A Regex Minimization Benchmark: A PSPACE-Complete Challenge for Language Models

Hyundong Jin **Joonghyuk Hahn** **Yo-Sub Han**⋆
Yonsei University, Seoul, Republic of Korea
{tuzi04, greghahn, emmous}@yonsei.ac.kr

Abstract

Language models (LMs) have shown impressive reasoning capabilities across various domains. A fundamental question is the extent of their reasoning power. While recent studies show that LMs can solve NP-complete problems, their ability to handle PSPACE-complete problems remains underexplored. We investigate regex minimization as a PSPACE-complete challenge for LMs to address this issue. Regexes, formal expressions for regular languages widely used in NLP, software engineering (SE), and programming language (PL), are supported by several efficient tools for their manipulation grounded in theoretical backgrounds. Inspired by this, we introduce the first benchmark for regex minimization containing over a million regexes paired with their minimal equivalents. Through extensive experiments with two LMs trained on our dataset and five open-source large language models (LLMs), we analyze how well LMs perform on PSPACE-complete problems, highlighting their capabilities of generalization and limitations in reasoning. To the best of our knowledge, this is the first study to systematically evaluate LM reasoning in regex minimization and establish a foundation for solving PSPACE-complete problems with LMs. Our code is available at https://github.com/hyundong98/RegexPSPACE.

1 Introduction

Language models (LMs) have demonstrated reasoning capabilities across various domains. However, the extent of their computational reasoning remains an open question. While recent studies suggested that LMs can solve NP-complete problems (Fan et al., 2024a; Bampis et al., 2024), NP-completeness is not the upper bound of computational reasoning. PSPACE-complete problems, which require polynomial space, are strictly harder than NP-complete problems unless PSPACE = NP, offering a more rigorous evaluation of reasoning capability. Despite the rapid advancements of LMs, their ability to handle PSPACE-complete problems remains underexplored.

Addressing this gap, we introduce a new benchmark of regular expression (regex) minimization, a typical PSPACE-complete problem, to evaluate the reasoning capabilities of LMs. Regex minimization, the task of reducing a regex to its smallest equivalent form, is practically essential in domains like natural language processing (NLP), software engineering (SE), and programming languages (PL) (Davis et al., 2018; Shen et al., 2018; Li et al., 2022; Siddiq et al., 2024) for optimizing search engines (Thompson, 1968), lexical analysis (DeRemer, 1974; Sproat, 2000; Sposito et al., 2021), and data processing pipelines. However, regex minimization lacks an efficient algorithm, requiring an exhaustive search over all possible expressions (Stockmeyer and Meyer, 1973; Hunt, 1973). Unlike finite automata minimization, which has a polynomial-time solution (Hopcroft, 1971; Sipser, 1997; Hopcroft et al., 2007), regex minimization remains intractable under conventional algorithmic paradigms. This intractability and vast combinatorial search space make regex minimization a strong candidate for evaluating how well LMs can reason on PSPACE-complete challenges.

We construct the first large-scale benchmark dataset for regex minimization, consisting of over a million regexes paired with their minimal equivalents. Through extensive experiments, we compare LMs trained on our dataset and pretrained large language models (LLMs) to examine how different models approach the task. We further analyze whether LMs properly handle the inherent reasoning challenges of regex minimization. Our results show that while LLMs exhibit stronger generalization, they often fail to apply simple transformation

⋆ Corresponding author.

Proceedings of the 19th Conference of the European Chapter of the Association for Computational Linguistics
Volume 1: Long Papers, pages 8020–8048

rules in regex minimization. Our findings highlight the need for explicit reasoning mechanisms to enhance LMs' ability to handle PSPACE-complete problems, establishing a foundation for future research on evaluating the reasoning ability of LLMs.

2 Related Works

2.1 Real-world Applications of Regexes

Among the many PSPACE-complete problems, we focus on regex-related tasks for several reasons. Most importantly, regexes have wide-ranging real-world applications, making the tasks inherently important. They are practically relevant in natural language processing, software engineering, and programming languages (Davis et al., 2018; Shen et al., 2018; Li et al., 2022; Siddiq et al., 2024). In particular, finding concise and safe regex representations has always been essential in optimizing search engines (Thompson, 1968). Because of their prevalence in search, preprocessing, and other applications, most LLMs have already been exposed to regexes during pretraining, enabling regex tasks to serve as natural benchmarks without requiring additional fine-tuning.

2.2 Rule-based Regex Simplifications

Prior to exploring LLM capabilities, it is essential to consider traditional algorithmic approaches to regex reduction. To the best of our knowledge, no prior work has specifically addressed regex minimization. Instead, several studies have proposed simplification rules under their own objectives. Ellul et al. (2004); Lee and Shallit (2005); Kahrs and Runciman (2022); Gruber and Gulan (2010) simplified regexes by reducing the number of nodes in the expression tree. Brüggemann-Klein (1993) introduced star normal form, and Gruber and Gulan (2010) proposed strong star normal form as a basis for further simplification. Kahrs and Runciman (2022) used substring-level information to lift redundant alternatives or sequence items. Relatedly, automata-based approaches aim for concise regexes. Han and Wood (2005) proposed expression automata and studied determinism and minimization for that representation, and Han and Wood (2007) proposed structural decompositions for state-elimination-based automata-to-regex conversion to mitigate expression blow-up. Overall, these methods are complementary but do not guarantee globally minimal regexes.

2.3 Benchmarks for Evaluating Computational Power of LLM

Benchmarks designed to evaluate the capabilities of LLMs continue to be proposed. Fan et al. (2024a) introduced NPHardEval, a benchmark for evaluating LLMs on NP-hard problems, which includes P, NP-complete, and NP-hard problems. As a follow-up, Fan et al. (2024b) proposed NPHardEval4V, a benchmark designed to assess the reasoning capabilities of multimodal LLMs. This benchmark serves as an image-based extension of the previous work. Additionally, Qi et al. (2024) introduced SCYLLA, a benchmark incorporating problems with various time complexities, aiming to demonstrate that LLMs exhibit generalizability beyond mere memorization. Our Regex Minimization Task takes a step further by evaluating the reasoning ability of language models on PSPACE-complete problems and providing experimental settings to assess their generalizability.

3 Preliminary

3.1 Regular Expressions

Regexes define regular languages, the most fundamental formal languages formed by sets of strings over a finite alphabet Σ. While the classical definition relies on concatenation, union, and Kleene star, we adopt the extended setting of (Kahrs and Runciman, 2022) which adds the option operator. Using the notation from Gruber and Gulan (2010), our expressions take the form $\epsilon, s, x + y, x \cdot y, x^*, x^?$, where $s \in \Sigma$, and x, y are any finite expressions. For an expression x, the corresponding language $L(x)$ is defined as follows:

- $\emptyset = \{\}$ (An empty language)
- $L(\epsilon) = \{\epsilon\}$ (An empty string)
- $L(s) = \{s\}\ \forall s \in \Sigma$ (A character)
- $L(x + y) = L(x) \cup L(y)$ (Union)
- $L(x \cdot y) = \{vw \mid \forall v \in L(x), \forall w \in L(y)\}$ (Concatenation)
- $L(x^?) = L(x) \cup L(\epsilon)$ (Option)
- $L(x^*) = \{x_1 \cdots x_n \mid \forall n \in \mathbb{N}_0,$ $\forall x_1, \ldots, x_n \in L(x)\}$ (Kleene Star)

By definition, Σ^* denotes the set of all strings on the alphabet Σ including ϵ. Since regex is defined using unary and binary operations, every regular expression has a binary expression tree.

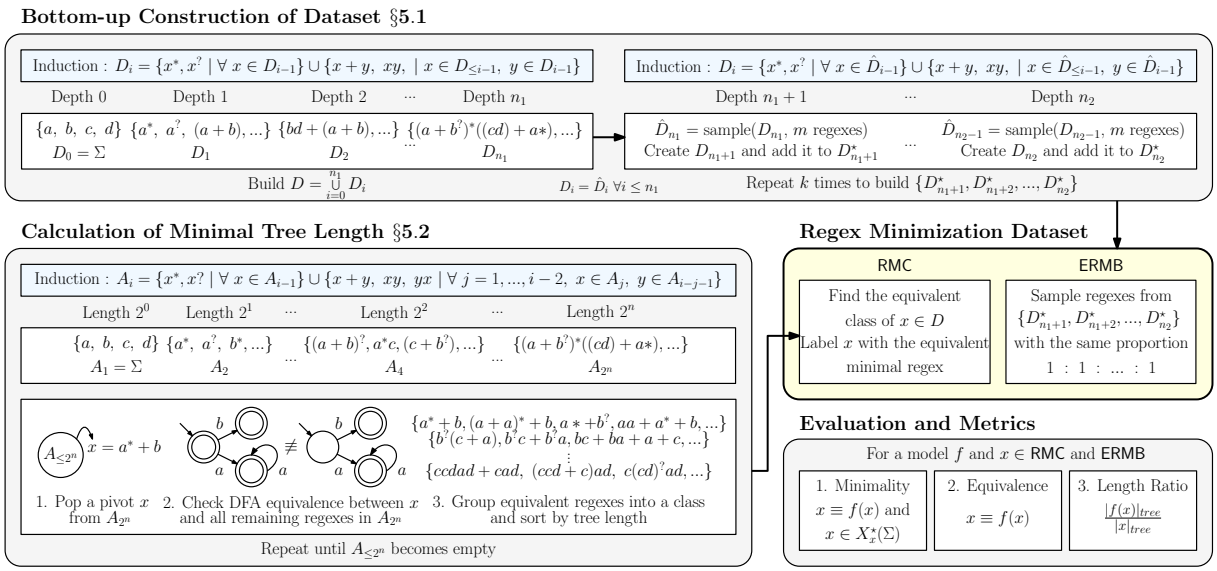


Figure 1: The overview of our dataset construction and evaluation. Our dataset consists of regex minimization corpus (RMC) and extended regex minimization benchmark (ERMB), which are constructed using a bottom-up approach over tree depth. RMC is labeled using the minimal tree length calculated in Section 4.2.

3.2 Complexity Classes

Computational complexity is defined by the time and space resources required by Turing machines. P and NP denote sets of problems solvable in polynomial time by deterministic and non-deterministic Turing machines, respectively, while PSPACE refers to problems solvable within polynomial space. By Savitch's theorem, which equates PSPACE with NPSPACE and the fact that a machine cannot access more space than the number of time steps, we have the inclusion relationship P $\subseteq$ NP $\subseteq$ PSPACE. Hardness is defined via polynomial-time reduction. A problem is NP-hard (or PSPACE-hard) if every problem in NP (or PSPACE) can be reduced to it. A problem is termed NP-complete (or PSPACE-complete) if it is both in the class and hard for that class. Consequently, unless NP $=$ PSPACE, PSPACE-complete problems represent a strictly harder computational tier than NP-complete ones.

3.3 Comparison between NP and PSPACE

A critical distinction between NP and PSPACE lies in the complexity of verification. While NP-complete problems allow for efficient and polynomial-time verification of a candidate solution, PSPACE-complete problems lack this asymmetry because determining the correctness of an answer often requires traversing the same enormous search space used to find it. This characteristic introduces a novel dimension to LLM evaluation by testing reasoning capabilities in scenarios where validating the result is as intractable as the problem itself. Consequently, this pushes the boundaries of self-verification and rigorous logical consistency beyond standard NP benchmarks.

3.4 Problem Formulation

For a fixed alphabet Σ, let $X(\Sigma)$ denote the set of all regexes over Σ. Since regexes are constructed using unary and binary operations, each can be represented as a binary expression tree. We define the length of a regex r, denoted by $|r|_T$, as the number of nodes in its expression tree. Throughout this paper, unless stated otherwise, the length of a regex refers to the tree length. The equivalence between two regexes x and y is represented as $x \equiv y$. Conventionally, we define minimal regexes as regexes with minimal tree lengths among the equivalent regexes. For $r \in X(\Sigma)$, the set of regexes equivalent to r is

$$X_r(\Sigma) = \{x \in X(\Sigma) \mid x \equiv r\},$$

and the subset of $X_r(\Sigma)$ with minimal regexes is

$$X^\star_r(\Sigma) = \{x \equiv r \mid |x|_T \leq |y|_T, \forall y \equiv r\}.$$

Given a regex $r \in X(\Sigma)$, our task is to find a minimal equivalent regex $x \in X^\star_r(\Sigma)$. This is equivalent to learning a function $f : X(\Sigma) \to X(\Sigma)$ such that $f(r) \in X^\star_r(\Sigma)$. For a parameterized model f_θ, the goal is to find the optimal parameter $\theta^\star$, where

$$\theta^\star = \operatorname*{argmax}_\theta \mathop{\mathbb{E}}_{x \sim X_r(\Sigma)} \big[\log P(f_\theta(x) \in X^\star_r(\Sigma))\big].$$

Algorithm 1 Construction of RMC

Require: Maximum depth n, alphabet Σ
  Initialize $D_0 = \Sigma$
  for $i = 1 \dots n$ **do**
    $D_i \leftarrow \emptyset$
    for $x \in D_{i-1}$ **do**
      $D_i \leftarrow D_i \cup \{\text{concat}(x, \text{`?'})\}$
      $D_i \leftarrow D_i \cup \{\text{concat}(x, \text{`*'})\}$
    end for
    for $x \in D_{\leq i-1}$ **do**
      for $y \in D_{i-1}$ **do**
        $D_i \leftarrow D_i \cup \{\text{concat}(x, \text{`+'}, y)\}$
        $D_i \leftarrow D_i \cup \{\text{concat}(x, y)\}$
      end for
    end for
  end for

4 Dataset Construction

We introduce a dataset for regex minimization, structured into two components: the Regex Minimization Corpus (RMC) and the Extended Regex Minimization Benchmark (ERMB). RMC is a labeled dataset of regexes and their minimal equivalents, partitioned into train, validation, and test sets. Given that verifying minimality requires an exhaustive comparison against all shorter regexes, which practically limits the size of labeled data, we also construct ERMB as a challenging out-of-distribution (OOD) benchmark containing longer, unlabeled regexes to evaluate generalization. The dataset construction involves a bottom-up approach to generate regexes followed by the computation of minimal tree lengths. The alphabet Σ is fixed at four symbols $\{a, b, c, d\}$, and Figure 1 illustrates the overall dataset construction procedure.

4.1 Bottom-up Construction of Dataset

RMC Construction. RMC is constructed using a bottom-up approach by representing regexes as binary trees. We define the initial set D_0 as Σ, and recursively build the set of regexes D_n, where n denotes the tree depth of a regex in D_n. When constructing D_n, we apply unary operations to elements in D_{n-1} and binary operations between elements from $D_{\leq n-1}$ and D_{n-1}. Binary operations are restricted to cases where the operand from $D_{\leq n-1}$ comes first in binary operations to control dataset growth. The construction process is described in Algorithm 1. Due to double-exponential growth with the depth, we limit the depth of RMC to 3. The detailed dataset construction cost analysis is in Appendix A. The dataset is partitioned into train, validation, and test sets in a ratio of 20:2:1. The test set size is relatively smaller to reduce computational costs during LLM inference. Table 1 provides dataset statistics.

Algorithm 2 Construction of ERMB

Require: Number of iterations k, sample size m, depth range $[n_1, n_2]$, alphabet Σ
  for $i = 1 \dots n_2$ **do**
    Initialize $D_i^\star \leftarrow \emptyset$
  end for
  for $t = 1 \dots k$ **do**
    Initialize $\hat{D}_0 = \Sigma$
    for $i = 1 \dots n_2$ **do**
      Initialize $D_i \leftarrow \emptyset$
      for $x \in \hat{D}_{i-1}$ **do**
        $D_i \leftarrow D_i \cup \{\text{concat}(x, \text{`?'})\}$
        $D_i \leftarrow D_i \cup \{\text{concat}(x, \text{`*'})\}$
      end for
      for $x \in \hat{D}_{\leq i-1}$ **do**
        for $y \in \hat{D}_{i-1}$ **do**
          $D_i \leftarrow D_i \cup \{\text{concat}(x, \text{`+'}, y)\}$
          $D_i \leftarrow D_i \cup \{\text{concat}(x, y)\}$
        end for
      end for
      $D_i^\star \leftarrow D_i^\star \cup D_i$
      $\hat{D}_i \leftarrow$ sample m regexes from D_i
    end for
  end for
  for $i = n_1 \dots n_2$ **do**
    $D_i^{final} \leftarrow$ sample from $D_i^\star$
  end for

	Train	Validation	Test
RMC	1,100,000	116,752	50,000
ERMB	-	-	50,000

Table 1: Dataset statistics of RMC and ERMB. RMC consists of train, validation, and test sets, while ERMB only consists of test set.

ERMB Construction. The dataset $D_{\leq 3}$ consists of short regexes, making it necessary to evaluate performance on longer, unseen examples to assess generalizability. We provide an unlabeled benchmark with regexes of depths 4 to 6 to address this. For depths 4 to 6, we construct the sets by sampling $m(= 1000)$ regexes from the previous depth before applying binary operations. However, this

direct sampling may cause repeated patterns, which distorts the distribution of the benchmark. In order to mitigate this, we repeat the construction process k $(= 10)$ times, combine the results, and then sample regexes in a 1:1:1 ratio by depth. The construction process is described in Algorithm 2.

4.2 Calculation of Minimal Tree Length

Given a regex, all smaller regexes must be examined to determine the minimal tree length. Similar to the construction of D_n, we initialize $A_1 = \Sigma$ and recursively construct A_n, where n represents the tree length of regexes in A_n. Unary operations are applied to A_{n-1}, while binary operations are applied to all possible pairs between elements from A_i and A_{n-i-1}. The detailed process is described in Algorithm 3. Once $A_{\leq n}$ is constructed, regexes are partitioned into equivalence classes. We select a regex from $A_{\leq n}$ as a pivot, and all equivalent regexes are grouped into the same class. The minimal tree length for each class is determined as the smallest tree length within the group. The proof of minimality and the equivalence class partitioning algorithm are in Appendix B.

This approach requires a quadratic number of comparisons, each taking exponential time, making it computationally infeasible. We reduce the computation by applying a heuristic based on string acceptance. This method generates sequences of strings that a regex may accept, allowing regexes to be partitioned based on acceptance and rejection. Once the regex set is sufficiently reduced, the remaining comparisons are performed to determine equivalence. This method also helps in determining whether a regex x has an equivalent regex of length n or less by narrowing down candidate equivalent regexes. Using this approach, RMC is labeled. The detailed dataset construction procedure is in Appendix C. Additionally, examples of our regex minimization dataset are in Appendix D.

5 Experiments

5.1 Experimental Settings

We first conduct baseline experiments using zero-shot and five-shot prompting on five open-source instruction-tuned LLMs, specifically Llama (3.1:8B, 3.3:70B) and Qwen (2.5:7B, 2.5:72B, 2.5-coder) families. The code-specialized model is included to assess the impact of practical coding knowledge. However, as shown in Table 2, prompting performance is unsatisfactory. In order to investigate whether this stems from data scarcity or task difficulty, we train smaller models like BART and T5 from scratch. Training details are provided in Appendix E.

Due to the constraints of evaluating commercial LLMs on the full dataset, we alternatively present case studies using state-of-the-art models such as GPT-4o and Gemini-2.5-pro in Appendix F. The prompt templates used across all evaluated models are detailed in Appendix G.

5.2 Evaluation Metrics

We employ three evaluation metrics. The most intuitive metric is minimality, which represents the percentage of cases where the model successfully generates an equivalent minimal regex. The second metric is equivalence, which measures the proportion of outputs that are equivalent to the input regex. Since many generated responses are either non-equivalent or syntactically invalid, we report the equivalence metric separately for a more detailed analysis. The minimality is stricter than the equivalence, as it considers only minimal and equivalent regexes as success. Additionally, we use Length Ratio, a metric adopted from previous work for regex simplification. The metric is based on the geometric mean of the tree length ratio between the original regex and minimized regex. For the geometric mean calculation, the length ratio is set to 1 if the minimized regex is not equivalent to the original or becomes longer. Unlike minimality, which is a binary measure, the Length Ratio quantifies the degree of compression, offering insight into partial optimization. Since ERMB does not contain ground-truth minimized regexes, minimality cannot be measured. Instead, we report equivalence and length ratio for ERMB.

$$\text{(Minimality)} = \frac{\left|\{r \in D | f(r) \in X_r^\star(\Sigma)\}\right|}{|D|}$$

$$\text{(Equivalence)} = \frac{\left|\{r \in D | f(r) \in X_r(\Sigma)\}\right|}{|D|}$$

$$\text{(Length Ratio)} = \Big(\prod_{x \in D} \frac{|f(x)|_T}{|x|_T}\Big)^{\frac{1}{|D|}}$$

5.3 Main Results

5.3.1 Experimental Results on RMC

Table 2 presents the main results on RMC. Overall, LLMs demonstrate rather low performance. Models with 7-8B parameters produce minimal regexes

Model	Prompting	Minimality ↑	Equivalence ↑	Length Ratio ↓
BART 139M	-	**0.8589**±0.0007	**0.8712**±0.0014	**0.7966**±0.0002
T5 223M	-	0.8556±0.0113	0.8671±0.0116	0.7972±0.0015
Llama3.1:8B	Zero-shot	0.0172	0.0588	0.9939
	5-shot	0.0395	0.1082	0.9854
Llama3.3:70B	Zero-shot	0.1345	0.2427	0.9733
	5-shot	0.2115	0.3982	0.9430
Qwen2.5:7B	Zero-shot	0.0763	0.1651	0.9891
	5-shot	0.0675	0.1566	0.9817
Qwen2.5-coder:7B	Zero-shot	0.1048	0.1843	0.9785
	5-shot	0.1067	0.2469	0.9726
Qwen2.5:72B	Zero-shot	0.2031	**0.5452**	0.9537
	5-shot	**0.2306**	0.5433	**0.9309**

Table 2: Main results on RMC. We evaluate two LMs trained on RMC from scratch and five LLMs from Llama and Qwen family with zero-shot and five-shot prompting. We report the minimality, equivalence, and length ratio as metrics. The best performance is bolded in each of the trained LMs and the pre-trained LLMs.

Model	Prompting	Equivalence ↑	Length Ratio ↓
BART 139M	-	**0.0008**±0.0001	0.**9993**±0.0001
T5 223M	-	0.0005±0.0001	0.9995±0.0001
Llama3.1:8B	Zero-shot	0.0151	0.9996
	5-shot	0.0026	0.9999
Llama3.3:70B	Zero-shot	0.0716	0.9960
	5-shot	0.0257	0.9973
Qwen2.5:7B	Zero-shot	0.0179	0.9990
	5-shot	0.0066	0.9995
Qwen2.5-coder:7B	Zero-shot	0.0328	0.9985
	5-shot	0.0136	0.9995
Qwen2.5:72B	Zero-shot	**0.1399**	**0.9928**
	5-shot	0.0644	0.9950

Table 3: Main results on ERMB. We evaluate two LMs trained on RMC from scratch and five LLMs from Llama and Qwen family with zero-shot and five-shot prompting. We report the equivalence, and length ratio as metrics. The best performance is bolded in each of the trained LMs and the pre-trained LLMs.

in only 10% of cases and equivalent regexes in less than 25% of cases. Consistent with our intuition, larger models outperform smaller ones, and five-shot prompting surpasses zero-shot prompting. Nevertheless, even the best-performing model, Qwen2.5:72B, achieves only around 23% minimality, highlighting the difficulty of regex minimization for current LLMs.

In contrast, the trained BART and T5 substantially outperform LLMs, due to the lack of regex-specific train data for the LLMs. Further supporting this analysis, the code-specialized model Qwen2.5-coder outperforms models of similar size. This performance difference indicates that exposure to regex-related data during code-specialized pre-training positively impacts regex minimization.

5.3.2 Experimental Results on ERMB

Similarly, Table 3 presents the results on the ERMB. Overall, all models generally exhibit poor performance. In particular, the trained BART and T5 models experience severe performance degradation, performing worse than most LLMs. Even for regexes within the length range seen during training, performance remains low. This suggests that these models rely more on overfitting in the data rather than actual regex minimization. LLMs also demonstrate a significantly lower rate of producing equivalent regexes. These results indicate that handling longer regexes is challenging not only for minimization but also for equivalence preserva-

tion. Notably, five-shot prompting, which improves RMC performance, leads to a decline on ERMB. This may be due to the varying regex depths in ERMB, where exposure to regexes of different complexities interferes with generalization. These results highlight the limitations of LLM inference, particularly for regexes beyond the training distribution. We provide a deeper analysis of these limitations, including experiments on fine-tuning reliability and reasoning capabilities via CoT prompting, in Appendix H.

5.4 Further Analysis

We give a detailed analysis of the failures of various models by evaluating them from multiple perspectives, providing insights into how these models approach the problem. We present four research questions, along with the experimental results and corresponding analyses for each. RQ1 involves a type of failure analysis. RQ2 is an analysis based on the observation that LLMs tend to produce equivalent regexes more effectively than minimal regexes. RQ3 and RQ4 examine whether the model predictions accurately reflect the theoretical properties of regexes. Each research question aims at assessing the degree of mismatch between the performance of models and human insights, ultimately identifying the limitations inherent in each model. Each experiment is conducted using either the RMC test set or its variations, and we report the experimental results obtained with LLMs as well as the BART and T5 models that we trained.

RQ1: Which Type of Failure Do Models Experience? We categorize the failure cases of LMs into three types based on manual analysis: invalid syntax, inequivalent, and non-minimal. More specifically, invalid syntax refers to cases where the output remains syntactically invalid even after post-processing. Inequivalent refers to cases where the output is valid but not equivalent to the input regex. Non-minimal refers to cases where the output is valid and equivalent to the input regex but not minimal. Appendix I presents the failure type distributions for all models on the RMC test set. Figure 2 reports the failure proportions for BART, T5, and LLMs with over 70B parameters in the RMC test set, excluding successfully minimized regexes. Since BART and T5 perform better in minimization, their x-scale in the figure is significantly smaller compared to LLMs. Across all models, inequivalent cases account for the largest proportion of failures. This indicates that many generated regexes are valid but not equivalent to the input. Among LLMs, nearly half of the RMC test set fall into this category, showing that even maintaining equivalence is highly challenging. However, an encouraging observation is that the second most frequent failure type of Qwen2.5:72B is non-minimal. This means nearly half of its outputs are at least equivalent to the input regex. These observations motivate the next research question.

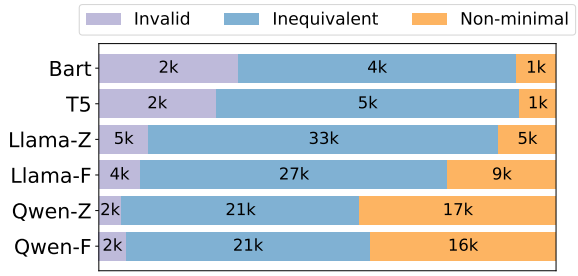
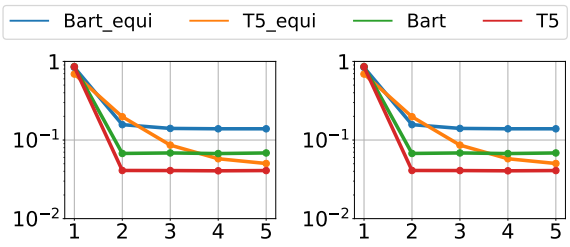


Figure 2: The proportion of each failure type on the RMC test set. The plot shows the proportion of each failure type of BART, T5, Llama3.3:70B and Qwen2.5:72B. The types of failures are classified into three cases: invalid, inequivalent and non-minimal. The number of regexes corresponding to each case is written on the bar.

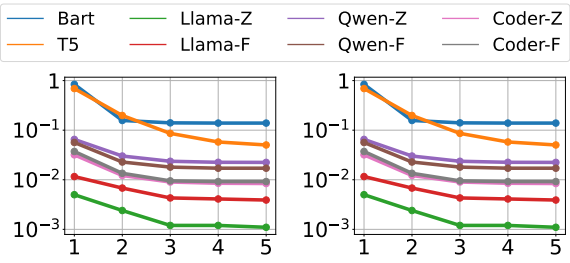
RQ2: Can Recursive Inference of Models Improve the Performance? The previous analyses show that minimizing regexes is challenging, whereas generating equivalent regexes is relatively easier. This leads to non-minimal failures, raising the question of whether recursive inference could improve performance. We evaluate recursive inference on BART and T5 using the RMC test set where each model's output is fed back as input without any post- or pre-processing. As shown in Figure 3, we observe severe performance degradation due to the accumulation of invalid or inequivalent regexes, preventing the provision of proper input as recursive steps progress.

We conduct equivalence-filtered recursive inference experiments to assess the influence of excluding invalid inputs on performance. Before feeding the output back as input, we perform an equivalence check and if an output is inequivalent, the previous input is reused instead. We use a temperature greater than zero to encourage variation because if the output is deterministic when reusing the previous input, there would be no advantage in performing another inference. The results in Figure 3 indicate that although the initial performance is worse, performance of repeated recursive steps 2



(a) Minimality (b) Equivalence

Figure 3: The result of recursive inference on RMC test set. We compare BART and T5 using recursive inference with and without equivalence check.



(a) Minimality (b) Equivalence

Figure 4: The result of recursive inference with equivalent check on RMC test set. We compare two LMs and LLMs with 7-8B parameters using zero-shot and five-shot prompting.

to 5 is better when compared to recursive inference without equivalence filtering. However, the benefits of equivalence filtering are outweighed by the negative impact of higher temperature settings, resulting in poor overall performance. Similar performance degradation due to recursive steps was consistently observed across LLMs, as described in Figure 4. Due to the computational cost of recursive inference, we conduct experiments using models with 7-8B parameters and report these results. These results suggest that LLMs are vulnerable to non-zero temperature settings in tasks that require high accuracy, such as regex minimization.

RQ3: For Equivalent Regexes as Inputs, Do Models Show Consistency in Output? Multiple equivalent regexes that represent the same regular language share the same set of minimal regexes. Following this theoretical insight, we investigate whether language models exhibit consistent performance when minimizing the set of equivalent regexes. For this analysis, we randomly sample 50 equivalent regex classes from the RMC test set, each containing at least 20 equivalent regexes. We retain 20 regexes per class to standardize evaluation, forming a benchmark of 1,000 regexes. Model evaluation is conducted based on the average number of regexes successfully minimized within each class. The models used for evaluation include BART and T5, as well as LLMs with over 70B parameters using both zero-shot and five-shot prompting. The distribution of correctly minimized regexes can be found in Appendix J.1.

The results show that BART and T5 achieved an average of 19.97 and 19.99 correctly minimize regexes per class, respectively, indicating near-perfect minimization performance across all sampled regexes. In contrast, Llama and Qwen minimize an average of 4.46 and 3.80 regexes per class in five-shot prompting, respectively. Furthermore, the five-shot prompting outperformed zero-shot. It suggests that while LLMs successfully minimize certain regexes, they fail to do so consistently across equivalent expressions.

A manual analysis of failure cases is conducted to understand how LLMs approach regex minimization. The results indicate that LLMs performed relatively well on simple transformations involving a single character $\sigma \in \Sigma$, such as reducing $\sigma + \sigma$, and $\sigma^*\sigma^*$. Additionally, regexes composed solely of unions of single characters are handled correctly. However, LLMs struggled with a more complex regex x, instead of σ. In some cases, the models incorrectly remove alphabets or operators without justification. Notably, LLMs often fail to minimize expressions like $x^* + xx$ correctly, failing to eliminate redundant xx. This suggests that LLMs do not fully grasp the interactions between option, union, and Kleene star operators. These findings indicate that LLMs currently struggle with even basic regex simplifications and lack a fundamental understanding of regex properties, particularly recursion and union. Additional experiments are conducted in Appendix J.2 to further investigate whether commercial LLMs better address these challenges.

RQ4: How Sensitive are Models to the Homomorphic Regexes? A regular language is closed under homomorphism. This means that a regex minimization model should operate symmetrically, which implies regardless of any permutation applied to the input alphabet. If this condition is not met, the model's performance would vary depending on the ordering of the alphabet. It implies that the LLM is overfitted to specific alphabet sequences, which might negatively impact generalizability. We examine the symmetry of the models which is represented by robustness on the permuta-

Model	Minimality ↑	Equivalence ↑	Length Ratio ↓
BART	**0.6882**	**0.7005**	**0.8837**
T5	0.3535	0.3656	0.9355
Llama-Z	0.1766	0.2894	0.9822
Llama-F	0.2493	0.4194	0.9600
Qwen-Z	0.2593	**0.5963**	0.9680
Qwen-F	**0.2839**	0.5631	**0.9524**

Table 4: The result on the set of regexes constructed using permutation. We report the performance our trained LMs and LLMs with more than 70B parameters. The best performances are highlighted with bold.

tion of alphabet.

We randomly sample 1,000 regexes from the RMC test set that contain all four symbols (a,b,c,d), and are not equivalent under any permutation. We then apply all 24 possible permutations to these alphabets, generating a total of 24,000 regexes. Then, we removed 2,556 regexes that appeared in the train and validation sets for ensuring fairness in evaluation. The resulting dataset consists of 21,444 regexes. The models used for evaluation include the fine-tuned BART and T5, as well as LLMs with over 70B parameters. Both zero-shot and five-shot prompting are tested for LLMs. The performance results are presented in Table 4.

Compared to the baseline RMC results in Table 2, BART and T5 exhibit significant performance drops of 20% and 60%, respectively. This degradation highlights the models' inability to capture algebraic symmetry, suggesting they have overfitted to specific sequential patterns rather than learning generalizable rules. Our dataset design intentionally minimizes redundancy by selecting specific binary operations. For instance, given regexes x and y, we include $x + y$ and xy while excluding their commutative counterparts $y + x$ and yx. Although this creates an asymmetric distribution, we expect robust models to generalize across these variations. Therefore, the observed failure is not an inherent flaw of the dataset but a limitation of the training process, which can be mitigated through simple data augmentation such as permutation. In contrast, LLMs maintain consistent performance in this setting, suggesting they possess a stronger capacity for symbolic abstraction rather than relying solely on sequential representation.

6 Conclusion

We present regex minimization as a new challenge for LLMs by constructing the first dataset for this PSPACE-complete problem and comparing the performance of various models. Additional analysis reveals that while LLMs currently exhibit poor performance and fail to apply simple rules, they demonstrate greater generalizability compared to trained language models. This suggests that our benchmark highlights the necessity for symbolic operations and precise reasoning processes, serving as a key direction for future advancements. Consequently, we propose the development of dedicated frameworks that incorporate regex minimization into LLM reasoning, such as Chain-of-Thought or multi-agent systems. Furthermore, we believe that reconstructing the dataset to explicitly integrate step-by-step reasoning processes represents a promising direction for future research. While collecting minimal regex data is inherently difficult, previous studies on regex simplification provide a solid foundation to overcome this hurdle. By leveraging these techniques as intermediate steps, regex minimization can be developed into a highly challenging yet analytically rich reasoning task.

Limitations

Our dataset construction process is highly labor-intensive, and as the construction steps increase, the amount of data that needs to be processed grows double-exponentially. This imposes constraints on dataset expansion. Given the PSPACE-complete nature of regex minimization, addressing this issue remains a significant challenge. Additionally, as mentioned in RQ4, our dataset contains partially asymmetric elements. However, this limitation can likely be mitigated through simple data augmentation. Another limitation of our dataset is that it is generated using a restricted set of alphabet symbols. Nevertheless, we have provided a detailed explanation of the dataset construction process, ensuring that it can be reproduced if necessary. Since our work focuses on theoretical regex rather than practical regex, it may not be directly applicable to real-world applications. However, as mentioned in Section 2, this aspect is beyond the scope of our study.

Acknowledgements

This research was supported by the NRF grant (RS-2025-00562134) and the AI Graduate School Program (RS-2020-II201361) funded by the Korean government.

References

Evripidis Bampis, Bruno Escoffier, and Michalis Xefteris. 2024. Parsimonious learning-augmented approximations for dense instances of $\mathcal{NP}$-hard problems. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235, pages 2700–2714.

Anne Brüggemann-Klein. 1993. Regular expressions into finite automata. *Theoretical Computer Science*, 120(2):197–213.

James C. Davis, Christy A. Coghlan, Francisco Servant, and Dongyoon Lee. 2018. The impact of regular expression denial of service (redos) in practice: an empirical study at the ecosystem scale. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, page 246–256.

Franklin L DeRemer. 1974. Lexical analysis. *Compiler Construction: An Advanced Course*, pages 109–120.

Keith Ellul, Bryan Krawetz, Jeffrey Shallit, and Mingwei Wang. 2004. Regular expressions: new results and open problems. *J. Autom. Lang. Comb.*, 9(2–3):233–256.

Lizhou Fan, Wenyue Hua, Lingyao Li, Haoyang Ling, and Yongfeng Zhang. 2024a. NPHardEval: Dynamic benchmark on reasoning ability of large language models via complexity classes. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 4092–4114.

Lizhou Fan, Wenyue Hua, Xiang Li, Kaijie Zhu, Mingyu Jin, Lingyao Li, Haoyang Ling, Jinkui Chi, Jindong Wang, Xin Ma, and Yongfeng Zhang. 2024b. NPHardEval4V: A dynamic reasoning benchmark of multimodal large language models. *arXiv preprint arXiv:2403.01777*.

Hermann Gruber and Stefan Gulan. 2010. Simplifying regular expressions. In *Language and Automata Theory and Applications, 4th International Conference, LATA Proceedings*, volume 6031, pages 285–296.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*.

Yo-Sub Han and Derick Wood. 2005. The generalization of generalized automata: Expression automata. *International Journal of Foundations of Computer Science*, 16(03):499–510.

Yo-Sub Han and Derick Wood. 2007. Obtaining shorter regular expressions from finite-state automata. *Theoretical Computer Science*, 370(1-3):110–120.

John Hopcroft. 1971. An n log n algorithm for minimizing states in a finite automaton. In *Proceedings of an International Symposium on the Theory of Machines and Computations*, pages 189–196.

John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. 2007. *Introduction to automata theory, languages, and computation, 3rd Edition*. Addison-Wesley.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, et al. 2024. Qwen2.5-coder technical report. *arXiv preprint arXiv:2409.12186*.

H. B. Hunt. 1973. On the time and tape complexity of languages i. In *Proceedings of the Fifth Annual ACM Symposium on Theory of Computing*, page 10–19.

Stefan Kahrs and Colin Runciman. 2022. Simplifying regular expressions further. *Journal of Symbolic Computation*, 109:124–143.

Jonathan Lee and Jeffrey Shallit. 2005. Enumerating regular expressions and their languages. In *Implementation and Application of Automata*, pages 2–22.

Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. 2020. Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. In *Proceedings of the 58th annual meeting of the association for computational linguistics*, pages 7871–7880.

Yeting Li, Yecheng Sun, Zhiwu Xu, Jialun Cao, Yuekang Li, Rongchen Li, Haiming Chen, Shing-Chi Cheung, Yang Liu, and Yang Xiao. 2022. RegexScalpel: Regular expression denial of service (ReDoS) defense by Localize-and-Fix. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 4183–4200.

Zhenting Qi, Hongyin Luo, Xuliang Huang, Zhuokai Zhao, Yibo Jiang, Xiangjun Fan, Himabindu Lakkaraju, and James Glass. 2024. Quantifying generalization complexity for large language models. *arXiv preprint arXiv:2410.01769*.

Qwen, An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, et al. 2024. Qwen2.5 technical report. *arXiv preprint arXiv:2412.15115*.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of machine learning research*, 21(140):1–67.

Yuju Shen, Yanyan Jiang, Chang Xu, Ping Yu, Xiaoxing Ma, and Jian Lu. 2018. ReScue: crafting regular expression DoS attacks. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, page 225–235.

Mohammed Latif Siddiq, Jiahao Zhang, Lindsay Roney, and Joanna C. S. Santos. 2024. Re(gEx|DoS)Eval: Evaluating generated regular expressions and their proneness to DoS attacks. In *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*, page 52–56.

Michael Sipser. 1997. *Introduction to the theory of computation*. PWS Publishing Company.

Osvaldo Mario Spositto, Julio César Bossero, Edgardo Javier Moreno, Viviana Alejandra Ledesma, and Lorena Romina Matteo. 2021. Lexical analysis using regular expressions for information retrieval from a legal corpus. In *Argentine Congress of Computer Science*, pages 312–324. Springer.

Richard Sproat. 2000. Lexical analysis. *Handbook of Natural Language Processing, 2nd Edition, Marcel Dekker Inc*, pages 37–57.

L. J. Stockmeyer and A. R. Meyer. 1973. Word problems requiring exponential time(preliminary report). In *Proceedings of the Fifth Annual ACM Symposium on Theory of Computing*, page 1–9.

Ken Thompson. 1968. Programming techniques: Regular expression search algorithm. *Communications of the ACM*, 11(6):419–422.

A Computational Cost of Dataset Construction

Our dataset size is determined by the need to exhaustively compare each regex with all shorter expressions to verify minimality. As the depth or length of regexes increases, the number of candidate expressions grows double-exponentially due to our inductive construction. We report the computational cost over different sizes of alphabet and different depth. Each following Tables 5 and 6 contains the number of regexes in the dataset and the number of regexes we need to investigate to find the minimal regexes. Note that the number of regexes grows in double-exponential which makes it almost impossible to increase depth more than 3. We construct the dataset inductively using a fixed alphabet of size 4 ($\{a, b, c, d\}$) and restrict the tree depth to 3, resulting in regular expressions with a maximum length of 15. Although the dataset may appear small, we exhaustively compare a vast number of regexes to identify the minimal ones, a process that is extremely labor intensive.

B Proofs for Minimality of the Constructed Datasets in Section 4

In this section, we discuss algorithms and proofs that were not covered in Section 4. More specifically, our goal is to prove the minimality of the constructed dataset. The proof sketch is as follows: A_n generated by Algorithm 3, and M_n generated by Algorithm 4, contain all possible regexes of length n or less. Moreover, the sequences $M_1, M_2, \ldots, M_n$ follow an inclusion relationship $M_1 \subset M_2 \subset \ldots \subset M_n$, where it contains all M_i for $i \leq n$ as n increases. Thus, any regex of length n or less has an equivalent minimal regex included in some A_i for $i \leq n$, and by the inclusion relationship, it must be contained in M_n. This proves that M_n contains all minimal regexes of length n or less. Furthermore, based on the construction method, we can show that non-minimal regexes are not included in M_n. This implies that M_n represents the complete set of all minimal regexes of length n or less.

Lemma B.1. *Given an integer n and an alphabet Σ, the set A_n constructed by algorithm 3 contains all possible regular expressions on Σ, when considering the equivalence relation as the same.*

Proof. Assume that there exists a regular expression x over Σ such that $x \notin A_n$, and $|x|_T = n$.

Algorithm 3 An algorithm for building A_n

Require: integer n, alphabet Σ
  Initialize $A_1 = \Sigma$
  for $i = 2 \ldots n$ **do**
    $A_i \leftarrow \emptyset$
    for $x \in A_{i-1}$ **do**
      $A_i \leftarrow A_i \cup \{\text{concat}(x,\text{'?'})\}$
      $A_i \leftarrow A_i \cup \{\text{concat}(x,\text{'*'})\}$
    end for
    for $j = 1 \ldots \frac{i-1}{2}$ **do**
      for $x \in A_j$ **do**
        for $y \in A_{i-1-j}$ **do**
          if $x \equiv y$ **then**
            $A_i \leftarrow A_i \cup \{\text{concat}(x,y)\}$
          else
            $A_i \leftarrow A_i \cup \{\text{concat}(x,\text{'+'},y)\}$
            $A_i \leftarrow A_i \cup \{\text{concat}(x,y)\}$
            $A_i \leftarrow A_i \cup \{\text{concat}(y,x)\}$
          end if
        end for
      end for
    end for
  end for

Each regular expression can be represented by a binary tree. Denote the binary tree notation of a regular expression x be $T(x)$, and the left and right subtrees of x be x_{left} and x_{right}. Let $|x_{left}|_T$ be $j(\geq 1)$. Since $x \notin L_n$, at least one of $x_{left} \notin A_j$ or $x_{right} \notin A_{n-1-j}$ is true. By using the recursion, we can conclude that there exists a single character $c \in \Sigma$, which is included as a character in the regex x, but $c \notin A_1$. This contradicts the fact that $A_1 = \Sigma$. Thus, there is no regular expression $x \notin A_n$ with $|x|_T$=n. □

Lemma B.2. *Given the sets $A_1, A_2, \ldots, A_n$ constructed by algorithm 3 and an alphabet Σ, the sets M_n constructed by algorithm 4 contains all possible regular expressions on Σ, when considering the equivalence relation as the same.*

Proof. Assume that there is a regular expression $x \notin M_n$ with $|x|_T \leq n$. By lemma B.1, $x \in \bigcup_{j=1}^{n} A_j$, and it must be considered by the loop in line 4 of algorithm 4. Then, x should be in M_n, since the representative of each equivalent class in $\bigcup_{j=1}^{n} A_j$ is added to M_n. This contradicts the assumption that $x \notin M_n$. Thus, M_n contains all possible regular expressions on Σ. □

Size of Alphabet ($\|\Sigma\|$)	Depth					
	0	1	2	3	4	5
2	2	10	170	3.35E+04	1.13E+09	1.29E+18
3	3	18	486	2.58E+05	6.69E+10	4.47E+21
4	4	28	1092	**1.27E+06**	1.60E+12	2.57E+24
5	5	40	2120	4.69E+06	2.20E+13	4.85E+26
6	6	54	3726	1.43E+07	2.06E+14	4.23E+28
7	7	70	6090	3.80E+07	1.45E+15	2.10E+30
8	8	88	9416	9.05E+07	8.19E+15	6.71E+31
9	9	108	13932	1.97E+08	3.90E+16	1.52E+33

Table 5: The number of regexes contained in the dataset constructed by the proposed procedure depending on the size of alphabet and depth

Size of Alphabet ($\|\Sigma\|$)	Depth					
	0	1	2	3	4	5
2	2	20	6176	2.16E+09	8.46E+20	3.85E+44
3	3	39	20118	2.10E+10	7.63E+22	3.00E+48
4	4	64	48640	**1.15E+11**	2.21E+24	2.46E+51
5	5	95	98870	4.52E+11	3.31E+25	5.39E+53
6	6	132	179232	1.42E+12	3.20E+26	4.95E+55
7	7	175	299446	3.80E+12	2.26E+27	2.44E+57
8	8	224	470528	9.05E+12	1.26E+28	7.52E+58
9	9	279	704790	1.96E+13	5.86E+28	1.61E+60

Table 6: The number of regexes to be investigated to find minimal by the proposed procedure depending on the size of alphabet and depth

Algorithm 4 An algorithm for building M_n

Require: $A_1, A_2, \ldots, A_n$
  $A \leftarrow \bigcup_{i=1}^{n} A_i$
  $M_n \leftarrow List()$
  while $|A| > 0$ **do**
    $x \leftarrow A$.pop()
    for $y \in A$ **do**
      if $y \equiv x$ **then**
        A.remove(y)
        if $|y|_T < |x|_T$ **then**
          $x \leftarrow y$
        end if
      end if
    end for
    $M_n \leftarrow M_n \cup \{x\}$
  end while

Lemma B.3. *Given an alphabet Σ and the sets of regexes $A_1, A_2, \ldots, A_i, \ldots, A_n$ constructed by algorithm 3, the sets M_i and M_n are constructed by algorithm 4. Then, $M_i \subset M_n \; \forall \; i = 1, \ldots, n$, when considering the equivalence relation with the same tree length as the same.*

Proof. Assume that there is a regular expression $x \in M_i$ with $|x|_T = i \leq n$, but $x \notin M_n$, when considering the equivalence relation with the same tree length as the same. By the lemma B.2, there exists an equivalent regular expression $x' \in M_n$ with $|x'|_T = j \neq i$. Note that $i, j \leq n$.

Case1: $i < j$ By the lemma B.1, $x \in \bigcup_{k=1}^{i} L_k \subset \bigcup_{k=1}^{n} L_k$. When we compare the tree lengths of regular expressions during the construction of M_n by algorithm 4, x' must be discarded and replaced by x. Thus, j cannot be larger than i.

Case2: $j < i$ By the lemma B.2, $x' \in \bigcup_{k=1}^{i} A_j \subset$

$\bigcup_{k=1}^{i} A_k$. When we compare the tree lengths of regular expressions during the construction of M_i by algorithm 4, x must be discarded and replaced by x'. Thus, i cannot be larger than j

Since $i \geq j$ and $i \leq j$, i and j must be the same, which contradicts the assumption. Thus, $M_i \subset M_n$ $\forall i = 1, \ldots, n$. □

Corollary B.4. *Given the sets $A_1, A_2, \ldots, A_n$ constructed by algorithm 3 and an alphabet Σ, the set M_n constructed by algorithm 4 contains all possible minimal regular expressions on Σ with $|x|_T \leq n$, when considering the equivalence relation with the same tree length as the same.*

Proof. Assume that there is a minimal regular expression x with $|x|_T = i \leq n$ such that $x \notin M_n$. By the lemma B.2, $x \in M_i$. Then, by the lemma B.3 $x \in M_n$. This contradicts the assumption. Thus, there is no such x. □

Corollary B.5. *Given the sets $A_1, A_2, \ldots, A_n$ constructed by algorithm 3 and an alphabet Σ, the set M_n constructed by algorithm 4 contains only minimal regular expressions on Σ with $|x|_T \leq n$, when considering the equivalence relation with the same tree length as the same.*

Proof. Assume that there is a non-minimal regular expression $x \in M_n$. There is a minimal regular expression x' of x with $|x'|_T = i \leq n$. Then, x' must be in M_n by corollary B.4. It contradicts the assumption because both x and x' cannot be in M_n by the algorithm 4. Thus, there is no such x. □

Theorem B.6. *Given the sets $A_1, A_2, \ldots, A_n$ constructed by algorithm 3, an alphabet Σ, and the set M_n constructed by algorithm 4, the following statement holds when considering the equivalence relation with the same tree length as the same.*

$$x \in M_n \Leftrightarrow x \in X_x^{\star}(\Sigma), \text{ where } |x|_T \leq n.$$

Proof. It is proved by combining Corollary B.4 and B.5. □

C Calculation of Minimal Tree Length using String Acceptance

As mentioned earlier in Section 4.2, the process of calculating the equivalence of all regex pairs requires $O(n^2)$ equivalence comparisons. We utilize string acceptance in order to reduce the number of candidates within a group that can be equivalent to one another. The detailed algorithm is described in Algorithm 5. For a given string s, a regex that accepts s and another regex that rejects s cannot be equivalent. Based on this observation, we use string acceptance as a query to partition the regex group. First, we categorize regexes based on the alphabet character set they contain. Then, we check the acceptance of a predefined string sequence. Using the acceptance information of the string sequence as a binary encoding, we further divide the regexes into potentially equivalent classes. Once the group size becomes sufficiently small, we perform the $O(n^2)$ equivalence comparison to construct the equivalent classes. Although this procedure does not reduce the time complexity, it reduces the practical running time of partitioning regex groups into equivalent classes. Furthermore, when a new regex to minimize is given as a query, we can find the candidate equivalent class by checking string acceptance of the given regex on the predefined string sequence.

D Examples of Our Regex Minimization Dataset

Table 7 presents examples from our constructed regex minimization dataset, divided into RMC and ERMB subsets. We report query regex examples for depths ranging from 0 to 6, covering a wide spectrum of difficulty. As shown in the table, the query length and complexity increase significantly as the tree depth grows, posing a greater challenge for minimization. This demonstrates our effort to create a comprehensive benchmark that tests the model's capability across varying levels of structural complexity.

For each entry, we provide the query regex (the input expression) and its query length. In the RMC subset, we additionally list the minimal regex and minimal length, which serve as the ground-truth shortest equivalent expression and its character count, respectively. We do not provide the depth of the minimal regex, since tree depth can vary even for identical regexes depending on how parentheses are structured.

Algorithm 5 An algorithm for building M_n

Require: an alphabet Σ, $A_1, A_2, \ldots, A_n$, an integer m and the fixed sequence of strings S

$A \leftarrow \bigcup_{i=1}^{n} A_i$ ▷ Merge all input regex sets
$G \leftarrow map()$
for $\sigma \in 2^{\Sigma}$ **do**
  $G[\sigma] \leftarrow \emptyset$
  label $G[\sigma]$ as 1 ▷ Initialize bit for each alphabet subset
end for
while $|A| > 0$ **do**
  $x \leftarrow A$.pop()
  $\sigma_x \leftarrow$ the alphabet of x
  $G[\sigma_x] \leftarrow G[\sigma_x] \cup \{x\}$ ▷ Group regexes by their used alphabet
end while
$H \leftarrow G.values()$
while $|H| > 0$ **do**
  $X \leftarrow H$.pop()
  if $|X| > m$ **then**
    $s \leftarrow S[X.label]$ ▷ Case 1: Group is too large, split using distinguishing string
    $X_{success} \leftarrow \emptyset$
    label $X_{success}$ as X.label $\ll$ 1+1 ▷ Update label for success branch
    $X_{fail} \leftarrow \emptyset$
    label X_{fail} as X.label $\ll$ 1 ▷ Update label for fail branch
    for $y \in X$ **do**
      if y accepts s **then**
        $X_{success} \leftarrow X_{success} \cup \{y\}$
      else
        $X_{fail} \leftarrow X_{fail} \cup \{y\}$
      end if
    end for
    $H \leftarrow H \cup \{X_{success}, X_{fail}\}$ ▷ Push split groups back to H
  else
    while $|X| > 0$ **do**
      $x \leftarrow X$.pop() ▷ Case 2: Group is small enough for costly equivalence check
      for $y \in X$ **do**
        if $y \equiv x$ **then**
          X.remove(y) ▷ Check if equivalent
          if $|y|_T < |x|_T$ **then**
            $x \leftarrow y$ ▷ Keep the smaller regex
          end if
        end if
      end for
      $M_n \leftarrow M_n \cup \{x\}$ ▷ Add minimal regex to result
    end while
  end if
end while

RMC		
Depth	Field	Content
0	Query Regex	b
	Query Length	1
	Minimal Regex	b
	Minimal Length	1
1	Query Regex	ab
	Query Length	3
	Minimal Regex	ab
	Minimal Length	3
2	Query Regex	$a^* + a + d$
	Query Length	6
	Minimal Regex	$d + a^*$
	Minimal Length	4
3	Query Regex	$a + b^* + a^? + b^?$
	Query Length	10
	Minimal Regex	$a + b^*$
	Minimal Length	4
ERMB		
Depth	Field	Content
4	Query Regex	$(d + c)c^? + ca + c + c + b^* + a^* + bd(b + a)$
	Query Length	28
5	Query Regex	$(ac + a + d + (a + d)c)adaacdc$ $(dda^? + b + d + b^* + (a + b)c^* + b + c + a^?)$
	Query Length	55
6	Query Regex	$(b^* + c + b + ba^* + c^?(b + b) + (a + c)b^?)$ $(c^*cb(a + c + cb) + a + c + c^* + bd^*)$ $+(c^* + a + a)(d + b)ccc^?(b + db)$ $+dd + c^* + d + aba^* + c + d + d$
	Query Length	98

Table 7: Examples of our regex minimization dataset. We report the example regexes of depth 0 to 6 sampled from RMC and ERMB. RMC contains the minimal equivalent of a regex and the minimal length, while ERMB only contains regexes with their lengths.

E Experimental Details

E.1 Model Details

This section describes all models used in our main experiments, analyses, and appendix, and we use each model in accordance with its intended use.

Llama 3.1 and 3.3. We use two open-source models from Meta's Llama family: Llama3.1:8B[2] and Llama3.3:70B[3], both in their instruction-tuned, text-only variants. Llama3.1:8B serves as a smaller-parameter baseline, while Llama3.3:70B serves as a larger-parameter baseline for assessing scaling effects on regex minimization performance.

Qwen 2.5. We use Qwen2.5 (Qwen et al., 2024) in both 7B[4] and 72B[5] sizes. Qwen2.5 is an open-source model family available in multiple model scales, enabling controlled comparisons across parameter sizes.

Qwen 2.5-Coder. We use Qwen2.5-Coder[6] (Hui et al., 2024), a code-specialized series built on the Qwen2.5 architecture and trained with coding-focused data and evaluations. We include this model to test whether code-oriented training improves systematic string/grammar transformations relevant to regex simplification.

BART. We use BART[7] (Lewis et al., 2020) as a classic encoder-decoder seq2seq baseline. BART is widely used for text generation and transformation tasks, making it an LM baseline for regex-to-regex rewriting.

T5. We use T5[8] (Raffel et al., 2020) as another encoder–decoder baseline that frames tasks in a unified text-to-text format. This setup aligns well with regex minimization as a pure string-to-string transformation problem.

GPT-4o. We use GPT-4o[9], OpenAI's flagship multimodal model. In our experiments it served as a strong proprietary baseline for general-purpose generation and reasoning.

OpenAI o1. We use OpenAI o1[10], a reasoning-oriented model series as spending more time before responding. We include it to measure how a state-of-the-art reasoning model behaves on a PSPACE-complete rewriting task.

DeepSeek-R1. We use DeepSeek-R1 (Guo et al., 2025), a reasoning-focused model with a multi-stage training framework incorporating reinforcement learning and alignment steps. We include it as a representative reasoning-optimized open model in our comparison.

Gemini-2.5-Pro. We use Gemini-2.5-Pro[11], a reasoning-focused model developed by Google. At the time of our experiments, it was Google's most advanced reasoning model, and we include it as a strong proprietary reasoning baseline.

E.2 Hyperparameters for Training

Table 8 presents the hyperparameters used for training. Most of the hyperparameters are set to their default values, with only the tokenizer and learning rate adjusted for training. We construct the tokenizer word list using only the alphabet, operations, and parentheses used in actual regexes, employing character-based tokenization. This heuristic is based on the insight that when using methods like byte pair encoding (BPE) to train a tokenizer, character n-grams that do not form valid regexes may lead to invalid syntax during decoding. We report the main results aggregated over three independent runs with different random seeds. The experiments were conducted using NVIDIA A6000 GPUs, RTX 3090 GPUs, AMD Ryzen Threadripper 3960X CPUs, and Python 3.10.

E.3 Sensitivity Analysis for Training BART and T5

We conduct a sensitivity analysis in order to determine the appropriate learning rate. We train BART and T5 with learning rates of 1e-04, 5e-05, 1e-05, and 5e-06, respectively, and report the results. For all learning rates, the train loss decreases to nearly zero as training progresses, and the validation loss also converges well, as shown in Figure 5. This stable training trend suggests that the regex minimization task is well-suited for LMs. However, the trends of validation accuracy and validation loss do not completely align. Based on these findings, we

[2] meta-llama/Llama-3.1-8B-Instruct
[3] meta-llama/Llama-3.3-70B-Instruct
[4] Qwen/Qwen2.5-7B-Instruct
[5] Qwen/Qwen2.5-72B-Instruct
[6] Qwen/Qwen2.5-Coder-7B-Instruct
[7] facebook/bart-base
[8] google-t5/t5-base
[9] gpt-4o-2024-11-20
[10] o1-2024-12-17
[11] gemini-2.5-pro-preview-03-25

	BART	T5
The number of parameters	139M	223M
Tokenizer	Character-based	Character-based
Vocabulary Size	16	16
Optimizer	AdamW	AdamW
Learning rate	5e-05	1e-05
Weight decay	0.1	0.1
Learning rate schedule	Cosine schedule with warmup	Cosine schedule with warmup
Warmup steps	1000	1000
Batch Size	512	512
Number of Epochs	1000	1000

Table 8: Implementation details for the BART and T5 models. The table lists the architectural parameters and specific optimization settings employed to ensure the reproducibility of our results.

select a learning rate of 5e-5 for BART and 1e-5 for T5 to report the main results, although they do not achieve the lowest validation loss.

F Empirical Case Study of State-of-the-Art LLMs

F.1 Quantitative Analysis

We evaluate the performance of state-of-the-art closed-source LLMs, specifically GPT-4o, OpenAI o1, and Gemini-2.5-pro. These models were selected to represent the forefront of current capabilities, with OpenAI o1 and Gemini-2.5-pro being particularly optimized for complex reasoning tasks. Our evaluation dataset consists of 200 regexes, composed of 100 randomly sampled instances from the RMC and ERMB subsets, respectively. We employ five-shot prompting, as this setting demonstrated superior performance in our prior open-source model evaluations.

The results are summarized in Table 9. Gemini-2.5-pro outperforms other models across both benchmarks. Notably, in the RMC subset, it achieves perfect equivalence (1.00) and the highest minimality score (0.91), while also producing the most concise expressions (lowest length ratio) in the more challenging ERMB subset.

F.2 Qualitative Analysis

This section examines how well state-of-the-art LLMs perform in regex minimization based on a small test set. Using the five-shot prompt illustrated in Appendix G, we evaluate GPT-4o, one of the most popular LLMs, OpenAI o1, known for its reasoning capabilities, and DeepSeek-R1, which has recently gained significant attention as a competitive open LLM. The evaluation set consists of five regexes from the RMC test set, selected to highlight common failure patterns of LLMs observed in Section 5.4, along with two regexes from each of depths 4, 5, and 6 in ERMB, resulting in a total of eleven test cases.

The experimental results are in Table 10 and 11. GPT-4o successfully produces equivalent regexes in most cases from RMC but reveal clear limitations in regex minimization. As the regex length increases, it fails to generate equivalent outputs. DeepSeek-R1 successfully minimizes all regexes in RMC and handles equivalent regex generation up to depth 4. However, for regexes of depth 5 or higher, it struggles to produce equivalent outputs. OpenAI o1 successfully minimizes all regexes in RMC and generates equivalent regexes for all ERMB examples.

Despite being considered models with strong reasoning capabilities, both GPT-4o and DeepSeek-R1 exhibit clear limitations in reasoning when applied to regex minimization. This raises the question of whether existing LLMs, even those optimized for reasoning, may not be truly reasoning in the context of structured, algorithmic tasks. At the same time, these findings demonstrate that the consistent performance degradation observed from long regexes in ERMB underscores the complexity of regex minimization and further demonstrates the difficulty of our dataset as a valuable benchmark. The results highlight the need for explicit mechanisms that enhance structured reasoning, as LLMs struggle with systematic transformations required for regex minimization.

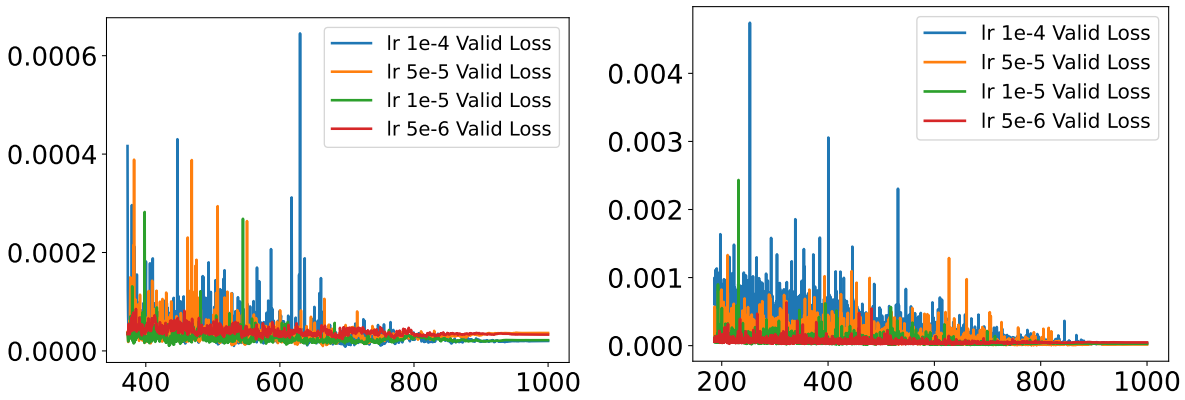


Figure 5: Validation loss over training epochs. We conduct this experiment as a sensitivity study for BART and T5. We train two LMs on four different learning rates for 1000 epochs.

Benchmark	Model	Minimality	Equivalence	Length ratio
RMC	GPT-4o	0.29	0.47	0.9390
	OpenAI o1	0.79	0.98	0.8091
	Gemini-2.5-pro	0.91	1.00	0.7905
ERMB	GPT-4o	-	0.02	0.9961
	OpenAI o1	-	0.86	0.8403
	Gemini-2.5-pro	-	0.87	0.8153

Table 9: Performance comparison of state-of-the-art closed-source LLMs on the RMC and ERMB datasets. We report Minimality, Equivalence (accuracy), and Length Ratio using five-shot prompting. Note that Minimality is not applicable to ERMB as the ground-truth minimal regexes are unknown.

RMC	
Field	Content
Query Regex	$(((a+b)+(b+c))+((b+d)+(c+c)))$
Minimal Regex	$(a+b+c+d)$
GPT-4o Output	$(a+b+c+d)$
Evaluation	Minimal, Length Ratio: 0.4667
DeepSeek-R1 Output	$(a+b+c+d)$
Evaluation	Minimal, Length Ratio: 0.4667
OpenAI o1 Output	$(a+b+c+d)$
Evaluation	Minimal, Length Ratio: 0.4667
Query Regex	$((a+(a+b))+((a+b)(b+c)))$
Minimal Regex	$((a+b)(b+c)^?)$
GPT-4o Output	$(a+(a+b)(b+c))$
Evaluation	Inequivalent
DeepSeek-R1 Output	$(a+b)(b+c)^?$
Evaluation	Minimal, Length Ratio: 0.6154
OpenAI o1 Output	$(a+b)(b+c)^?$
Evaluation	Minimal, Length Ratio: 0.6154
Query Regex	$(((a^?)(b^*))+((b^*)(b^?)))$
Minimal Regex	$(a^?b^*)$
GPT-4o Output	$(a^?b^*+b^*b^?)$
Evaluation	Equivalent, Length Ratio: 1.0000
DeepSeek-R1 Output	$(a^?b^*)$
Evaluation	Minimal, Length Ratio: 0.4545
OpenAI o1 Output	$a^?b^*$
Evaluation	Minimal, Length Ratio: 0.4545
Query Regex	$((d+(a^?))+((b^*)+(b^?)))$
Minimal Regex	$(a+d+b^*)$
GPT-4o Output	$(d+a^?+b^*)$
Evaluation	Equivalent, Length Ratio: 0.7000
DeepSeek-R1 Output	$(a+d+b^*)$
Evaluation	Minimal, Length Ratio: 0.6000
OpenAI o1 Output	$(a+d+b^*)$
Evaluation	Minimal, Length Ratio: 0.6000
Query Regex	$(((b^*)(bb))((d^*)(d^*)))$
Minimal Regex	(bbb^*d^*)
GPT-4o Output	$(b^*bb)(d^*)$
Evaluation	Minimal, Length Ratio: 0.7500
DeepSeek-R1 Output	$((b^*bb)d^*)$
Evaluation	Minimal, Length Ratio: 0.7500
OpenAI o1 Output	$((b^*bb)d^*)$
Evaluation	Minimal, Length Ratio: 0.7500

Table 10: Case study of state of the art LLMs on RMC. We report a case study using GPT-4o, OpenAI o1, and DeepSeek-R1 on RMC test set. We report the type of failure and length ratio for each example. GPT-4o, which is not specifically optimized for reasoning, shows the lowest performance while others successfully minimize all the examples.

ERMB	
Query Regex	$((((b+a)(ca))((b^*)+c))+(((da)^*)((da)+d)))$
GPT-4o Output	$((a+b)ca(b^*+c)+(da)^*d^?)$
Evaluation	Inequivalent
DeepSeek-R1 Output	$((b+a)ca(b^*+c)+d(ad)^*a^?)$
Evaluation	Equivalent, Length Ratio: 0.9565
OpenAI o1 Output	$((b+a)ca(b^*+c)+(da)^*da^?)$
Evaluation	Equivalent, Length Ratio: 0.9565
Query Regex	$((((cd)(a+b))+((c^?)(ab)))+(((bd)^*)((b+b)(bc))))$
GPT-4o Output	$(cd(a+b)+c^?ab+bd^*(b+b)bc)$
Evaluation	Inequivalent
DeepSeek-R1 Output	$(cd(a+b)+c^?ab+b(db)^*bc)$
Evaluation	Equivalent, Length Ratio: 0.9259
OpenAI o1 Output	$(cd(a+b)+c^?ab+(bd)^*bbc)$
Evaluation	Equivalent, Length Ratio: 0.9259
Query Regex	$(((((a+a)(bd))+((cc)^?))(((ad)(b+b))((ad)(c^*))))((((cc)(a+a))+(d^*))$ $+(((c+a)(bc))((aa)(bd)))))$
GPT-4o Output	$(a+b)bd+(cc)^?(ad(b+b)adc^*+(cc(a+a)+d^*+(c+a)bcaabd))$
Evaluation	Inequivalent
DeepSeek-R1 Output	$((abd+cc)^?adbadc^*(cca+d^*+(c+a)bcaabd))$
Evaluation	Equivalent, Length Ratio: 0.8889
OpenAI o1 Output	$((abd+(cc)^?)adbadc^*(cca+d^*+(c+a)bcaabd))$
Evaluation	Equivalent, Length Ratio: 0.8889
Query Regex	$(((((c+c)^*)+((a^*)(b^*)))+(((dd)(b^?))(d+d)))+((((d+d)+(ad))+(d+(b+b)))$ $+((b+(dd))((d+d)+(d+d)))))$
GPT-4o Output	$((c+c)^*+a^*b^*+dd(b^?)(d+d)+b+dd(d+d))$
Evaluation	Inequivalent
DeepSeek-R1 Output	$(c^*+ab+d(a^?+b+db^?d))$
Evaluation	Inequivalent
OpenAI o1 Output	$(c^*+a^*b^*+ddb^?d+a^?d+bd)$
Evaluation	Equivalent, Length Ratio: 0.5306
Query Regex	$((((((d+a)(b^*))^*)+(((b+a)+(a+a))((b+b)+(bb))))(((a(ba))+((ba)(ba)))$ $+(((c^?)(c+c))+((d+c)+(da)))))((((((d+c)^*)+((d+b)+(d^*)))+(((b+a)(c^*))$ $+((a^?)(c^?))))+((((dc)+(bb))((d+b)(ac)))+(((db)(d^*))((bc)^*)))))$
GPT-4o Output	$((d+a)(b^*)^*)+((a+b)(b+bb)+(ba)^*(c^?c+d+a))(c^*+d+b+d^*)(dc+bb)$ $(d+b)ac+(dbd^*)(bc)^*$
Evaluation	Inequivalent
DeepSeek-R1 Output	$((((a+d)b^*)^*+(a+b)bb^?)((a+ba)ba+cc^?+da^?))+(d+c)^*+b$ $+(a+b)^?c^*+(dc+bb)(d+b)ac+dbd^*(bc)^*$
Evaluation	Inequivalent
OpenAI o1 Output	$(((((a+d)b^*)^*+(a+b)bb^?)(b^?aba+cc^?+da^?))((c+d)^*+(b+d^*)$ $+(a+b)c^*+a^?c^?+((bb+dc)(b+d)ac)+(dbd^*)(bc)^*))$
Evaluation	Equivalent, Length Ratio: 0.8190
Query Regex	$(((((((c+d)d)+((c^*)+b))(((c^*)+(d^*))((a+d)(c+c))))((((c+c)^?)$ $+((bd)(c+d)))+(((b^*)a)+((bb)(c+c)))))+(((((dd)+(b^*))$ $+((c+d)(c+d)))^?)((((c+d)(bc))((dd)(ad)))+(((dd)(a+b))((c+d)(a^?))))))$
GPT-4o Output	$((c+d)d+c^*+b)((c^*+d^*)(a+d)cc)+(c+c)^?+bd(c+d)+(b^*a+bbcc)$ $+((dd+b^*+(c+d)(c+d))^?((c+d)bcddad+dda+b(c+d)a^?))$
Evaluation	Inequivalent
DeepSeek-R1 Output	$((c^*+b+d(c+d))(c^*+d^*)(a+d)c(c^?+bd(c+d)+ba^*+bbc)$ $+(b^*+(c+d)(c+d))^?((c+d)bcddad+dd(a+b)(c+d)a^?))$
Evaluation	Inequivalent
OpenAI o1 Output	$(((((c+d)d+(c^*+b))((c^*+d^*)(a+d)c))((c^?+bd(c+d))+((b^*)a+bbc)))$ $+((((dd+b^*)+(c+d)(c+d)))^?(((c+d)bcddad)+((dd(a+b))((c+d)a^?)))))$
Evaluation	Equivalent, Length Ratio: 0.9381

Table 11: Case study of state of the art LLMs on ERMB. We report case studies using GPT-4o, OpenAI o1, and DeepSeek-R1 on ERMB. We report the type of failure and length ratio for each example. GPT-4o shows the lowest performance while OpenAI o1 surpasses all other models.

G Prompts Used for Zero-shot and Few-shot Prompting of LLMs

In our experiments using LLMs, we employ zero-shot and five-shot prompting. Our prompts are structured as shown in the examples below. Figure 7 is the prompt template used for zero-shot prompting and Figure 6 is the prompt template used for five-shot prompting. The same prompt is used across all LLMs. For the five-shot examples used in ERMB, since minimized answers are not available, we use manually simplified examples based on rules. Different examples are used for RMC and ERMB. For ERMB, the examples are composed to include two regexes with depth 4, two regexes with depth 5, and one regex with depth 6. Each prompt contains a brief explanation of formal regex and explicitly instructs the model not to use practical regex notation while generating only the regex output without additional explanations.

H Evaluation on LLM Fine-tuning and Chain-of-Thought Prompting

In this section, we explore methods to enhance the capabilities of open-source LLMs through fine-tuning and advanced prompting strategies. We report the experimental results on RMC in Table 12 and on ERMB in Table 13.

H.1 Fine-tuning Open-Source LLMs

We first investigate whether fine-tuning can bridge the gap between open-source models and the task requirements. We fine-tuned Qwen2.5:7B and Llama3.1:8B on the RMC corpus. In order to ensure efficient training, we utilized LoRA with a rank of 16 and alpha of 16. The optimization was performed using AdamW with a weight decay of 0.01 and a cosine learning rate schedule with learning rate $= 1e^{-4}$ and 5 warmup steps. The models were trained for 10 epochs with a batch size of 16. For evaluation, we employed zero-shot prompting to align with the training data format. As shown in Table 12, fine-tuning yields remarkable improvements on the RMC dataset. Both Llama3.1:8B and Qwen2.5:7B achieve near-perfect equivalence scores (100.00% and 98.93%, respectively) and significantly reduced length ratios compared to the scratch-trained BART and T5 baselines. However, Table 13 reveals a critical limitation regarding generalization. When evaluated on the ERMB dataset, which consists of longer and more complex regexes, the performance of fine-tuned models drops drastically (e.g., Qwen2.5:7B equivalence drops to 1.23%). This indicates that while fine-tuned models excel at distributions similar to their training data, they fail to generalize to longer inputs, suggesting they learn statistical patterns rather than the underlying logic of minimization.

H.2 Chain-of-Thought Prompting

Next, we examine the impact of Chain-of-Thought (CoT) prompting on the reasoning capabilities of larger models, specifically Qwen2.5:72B and Llama3.3:70B. We utilized both zero-shot and five-shot settings, explicitly instructing the models to "think step by step." We post-processed the outputs to extract the final regex from the generated reasoning traces. The results in Table 12 demonstrate that CoT prompting can enhance performance. For instance, Llama3.3:70B shows a significant improvement in the zero-shot setting, with equivalence increasing from 13.45% to 23.25%. However, despite gains of approximately 50% improvement in some cases, the overall performance remains significantly lower compared to the fine-tuned models on RMC or the proprietary models.

H.3 Analysis of Generalization and Reasoning Capabilities

The overall results reveal distinct characteristics of each approach. Pretrained open-source LLMs generally exhibit poor performance on regex minimization. Fine-tuned and scratch-trained models perform reasonably well on regexes of lengths similar to those seen during training but fail to generalize to longer inputs. Notably, although performance improves with larger model sizes, open-source models still fail to minimize effectively beyond a few simple heuristics. In contrast, our case study on proprietary LLMs demonstrates that some models can solve minimization tasks without additional training, and proprietary reasoning models, in particular, perform well on longer regexes. This suggests that the ability to handle longer contexts and complex minimization is closely tied to reasoning capacity, highlighting a substantial margin for improvement between current open-source and proprietary LLMs.

“““You are a regex minimization tool. Your task is to simplify a given formal regular expression over the fixed alphabet ‘a’, ‘b’, ‘c’, ‘d’. The minimized regex must have the smallest possible number of nodes in its expression tree while remaining functionally equivalent to the input.

The input regex follows these rules:
- Allowed operations: concatenation, union (‘+’), Kleene star (‘*’), and option (‘?’).
- Parentheses ‘()’ indicate operation precedence.
- The union operation is represented by ‘+’ instead of ‘|’.
- You must not use practical regex notations like ‘|’ (alternative) or Kleene plus (‘+’ for repetition).

Do not provide any explanations. Return only the minimized regex.

Examples:
Input: **[EXAMPLE1]**
Output: **[ANSWER1]**

Input: **[EXAMPLE2]**
Output: **[ANSWER2]**

Input: **[EXAMPLE3]**
Output: **[ANSWER3]**

Input: **[EXAMPLE4]**
Output: **[ANSWER4]**

Input: **[EXAMPLE5]**
Output: **[ANSWER5]**

Now simplify the following regex:
Input: **[REGEX]**
Output:”””

Figure 6: Prompt for five-shot regex minimization using LLMs. We utilize LLMs to inference the minimal regex ([REGEX]) along with five example-answer pairs ([EXAMPLE 1-5], [ANSWER 1-5]) into this prompt template.

“““You are a regex minimization tool. Your task is to simplify a given formal regular expression over the fixed alphabet ‘a’, ‘b’, ‘c’, ‘d’. The minimized regex must have the smallest possible number of nodes in its expression tree while remaining functionally equivalent to the input.

The input regex follows these rules:
- Allowed operations: concatenation, union (‘+’), Kleene star (‘*’), and option (‘?’).
- Parentheses ‘()’ indicate operation precedence.
- The union operation is represented by ‘+’ instead of ‘|’.
- You must not use practical regex notations like ‘|’ (alternative) or Kleene plus (‘+’ for repetition).

Do not provide any explanations. Return only the minimized regex.

Input: **[REGEX]**
Output:”””

Figure 7: Prompt for zero-shot regex minimization using LLMs. We utilize LLMs to inference the minimal regex ([REGEX]) into this prompt template.

Approach	Model	Size	Shot	Min. (↑)	Equiv. (↑)	Ratio (↓)
Training	BART	139M	-	**85.89**±0.07	**87.12**±0.14	**79.66**±0.02
	T5	223M	-	85.56±1.13	86.71±1.16	79.72±0.15
Finetuning	Llama3.1	8B	-	**99.99**±0.00	**100.00**±0.00	**77.94**±0.00
	Qwen2.5	7B	-	98.93±0.19	98.93±0.19	78.15±0.04
Prompting	Llama3.1	8B	Zero	1.72	5.88	99.39
			Five	3.95	10.82	98.54
	Llama3.3	70B	Zero	13.45	24.27	97.33
			Five	21.15	39.82	94.30
	Qwen2.5	7B	Zero	7.63	16.51	98.91
			Five	6.75	15.66	98.17
	Qwen2.5-coder	7B	Zero	10.48	18.43	97.85
			Five	10.67	24.69	97.26
	Qwen2.5	72B	Zero	20.31	**54.52**	95.37
			Five	**23.06**	54.33	**93.09**
CoT	Llama3.3	70B	Zero	23.25	37.79	93.31
			Five	29.07	41.82	92.48
	Qwen2.5	72B	Zero	19.09	44.41	91.82
			Five	**31.13**	**46.87**	**90.70**

Table 12: Performance comparison on the RMC dataset. We evaluate the effectiveness of training from scratch, fine-tuning, standard prompting, and Chain-of-Thought (CoT) prompting across various model sizes. Fine-tuned models demonstrate near-perfect performance on in-distribution data.

Approach	Model	Size	Shot	Equiv. (↑)	Ratio (↓)
Training	BART	139M	-	**0.08**±0.01	**99.93**±0.01
	T5	223M	-	0.05±0.01	99.95±0.01
Finetuning	Llama3.1	8B	-	0.86±0.09	99.44±0.04
	Qwen2.5	7B	-	**1.23**±0.07	**99.34**±0.01
Prompting	Llama3.1	8B	Zero	1.51	99.96
			Five	0.26	99.99
	Llama3.3	70B	Zero	7.16	99.60
			Five	2.57	99.73
	Qwen2.5	7B	Zero	1.79	99.90
			Five	0.66	99.95
	Qwen2.5-coder	7B	Zero	3.28	99.85
			Five	1.36	99.95
	Qwen2.5	72B	Zero	**13.99**	**99.28**
			Five	6.44	99.50

Table 13: Generalization performance on the ERMB dataset. This table reports the results of trained, fine-tuned, and prompted models on longer and more complex regexes. Despite the high performance on RMC, fine-tuned models show limited generalization capabilities on ERMB.

I Type of Failures Details

I.1 Proportions of Type of Failures for All Models

In this section, we report the results of the type of failure analysis when evaluating all models we utilize on the RMC test set, as shown in Figure 8. Unlike Figure 2 in Section 5.4, this figure includes the Minimal category, providing a comprehensive view of the actual results across 50,000 evaluated regexes. As discussed earlier, the primary source of failures for most models is producing valid but inequivalent regexes. This pattern was especially evident in 7-8B models, where their smaller size likely makes it even more challenging to generate equivalent regexes. Among the LLMs, Qwen2.5:72B achieved the best performance. It notably exhibits stronger capability in generating equivalent regexes, and when provided with five-shot examples, the proportion of minimal regexes increased. Additionally, models with more than 70B parameters generally produce fewer invalid regexes. However, an interesting observation is the behavior of Qwen2.5-coder. Compared to other LLMs and even our trained models, Qwen2.5-coder produces significantly fewer invalid regexes. This behavior is likely due to its code-specialized nature, which may have exposed the model to a larger variety of practical regexes compared to other models. As a result, it appears to be more resistant to generating syntactically invalid outputs. However, its overall performance is not significantly better than other 7B or 8B models, highlighting the challenging nature of regex minimization.

I.2 Case Study on Type of Failures

Following Table 14 presents examples of failure types observed in the inference results of each language model. We report output examples for all LLMs as well as the trained BART and T5 models, with LLM results based on five-shot prompting. For post-processing, the symbol 'ǀ' is treated as a practical expression and replaced with '+', while transformation steps such as "->" and tags like "Answer:" are post-processed. As shown in the table, most syntax errors occur due to mismatched parentheses or missing operands after the '+' symbol. However, the '+' symbol is sometimes used as a Kleene plus unary operation, where x is a regex and $x^+ = xx^*$. Even if we interpret such cases as intended Kleene plus expressions, they still do not represent a valid minimization.

J Detailed Results of Consistency on Equivalent Classes

J.1 Experimental Results on Equivalent Regex Classes

In Section 5.4, we conduct an experiment under our research question to examine whether language models produce consistent results for equivalent regexes within the same equivalent class. Here, in Figure 9, we report the distribution of LLMs with more than 70B parameters. BART and T5 are excluded from the figure since their strong performance in consistently producing minimal regexes results in a trivial distribution. In the figure, the x-axis represents the number of regexes successfully minimized within a class of 20 equivalent regexes, while the y-axis indicates the number of classes where the model successfully minimized the corresponding x-amount of regexes. The sum of all y-values for a given model is 50, corresponding to the total number of sampled classes. As shown in the figure, the performance is generally poor, with most results concentrated in the 0-10 range. The models successfully minimized only about four regexes per class on average. This finding suggests that even within the same equivalent class, LLMs minimize some regexes successfully while failing to do so for others.

J.2 Case Studies on Failure of Minimizing Equivalent Regex Classes

Table 15 presents examples of LLM failure cases discussed in Section 5.4, reporting specific instances where models struggled with regex minimization. Here, $\sigma \in \Sigma$ represents a single character, while x and y denote regexes. As shown in the table, LLMs generally perform well in applying simplifying rules at the character level. However, as the expressions become more complex, failures in minimization occur more frequently. Notably, in some cases such as the final example for Llama, the model successfully simplifies the expression but fails to apply further optimizations even though additional reductions are possible. This observation suggests that LLMs tend to apply rules only within local contexts rather than considering the overall structure. Such limitations indicate that LLMs struggle to fully understand and process long regexes. Additionally, these cases of partially minimized outputs, where further simplification is possible, are one of the key motivations for conducting our recursive inference experiments.

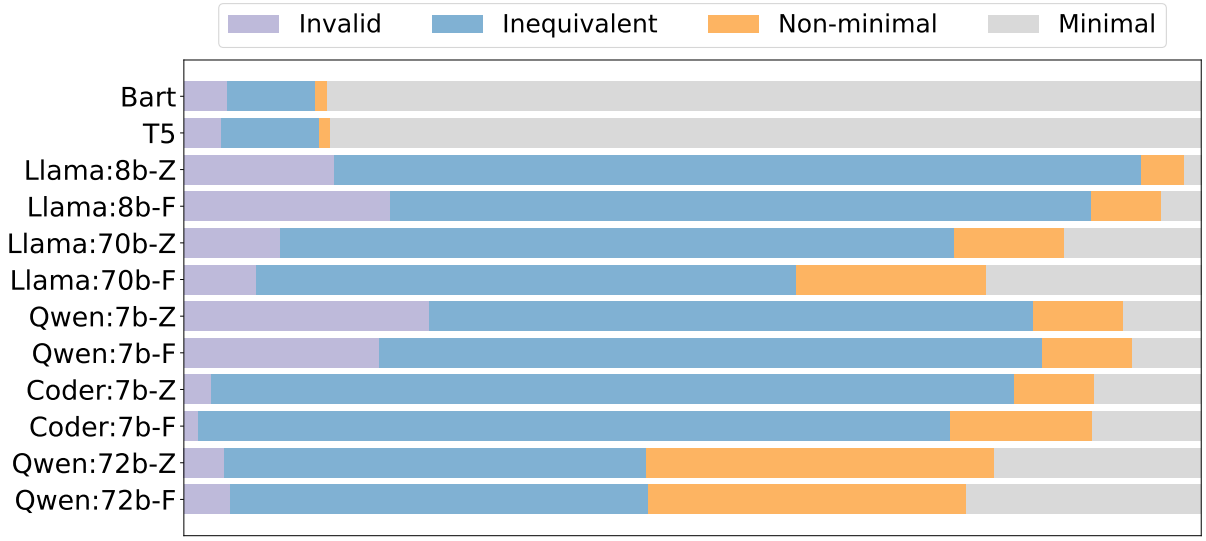


Figure 8: The proportion of each failure type on RMC test set. The plot shows the proportion of each failure type of all models we used. zero-shot and five-shot prompting are applied to each LLM. The types of failures are classified into three cases: invalid, inequivalent and non-minimal. Additionally, we report the success cases together in the plot by labeling them with "minimal" in order to visualize the total distribution.

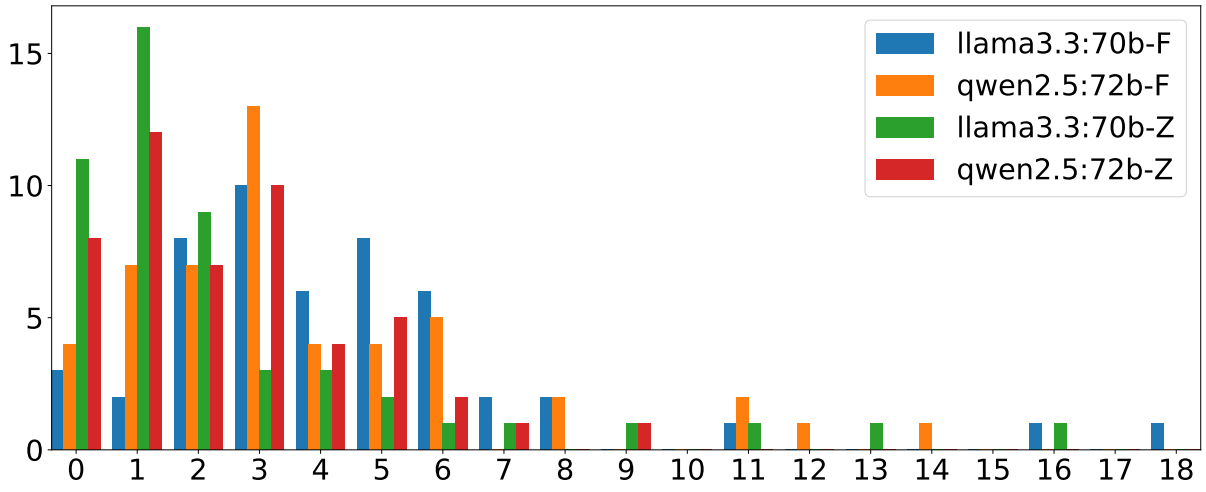


Figure 9: The distribution of the number of successfully minimized regexes in each equivalent class. We report zero-shot and five-shot prompting results on LLMs with more than 70B parameters. The x-axis represents the number of regexes in the same equivalent class that are successfully minimized. The y-axis represents the number of equivalent classes where the model successfully minimized in each class.

Model	Type of Failure	Input Regex	Output Regex	Minimal Regex
BART:139M	Invalid	$(((a+c)(a+d))((c+c)(d^?)))$	$((a(a+d)cd^?)$	$((a+c)(a+d)cd^?)$
	Inequivalent	$(((aa)+(a+b))((b+d)(dd)))$	$(c(a+b)^?(b+d)dd)$	$((b+aa^?)(b+d)dd)$
	Non-minimal	$((c(ab))+((b^?)+(c+d)))$	$((b+d+c(ab)^?)^?+c)$	$((b+d+c(ab)^?)^?)$
T5:223M	Invalid	$(((b+d)(c+d))((cd)^?))$	$c(b+d)(cd)^?)$	$((b+d)(c+d)(cd)^?)$
	Inequivalent	$(((aa)+(a+b))((b+d)(dd)))$	$((b+aa^?)(bd)dd)$	$((b+aa^?)(b+d)dd)$
	Non-minimal	$((b(b+c))+((a+d)(bb)))$	$(b(b+c)+(a+d)bb)$	$(bc+(a+d)^?bb)$
Llama3.1:8B	Invalid	$(((ab)+(ac))+((ab)+(bb)))$	$(ab+ac)+((ab)^?(b+))$	$(ac+(a+b)b)$
	Inequivalent	$((d+(b^*))+((b^?)+(d^?)))$	$((b+d)(b^?+d^?))$	$(d+b^*)$
	Non-minimal	$(((a^?)(d^*))((b+d)+(d+d)))$	$(((a^?)(d^*))((b+d)+d))$	$(a^?d^*(b+d))$
Llama3.3:70B	Invalid	$((d(b+b))((a+a)(ad)))$	$(db+)((a+)ad)$	$(dbaad)$
	Inequivalent	$(((ac)+(ac))+((d^?)+(dd)))$	$(ac+(dd)^?d^?)$	$((ac+dd^?)^?)$
	Non-minimal	$(((b^*)(cd))((c^?)+(d^?)))$	$(b^*(cd))(c^?+d^?)$	$(b^*cd(c+d)^?)$
Qwen2.5:7B	Invalid	$(((a^*)+(d^?))((ab)(b+b)))$	$(a^*d^?ab(b+)^?)$	$((d+a^*)abb)$
	Inequivalent	$(((ab)(b^?))+((ac)+(a+d)))$	$(ab(b^?)^?+a(c+d))$	$(d+a(c+bb^?)^?)$
	Non-minimal	$((a(a+d))+(c+(ab)))$	$(aa+ad+c+ab)$	$(c+a(a+b+d))$
Qwen2.5-coder:7B	Invalid	$(((ad)(bd))((ad)(d+d)))$	$(ad(bd)^?(d+))$	$(adbdadd)$
	Inequivalent	$((a(d+d))+(d+(a+a)))$	$(a+d)(a+d)$	$(a+a^?d)$
	Non-minimal	$((d(c+c))((c^*)+(c+d)))$	$(d(c+c))(c^*+c+d)$	$(dc(d+c^*))$
Qwen2.5:72B	Invalid	$((c(ab))+((b^?)+(c+d)))$	$(cab+)^?(b^?+c+d)$	$((b+d+c(ab)^?)^?)$
	Inequivalent	$(((b^*)(bc))+((c^?)(d^*)))$	$(b^*c^?+cd^*)$	$(b^*c+c^?d^*)$
	Non-minimal	$((a(d+d))+(d+(a+a)))$	$(a(d+d)+d+a)$	$(a+a^?d)$

Table 14: Type of failure case study on RMC test set. We evaluate our two LMs and LLMs with five-shot prompting only. All LLMs are instruction-tuned models. We report each failure case of each model associated with input, output and minimal regex.

Model	Simplifying Rules	Input Regex	Output Regex	Minimized
Llama3.3:70B	$\sigma+\sigma\to\sigma$	$(((a+d)+(d+d))+((c^?)+(c^?)))$	$(d+a+(c^?))$	O
	$\sigma^*\sigma^*\to\sigma^*$	$(((a^*)(a^*))+((a^?)+(b+c)))$	$(a^*+(b+c))$	O
	$x+x\to x$	$(((a^*)+(a^*))+((a+b)+(a+b)))$	$(a+b)$	X
	$x^*+xx\to x^*$	$(((b^*)+(c+c))+((c^*)+(cc)))$	$(c+(bc^*)^?)$	X
	$x^?+y^*\to x+y^*$	$(((a^?)+(b^*))+((a+c)+(d^?)))$	$(a^?+b^*)+(a+c+d^?)$	X
Qwen2.5:72B	$\sigma+\sigma\to\sigma$	$(((a+d)+(d+d))+((c^?)+(c^?)))$	$(a+d+c^?)$	O
	$\sigma^*\sigma^*\to\sigma^*$	$(((a^*)(a^*))+((a^?)+(b+c)))$	(a^*+b+c)	O
	$x+x\to x$	$(((a^*)+(a^*))+((a+b)+(a+b)))$	$(a^*+(a+b))$	X
	$x^*+xx\to x^*$	$(((b^*)+(c+c))+((c^*)+(cc)))$	$(b^*+c+c^?)$	X
	$x^?+y^*\to x+y^*$	$(((a^?)+(b^*))+((a+c)+(d^?)))$	$(a^?+b^*(a+c+d^?))$	X

Table 15: LLM performance case study on equivalent regex classes in the perspective of simplifying rules. We analyze the results of LLMs with more than 70B parameters. We choose five representative simplifying rules manually and report the behaviors of LLMs on those cases.

K AI Assistant Usage Statement

We utilize GPT-5.2 and Gemini-3 solely for the purpose of refining grammar, punctuation, and phrasing in this manuscript. These tools are not used to generate any of the core scientific content, such as the methodology, experiments, or conclusions. The authors verified the output and hold full responsibility for the final text.