Optimising Metamath Proofs for Human Working Memory

Jeremy Lindsay, Cezary Kaliszyk, and Christine Rizkallah

University of Melbourne, Australia
jeremy.n.lindsay@student.unimelb.edu.au
cezary.kaliszyk@unimelb.edu.au
christine.rizkallah@unimelb.edu.au

Abstract. Mathematical proofs vary in legibility. While most proof optimisation techniques seek to minimise proof size, the strategic reordering of inferences can reduce the working memory demand of proof checking without altering overall size. Metamath serves as a prime case study for this approach: its verification architecture requires proof steps to be ordered in a manner that prioritises algorithmic efficiency over readability. In this paper, we introduce algorithms to minimise both peak and cumulative memory consumption, applying the latter as a novel proxy for sustained human cognitive effort. We achieve this by representing proofs as directed acyclic graphs and modelling their execution as a pebbling game. Finding an optimal ordering via brute force is computationally infeasible, so we use heuristics to provide approximations. We apply these algorithms across Metamath's ZFC set theory library and present case studies demonstrating how automated reordering systematically improves the presentation of formal mathematics.

Keywords: Metamath · Proof reordering · Working memory · Pebbling games · Legibility · Cumulative cost · Set.mm

1 Introduction

An important aspect of proof readability is the demand it places on a reader's working memory. Consider a mathematician presenting a proof on a chalkboard, writing down intermediate results and erasing them when they are no longer needed. The total amount of chalk used reflects the proof's length, and the size of the board limits the number of results that can be simultaneously retained for later reference. Yet, if even a small chalkboard is continuously cluttered with results awaiting later reference, the audience may become overwhelmed. We, therefore, prefer proofs that are not only short and space-efficient, but that also keep the running total of retained intermediate results low. In this paper, we introduce algorithms to reorder proof steps that minimise the latter two metrics, while preserving the proof length. Though legibility is subjective and multifaceted, we target these metrics as concrete, quantifiable proxies for human cognitive load.

Formal mathematics libraries of proof assistants like Lean [13], Rocq [24], and Metamath [12], provide a natural setting for this problem. Their proofs have explicit dependency structure, so valid reorderings can be specified precisely rather than judged informally. The methods we present in this paper are general-purpose and require only abstract graph-based representations of proof structure. Nevertheless, we perform our experiments on Metamath proofs in particular, for a few reasons. First, Metamath's syntax and verification mechanisms are simple, making it straightforward to export proofs into a graph format. Recent tooling such as LeanTree [9] demonstrates the feasibility of extracting structured proof representations from other proof assistants, but we leave this for future work. Second, Metamath requires proof steps to be ordered in a highly specific manner that is optimised for verification rather than readability. By default, Metamath proof editors display proofs in this native format, though they allow steps to be reordered during the editing stage. Thus Metamath provides a clear testbed for studying whether automatic reordering can improve the presentation of formal proofs.

Our approach is as follows. We represent a proof as a *directed acyclic graph (DAG)* whose vertices are distinct inferences and edges represent logical dependencies. A traditional, sequential presentation of the proof is then a *topological order* on this DAG. To model memory demand as the proof is followed step by step, we use the formalism of *pebbling games*. It has been shown for a variety of metrics that brute force methods for computing an optimal topological order are infeasible in practice [3, 17, 20]. We therefore turn to heuristic algorithms.

Our contributions are the following:

- We formulate the problem of optimising proofs for *sustained* demand on working memory. This is captured by cumulative memory use across the proof, which we measure alongside the better-studied objective of minimising peak memory consumption.
- We introduce metrics closely related to Karol Pąk's legibility metrics [16, 17], and show where our memory consumption metrics diverge.
- We introduce a novel variant of the pebbling game that treats unpebbling as an automatic process, simplifying memory calculations as a result.
- We develop heuristic reordering methods that combine ideas from Fellner and Paleo's greedy algorithm [3], Sethi–Ullman numbering [21], and other lightweight ordering criteria. We apply these algorithms to Metamath's ZFC set theory library `set.mm`, analyse the results, and present case studies.

More broadly, this contributes towards improving proof legibility for humans, a problem that is likely to become more important as LLM-based proof generation and proof transformation become more common [4, 26, 28].

2 Related Work

Most proof optimisation work seeks to minimise proof length or verification time through techniques like definition invention [25], redundancy pruning [16], LLM-based refactoring [1, 6, 29], and abstraction discovery [7, 19]. Instead, we consider

the problem of how the *ordering* of proof steps affects memory consumption and legibility, while the overall proof length is held constant.

Karol Pąk was the first to consider the impact of proof step ordering on legibility. His work encompasses novel optimisation algorithms [15], experimental studies with human subjects [18], and theoretical results showing that various proof reordering problems are NP-complete [17]. Pąk's metrics are based on the *close reference principle*, which recommends that premises should be introduced soon before they are required in a logical inference. These metrics can be calculated on a static proof graph; our work diverges in that we model the evolution of memory consumption over time during proof checking. Nevertheless, we introduce two metrics, the *mean index distance*, and the *adjacency rate*, that are inspired by Pąk's *3rd Method of Improving Legibility* [17] and the *4th Method of Improving Legibility* [15] respectively. Using these metrics, we present case studies demonstrating how our memory consumption metrics can diverge from metrics based solely on the close reference principle.

Pebbling games are used in the field of *proof complexity* to analyse the space complexity of proof verification [14]. Proof complexity is a theoretical discipline concerned with establishing asymptotic bounds on the size and space requirements of various proof systems [8]. In contrast, our work is applied: rather than analysing the theoretical limits of Metamath, our focus is the efficient optimisation of concrete proofs. The work of Fellner and Paleo [3] most closely resembles ours: they study space optimisation for machine-generated resolution proofs using both an exact SAT formulation and a greedy algorithm. Our depth-first search approach is directly inspired by the latter, but they focus only on peak memory consumption, whereas we view cumulative memory consumption as a proxy for sustained cognitive load in humans.

3 Metamath

Metamath is a logic-agnostic proof assistant: without provided axioms, it contains no inherent mathematical content. Instead, users specify Hilbert-style axioms as explicit foundations for different mathematics libraries. In this paper, we restrict ourselves to the ZFC set theory library `set.mm`, the largest and most mature Metamath library.

During verification, Metamath uses a *stack* to keep track of proof states. Each proof step either introduces a given assumption or applies an inference rule, producing a new mathematical expression[1]. In `set.mm`, *modus ponens* appears as the axiom `ax-mp`, which states that given P and $P \to Q$, we can derive Q. Crucially, Metamath theorems, like functions in programming languages, expect arguments to be specified in a particular order. Theorems are assigned abbreviated mnemonic labels: `mp2`, for instance, stands for *double modus ponens*, and it has three ordered premises:

[1] This is a simplification: like the Metamath Proof Explorer, we abstract away purely syntactic construction steps and track only expressions with typecode `|-`.

premise 1	`mp2.1`	φ
premise 2	`mp2.2`	ψ
premise 3	`mp2.3`	$\varphi \to (\psi \to \chi)$
conclusion	`mp2`	χ

To prove `mp2`, we apply modus ponens twice. Metamath's stack, as depicted in Table 1, evolves as follows:

1. Initially, the stack is empty.
2. Premise 2, ψ, is pushed to the stack.
3. Premise 1, φ, is pushed.
4. Premise 3, $\varphi \to (\psi \to \chi)$, is pushed.
5. Modus ponens is applied to φ and $\varphi \to (\psi \to \chi)$, popping both and pushing the result $\psi \to \chi$.
6. Modus ponens is applied again to ψ and $\psi \to \chi$, popping both and pushing the result χ. The stack now contains exactly the goal formula, so the proof is complete.

In general, the application of a theorem with n premises corresponds to mechanically popping the top n elements of the stack, applying the theorem, and pushing the result onto the stack.

Table 1. Metamath's stack during the proof of `mp2`. Each column corresponds to a step in the proof, and each row corresponds to a position on the stack.

			$\varphi \to (\psi \to \chi)$		
		φ	φ	$\psi \to \chi$	
	ψ	ψ	ψ	ψ	χ
1	**2**	**3**	**4**	**5**	**6**

Metamath requires inferences to be ordered in the precise sequence in which they are evaluated on the stack. (If an expression is required multiple times in a proof, Metamath will cache it and push it back onto the stack when necessary. We do not consider this operation a new inference.) This makes the native Metamath format unusually laborious to read. The proof of `prmunb` (unboundedness of primes) is paradigmatic: the first step introduces an expression that is not referenced until the very end of the proof, a full 47 steps later. In practice, proof editors like `mmj2` can be used to present proof steps in any order, so long as their premises are referenced appropriately. However, proofs are initially loaded from `set.mm` with the original step order, and are also compiled back into the native format.

4 Proof DAGs and Topological Orders

In this section, we show how proofs can be represented as directed acyclic graphs (DAGs), with vertices representing inferences, and edges representing logical de-

pendencies. The acyclic condition prohibits circular reasoning, whereby a mathematical expression could be derived from itself.

Definition 1 (Proof DAG). *Let $\mathcal{E}$ denote the set of all valid mathematical expressions, and let P be a formal proof consisting of a sequence of inferences. A* proof DAG *is a tuple $(V, E, \varepsilon, \lambda)$ where:*

1. *The pair (V, E) is a directed acyclic graph with a unique sink. The vertices V represent the distinct inferences in P, and an edge $(u, v) \in E$ indicates that inference v directly consumes the output of inference u.*
2. *The function $\varepsilon : V \to \mathcal{E}$ maps each vertex to the mathematical expression it derives. (Injectivity is not required, so expressions may be rederived.)*
3. *The function $\lambda : E \to \mathbb{Z}^+$ assigns to each edge $(u, v) \in E$ the premise index of u within the theorem applied at v.*

In Figure 1, for example, the three sources are mapped by ε to the three premises of `mp2`, and the sink is mapped to the conclusion χ. The edges are labelled by λ according to `ax-mp`'s ordered premises: the minor premise P followed by the major premise $P \to Q$.

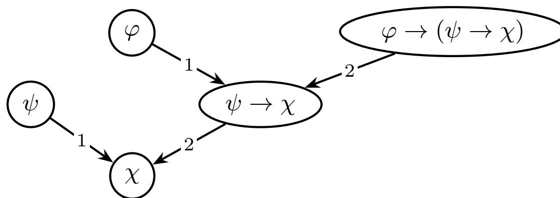

Fig. 1. Proof DAG for the double modus ponens theorem (`mp2`). Vertices v represent distinct inferences and are labelled with their derived expressions $\varepsilon(v)$.

Given a proof DAG, there are in general multiple valid sequences (linearisations) of proof steps that can be constructed. Each of these corresponds to a *topological order* $\prec$ on the DAG; we write $V_\prec$ for the induced sequence of vertices in V. Since an edge $(u, v) \in E$ represents a logical dependency on u by v, ideally u should be executed not long before v. The average of this dependency distance across all edges is thus a natural proxy metric for legibility.

Definition 2 (Mean index distance). *Let $G = (V, E)$ be a DAG, and let $\prec$ be a topological order on G. The* index distance *of any two vertices $u, v \in V$ is defined as $d_\prec(u, v) := |\mathrm{idx}_\prec(u) - \mathrm{idx}_\prec(v)|$, where $\mathrm{idx}_\prec(v)$ is the index of v in $V_\prec$. The* mean index distance (MID) *of G with respect to $\prec$ is*

$$\mathrm{MID}_\prec(G) := \frac{1}{|E|} \sum_{(u,v) \in E} d_\prec(u, v).$$

In the limit, legibility is optimised when $d_\prec(u,v) = 1$ for every edge $(u,v) \in E$. The following definition captures the extent to which this ideal is achieved.

Definition 3 (Adjacency rate). *Let $G = (V,E)$ be a DAG, and let $\prec$ be a topological order on G. The* adjacency rate *of G with respect to $\prec$ is defined as*

$$\mathrm{AdjRate}_\prec(G) \coloneqq \frac{|\{(u,v) \in E : d_\prec(u,v) = 1\}|}{|E|}.$$

5 Pebbling Games

To compare the working memory used while checking different linearisations of the same proof, we need a way to track which intermediate expressions must remain available at each step. The formalism of *pebbling games* played on a DAG provides a natural abstraction for this. Intuitively, placing a pebble on a vertex v indicates that the expression $\varepsilon(v)$ that it derives is currently stored in working memory. Only after this expression is no longer required should v be "unpebbled" and released from memory. From this formalism, we can derive metrics that model *peak* and *cumulative* memory demand across the whole proof.

Definition 4 (Pebbling game, adapted from Nordström [14]). *Let $G = (V,E)$ be a DAG. The* pebbling game *on G is a one-player game where each legal move is a pebbling or unpebbling of exactly one vertex as follows:*

1. *Any unpebbled source may be pebbled.*
2. *Any unpebbled vertex whose predecessors are pebbled may be pebbled.*
3. *Any pebbled vertex may be unpebbled.*

To begin, the DAG is entirely unpebbled, and the goal of the game is to pebble all sinks. Formally, a pebbling strategy *for G is a sequence $P = (P_0, P_1, \ldots, P_n)$ of* pebble configurations *$P_t \in \mathcal{P}(V)$ such that $P_0 = \emptyset$ and $P_n = S$, where $S \subseteq V$ is the set of all sinks; and such that each P_t follows from P_{t-1} according to a legal move. The* cost *of a configuration P_t is $|P_t|$, the number of active pebbles on the DAG.*

When G is a proof DAG, the goal is to pebble the unique sink, i.e., its conclusion. Consider Figure 1. Before each application of modus ponens, its two premises must be pebbled; the game ends once χ is pebbled. Standard pebbling, however, leaves the timing of unpebbling to the player. For proof linearisation this is an unnecessary degree of freedom: once an order is fixed, we want memory usage to be determined by which expressions still have future uses. This motivates the following restricted variant.

Definition 5 (One-shot pebbling game with implicit unpebbling). *Definition 4 is augmented with the following rules to form a variant of the pebbling game.*

1. *(*One-shot*.) Each vertex can be pebbled at most once.*

2. *(*Implicit unpebbling.*) If, after a move, all successors of a vertex v have at some point been pebbled, then v is automatically unpebbled. This does not count as a move.*

The one-shot rule stipulates that each inference should be evaluated at most once in a proof. We view the strategic re-evaluation of past inferences as a separate theoretical problem and prohibit it here. (If a proof inefficiently derives the same mathematical expression via two distinct inferences, e.g., $\varepsilon(v_1) = \varepsilon(v_2)$, our model still permits both to be pebbled.) The implicit unpebbling rule captures the idea that premises do not (in general) need to be held in memory together with the conclusion once it is established. By analogy, this is like erasing premises on the chalkboard to make space for the conclusion.

There are caveats to using pebbling games for measuring cognitive load. First, in the chalkboard analogy, they measure demand on the *audience's* memory rather than the presenter's—the presenter has decided in advance how long each mathematical expression must stay on the board, relieving the audience of having to perform explicit memory management. In other words, pebbling games use "live mathematical expressions" only as a proxy for working memory load. Second, in cases where the optimal space or cumulative cost of a proof is very high, it is debatable whether readability can be noticeably improved at all. Nevertheless, we speculate that optimising proofs with respect to these metrics improves dataset quality for neural and LLM-based methods.

Given a pebbling strategy, two metrics are of particular interest: the *maximum*, and the *cumulative* cost across all configurations.

Definition 6 (Pebbling space and cumulative pebbling cost). *Let $G = (V, E)$ be a DAG, and let P be a pebbling strategy on G. The* pebbling space *of P, and the* cumulative pebbling cost *of P, are defined respectively as*

$$\mathrm{sp}(P) \coloneqq \max_t |P_t| \qquad \textit{and} \qquad \mathrm{cc}(P) \coloneqq \sum_t |P_t|.$$

Tables 2 and 3 demonstrate calculations of space and cumulative cost for the two DAGs in Figure 2. Two pebbling strategies with different pebbling spaces are presented for Figure 2 (a). For Figure 2 (b), two strategies are presented that require the same space but have different cumulative costs. Thus optimising for space and optimising for cumulative cost can lead to different strategies.

Fig. 2. Two simple DAGs. DAG (a) has the same structure as that of `mp2` in Figure 1.

Table 2. Two possible pebbling strategies for Figure 2 (a). Strategy 1 requires space 3 whereas strategy 2 requires space 2.

Step	$\prec_1$	Configuration	Cost	$\prec_2$	Configuration	Cost
1	A	$\{A\}$	1	B	$\{B\}$	1
2	B	$\{A, B\}$	2	C	$\{B, C\}$	2
3	C	$\{A, B, C\}$	3	D	$\{D\}$	1
4	D	$\{A, D\}$	2	A	$\{D, A\}$	2
5	E	$\{E\}$	1	E	$\{E\}$	1

Table 3. Two possible pebbling strategies for Figure 2 (b). Both require space 2, but strategy 1 has cumulative cost 6 while strategy 2 has cumulative cost 5.

Step	$\prec_1$	Configuration	Cost	$\prec_2$	Configuration	Cost
1	A	$\{A\}$	1	B	$\{B\}$	1
2	B	$\{A, B\}$	2	C	$\{C\}$	1
3	C	$\{A, C\}$	2	A	$\{C, A\}$	2
4	D	$\{D\}$	1	D	$\{D\}$	1

Our proof optimisation algorithms do not compute pebbling strategies directly, but rather topological orders. Each topological order on a DAG, however, induces the obvious pebbling strategy as follows.

Definition 7 (Canonical pebbling strategy). *Let $G = (V, E)$ be a DAG, and let $\prec$ be a topological order on G such that $V_\prec = (v_1, v_2, \ldots, v_n)$. The* canonical pebbling strategy *$P_\prec$ is the unique pebbling strategy constructed as follows:*

- $P_0 \coloneqq \emptyset$
- $P_t \coloneqq \mathrm{UnPeb}\,(P_{t-1} \cup \{v_t\})$ *for* $t = 1, 2, \ldots, n$,

where $\mathrm{UnPeb} : \mathcal{P}(V) \to \mathcal{P}(V)$ *unpebbles vertices according to the implicit unpebbling rule in Definition 5.*

6 Proof Reordering Algorithms

Given a proof DAG, our goal is to find a pebbling strategy whose space and cumulative cost is minimised. In this section we present Algorithm 1, a bottom-up depth-first search (DFS) approach adapted from Fellner and Paleo [3]. During search, the order in which the predecessors of each vertex are visited is determined by an *ordering policy.*

Definition 8 (Ordering policy). *Let $G = (V, E, \varepsilon, \lambda)$ be a proof DAG. An* ordering policy *π is a rule that assigns to each vertex $v \in V$ a total preorder $\geq_v^\pi$ or total order $>_v^\pi$ on $\mathrm{pred}(v)$. For distinct $u, w \in \mathrm{pred}(v)$, we define the following policies:*

Ordering policy π ***(strict)***	***Condition for*** $u >_v^\pi w$
Original	$\lambda(u,v) < \lambda(w,v)$
Random	*Holds with probability* $1/2$
Ordering policy π ***(non-strict)***	***Condition for*** $u \geq_v^\pi w$
SU	$W_{\mathrm{SU}}(u) \geq W_{\mathrm{SU}}(w)$
CumulSU	$W_{\mathrm{CumulSU}}(u) \geq W_{\mathrm{CumulSU}}(w)$
CumulPred	$W_{\mathrm{CumulPred}}(u) \geq W_{\mathrm{CumulPred}}(w)$
NumSucc	$\lvert\mathrm{succ}(u)\rvert \geq \lvert\mathrm{succ}(w)\rvert$
LastSucc	$W_{\mathrm{LastSucc}}(u) \geq W_{\mathrm{LastSucc}}(w)$

where the weights W_π *are defined as follows: for all non-source vertices* $v \in V$*,*

$$W_{\mathrm{SU}}(v) \coloneqq \max_{1\leq i\leq k} \left(W_{\mathrm{SU}}(u_{(i)}) + i - 1\right)$$

$$W_{\mathrm{CumulSU}}(v) \coloneqq W_{\mathrm{SU}}(v) + \sum_{u\in\mathrm{pred}(v)} W_{\mathrm{CumulSU}}(u)$$

$$W_{\mathrm{CumulPred}}(v) \coloneqq 1 + \sum_{u\in\mathrm{pred}(v)} W_{\mathrm{CumulPred}}(u)$$

$$W_{\mathrm{LastSucc}}(v) \coloneqq \left|\{(u,v) \in E : \lambda(u,v) > \lambda(u,w) \text{ for all } w \in \mathrm{succ}(u)\}\right| .$$

Here, $(u_{(1)}, \ldots, u_{(k)})$ *is* $\mathrm{pred}(v)$ *sorted so that* $W_{\mathrm{SU}}(u_{(1)}) \geq \cdots \geq W_{\mathrm{SU}}(u_{(k)})$*. For* $\pi = \mathrm{LastSucc}$*, all source weights are set to zero; for all other policies, source weights are set to one.*

The initialism SU stands for *Sethi-Ullman* numbering, originally developed for efficient evaluation of binary computation trees [21] and later generalised by Appel and Supowit [2]. The LastSucc heuristic is taken directly from Fellner and Paleo [3], which was their best-performing heuristic. We include CumulSU and CumulPred as simple adaptations meant to target cumulative cost. These ordering policies are driven by the intuition that it is generally better to evaluate heavier branches before lighter ones, thereby minimising the duration that intermediate results must remain in memory. In Section 8 we present examples that support this intuition.

Given a sequence $(\pi_1, \ldots, \pi_n)$ of ordering policies, we can combine them lexicographically into a single policy such that later policies break ties in earlier ones. Algorithm 1 requires all input orders to be strict, which we achieve by appending the Original policy as a final tie-breaker to all non-strict policies.

7 Experiments

Our experiments are run on Metamath's `set.mm` library. Rather than parsing proofs directly, we modify the `mmverify.py` Python verifier [10, 27] so that at each stack operation the context is logged to JSON. Each proof is then converted into a DAG, on which we run Algorithm 1 and calculate metrics. Figure 3 presents some basic statistics about `set.mm`.

Algorithm 1 Bottom-up DFS linearisation of a DAG

Require: A DAG $G = (V, E)$ with sink s. A family $(>^{\pi}_{v})_{v \in V}$ of predecessor orders.
Ensure: A list *result*, an ordering of V.
1: $\mathit{visited} \leftarrow \emptyset$
2: $\mathit{result} \leftarrow [\ ]$
3: **function** DFS(v)
4:     **if** $v \notin \mathit{visited}$ **then**
5:         $\mathit{visited} \leftarrow \mathit{visited} \cup \{v\}$
6:         **for all** $u \in \mathrm{pred}(v)$ sorted by $>^{\pi}_{v}$ descending **do**
7:             DFS(u)
8:         **end for**
9:         append v to *result*
10:     **end if**
11: **end function**
12: DFS(s)
13: **return** *result*

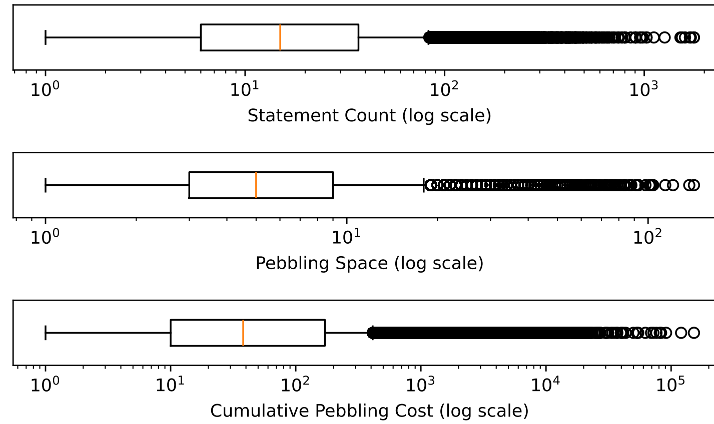

Fig. 3. Box plots of statement counts, pebbling spaces, and cumulative pebbling costs of the original `set.mm` proofs (log scale). In all three cases, the distributions are heavily right-skewed: the median statement count is 15, the median space is 5, and the median cumulative cost is 38.

7.1 Methodology

For each proof DAG G in `set.mm`, we use Algorithm 1 to compute a topological order $\prec$ on G for each of the ordering policies below:

- Original
- Random[2]
- (SU, Original)
- (SU, CumulSU, Original)
- (CumulPred, Original)
- (NumSucc, Original)

The Random ordering policy serves as a baseline for comparison: intuitively, it is reasonable to expect it to perform similarly to Metamath's original ordering. This is likely because the premise orderings $\lambda(e)$ of each edge $e \in E$ in a proof DAG are effectively arbitrary. By reordering premises in `set.mm`'s theorem definitions, one could theoretically make Original coincide with other policies on certain proofs.

Once a topological order $\prec$ is obtained for each policy, we calculate the pebbling space $\mathrm{sp}(P_\prec)$ and cumulative pebbling cost $\mathrm{cc}(P_\prec)$ of the canonical pebbling strategy $P_\prec$. We also calculate the mean index distance $\mathrm{MID}_\prec(G)$ and adjacency rate $\mathrm{AdjRate}_\prec(G)$. For space and cumulative cost, we calculate the relative improvement as

$$\Delta_{\mathrm{metric}} \coloneqq \frac{\mathrm{metric}(P_{\mathrm{Original}}) - \mathrm{metric}(P_\prec)}{\mathrm{metric}(P_{\mathrm{Original}})}.$$

Algorithm 1 may produce strictly worse pebbling strategies; for each ordering policy we calculate the percentage of proofs in `set.mm` where this occurs. At the other extreme, we evaluate the percentage of proofs where space attains the theoretical lower bound—that is, where $\mathrm{sp}(P_\prec) = \max_{v \in V} |\mathrm{pred}(v)|$. Finally, we record the execution time of Algorithm 1 for each ordering policy across all proofs in `set.mm`. Experiments were run with Python 3.14.3 on an Apple M5 MacBook Pro with 16GB unified memory, running macOS 26.4 (Tahoe). The code for our experiments is available online [11].

Sethi-Ullman numbering and its generalisation by Appel and Supowit are optimal for trees with respect to pebbling space [2, 21]. We therefore separate treelike and non-treelike proofs in our results. Out of a total of 47,248 proofs, 29,338 of them are treelike.

7.2 Results

Table 4 compares the relative improvement in space and cumulative cost of each reordering policy compared to the original Metamath ordering, averaged across `set.mm`. Despite CumulPred's relative simplicity, CumulPred and CumulSU are very close in performance with respect to cumulative cost. This suggests that

[2] For reproducibility, the Random policy uses a fixed pseudorandom seed of 2026.

simple heuristics for recursive branch heaviness are effective for reducing cumulative cost. For trees, (NumSucc, Original) defers to Original since every vertex has at most one successor. Overall, a mixture of SU and either of CumulSU and CumulPred seems to be the best approach for optimising both space and cumulative cost.

Table 4. Average pebbling metrics for each reordering policy, broken down by treelike (†) and non-treelike (*) proofs.

Ordering policy	$\Delta^\dagger_{sp}$ (%)		$\Delta^\dagger_{cc}$ (%)		Δ^*_{sp} (%)		Δ^*_{cc} (%)	
	Median	Mean	Median	Mean	Median	Mean	Median	Mean
Random	0.00	-1.03	0.00	-0.62	0.00	0.04	2.96	0.74
(SU, Original)	0.00	**12.48**	0.00	11.10	22.86	23.12	21.99	23.12
(CumulSU, Original)	0.00	11.80	**9.09**	14.32	22.22	22.31	25.00	25.76
(CumulPred, Original)	0.00	11.53	**9.09**	**14.36**	22.22	22.09	**25.14**	**25.81**
(NumSucc, Original)	0.00	0.00	0.00	0.00	0.00	0.13	0.00	0.33
(LastSucc, Original)	0.00	10.16	1.64	11.42	14.29	14.23	14.29	15.61
(SU, CumulSU, Original)	0.00	**12.48**	8.33	14.05	**24.14**	**23.58**	24.14	24.99
(SU, CumulPred, Original)	0.00	**12.48**	8.33	14.05	24.00	23.56	24.14	24.98
(SU, NumSucc, Original)	0.00	**12.48**	0.00	11.10	22.86	23.13	22.02	23.16

Table 5 breaks down the percentage of proofs across `set.mm` where the heuristic policies perform worse than the original ordering. Recall that SU is optimal for treelike proofs. For non-treelike proofs, Figure 4 (d) presents a representative example showcasing an inherent limitation of DFS. Note also that AdjRate is invariant for trees—this is because the adjacency rate of any (non-source) vertex's n incoming edges is exactly $1/n$ regardless of the order in which DFS visits its predecessors.

Table 5. Percentage of proofs for which each policy performs worse than Original, broken down by treelike (†) and non-treelike (*) proofs.

Ordering policy	sp (%)		cc (%)		MID (%)		AdjRate (%)	
	$\dagger$	$*$	$\dagger$	$*$	$\dagger$	$*$	$\dagger$	$*$
Random	20.85	34.33	29.87	42.84	29.87	41.18	0.00	22.20
(SU, Original)	**0.00**	**2.04**	0.62	4.14	0.62	5.25	0.00	18.41
(CumulSU, Original)	0.80	3.75	0.03	3.81	0.03	4.58	0.00	23.04
(CumulPred, Original)	1.06	3.80	**0.00**	**3.66**	**0.00**	**4.41**	0.00	23.37
(NumSucc, Original)	**0.00**	2.92	**0.00**	16.97	**0.00**	48.64	0.00	**1.69**
(LastSucc, Original)	1.28	8.97	2.22	11.88	2.22	11.92	0.00	15.83
(SU, CumulSU, Original)	**0.00**	2.52	0.46	4.31	0.46	5.28	0.00	21.76
(SU, CumulPred, Original)	**0.00**	2.49	0.46	4.27	0.46	5.26	0.00	21.74
(SU, NumSucc, Original)	**0.00**	**2.04**	0.62	4.26	0.62	8.21	0.00	18.37

Table 6 reports the percentage of proofs where the theoretical lower bound of pebbling space is attained. Since SU is optimal with respect to space, it acts as a baseline for comparison for trees; the wide margin of improvement over Original (>30%) highlights the inherent inefficiency of the native Metamath format. For non-treelike proofs, the best-performing policies achieve the lower bound almost six times as often as the original ordering.

Table 6. Percentage of proofs where space attains the theoretical lower bound.

Ordering policy	Treelike (%)	Non-treelike (%)
Original	54.57	3.74
Random	56.93	4.28
(SU, Original)	**85.98**	20.90
(CumulSU, Original)	82.75	15.85
(CumulPred, Original)	81.72	15.38
(NumSucc, Original)	54.57	3.80
(LastSucc, Original)	78.50	12.88
(SU, CumulSU, Original)	**85.98**	21.04
(SU, CumulPred, Original)	**85.98**	**21.05**
(SU, NumSucc, Original)	**85.98**	20.91

Finally, Table 7 ranks the ordering policies by their execution time. Regardless of the policy, execution time across the entire `set.mm` library is less than four seconds. Even for very large proofs, step reordering is virtually instant.

Table 7. Execution times of each ordering policy across `set.mm`.

Ordering policy	Time per Statement (ns)	Time per Edge (ns)
(SU, NumSucc, Original)	2404.30	2139.58
(SU, CumulSU, Original)	2261.31	2012.33
(SU, CumulPred, Original)	1994.49	1774.89
(LastSucc, Original)	1913.14	1702.49
(CumulSU, Original)	1828.68	1627.33
(SU, Original)	1472.77	1310.61
(CumulPred, Original)	1294.81	1152.25
(NumSucc, Original)	1094.66	974.13
Random	801.13	712.92

8 Case Studies

To show the effect of the heuristics, we begin with the previously mentioned larger example `prmunb`, and then turn to smaller proofs whose DAGs make the

structural reason for improvements easy to see. In the proof of `prmunb`, the assumption `nnnn0` is introduced at the first step of the proof even though it is first used only in step 48:

```
STEP   PREMISES   LABEL   EXPRESSION
1                 nnnn0   |- ( N e. NN -> N e. NN0 )
...
48     1,47       syl     |- ( N e. NN -> E. p e. Prime N < p )
```

Applying the ordering policy (SU, CumulSU, Original) delays `nnnn0` until step 47, reducing the space from 9 to 6 and cumulative cost from 224 to 137. In the rest of this section we present smaller examples where improvements are visually apparent.

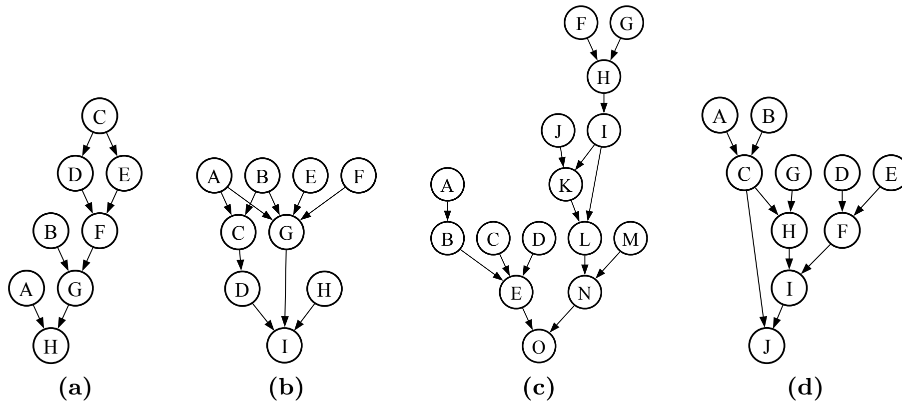

Fig. 4. Proof DAGs for theorems `eldisjs5`, `subcan`, `neifval`, and `knoppndvlem3`, with vertices relabelled alphabetically according to the original Metamath step order.

Consider Figure 4 (a). Inference A is executed 7 steps before it is required by inference H. Inference B is also executed unnecessarily early. The policy (SU, CumulSU, Original) instead computes the ordering (C, D, E, F, B, G, A, H), executing inferences B and A just as they are required, i.e., the edges (A, H) and (B, G) now have unit index distance. Pebbling space is reduced from 4 to the theoretical lower bound of 2, and cumulative cost is reduced from 20 to 12.

Adjacency rate and pebbling space are not always positively correlated, however. In Figure 4 (b), the policy (SU, CumulSU, Original) computes the ordering $(A, B, E, F, G, C, D, H, I)$. The original ordering has four edges with unit index distance: $(B, C), (C, D), (F, G)$, and (H, I). The new ordering, however, has only three: $(F, G), (C, D)$, and (H, I). Overall, the MID worsens from 2.6 to 2.7. Despite this, space improves from 5 to 4 and cumulative cost from 24 to 21. Thus there are cases where tradeoffs between memory consumption and index distance may occur.

Recall from Table 3 that cumulative cost can improve even when space is unchanged. Figure 4 (c) shows a more dramatic example where space actually worsens. The policy (SU, CumulSU, Original) computes

$$(\underbrace{F, G, H, I, J, K, L, M, N}_{\text{right branch}}, \underbrace{A, B, C, D, E}_{\text{left branch}}, O).$$

This is identical to the original ordering except that the right branch ending in N is traversed before the left branch ending in E. In this case, cumulative cost improves from 31 to 27 even though space worsens from 3 to 4. This is possible because in the original ordering, $\varepsilon(E)$ must be held in memory while the larger branch is traversed. In the new ordering, $\varepsilon(N)$ is held in memory for far fewer steps, although it does force the bottleneck configuration $\{N, B, C, D\}$. Arguably, a brief spike in configuration cost could be a small price to pay for a significantly reduced cumulative cost.

Finally, Figure 4 (d) presents a rare example (4.31%) where the policy (SU, CumulSU, Original) computes a worse ordering $(A, B, C, G, H, D, E, F, I, J)$. Space is inflated from 3 to 4, and cumulative cost from 20 to 21. This is a limitation of DFS: predecessors of different vertices cannot be interleaved out of order. Thus the edge (G, H) must be traversed before any predecessors of F can be visited. In this case, the bottleneck configuration for space is $\{C, G, D, E\}$.

9 Future Work

Future directions include adapting proof DAGs to account for backwards reasoning, which could possibly be modelled using *black-white* pebbling games. This may enable adapting our methods for application in other mathematical libraries such as Mizar [5], Lean's Mathlib [23], and Isabelle/HOL's AFP [22]. To capture variations in the complexity of mathematical expressions, future work could explore using *weighted* pebbling games and assign higher weights to expressions with more symbols or deeper nested structure.

10 Conclusion

Using pebbling games, we studied how proof step reordering in Metamath influences the memory consumption of proof verification. Our heuristic DFS algorithms improve peak and cumulative memory consumption on many proofs in Metamath's ZFC set theory library (`set.mm`) while preserving proof length. Case studies illustrate how these improvements arise and where they may fail, e.g., as an inherent limitation of DFS. Overall, our results support proof reordering as a practical optimisation target for legibility proxy metrics, and motivate future evaluation on other formal libraries beyond Metamath.

Acknowledgments. This research was supported by an Australian Government Research Training Program Scholarship and the Renaissance Philanthropy grant Deeper.

References

1. Ahuja, R., Avigad, J., Tetali, P., Welleck, S.: Improver: Agent-based automated proof optimization. In: The Thirteenth International Conference on Learning Representations (2025), https://openreview.net/forum?id=dWsdJAXjQD
2. Appel, A.W., Supowit, K.J.: Generalizations of the Sethi-Ullman algorithm for register allocation. Software: Practice and Experience **17**(6), 417–421 (1987). https://doi.org/10.1002/spe.4380170607
3. Fellner, A., Woltzenlogel Paleo, B.: Greedy pebbling for proof space compression. International Journal on Software Tools for Technology Transfer **21**, 71–86 (2017). https://doi.org/10.1007/s10009-017-0459-0
4. Gallardo, P., Raissi, M., Zhang, K., Murthy, S.: Agentic lean auformalization (ALA): An LLM collaborative approach to autoformalization in LEAN. In: NeurIPS 2025 Workshop on Evaluating the Evolving LLM Lifecycle: Benchmarks, Emergent Abilities, and Scaling (2025), https://openreview.net/forum?id=DNunBkhEUx
5. Grabowski, A., Kornilowicz, A., Naumowicz, A.: Mizar in a nutshell. Journal of Formalized Reasoning **3**(2), 153–245 (2010). https://doi.org/10.6092/issn.1972-5787/1980
6. Gu, A., Piotrowski, B., Gloeckle, F., Yang, K., Markosyan, A.H.: ProofOptimizer: Training language models to simplify proofs without human demonstrations (2026). https://doi.org/10.48550/arXiv.2510.15700
7. Kaliszyk, C., Urban, J.: Learning-assisted theorem proving with millions of lemmas. Journal of Symbolic Computation **69**, 109–128 (2015). https://doi.org/10.1016/j.jsc.2014.09.032
8. Krajíček, J.: Proof Complexity. Encyclopedia of Mathematics and its Applications, Cambridge University Press (2019)
9. Kripner, M., Šustr, M., Straka, M.: LeanTree: Accelerating white-box proof search with factorized states in Lean 4 (2025), https://arxiv.org/abs/2507.14722
10. Levien, R., Wheeler, D.A., Lindsay, J.: Metamath proof structure exporter to JSON (2026), https://github.com/jnlindsay/mmverify.py-json-exporter/releases/tag/v1.0
11. Lindsay, J., Kaliszyk, C., Rizkallah, C.: Optimising Metamath Proofs for Human Working Memory — Accompanying Code (2026). https://doi.org/10.26188/32018343.v1
12. Megill, N.D., Wheeler, D.A.: Metamath: A Computer Language for Mathematical Proofs. Lulu Press, Morrisville, North Carolina (2019), http://us.metamath.org/downloads/metamath.pdf
13. de Moura, L., Ullrich, S.: The Lean 4 theorem prover and programming language. In: Automated Deduction – CADE 28. pp. 625–635. Lecture Notes in Computer Science, Springer International Publishing (2021). https://doi.org/10.1007/978-3-030-79876-5_37
14. Nordstrom, J.: Pebble games, proof complexity, and time-space trade-offs. Logical Methods in Computer Science **9**(3) (2013). https://doi.org/10.2168/LMCS-9(3:15)2013
15. Pąk, K.: Automated improving of proof legibility in the Mizar system. In: Intelligent Computer Mathematics. pp. 373–387. Springer International Publishing, Cham (2014)
16. Pąk, K.: The algorithms for improving and reorganizing natural deduction proofs. Studies in Logic, Grammar and Rhetoric **22**(35) (2010)

17. Pąk, K.: Improving legibility of formal proofs based on the Close Reference Principle is NP-Hard. Journal of Automated Reasoning **55**(3), 295–306 (2015). https://doi.org/10.1007/s10817-015-9337-1
18. Pąk, K., Schubert, A.: The impact of proof steps sequence on proof readability – experimental setting (2016), https://cicm-conference.org/2016/ceur-ws/W47.pdf, submitted to the 9th Conference on Intelligent Computer Mathematics. Work in Progress.
19. Rahul, S.P., Necula, G.C.: Proof Optimization Using Lemma Extraction (2001), https://www2.eecs.berkeley.edu/Pubs/TechRpts/2001/Archive/CSD-01-1143.pdf
20. Sethi, R.: Complete register allocation problems. In: Proceedings of the Fifth Annual ACM Symposium on Theory of Computing. pp. 182–195. STOC '73, ACM Press, Austin, Texas, United States (1973). https://doi.org/10.1145/800125.804049
21. Sethi, R., Ullman, J.D.: The Generation of Optimal Code for Arithmetic Expressions. J. ACM **17**(4), 715–728 (1970). https://doi.org/10.1145/321607.321620
22. The Archive of Formal Proofs contributors: The Archive of Formal Proofs (2004), https://isa-afp.org/, accessed: 2026-06-21
23. The mathlib Community: The Lean mathematical library. In: Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs. pp. 367–381. CPP 2020, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3372885.3373824
24. The Rocq Development Team: The Rocq Prover (2025). https://doi.org/10.5281/zenodo.15149629
25. Vyskočil, J., Stanovský, D., Urban, J.: Automated Proof Compression by Invention of New Definitions. In: Logic for Programming, Artificial Intelligence, and Reasoning, Lecture Notes in Computer Science, vol. 6355, pp. 447–462. Springer Berlin Heidelberg, Berlin, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17511-4_25
26. Wang, H., et al.: Kimina-prover preview: Towards large formal reasoning models with reinforcement learning (2025), http://arxiv.org/abs/2504.11354
27. Wheeler, D.A., Levien, R.: mmverify.py: Metamath verifier in Python. https://github.com/david-a-wheeler/mmverify.py, accessed: 2026-04-14
28. Xin, H., et al.: Deepseek-prover-v1.5: Harnessing proof assistant feedback for reinforcement learning and monte-carlo tree search. In: The Thirteenth International Conference on Learning Representations (2025), https://openreview.net/forum?id=I4YAIwrsXa
29. Zhou, J., Wu, Y., Li, Q., Grosse, R.: Refactor: Learning to extract theorems from proofs. In: Kim, B., Yue, Y., Chaudhuri, S., Fragkiadaki, K., Khan, M., Sun, Y. (eds.) International Conference on Learning Representations. vol. 2024, pp. 40601–40621 (2024)