
Tree-Based Premise Selection for Lean4 (Appendix)

1	Contents	
2	A Data Extraction and Theorem Library Construction	2
3	B WL Kernel Encoding Process	3
4	C Clustering Process	3
5	D CM Similarity and Jaccard Similarity of Const nodes	5
6	E The construction of test sets	6
7	E.1 Test set A	6
8	E.2 Test set B	6
9	E.2.1 Overview of Lean Tactics	6
10	E.2.2 Extraction and Transformation of exact Invocations	7
11	E.2.3 Implementation Details of the Data Transformation and Extraction Pipeline .	8
12	F Detailed Visualization of Search Results	9
13	G Limitations	12
14	H Broader Impacts	13
15	I Extended Performance Evaluation	14
16	I.1 Analysis of Efficiency and Performance Trade-off	14
17	I.2 Domain-Specific Performance Comparison	14

18 A Data Extraction and Theorem Library Construction

19 The theorem library used in this study is sourced from Lean4’s Mathlib4 version v4.18.0-rc1. An
 20 initial extraction yields 339,746 theorems. To construct a high-quality, non-redundant theorem
 21 library suitable for the premise selection task, we apply a series of filtering rules to remove theorems
 22 that do not meet the requirements, such as internal implementation details, auxiliary lemmas, and
 23 theorems from specific namespaces. These filtering rules include excluding theorems from `Lean`, `Std`,
 24 `Mathlib.Tactic` namespaces or those whose names contain patterns like `.proof_`, `._auxLemma.`,
 25 `.Lemmas.`, or `_private.`. After this filtering process, the final library comprises 217,555 valid
 26 mathematical theorems. Each theorem in the library is parsed and converted into an expression tree
 27 structure, and stored in a database with key fields including `id` (unique identifier), `name` (theorem
 28 name), `statement_str` (formalized statement at Lean4’s underlying level), `expr_json` (raw ex-
 29 pression tree JSON), `expr_cse_json` (CSE-processed tree JSON), and `node_count` (number of
 30 nodes). To assess data quality and understand the characteristics of the theorem library, we conduct
 31 statistical analysis on the final set of theorems, focusing on the node count and depth distributions
 32 of the theorem expression trees. The node count distribution histogram is shown in Figure 1, and
 33 the depth distribution histogram is shown in Figure 2. Statistics on the distribution and structural
 characteristics of the theorem library across different mathematical domains are detailed in Table 1.

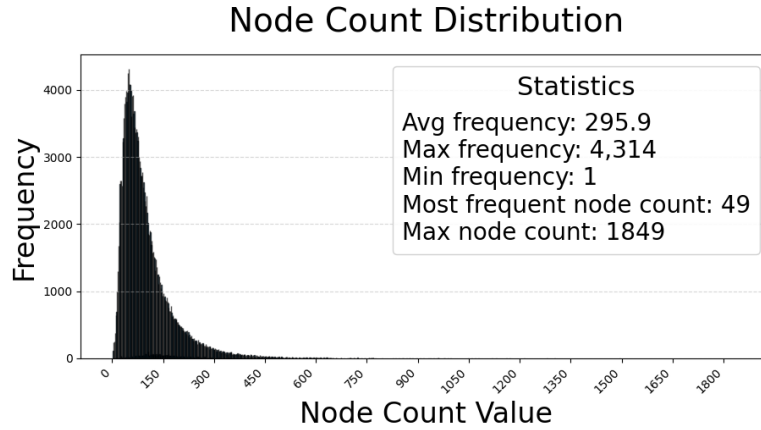


Figure 1: Node Count Distribution Histogram

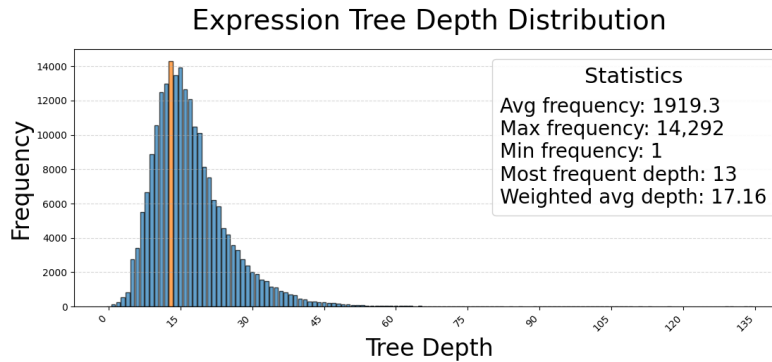


Figure 2: Depth Distribution Histogram

Table 1: Mathematical Domain Analysis (partial) in mathlib

Domain	Count	Node Count			Depth			Features
		Max	Min	Median	Max	Min	Median	
Algebra	11718	1325	5	95.0	130	2	17.0	Polynomial/Ring
Analysis	8689	1147	5	125	78	2	20	Convex/InnerProductSpace
Geometry	1595	973	11	165	95	4	25	Euclidean/Manifold
Topology	5720	819	5	75.0	72	2	16.0	Bornology/UniformSpace
NumberTheory	2551	986	3	85	59	1	15	ClassNumber/Harmonic
CategoryTheory	5570	1077	5	113.0	84	2	20.0	Adjunction/Functors
Logic	535	707	3	51	35	1	13	Embedding/Denumerable

B WL Kernel Encoding Process

This section describes the WL kernel method used for encoding expression trees in this paper. The WL kernel is an effective technique for extracting graph structural features, generating compact vector representations that capture the tree’s structural properties by iteratively aggregating neighborhood information of nodes. The encoding process can be broken down into the following steps:

- 1. Label Initialization** (`initialize_labels`): Initial labels are assigned to each node in the expression tree. The initial label is based on the node’s original type; for specific types like `BVar`, `FVar`, `MVar`, `Sort`, `Const`, the label is simplified to the type prefix. Furthermore, each node’s label is combined with its depth information in the tree.
- 2. Iterative Label Refinement** (`wl_iteration`): h iterations are performed. In each iteration, a node’s label is updated based on its label from the previous round and the labels of all its children from the previous round via a hashing function. Specifically, the children’s previous labels are collected, sorted lexicographically, concatenated with the current node’s previous label to form a string, and finally, the MD5 hash of this string is computed as the node’s new label for the current round.
- 3. Generate Label Histogram** (`get_label_histogram`): After each iteration, the distribution of all nodes’ new labels for the current round is counted to generate a label histogram.
- 4. Combine Histograms** (`compute_wl_encoding`): The final tree encoding is generated by combining the label histograms from all h iterations. To distinguish identical labels originating from different iteration rounds, each histogram’s labels are prefixed with the corresponding iteration number. The number of iterations h is determined based on the tree’s depth and a preset maximum number of iterations, taking the minimum of the two. These combined label counts form the final feature vector used to represent the expression tree.

Through this iterative aggregation process, the WL kernel method is able to capture information about different structural patterns within the expression tree and encode it into a fixed-dimensional feature vector, facilitating subsequent similarity calculations.

C Clustering Process

This section describes the process of clustering the WL-encoded theorem vectors. The core objective of clustering is to group theorems with similar structural features to accelerate the preliminary screening speed during subsequent theorem search. We first read the pre-processed theorem trees from the database. These trees are transformed into a more canonical and compact form by applying structural simplification techniques, such as common subexpression elimination (CSE). Subsequently, we apply the Weisfeiler-Lehman (WL) algorithm to encode these simplified trees. These encodings are converted into high-dimensional numerical vectors using a bag-of-features approach, where features correspond to the structural patterns extracted during the WL encoding process. In the feature extraction stage, we select the top 8000 most frequent features as the final feature set by counting feature occurrences across the entire dataset. To measure the distinctiveness of these features and reduce the influence of high-frequency generalized features, we apply an IDF-like weighting scheme to the feature counts in each vector. Considering the high dimensionality of the generated feature vectors, we use Principal Component Analysis (PCA) for dimensionality reduction to mitigate the ‘curse of dimensionality’ and reduce retrieval computational cost. PCA aims to find

directions of maximum variance in the data, projecting the vectors into a lower-dimensional space comprising 1200 principal components, thereby removing some noise and compressing the data representation. The reduced-dimensional vectors are then clustered using the Mini-Batch K-Means algorithm, an optimized variant of the K-Means algorithm particularly suitable for handling large datasets. Clustering is configured to find 10,000 clusters, utilizing an incremental learning approach where Mini-Batches of 10,000 samples are processed iteratively to update cluster centers. The algorithm converges after a set number of iterations. Upon completion of training, each theorem's vector in the database is assigned the cluster ID corresponding to its nearest learned cluster center, and the results are stored.

The entire clustering pipeline is implemented in Python, primarily utilizing the PCA and MiniBatchKMeans modules provided by the scikit-learn library for core computation, and psycopg2 for database interaction with the PostgreSQL database. Table 2 presents illustrative examples of clustering results, showing theorems grouped into the same clusters due to structural or semantic similarities, such as the cluster containing `Nat.add_comm` which includes several theorems related to the commutative property of addition. The effectiveness and characteristics of the clustering results are further visualized in Figure 3 and Figure 4. Specifically, Figure 3 presents the sizes of the top-N largest clusters, illustrating the scale of the most populated groups. Complementing this view, Figure 4 shows the histogram of cluster sizes across various predefined bins, providing insight into the overall distribution of cluster memberships, particularly the prevalence of smaller clusters. As is common in clustering diverse datasets, the distribution reveals a substantial number of small clusters, with a rapid decrease in frequency for larger cluster sizes.

Table 2: Display of clustering results for some classical theorems

Cluster ID 9881 <code>Nat.add_comm</code>	Cluster ID 9182 <code>Nat.add_left_cancel_iff</code>	Cluster ID 6559 <code>Nat.eq_zero_of_add_eq_zero_left</code>
<code>Nat.add_comm</code>	<code>Nat.add_le_add_iff_left</code>	<code>Nat.eq_one_of_mul_eq_one_left</code>
<code>Nat.add_left_comm</code>	<code>Nat.add_left_cancel_iff</code>	<code>Nat.eq_zero_of_add_eq_zero_left</code>
<code>Nat.land_comm</code>	<code>Nat.add_lt_add_iff_left</code>	<code>Nat.ne_zero_of_mul_ne_zero_right</code>
<code>Nat.min_comm</code>	<code>Nat.add_right_inj</code>	<code>AddSubgroup.Normal.conj_mem</code>
<code>Nat.min_left_comm</code>	<code>Nat.xor_right_inj</code>	<code>Filter.mem_inf_of_right</code>
<code>Nat.mul_left_comm</code>	<code>BitVec.neg_inj</code>	<code>Function.IsPeriodicPt.const_mul</code>
<code>Bool.or_left_comm</code>	<code>BitVec.neg_mul_neg</code>	<code>Int.emod_lt_of_pos</code>
<code>Bool.xor_left_comm</code>	<code>BitVec.xor_right_inj</code>	<code>List.append_ne_nil_of_ne_nil_right</code>
<code>Int.max_comm</code>	<code>Bool.bne_right_inj</code>	<code>List.append_ne_nil_of_right_ne_nil</code>
<code>Int.min_comm</code>	<code>Bool.xor_right_inj</code>	<code>List.disjoint_of_disjoint_append_left_right</code>

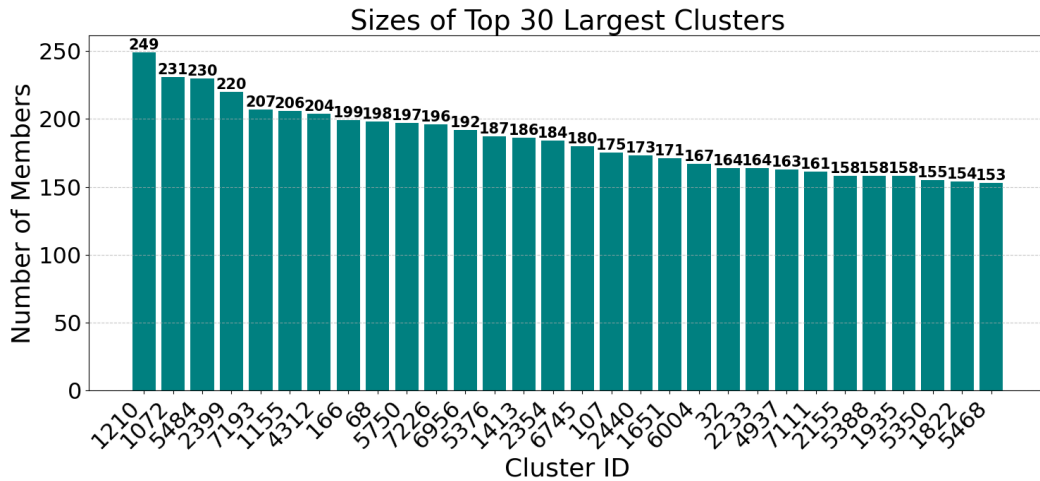


Figure 3: Sizes of the largest clusters (Top N)

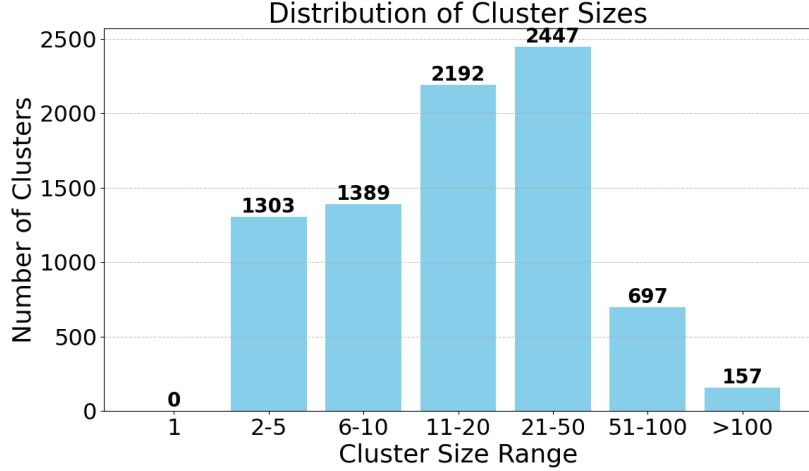


Figure 4: Histogram of cluster size distribution (by bin)

97 D CM Similarity and Jaccard Similarity of Const nodes

98 This section details two additional similarity metrics used in our method: CM similarity and Const
99 node Jaccard similarity.

100 **CM Similarity.** This metric measures how a source expression tree conforms to the structure of a
101 target expression tree based on modified collapse rules. Its core relies on a recursive scoring function:
102 for the current node in the target tree, if it is a leaf node, it contributes 1.0 to the total alignment
103 score regardless of the corresponding node in the source tree; if the current target node is not a
104 leaf, it requires the corresponding node in the source tree to match exactly in label and number of
105 direct children. If they match, the current target node contributes 1.0, and the alignment scores of all
106 corresponding children are recursively computed and summed up; if the label or number of children
107 do not match, this target node and its subtree cannot align, contributing 0. The final CM similarity is
108 the total score obtained by recursively calculating the alignment of the source tree to the target tree
109 structure, normalized by dividing by the total number of nodes in the target expression tree, scaling
110 the similarity value to the $[0, 1]$ range.

111 **Adjustable Weighting and Application Tendency.** As a crucial metric for evaluating structural
112 conformity, the weight of CM similarity in the final relevance score is adjustable. Users can tune
113 this weight according to their specific proof requirements. For instance, when the objective is to
114 find theorems that can be directly applied (apply) to the current proof state, the weight of CM
115 similarity in the overall score can be increased, as this typically signifies a stricter structural match.
116 Conversely, if the goal is to explore broader conceptual associations or find theorems that require
117 multiple transformation steps to be applied, the weight of CM similarity can be reduced, thereby
118 allowing more structurally less-matching but potentially logically relevant theorems to be retrieved.
119 This flexibility enables our premise selection method to adapt to proof tasks of varying complexity
120 and style, offering a more customized search experience.

121 **Jaccard Similarity of Const Node.** This metric aims to capture the semantic overlap between
122 expressions based on the constants used. The calculation process is as follows: Traverse both
123 expression trees and extract the `declName` strings contained within nodes whose labels start with
124 `Const`, collecting them into two sets of unique `declNames`, S_1 and S_2 . Then, compute the Jaccard
125 similarity of these two sets, i.e., $|S_1 \cap S_2| / |S_1 \cup S_2|$. If both sets are empty, the similarity is defined as
126 1.0. This similarity score reflects the degree to which the two expressions share common constants.

E The construction of test sets

E.1 Test set A

Test set A comprises 100 small-scale problems, designed to evaluate the method’s premise selection capability when dealing with theorems exhibiting specific structural variations. These problems are carefully categorized into three types: substitution-type, condition-swapping-type, and mixed-type, respectively assessing the method’s ability to recognize structural similarities, adapt to premise reordering, and handle combined transformations. Examples in Lean4 syntax from Test set A are provided in Table 3. Each problem is manually verified to ensure its correctness and alignment with its designated category.

Table 3: Examples from Test Set A: Structured Problem Variants

Category	Original Theorem	Variant
Substitution	<code>theorem Nat.add_comm (n m : Nat) : n + m = m + n</code>	<code>example : ∀ (n m : Nat), (n + 1) + m = m + (n + 1)</code>
Condition-swapping	<code>theorem Nat.nextPowerOfTwo_dec {n power : Nat} (h1 : power > 0) (h2 : power < n) : n - power * 2 < n - power</code>	<code>example : {n power : Nat} → (h2 : power < n) → (h1 : power > 0) → n - power * 2 < n - power</code>
Mixed	<code>theorem Nat.dvd_gcd {k m n : Nat} : k m → k n → k m.gcd n</code>	<code>example : {k m n : Nat} → k.succ n → k.succ m → k.succ m.gcd n</code>

E.2 Test set B

We construct Test set B from Mathlib4 version 4.18.0 by collecting all `exact` tactic invocations occurring in the proofs of theorem declarations. This test set comprises $m = 6,119$ (state, theorem) pairs automatically extracted from the Lean proof environment.

For each invocation of the `exact` tactic, we capture the proof state, including local context, current goal, and relevant metadata, immediately before the tactic is applied, and pair it with the theorem supplied to `exact`. We then convert each proof state into a standardized problem statement. We use this dataset to compare our method with existing Lean premise retrieval tools, evaluating their ability to predict the next proof step (the theorem that advances the proof) given a well-defined proof context.

Since the extraction process modifies the Lean compiler, we provide a self-contained Git patch containing the required instrumentation and extraction logic. We have validated this pipeline on multiple Lean4 releases around version 4.18.0, confirming its stability and compatibility across minor revisions.

E.2.1 Overview of Lean Tactics

In Lean, tactics are high-level commands that guide the proof assistant by transforming the current goal into simpler subgoals or by directly supplying a proof term. For example, the `exact` tactic closes a goal immediately by providing a term whose type matches the goal’s statement: `exact h`.

Here, Lean checks that the hypothesis `h` has exactly the required type and uses it to discharge the goal in one step. See Figure 5 for an illustrative flow of how tactics like `exact` operate within a proof.

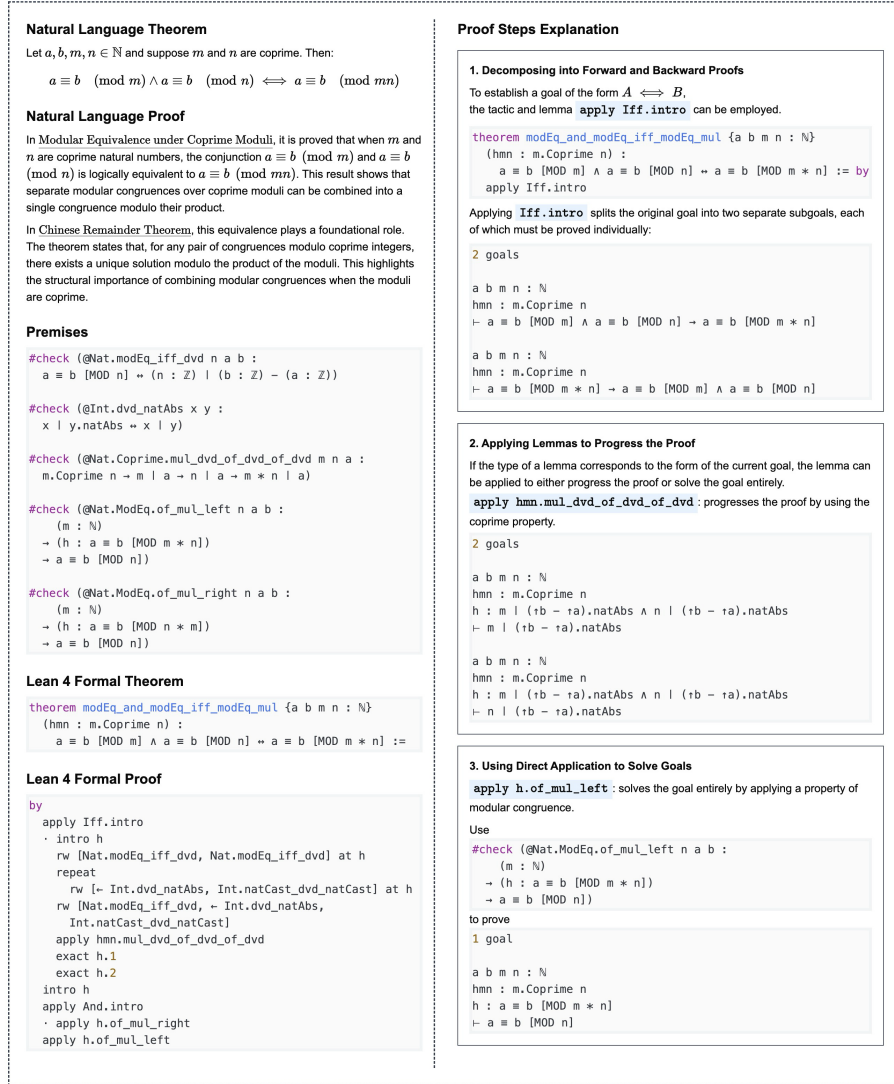


Figure 5: Proof workflow of the `modEq_and_modEq_iff_modEq_mul` lemma in Lean, showing how the two congruence goals are split, rewritten to divisibility, and reassembled under the coprimality assumption.

Furthermore, `exact` is closely tied to premise selection: it requires that the user or an earlier automation has already identified and brought into context a proof object (lemma, hypothesis, or previously constructed term) whose type exactly matches the goal. In practice, this means that successful use of `exact` often follows a search or filtering of available premises to find one that precisely fits, making it the final step in a proof-search or tactic pipeline.

E.2.2 Extraction and Transformation of `exact` Invocations

In Test set B, we focus on the `exact` tactic: it takes a proof term (typically an existing lemma or theorem) and closes the current goal when the term exactly matches the goal's type. For each invocation of `exact`, the underlying proof step may originate from a combination of higher-order function applications and auxiliary tactics. However, after Lean's elaboration phase, every such invocation is represented as an `Expr` of the form $(f\ x\ y\ z\ \dots)$ where f is the theorem being applied and x, y, z, \dots are its explicit arguments. In our Test set B generation pipeline, we collect the inferred types of all these explicit arguments and append them to the original goal, thereby yielding a fully instantiated goal proposition with a complete type signature that reflects the concrete instantiation scenarios encountered in premise selection. The transformation illustrated in Figure 6 exemplifies

171 how our extraction pipeline turns a raw `exact` invocation into a fully instantiated search query, by
 172 appending every explicit argument type to the original goal.

```
exact Nat.pow_le_pow_left
  (add_le_add_right (Nat.mul_div_le i j) _) -- explicit argument 1
  (·) -- explicit argument 2:
  | | -- underscores '_' are synthesized by Lean's elaborator via type inference.
```

(a) Paired theorem invocation

▼Tactic state

```
1 goal
i j : ℕ
hj : 0 < j
this : j * (j - 1) < j ^ 2
⊢ (j * (i / j).succ) ^ 2 ≤ (i + j) ^ 2
```

(b) Captured proof state

▼Tactic state

```
1 goal
i j : ℕ
hj : 0 < j
this : j * (j - 1) < j ^ 2
⊢ j * (i / j) + j ≤ i + j → ∀ (x1 : ℕ),
  (j * (i / j).succ) ^ 2 ≤ (i + j) ^ 2
```

(c) Transformed proof state

Figure 6: Example of transformation of a proof state under an `exact` invocation in Test set B. (a) displays the corresponding `exact` tactic invocation using a composed proof term. (b) shows the proof state immediately before the tactic is applied. (c) shows the reconstructed, logically equivalent but more structured goal that exposes the necessary conditions for the theorem application.

173 In our pipeline, each transformation as illustrated in Figure 6 is recorded and serialized as JSON.
 174 This machine-readable format makes it easy to generate valid search queries for different premise
 175 retrieval tools, allowing systematic testing of our method and other approaches.

176 This transformation is crucial for evaluating premise-retrieval tools: although the constant f often
 177 has a very general polymorphic type, in real proof developments the goals resolved by `exact` are
 178 highly instantiated and structurally complex. By exposing the concrete argument-type information,
 179 our dataset rigorously tests a tool’s ability to select the correct theorem under realistic, nontrivial
 180 instantiations.

181 E.2.3 Implementation Details of the Data Transformation and Extraction Pipeline

182 To support this extraction, Lean tactics operate within the `TacticM` monad, which provides access to:

- 183 • **Local context:** the list of hypotheses and their types;
- 184 • **Goal:** the current proposition to be proved;
- 185 • **Metadata:** such as proof position, the name of the invoking lemma or theorem, and
 186 environment declarations;
- 187 • **Elaboration:** converting `Syntax` objects into fully elaborated `Expr` terms;
- 188 • **Meta-variable resolution:** instantiating and solving meta-variables generated during proof
 189 construction;
- 190 • **Type inference:** inferring the type of arbitrary `Expr` terms within the current environment.

191 To construct the test set, we instrument the built-in `exact` tactic to capture structured information at
 192 each invocation. The extraction process operates as follows.

- 193 1. We extend the `TacticM` environment and context to record additional state required for the
 194 transformation with additional internal state, including elaboration results, meta-variable
 195 assignments, and relevant `Syntax` information required during reconstruction.
- 196 2. When `exact e` is executed, we first elaborate the term e to an expression whose type must
 197 match the current goal. We then attempt to extract the head constant name (i.e., the applied
 198 theorem) from the syntax and expression of e , using a combination of syntactic resolution
 199 and application spine inspection.

- 200 3. Once the theorem is identified, we retrieve its type, instantiate any meta-variables in the
201 argument terms, and collect the types of all explicitly passed arguments. We then append
202 these instantiated argument types to the current goal, producing a normalized target type
203 that captures both the proof context and the specific theorem application, while preserving
204 the original proof state unmodified.
- 205 4. We compute a simplified version of the goal by recursively reducing all subexpressions. This
206 normalization step helps eliminate intermediate constructs introduced by instance resolution
207 and other elaboration mechanisms, making the goal type easier to analyze and compare.
- 208 5. In addition to the core type-level information, we also collect auxiliary metadata such as
209 the source file URI, position range, command kind, and a reprint of the original tactic
210 syntax. All extracted information, including the normalized goal type and associated context,
211 is serialized into a JSON object, which is then uploaded to a PostgreSQL database for
212 downstream consumption.

213 F Detailed Visualization of Search Results

214 This section provides detailed visualizations of search outcomes to further elucidate the efficacy
215 and operational mechanism of the proposed premise selection methodology. Examples of query
216 expressions and their corresponding matched theorems retrieved by the system are visually depicted
217 as tree structures in Table 5. These examples, derived from variants of theorems like `Nat.add_comm`,
218 allow for a qualitative inspection of the structural similarities leveraged by the method for determining
219 relevance. Through these successful retrieval cases, the immense power of structural information
220 in identifying pertinent theorems is clearly demonstrated. Further concrete examples of premise
221 retrieval are detailed in Table 4, listing target expressions in Lean format alongside the names of the
222 top-5 theorems retrieved by the system. Collectively, these figures and tables offer readers detailed
223 qualitative and concrete examples for a deeper understanding of the search process’s operational
224 characteristics and the underlying principles of our method.

Table 4: Illustrative examples of retrieved theorems for selected target expressions.

Target Expression in Lean:

$(n\ m : \text{Nat}) \rightarrow (n + 1) + m = m + (n + 1)$

Top-5 Retrieved Theorem(s) (Name):

`Nat.max_comm`; `Nat.add_comm`; `Nat.min_comm`; `Nat.xor_comm`; `Nat.or_comm`;

Target Expression in Lean:

$\{a\ b\ c : \text{Nat}\} \rightarrow (h : a + c < b) \rightarrow \neg b < a + c$

Top-5 Retrieved Theorem(s) (Name):

`Nat.ModEq.symm`; `Batteries.UnionFind.Equiv.symm`; `Nat.lt_asymm`; `Nat.ModEq.comm`;
`Nat.le_total`;

Target Expression in Lean:

$\{b\ a : \text{Nat}\} \rightarrow (h_1 : 0 < b) \rightarrow (h_2 : b \leq a.\text{succ}) \rightarrow a.\text{succ} / b = (a.\text{succ} - b) / b + 1$

Top-5 Retrieved Theorem(s) (Name):

`Nat.div_eq_sub_div`; `Nat.choose_le_succ_of_lt_half_left`;
`Nat.le_two_mul_of_factorization_centralBinom_pos`; `Nat.log_mul_base`;

Target Expression in Lean:

$(m\ n\ k : \text{Nat}) \rightarrow (m.\text{succ} * n).\text{gcd}\ (m.\text{succ} * k) = m.\text{succ} * n.\text{gcd}\ k$

Top-5 Retrieved Theorem(s) (Name):

`Nat.gcd_mul_left`; `Int.gcd_natCast_natCast`; `Nat.gcd_assoc`; `Nat.lcm_mul_left`; `Nat.dvd_gcd`;

Target Expression in Lean:

$\{n : \text{Nat}\} \rightarrow \{a\ b : \text{Fin}\ n.\text{succ}\} \rightarrow (h : a < b) \rightarrow \uparrow a + 1 \leq \uparrow b$

Top-5 Retrieved Theorem(s) (Name):

`Fin.add_one_le_of_lt`; `Fin.cycleRange_of_gt`; `Fin.add_one_lt_iff`; `Fin.add_one_le_iff`;
`lt_finRotate_iff_ne_last`;

Table 5: Visual Trees of Proposition Search Results

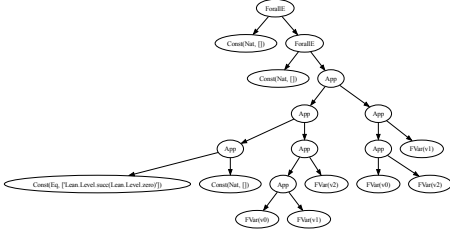
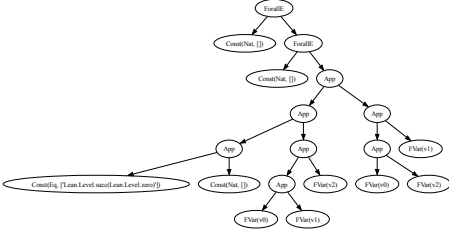
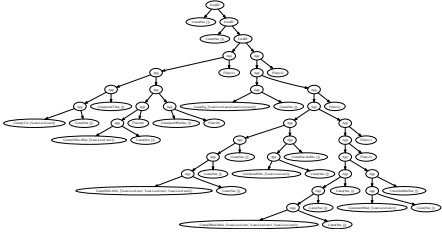
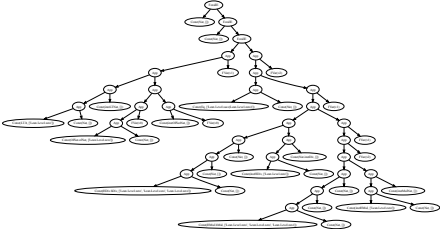
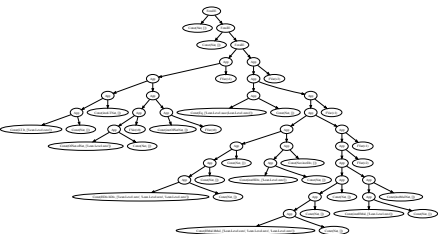
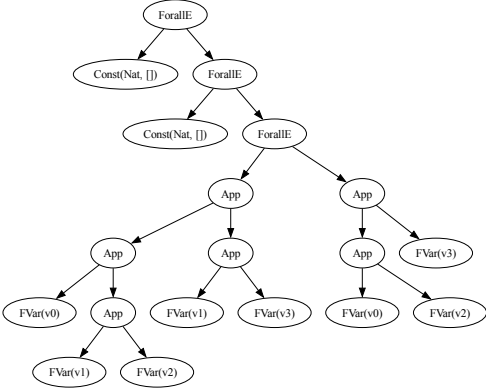
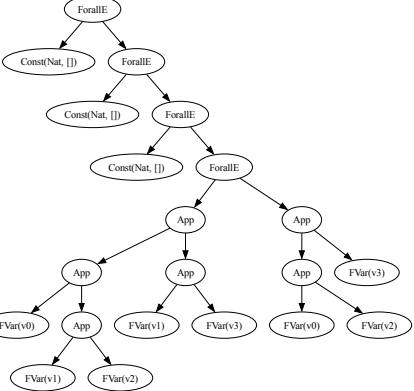
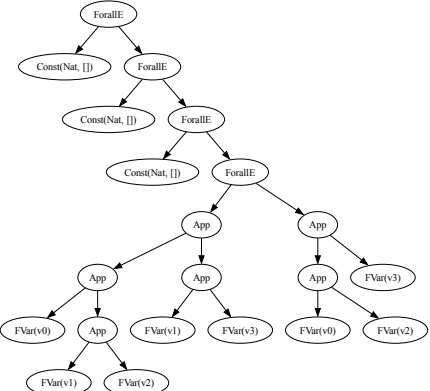
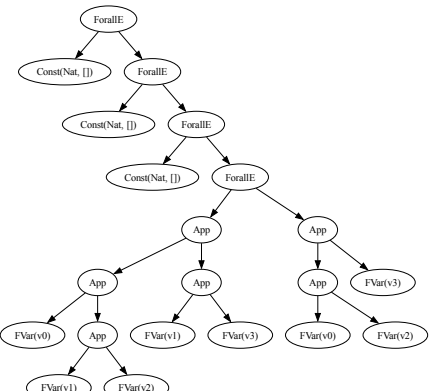
Query Proposition Tree	Matched Theorem Trees
$(n\ m : \text{Nat}) \rightarrow (n + 1) + m = m + (n + 1)$ 	<p>Nat.and_comm</p> 
$(m : \text{Nat}) \rightarrow n : \text{Nat} \rightarrow (H : 0 < n.\text{pred}) \rightarrow m * n.\text{pred} / n.\text{pred} = m$ 	<p>Nat.mul_div_cancel</p> 
	<p>Nat.mul_div_left</p> 

Table 5 (continued): Visual Trees of Proposition Search Results

Query Proposition Tree	Matched Theorem Trees
$\{a : \text{Nat}\} \rightarrow a + b = a + 1 \rightarrow b = 1$ 	$\text{Nat.add_left_cancel}$ 
	$\text{Nat.lt_of_add_lt_add_left}$ 
	$\text{Nat.lt_of_mul_lt_mul_left}$ 

225 G Limitations

226 Despite the significant effectiveness achieved by the proposed method in premise selection, certain
 227 limitations remain that can be addressed in future work. First, complex definition unfolding and
 228 implicit instance conversions in Lean can lead to expressions with similar mathematical meaning
 229 exhibiting considerable structural differences at the tree level. This structural divergence increases the
 230 difficulty of precise matching, necessitating further research into unifying or more robustly handling
 231 these structural variations. Second, there is still room for speed optimization. Current experiments are
 232 primarily conducted on local computing resources; on more powerful machines, retrieval efficiency
 233 can be further improved through enhanced parallelization and database-level optimizations (e.g.,
 234 placing hot data in an in-memory database).

235 A core challenge stems from the inherent complexity of Lean 4 expression trees and the dynamic
 236 nature of its proof environment. An illustrative example involves the query expression `padicValNat`
 237 `p n < padicValNat p n + 1` (depicted in Figure 9) and its relevant theorems such as `Nat.lt_succ_self`:
 238 `n < n.succ` (shown in Figure 7) and `Nat.lt_add_one`: `n < n + 1` (illustrated
 239 in Figure 8). While `n.succ` and `n + 1` are mathematically equivalent, Lean’s automatic implicit
 240 instance conversion for `+` results in structurally different expression trees. Lean’s exact tactic can
 241 automatically unfold definitions to handle such equivalences during theorem application, our method,
 242 by strictly adhering to the expression tree structure, perceives `n.succ` (representing the definition
 243 of successor) and `n + 1` as distinct sub-tree structures. This structural distinction, as evident when
 244 comparing Figure 7 and Figure 8, means that even theorems that are semantically identical to the
 245 query can exhibit significant structural differences in their corresponding expression trees.

246 Furthermore, Lean’s automatic implicit instance conversion mechanism poses another signifi-
 247 cant challenge for tree-based structural matching. For instance, what might appear as a simple
 248 `Const(instLTNat, [])` node (representing a type class instance) in a theorem’s simplified repre-
 249 sentation could expand into a large, complex instance conversion sub-tree within the target expression
 250 due to type inference and instance lookup. This structural inflation is particularly noticeable when
 251 observing the second node (or its corresponding sub-tree) from the left at the fourth level of the
 252 expression trees, as depicted in Figure 9 and Figure 8. These "large and complex instance conversion
 253 sub-trees" automatically inserted by the compiler and not directly related to the core mathematical
 254 semantics, drastically increase the structural discrepancy between query and target expression trees.
 255 Both the definition unfolding (e.g., `n.succ` vs. `n + 1`) and the implicit instance conversions con-
 256 tribute to these structural divergences, consequently impacting the retrieval accuracy of tree similarity
 257 algorithms. While our CM similarity metric was designed to partially mitigate this by introducing
 258 a match-degree based score rather than strict structural equality to enhance robustness, the overall
 259 structural inflation remains a critical area for further research.

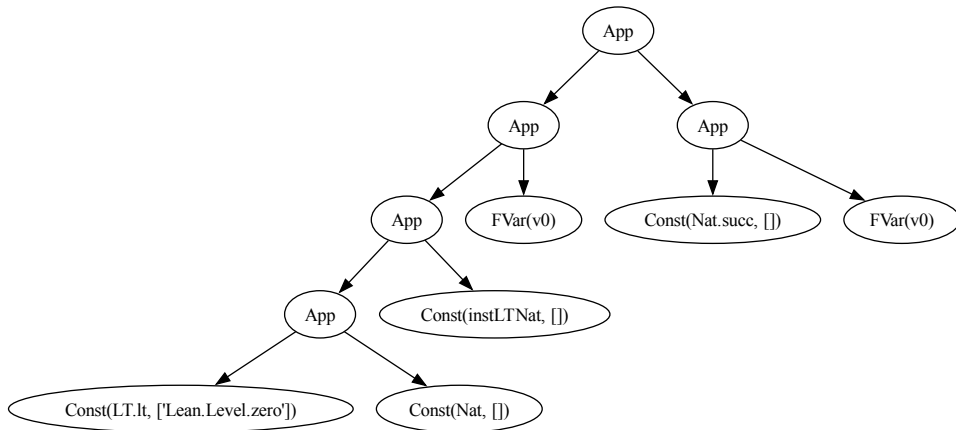


Figure 7: Simplified expression tree of theorem `Nat.lt_succ_self`

I Extended Performance Evaluation

I.1 Analysis of Efficiency and Performance Trade-off

This section focuses on analyzing the trade-off between efficiency and performance of the proposed clustering-based premise selection method. To quantify the impact of the clustering mechanism, we compare the full method ("With Clustering") against a baseline of full-library search without preliminary clustering ("Without Clustering"). All experiments were conducted on test set A, recording the query time distribution (Min, Average, Max), total time, speedup, and Recall@k at various k values for each method.

Table 6 presents a detailed comparison of the results. Regarding query time, the "With Clustering" method demonstrates a significant advantage in average query time. Its average query time of 14.574 seconds is substantially lower than the 30.973 seconds of the "Without Clustering" method. This translates to a remarkable speedup of **2.13x**, indicating that the clustering preliminary screening mechanism successfully narrows down the search space, thereby significantly boosting retrieval efficiency. Even in the worst-case scenario (Max Query Time), the "With Clustering" method exhibits superior performance (99.011 seconds vs. 110.329 seconds). However, this gain in efficiency comes with a certain degree of accuracy trade-off. From the Recall@k metrics, the "With Clustering" method shows a slight decrease in Recall@1, Recall@5, and Recall@10 compared to the "Without Clustering" baseline. For instance, Recall@10 drops from 95.0% to 86.0%. This suggests that clustering, as an approximate filtering method, might filter out a small number of truly relevant theorems that were not assigned to the target clusters, in exchange for accelerated retrieval. Despite the marginal decrease in recall, we argue that this efficiency-performance trade-off is highly practical and acceptable for interactive theorem proving environments. In the context of real-world theorem proving, users typically prioritize quickly obtaining a high-quality and explorable set of preliminary results to promptly proceed with their proof attempts or strategy adjustments, rather than spending extensive time waiting for a precise full-library ranking. The 2.13x speedup can significantly enhance user experience and improve the iterative efficiency of the entire proving process. This study successfully demonstrates that by incorporating structured representations and efficient clustering for preliminary screening, a qualitative leap in theorem retrieval efficiency can be achieved while maintaining sufficient accuracy.

Table 6: Significant Efficiency Gains of Clustering-based Premise Selection

Method	Query Time (s)			Total Time (s)	Speedup	Recall@k (%)		
	Min	Average	Max			@1	@5	@10
Without Clustering	2.295	30.973	110.329	3097.3	1.00x	90.0	91.0	95.0
With Clustering	1.760	14.574	99.011	1457.4	2.13x	82.0	84.0	86.0

I.2 Domain-Specific Performance Comparison

To comprehensively evaluate our proposed "Tree-Based Search" method, a comparison was conducted against several existing theorem retrieval approaches (including Lean Search, Moogler, Lean Search, Lean Search, Lean Search, and Lean Search) within a specific mathematical domain. This experiment focused on the Natural Number domain, with theorems further categorized by complexity into "Simple," "Medium," and "Complex" levels. The evaluation metric remained Recall@k (k=1, 5, 10), aiming to measure the system's ability to recall relevant theorems at various retrieval depths.

The retrieval performance across different complexity levels within the Natural Number domain is summarized in Table 7. From the results, it is evident that our "Tree-Based Search" method consistently and significantly outperforms all compared methods across all complexity levels. Notably, for both "Simple" and "Medium" complexity Natural Number theorems, the highest Recall@1, Recall@5, and Recall@10 were achieved by our method, with Recall@10 reaching 82.7% in the "Simple" category, far exceeding the next best approach.

As theorem complexity increases, a general decrease in retrieval performance is observed across all methods. However, even in the most challenging "Complex" category, a relatively significant advantage is maintained by "Tree-Based Search" (Recall@10 reaching 38.1%), whereas other methods

often show recall rates near or at 0%. This strongly demonstrates the robustness and effectiveness of our tree-based structural representation and similarity matching approach in handling mathematical expressions of varying complexity, especially in more complex theorems where its ability to capture deeper structural information becomes crucial.

Table 7: Retrieval performance (%) of different methods on natural number domain categorized by complexity

Domain	Complexity	Count	Method	Recall@k		
				@1	@5	@10
Natural Number	Simple	76	Lean Search	17.1	26.3	27.6
			MoogLe	7.9	14.5	22.4
			Lean Search Agu	16.0	20.0	20.0
			Lean Explore	2.7	20.0	38.7
			Lean State Search	30.3	50.0	63.2
			Tree-Based Search	68.0	81.3	82.7
	Medium	126	Lean Search	6.3	15.9	21.4
			MoogLe	0.0	4.8	7.9
			Lean Search Agu	7.2	15.2	20.0
			Lean Explore	0.0	2.4	9.5
			Lean State Search	10.3	23.8	25.4
			Tree-Based Search	38.9	42.1	44.4
	Complex	21	Lean Search	0.0	0.0	0.0
			MoogLe	0.0	4.8	4.8
			Lean Search Agu	0.0	0.0	9.5
			Lean Explore	0.0	0.0	0.0
			Lean State Search	4.8	14.3	14.3
			Tree-Based Search	23.8	38.1	38.1

This detailed performance is further illuminated through various visualizations. A comprehensive 4x4 comparison of our method against all other models across 'Recall@1', 'Recall@5', 'Recall@10', and 'MRR' metrics within the Natural Number domain is provided in Figure 11, clearly demonstrating our consistent lead. The distribution of search queries by difficulty level in this domain is presented in Figure 10, offering context on the dataset composition. Furthermore, the trends of key metrics (MRR, Recall@5, nDCG@10) across different difficulty levels are visualized in Figure 12, highlighting our method's robust performance even as complexity increases. A focused view on the '@1' metrics (Recall@1, Precision@1, F1@1, and nDCG@1) is provided in Figure 13, offering granular insights into the initial retrieval effectiveness. Collectively, these visualizations underscore the superior effectiveness of our Tree-Based Search method across different evaluation criteria and complexity levels.

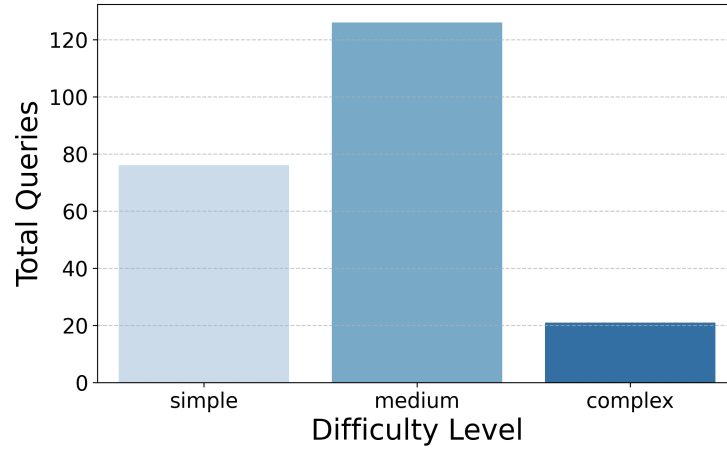


Figure 10: Total Queries by Difficulty Level in Natural Number Category.

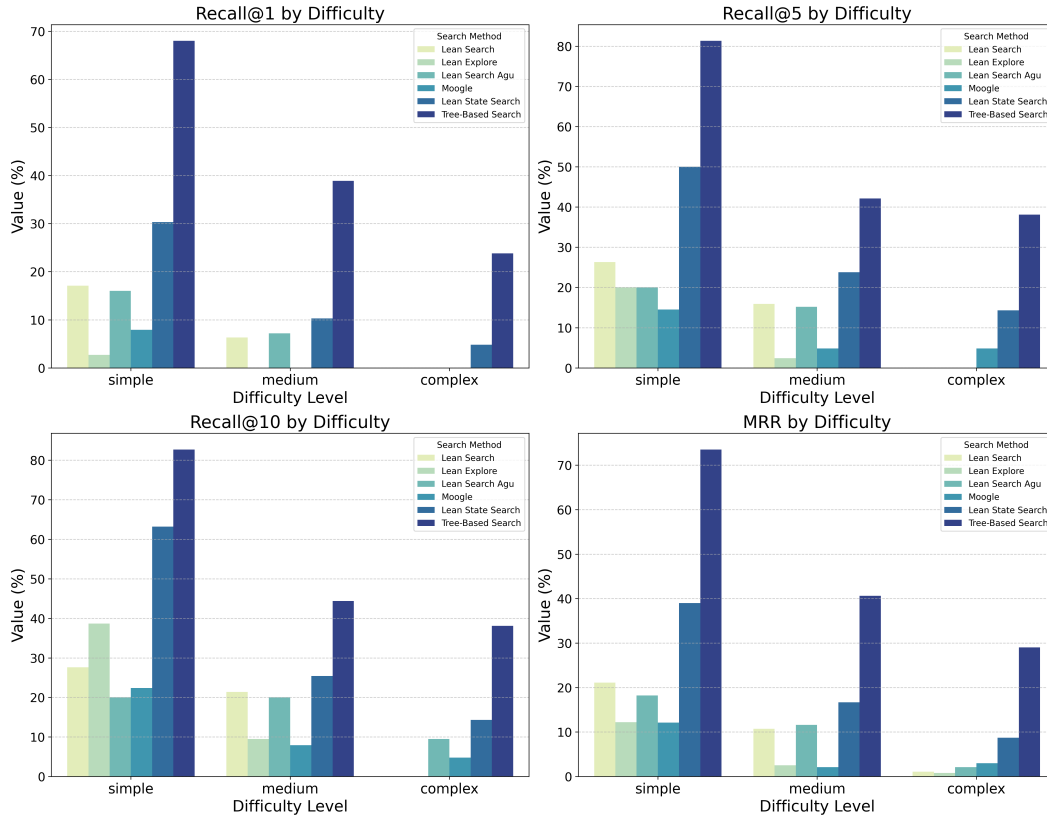


Figure 11: Performance comparison of search methods across Recall@k and MRR by difficulty level in Natural Number domain.

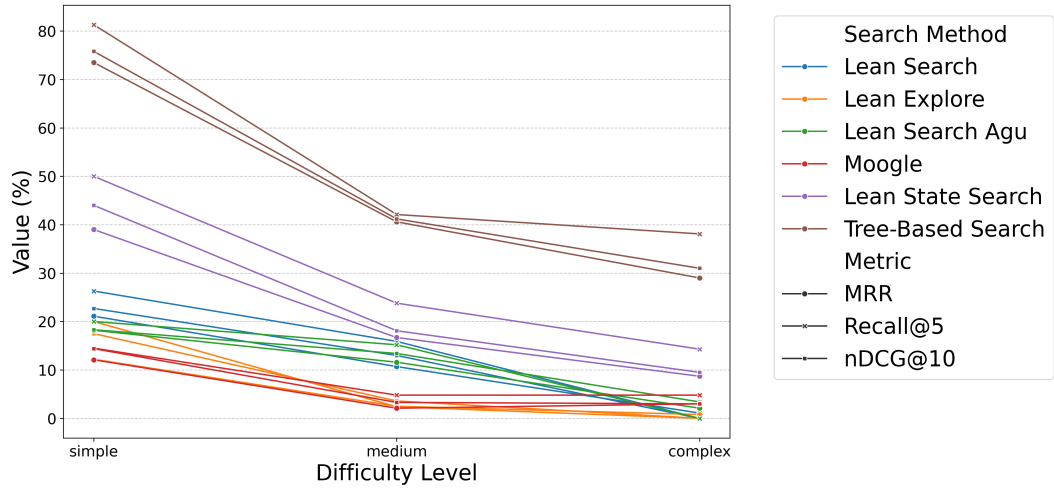


Figure 12: Key Metric Trends (MRR, Recall@5, nDCG@10) by Difficulty Level in Natural Number Category.

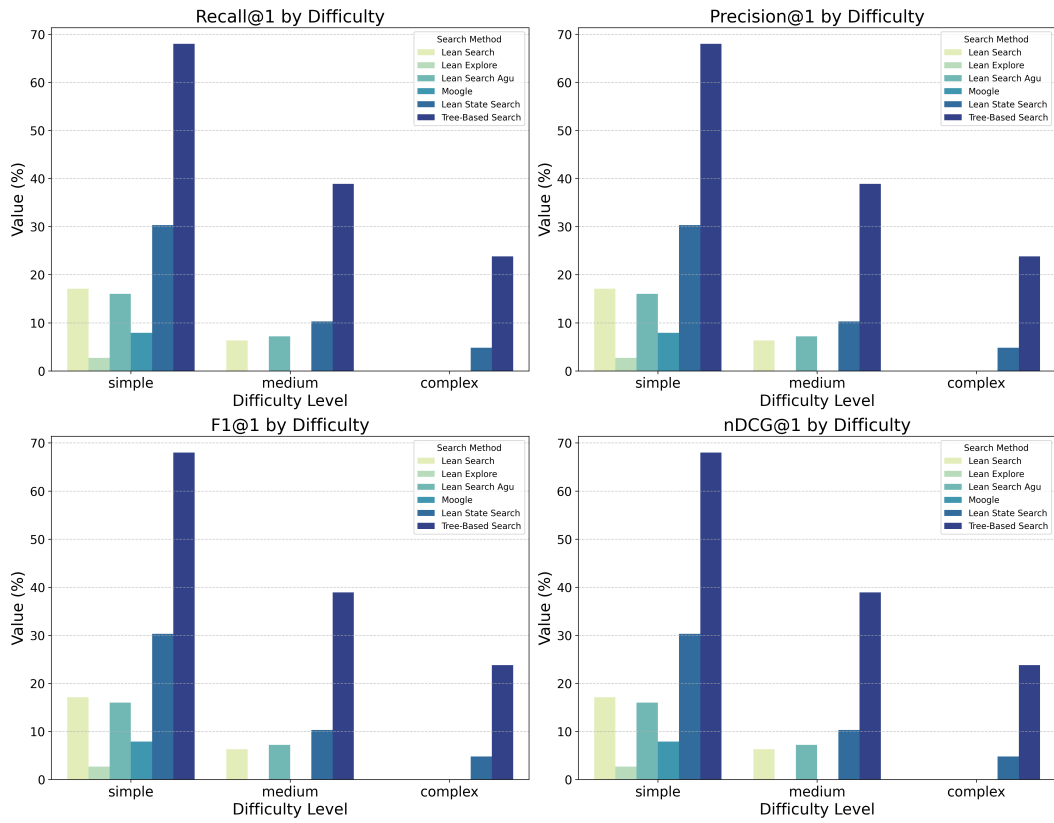


Figure 13: Grouped Bar Charts for @1 Metrics (Recall@1, Precision@1, F1@1, nDCG@1) by Difficulty Level.