-to-parent orientation are given in Table 8; every forward relation has a mirrored reverse relation, yielding $\mathcal{R}^{\pm}$.

Database graph payload. Each theorem entry stores a fully materialized graph payload (Appendix A.2); loading reconstructs its typed adjacency lists from the fixed inventory.

D.2 Node label normalization and vocabulary control

Node labels are standardized by a lightweight normalization that stabilizes symbol payloads while leaving expression structure unchanged. This normalization is applied during graph construction and during auxiliary structural computations (e.g., WL-guided negatives), so learned and symbolic components operate on aligned labels.

The node-label vocabulary $\mathcal{V}$ is constructed by scanning the entire theorem corpus and collecting all labels appearing in canonicalized and compressed expressions. We reserve a dedicated unknown token `<UNK>` and map unseen labels at inference time to `<UNK>`, stabilizing embeddings under minor library growth. The vocabulary is cached after construction and reused across training runs unless the theorem corpus is regenerated.

D.3 Training supervision and record-level splits

Training supervision combines original Mathlib tactic-trace records in `train_premise_pos` with syntactically perturbed records in `train_perturb_pos`. Each original record pairs the pre-tactic goal expression with the premise used in the corresponding proof step. Perturbations rewrite the query expression while retaining its premise label, for example by injecting neutral elements such as $x + 0$ or $x \cdot 1$. The final R-DGCN and matched Transformer supervision pool contains 40,000 original pairs and 4,000 perturbed pairs. The loader uses $r = 0.10$ to cap perturbations by $|\mathcal{D}_{\mathrm{pert}}| \le \frac{r}{1-r}|\mathcal{D}_{\mathrm{main}}|$; the selected 4,000 pairs occupy $4{,}000/44{,}000 \approx 9.09\%$ of the pool.

Query expressions are materialized from the source trace; premise expressions are read from `mathlib_filtered.expr_cse_json`. The source row identifier, parent-declaration name, and premise identifier are distinct fields in the extraction records. The theorem library is shared across splits.

Train/validation split. After perturbation subsampling, we combine the selected records from `train_premise_pos` and `train_perturb_pos`, retaining multiple premise observations and perturbed queries associated with the same query declaration. The training loader shuffles the combined record list using the configured random seed. For N records, the first $\lfloor 0.9N \rfloor$ form the training set and the remainder form the validation set, giving 39,600 training and 4,400 validation records. Validation is drawn from the same mixed pool, with no restriction to original records. Split

assignment operates on individual records, so records with the same query declaration name can occur in both partitions. Gradient updates use the training partition.

D.4 Batching by node-count buckets

To stabilize training cost and make negative construction size-compatible, we batch examples using node-count buckets derived from cached $n(e)$ after preprocessing. Each query is assigned to a bucket index $\lfloor n(e)/w \rfloor$ with fixed width $w = 25$, and each mini-batch is sampled entirely within one bucket. This reduces variance in message-passing cost and prevents mixing extremely small and extremely large graphs. The same bucket index is also used to sample additional same-bucket premises to enlarge the negative pool, so that negatives remain structurally comparable rather than trivially unrelated.

D.5 Encoder architecture: R-DGCN with dual pooling

Directed per-relation message passing. The encoder stacks L layers of a directed multi-relational GCN (R-DGCN) operating on the single node type `expr` with multiple edge relations. Each relation uses an independent linear transform and performs message passing along the directed edges (source→target), normalized by the *target in-degree* (a left normalization). We disable relation-level self-loops in these per-relation convolutions and instead inject self-information through a separate root residual pathway.

Root residual, normalization, and nonlinearity. Let $\mathbf{X}^{(\ell)}$ be the node features at layer ℓ. After aggregating relation-specific messages (with a fixed inter-relation aggregator, *sum* in our implementation), we add a learnable root residual projection of $\mathbf{X}^{(\ell)}$ to the aggregated result. Each layer then applies BatchNorm, ReLU, and dropout with a fixed rate. This design provides a separate "self" channel when a node has sparse incoming edges for a relation.

Graph-level embedding and similarity interface. Node features are initialized by an embedding lookup on discrete node-label ids. After L message-passing layers, we compute a graph embedding by concatenating mean pooling and an attention pooling (as defined in the main text), followed by a two-layer projection MLP.

Dual pooling (mean + attention). We reuse the notation from Section 4.1: $\mathbf{z}_{\mathrm{mean}}$ captures the global shape of the canonicalized role-aware graph, while $\mathbf{z}_{\mathrm{att}}$ weights selected nodes (e.g., constants, predicates, and literals) via an MLP-gated GlobalAttention mechanism. Concretely, the attention gate assigns each node v a scalar score s_v (computed from $\mathbf{h}_v^{(L)}$) and normalizes it into attention weights a_v via a softmax; this yields an attention-weighted summary $\mathbf{z}_{\mathrm{att}}$. The exact pooling definitions are given in the main text.

Projection and normalization. The concatenated vector $[\mathbf{z}_{\mathrm{mean}}; \mathbf{z}_{\mathrm{att}}]$ is mapped to the final embedding space by a lightweight two-layer projection MLP (with a ReLU nonlinearity), and the output is L_2-normalized to obtain $\hat{\mathbf{z}}$. This normalization makes cosine similarity equivalent to a dot product, which simplifies both training (InfoNCE) and online scoring.

D.6 Training objective and WL-guided structured negatives

The encoder uses the temperature-scaled InfoNCE objective given in the main text with $B = 64$ and $\tau = 0.07$. Query rows are grouped into node-count buckets of width 25. A training batch is drawn from one bucket; an incomplete bucket batch is dropped. The candidate columns contain the B in-batch rows and up to $K_{\mathrm{extra}} = 10$ additional premise rows sampled without replacement from the same bucket. If fewer than ten eligible rows exist, all are used; an empty pool contributes no extra columns.

For query i, the negative pool removes every candidate whose fully qualified premise identifier belongs to the known positive set P_i. Candidate columns are deduplicated by premise identifier before WL classification, so repeated rows for the same theorem do not create extra negative copies. The remaining columns are classified by WL similarity into easy $[0, 0.30)$, hard $[0.30, 0.70)$, and other. We sample up to 32 negatives without replacement, targeting a 1:1 hard/easy split (up to 16

from each group), then fill from the remaining groups in the order easy, other, and any remaining non-positive columns. If a group is too small, all of its available columns are used and no row is duplicated to reach the target count. If $|P_i| > 1$, all candidates in P_i are positive columns and the InfoNCE numerator is the log-sum-exp over that positive set; only the remaining premise identifiers can contribute to the denominator as negatives.

D.7 Optimization, early stopping, and training dynamics

Optimization uses AdamW with a fixed initial learning rate and weight decay. Gradients are clipped to a maximum ℓ_2 norm of 1.0. A plateau-based scheduler monitors validation loss and reduces the learning rate when progress stalls. Early stopping is enabled based on validation loss with a minimum number of epochs before activation and a fixed patience window. The checkpoint achieving the lowest validation loss is selected and reloaded for final evaluation. Figure 4 visualizes per-epoch training and validation losses for a representative run under the same structured-negative InfoNCE setup. Validation loss drives the scheduler, early stopping, and checkpoint selection.

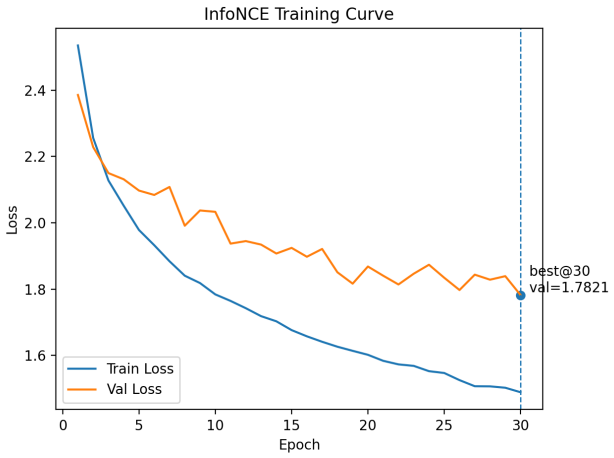

Figure 4: Training and validation loss curves for the temperature-scaled InfoNCE objective with structured negatives.

D.8 Reference encoder settings

Structural encoder controls. The directed untyped and role-only undirected controls use the same supervision, batching, optimizer, training schedule, negative-sampling policy, and evaluation protocol as R-DGCN. Let E_r and $E_{r^{\text{rev}}}$ denote the forward and reverse edges for constructor-field role r. The directed untyped GCN replaces the 22 typed relations with $E_{\uparrow} = \bigcup_r E_r$ and $E_{\downarrow} = \bigcup_r E_{r^{\text{rev}}}$, deduplicating ordered node pairs within each channel. Child-to-parent and parent-to-child messages retain separate transformations; constructor-field labels are merged within each direction.

The role-only undirected control retains the eleven constructor-field roles and bidirectional adjacency, but merges forward and reverse edges of each role into one symmetric channel. It removes the direction-specific relation distinction while preserving role identity.

Relation-agnostic tree encoder. The relation-agnostic tree encoder uses the same canonicalized expressions and tree nodes but omits constructor-field relation types. Each node label is mapped to a learned 128-dimensional vector; child vectors are averaged, with a zero vector for a leaf, then concatenated with the current label vector and passed through a ReLU linear layer and a 64-dimensional output projection. Query and theorem vectors are ranked by cosine similarity. The controlled run uses fixed seed zero and no CM reranking.

Structural fingerprint. The fingerprint rescorer traverses the same preprocessed and CSE-compressed expression tree. Its feature map contains counts of normalized node labels, parent–child label bigrams, a WL histogram with height $h = 1$, and log-scaled node-count and tree-depth features.

For each query q, $\mathcal{C}_q^{\mathrm{fp}} = \mathcal{C}_{\mathrm{emb}}^K(e_q)$, with $K = 2000$, is the fixed R-DGCN top-2,000 shortlist used by both fingerprint rows in Table 3. The vectorizer applies log-TF and computes shortlist-local weights

$$\mathrm{idf}_q(f) = \log \frac{|\mathcal{C}_q^{\mathrm{fp}}| + 1}{\mathrm{df}_q(f) + 1} + 1, \qquad \mathrm{df}_q(f) = \big|\{c \in \mathcal{C}_q^{\mathrm{fp}} : f \in \mathrm{supp}(c)\}\big|.$$

Here $\mathrm{supp}(c)$ is the set of feature keys in candidate c's fingerprint. The query contributes feature keys to the vocabulary but is excluded from the document count and document-frequency statistics. The same IDF weights are applied to the query and candidate vectors, followed by L_2 normalization and cosine scoring within $\mathcal{C}_q^{\mathrm{fp}}$. The statistics are recomputed for each query's shortlist, rather than over the full theorem library. The representation has no learned parameters; the fingerprint + CM row combines its cosine score with CM on the frozen shortlist.

Frozen sentence encoder. The frozen sentence baseline uses the pinned `sentence-transformers/all-MiniLM-L6-v2` checkpoint without fine-tuning. Each Lean statement is encoded as text, the output embedding is L_2-normalized, and theorem entries are ranked by cosine similarity. Theorem embeddings are precomputed once per evaluation run and query embeddings are computed with the same checkpoint.

Capacity-matched role-sequence Transformer. The Transformer baseline serializes the canonicalized expression into a deterministic depth-first sequence of constructor-role tokens. Its vocabulary has 1,576 entries and sequences are truncated or padded to length 1,024. The encoder has three Transformer layers with model width 96, four attention heads, feed-forward width 224, dropout 0.30, sinusoidal positional encoding, and a 64-dimensional output projection followed by L_2 normalization. The model has 400,576 trainable parameters, compared with 399,809 for R-DGCN. It is trained with the same 44,000-pair supervision split (39,600/4,400), seed 42, batch size 64, temperature 0.07, 32 structured negatives per query, AdamW with learning rate 5×10^{-4} and weight decay 5×10^{-4}, for at most 30 epochs with six minimum epochs and patience five. Transformer-only and Transformer+CM scores use the same fixed name-free scoring weights; CM is an inference-time reranking term and does not change Transformer training.

D.9 Model and training configuration

Table 10 summarizes the key hyperparameters required to reproduce training and the fixed online retrieval knobs used in evaluation.

Table 10: Final model, training, and online retrieval configuration.

Parameter	Value	Parameter	Value
Encoder depth	3 layers	Batch size	64
Hidden dimension	64	Node-count bucket width	25
Dropout	0.30	Maximum epochs	30
Training objective	InfoNCE (structured)	Temperature	0.07
Negatives per query	32	Hard/easy ratio	1:1
WL depth H	3	WL bins	$[0, 0.30)$ / $[0.30, 0.70)$
Extra pool K_{extra}	10	Pool size M	$M = B + K_{\mathrm{extra}}$
Optimizer	AdamW	Learning rate	5×10^{-4}
Weight decay	5×10^{-4}	Early stopping	patience 5, min 6 epochs
Online retrieval (fixed)			
Library size $\lvert\mathcal{D}\rvert$	218,415	CM shortlist size K	2000
Embedding dimension	64	Fusion evaluation set	$\mathcal{C}_{\mathrm{emb}}^K(e_q)$

Inference-time note. The theorem matrix is loaded once and reused across queries. Online execution encodes the query and scores the complete matrix; the fused mode additionally selects its semantic top-K and computes CM.

D.10 RANDOM-SEED SENSITIVITY ANALYSIS

We evaluated five seeded checkpoints (42, 43, 44, 100, and 1000) using the same 44,000-pair supervision split, 1,562-entry node vocabulary, 30-epoch budget, batch size 64, and learning rate 5×10^{-4}. Each checkpoint was evaluated on the identical 6,104-query Test-B manifest with cached theorem graphs; all runs completed with 6,104 result rows. The seed-42 checkpoint is the primary checkpoint used in the component tables. The CM reranking control in Table 4 is paired with this primary checkpoint; the seed tables report the matched encoder-only and fused runs.

Table 11: R-DGCN-only seed sensitivity on Test-B (%).

Seed	R@1	R@5	R@10	MRR
42	20.95	28.05	47.58	26.92
43	20.13	27.33	48.02	26.30
44	19.87	26.44	46.49	25.58
100	18.23	25.20	45.49	24.25
1000	18.66	25.84	46.30	24.85
Mean	19.57	26.57	46.78	25.58
Std.	1.11	1.14	1.02	1.07

Table 12: R-DGCN system seed sensitivity on Test-B (%).

Seed	R@1	R@5	R@10	MRR
42	29.51	41.84	64.40	37.91
43	23.26	34.53	58.39	31.54
44	24.13	34.01	55.41	31.61
100	31.45	41.50	63.06	39.05
1000	32.62	42.09	62.37	39.70
Mean	28.19	38.79	60.73	35.96
Std.	4.27	4.14	3.72	4.06