NFA vs DFA Representations for the Regular Constraint in Lazy Clause Generation: A Comparative Evaluation

Ross Mackay[1]
Supervisor(s): Emir Demirović[1]**, Imko Marijnissen**[1]
[1]EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 21, 2026

Name of the student: Ross Mackay
Final project course: CSE3000 Research Project
Thesis committee: Emir Demirović, Imko Marijnissen, Andreea Costea

Abstract

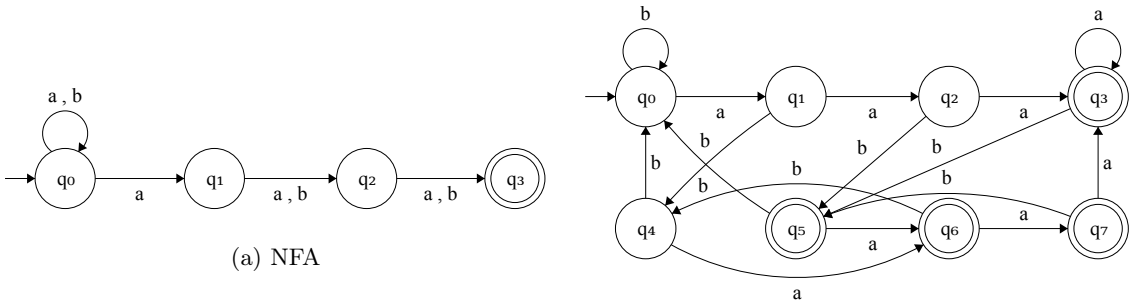
The regular constraint enforces that a sequence of variables forms a word accepted by a finite automaton, and is widely used in constraint programming (CP) for rostering, timetabling, and puzzle solving. While non-deterministic finite automata (NFAs) offer more compact representations than deterministic finite automata (DFAs), their relative performance within Lazy Clause Generation (LCG) remains underexplored. We adapt existing NFA and DFA-based LCG propagators, using a decomposition-based propagator as a baseline, and evaluate all three under identical solver conditions across instances stratified by DFA-to-NFA state blowup. At low blowup, NFA and DFA solve times are closely matched. At medium blowup, NFA remains faster by a factor of approximately three, and at high blowup it is approximately 22× faster than DFA, with DFA never clearly preferable. Although both compute the same filtering under generalised arc consistency, they can justify prunings differently, and thus are not guaranteed to explore identical search trees. Despite this potential for divergence, clause metrics are identical between NFA and DFA across every blowup bin, a result that may generalise more broadly but warrants further investigation. We also show that the currently used greedy explanations are close to optimal, within one percent of the exact minimum overall and within a few percent for any individual problem type measured. We thus provide the first direct evidence that NFA's compactness advantage carries over to propagation performance under LCG, establishing it as the default representation for the regular constraint in CP with LCG.

1 Introduction

Constraint Programming (CP) [1] solves combinatorial problems by modelling them as variables with finite domains and constraints over allowable value combinations. Since exhaustive search is typically infeasible, CP solvers rely on constraint propagation to remove values that cannot appear in any feasible solution. The effectiveness of a CP solver therefore depends on the strength and efficiency of its propagators.

The `regular` constraint, introduced by Pesant [2], enforces that a sequence of variables forms a word accepted by a finite automaton. Rather than decomposing a structured pattern into simpler constraints, `regular` propagates over the automaton directly, achieving stronger pruning than decomposition typically provides [2]. This makes it well suited to compact, expressive modelling of structured combinatorial problems, and is widely used in rostering [3], timetabling [4], and puzzle solving [5]. It can also encode other global constraints such as `slide`, `stretch`, and `pattern` [2, 6].

Propagation for the `regular` constraint has evolved since Pesant's original DFA formulation [2], with improvements in memory usage and runtime efficiency through techniques like bit-vector representations [5] and incremental propagation [7]. Additionally, NFAs have been adopted as a more compact alternative to DFAs [8], motivated by the potential exponential blow-up of DFAs derived from regular expressions [9], demonstrated in Figure 1. Furthermore, applying Lazy Clause Generation (LCG) [10] to the `regular` constraint has led to order-of-magnitude reductions in solve time [11]. Introduced by Ohrimenko et al. [10] and re-engineered by Feydy and Stuckey [12], LCG augments constraint propagation with clause learning by requiring propagators to produce explanations for their inferences, which may be reused as learned clauses during conflict-driven search.



(a) NFA

(b) DFA

Figure 1: An NFA and its minimal DFA both accepting strings over $\{a, b\}$ whose third-to-last symbol is 'a'. The NFA needs four states, the minimal DFA needs eight, a blowup ratio of two.

Despite these developments, the relative performance of NFA and DFA representations of the `regular` constraint in LCG solvers remains underexplored. Although both achieve generalized arc consistency and are therefore equivalent in inference strength, how the choice of representation and its compactness affects runtime efficiency and explanation quality within LCG remains unclear. Additionally, empirical studies of NFA-based `regular` propagation within LCG remain limited to the Nonograms benchmark family [13].

In this paper, we investigate how NFA and DFA representations of the `regular` constraint behave within LCG, examining when each is preferable. We adapt existing NFA and DFA-based propagators to accept a formal automaton tuple rather than a regular expression, and evaluate them against a decomposition-based baseline across instances stratified by blowup ratio. Concretely, we address three research questions: how the representation affects propagation efficiency, including per-call rebuild and construction overhead (RQ1), how it affects learned clause quality (RQ2), and how far the greedy explanations sit from the minimum-cardinality optimum (RQ3).

Our experiments show that at low blowup the two representations are closely matched. NFA remains faster by a factor of approximately three at medium blowup, and at high blowup it is approximately 22× faster. Although both achieve the same filtering under generalised arc consistency, they can justify their prunings differently, and thus are not guaranteed to explore identical search trees. Despite this potential divergence, clause metrics are identical between NFA and DFA across every blowup bin. We also show the greedy explanations used throughout are close to optimal, within one percent of the exact minimum overall and within a few percent for any individual problem type measured, though whether this generalises beyond these problem types remains open. These findings support NFA as the default representation for the `regular` constraint in CP with LCG.

The remainder of this paper is structured as follows. Section 2 reviews related work on the `regular` constraint and LCG. Section 3 covers preliminaries on constraint programming, LCG, and the `regular` constraint. Section 4 establishes the theoretical foundations for comparing automaton representations in the `regular` constraint, then describes the propagator implementations, problem selection, and instance generation procedures used to test these predictions empirically. Section 5 presents the experimental results and discussion. Section 6 discusses responsible research, and Section 7 concludes the paper.

2 Related Work

Research on the `regular` constraint can be divided into two related directions. The first concerns the design of the propagation algorithms for DFAs and NFAs. The second focuses on LCG and its integration into finite-domain solvers through explanation-based propagation and clause learning.

2.1 - Regular Constraint: Pesant [2] introduced the `regular` constraint, enforcing that a sequence of variables forms a word accepted by a finite automaton. It provides a compact representation for structured sequencing constraints and is widely used in rostering [3], timetabling [4], and puzzle solving [5]. It can also encode other constraints such as `slide`, `stretch`, and `pattern` [2, 6].

Pesant's original propagator enforces generalized arc consistency (GAC) [14] using a layered graph constructed from a DFA [2]. Each layer corresponds to a variable position, with nodes representing reachable automaton states and edges encoding feasible transitions under current domains. Propagation prunes values using reachability and co-reachability analysis over this layered structure.

Subsequent work has improved the efficiency of the `regular` propagator through incremental propagation and bit-vector representations. Makeeva and Szymanek [7] introduced an incremental approach that maintains a single layered graph and updates only the affected layers when variable domains change, avoiding full reconstruction at each propagation step. Zhen et al. [5] further improved efficiency using bit-vector representations, encoding state sets and variable domains as machine-word bit masks, performing transition checks and domain filtering via fast bitwise operations and reducing memory usage through constant-time set operations.

A separate line of work targets the representation itself. Lagerkvist observed that DFA-based propagation can suffer from exponential blow-up when converting a regular expression into a deterministic automaton [9], motivating the use of NFAs, which typically provide more compact representations. Cheng et al. [8] extended layered propagation to NFAs using the same construction as the deterministic case, at the cost of a denser transition structure that increases propagation overhead.

2.2 - Lazy Clause Generation: LCG [10] combines the inference power of constraint propagation with the learning capabilities of clause-based search. In CP, propagators remove values from variable domains that are inconsistent with a constraint [1]. In LCG, propagators must additionally justify each removal with an explanation, which is a set of variable assignments that together imply the removed value is infeasible. When a conflict is reached, the relevant explanations are combined into a learned clause, or nogood, recording why a particular combination of assignments failed and allowing the solver to avoid repeating the same failure later in the search.

The LCG framework was introduced by Ohrimenko et al. [10], who combined constraint propagation with conflict-driven clause learning to enable non-chronological backtracking over finite-domain (FD) problems. Feydy and Stuckey [12] later re-engineered this architecture, which had embedded a FD propagation engine inside a SAT (Boolean satisfiability) solver, by instead embedding a SAT solver inside an FD solver, making it easier to integrate arbitrary constraint propagators and yielding a more flexible and efficient system. While LCG has been applied to several global constraints, including `alldifferent` [15] and various scheduling constraints [12], explanation generation remains underexplored relative to propagation.

The `regular` constraint has proven particularly suitable for LCG because its layered propagation structure provides a natural basis for explanation generation. Gange et al. [11] incorporated explanations into `regular` propagation by working over multi-valued decision diagrams (MDDs), gener-

ating explanations greedily to favour speed over minimality. More recently, Gvozdiovas et al. [13] extended explanation-based propagation to NFA representations of the `regular` constraint. Their work demonstrates that NFA-based propagators can be competitive within LCG frameworks, particularly on structured benchmark classes such as Nonograms. However, how automaton representation affects explanation structure and overall solver performance remains an open question.

2.3 - Research Gaps: Despite progress in both propagation and explanation generation, the relative performance of NFA and DFA-based propagators for the `regular` constraint within LCG remains underexplored. Both representations enforce generalised arc consistency and are therefore equivalent in inference strength, yet how representations and their compactness affects runtime efficiency, propagation overhead, and explanation quality in practice remains unclear. Furthermore, empirical evaluation of NFA-based `regular` propagation within LCG has been limited to the Nonograms benchmark family [13], leaving broader performance characteristics unknown.

This paper addresses these gaps through a comparative evaluation of NFA, DFA, and decomposition-based `regular` propagators within a unified LCG framework, assessing both propagation efficiency and learned clause quality across a diverse set of benchmark instances.

3 Preliminaries

This section begins by outlining the CP framework, before describing LCG and its explanation requirements. We then define the `regular` constraint and its automaton-based representations, and explain propagation and explanation generation under both DFA and NFA representations.

3.1 - Constraint Programming: Formally, a *Constraint Satisfaction Problem* (CSP) is a tuple $\Pi = (\mathcal{X}, \mathcal{D}, \mathcal{C})$, where $\mathcal{X} = \{X_1, \ldots, X_n\}$ is a finite set of variables, $\mathcal{D} = \{D_1, \ldots, D_n\}$ is a corresponding set of finite domains such that each X_i takes values from D_i, and $\mathcal{C}$ is a set of constraints, each restricting the permissible combinations of values over some subset $\{X_{i_1}, \ldots, X_{i_k}\} \subseteq \mathcal{X}$. A solution assigns each variable a value from its domain such that every constraint in $\mathcal{C}$ is satisfied. A *Constraint Optimisation Problem* (COP) additionally specifies an objective $f : \mathcal{X} \to \mathbb{R}$ to be minimised or maximised subject to $\mathcal{C}$.

CP solvers prune the search space through *constraint propagation*. Each constraint $C \in \mathcal{C}$ is associated with one or more *propagators*, which are functions that remove values from variable domains that cannot participate in any solution given the current domains of the other variables. These propagators are applied repeatedly until no further removals are possible or a domain becomes empty, a state known as a *fixed point*. *Generalised Arc Consistency* (GAC) [14] is the strongest form of pruning a propagator can achieve for a single constraint in isolation, and guarantees that every value remaining in a domain is consistent with at least one solution to that constraint.

Propagation alone does not guarantee a solution is found. When a fixed point is reached with unassigned variables remaining, the solver *branches* by selecting a variable, partitioning its domain, and recursing on each sub-problem. Failures, detected when a domain becomes empty, trigger *backtracking* to a previous branch point.

3.2 - Lazy Clause Generation: Standard CP solvers discard all propagation work performed along a failed branch, allowing similar conflicts to be rediscovered repeatedly throughout search, with no mechanism to generalise from past failures. LCG addresses this by integrating *conflict-driven clause learning* (CDCL) [16] directly into CP, enabling the solver to learn from failures

and avoid rediscovering identical conflicts during search [10, 12]. Alongside the standard domain representation $\mathcal{D}$, LCG represents atomic constraints such as $[[x_i = v_j]]$ and $[[x_i \leq v_j]]$ as Boolean literals, introduced lazily during propagation or conflict analysis [10].

In LCG, every domain reduction performed by a propagator must be accompanied by an *explanation*, which is a justification for why that removal was necessary. Formally, an explanation is a 2-tuple (X, y) representing the implication $x_1 \wedge \cdots \wedge x_k \rightarrow y$, where $X = \{x_1, \ldots, x_k\}$ is a set of literals (the *antecedent*) encoding the domain events that forced the removal, and y (the *consequent*) is a single literal encoding the removed value [10].

When a contradiction is reached, the solver analyses the accumulated explanations to produce a *learned clause*, or *nogood*, characterising the conflict. This nogood is added to the solver's clause database and can trigger *non-chronological backtracking*, where the solver jumps back to the earliest branch point implicated in the conflict, potentially skipping large portions of the search tree.

The quality of the learned clause produced at a conflict depends directly on the quality of its contributing explanations, and has a significant impact on solver performance. Two common measures of explanation quality are nogood length and *Literal Block Distance* (LBD) [17], where shorter nogoods and lower LBD both indicate stronger, more focused pruning. Generating small, accurate explanations is therefore a central design challenge for LCG propagators [12].

3.3 - The Regular Constraint: A *finite automaton* is a tuple $A = (Q, \Sigma, \delta, q_0, F)$, where Q is a finite set of states, Σ is a finite alphabet corresponding to the values variables in the constraint can take, δ is a transition function, $q_0 \in Q$ is the initial state, and $F \subseteq Q$ is the set of accepting states. A word $w = w_1 \cdots w_n \in \Sigma^*$ is *accepted* by A if there exists a sequence of states $q_0, \ldots, q_n$ such that each $q_i \in \delta(q_{i-1}, w_i)$ and $q_n \in F$. The distinction between a *deterministic* (DFA) and *non-deterministic* (NFA) finite automaton lies solely in the transition function δ. For a DFA, $\delta : Q \times \Sigma \rightarrow Q$ maps each pair to exactly one successor, whereas for an NFA, $\delta : Q \times \Sigma \rightarrow \mathcal{P}(Q)$ maps each pair to a *set* of possible successors. Both accept the same class of languages, but NFAs can be exponentially more compact than equivalent DFAs [9], as shown in Figure 1.

The `regular` constraint, introduced by Pesant [2], is defined over a sequence of variables $X = \langle X_1, \ldots, X_n \rangle$ with domains $D_1, \ldots, D_n \subseteq \Sigma$ and a finite automaton A. It enforces that every complete assignment to X forms a word accepted by A. The constraint can be expressed using either a DFA or an NFA, since both represent the same class of languages.

3.4 - Propagating the Regular Constraint: Propagation for the `regular` constraint is built on a *layered graph* constructed from the automaton A and the variable sequence X [2] (Figure 2). The node set is $V = \{(i, q) \mid 0 \leq i \leq n,\ q \in Q\}$, where a node (i, q) represents automaton state q after processing the first i variables, and each layer corresponds to a position in the variable sequence. The edge set is $E = \{\, ((i, q),\ (i{+}1, q')) \mid \exists\, v \in D_{i+1} : q' \in \delta(q, v) \,\}$, so an edge from (i, q) to $(i{+}1, q')$ exists whenever some value $v \in D_{i+1}$ can transition A from q to q'. In other words, each edge represents a transition still available under X_{i+1}'s current domain.

Pesant's propagator [2] enforces GAC by computing *reachability* and *co-reachability* over this graph. A node (i, q) is *reachable* if there is a path from $(0, q_0)$ to (i, q), and *co-reachable* if there is a path from (i, q) to some accepting node (n, q_f) with $q_f \in F$. It follows that a node is on a valid accepting path if and only if it is both reachable and co-reachable.

Extending this propagator to NFAs uses the same construction, since E is defined via $q' \in \delta(q, v)$,

which holds whether $\delta(q, v) \in Q$ or $\delta(q, v) \in \mathcal{P}(Q)$. However, an NFA transition can yield multiple successors for a single value, so its layered graph generally has higher arc density than the equivalent DFA's, increasing propagation overhead [8]. Nonetheless, both propagators enforce GAC and so achieve identical *inference strength*, removing exactly the same values from each domain.

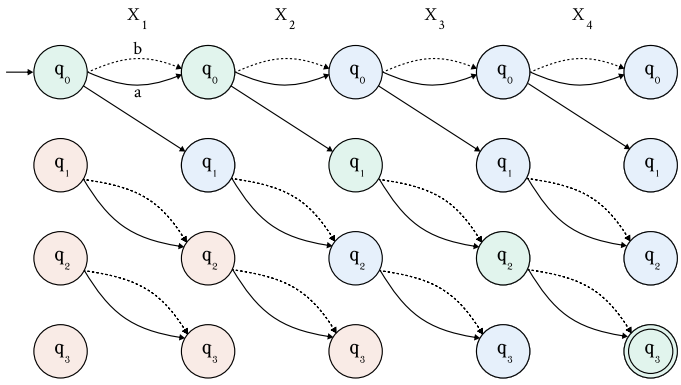


Figure 2: Layered graph for the NFA in Figure 1, coloured by reachability and co-reachability. Green is both, blue is reachable only, red is neither. In this example, every co-reachable node is also reachable, so none are co-reachable only. The corresponding DFA layered graph is in Appendix A.

3.5 - Explanations for the Regular Constraint: Within an LCG framework, the layered graph provides a natural structure for generating explanations. When a value v is removed from D_i, the propagator must produce an explanation justifying this removal in terms of prior domain reductions.

Explanation generation proceeds as a forward traversal from $(0, q_0)$ through the layered graph [11, 13], identifying previously-removed values whose absence collectively blocks every accepting path through the edge labelled v at layer i. At each layer other than i, if an edge is not currently in the domain but leads to a node from which an accepting state is reachable via v at layer i, that removed value is added to the explanation. The traversal continues along the target edge and any edge not yet blocked by an explanation literal. The result is a set of prior removals that, if reversed, would restore an accepting path through v at position i.

The automaton representation affects the structure and size of these explanations. Under a DFA, each state-value pair has exactly one successor, so each step of the traversal is unambiguous and an explanation corresponds to a single chain of blocking transitions. Under an NFA, a state-value pair may contribute multiple successors simultaneously, so the traversal must account for a wider set of reachable nodes per layer, and an explanation may need to block several parallel transitions at once. Whether this leads to larger or smaller explanations in practice, and how this affects clause learning and search performance, is a central empirical question of this paper.

4 Theoretical Analysis and Methodology

This paper sets up a controlled comparison in which the automaton representation is the only variable that differs between propagators, so any difference in solver behaviour is solely attributable to

the representation. We first establish theoretical foundations for the expected differences between representations and the explanations they generate, before describing the propagator implementations, problem selection, and instance generation procedures.

4.1 - Theoretical Propagation Differences: As established in Section 3.4, both propagators reduce the `regular` constraint to reachability in a layered graph (V, E) over the automaton A and the n-variable sequence, differing only in the automaton representation supplied. For a DFA, $\delta : Q \times \Sigma \to Q$ assigns exactly one successor per state-symbol pair, whereas for an NFA, $\delta : Q \times \Sigma \to \mathcal{P}(Q)$ assigns a set. Writing this as $s = \max_{q,a} |\delta(q, a)|$, the maximum number of successors any state-symbol pair can reach, the DFA is the case $s = 1$ and the NFA is the general case $s \geq 1$.

Both construction and a single propagation call run in $O(|V|+|E|)$. The two implementations share this pipeline, so the difference between them lies in the node count $|V|$ and edge count $|E|$ of their respective layered graphs. The node count is $|V| = O(n|Q|)$ in both cases, though $|Q|$ itself differs between the representations, since an NFA can be exponentially more compact than an equivalent DFA (Section 3.3). For the DFA, each state-symbol pair has exactly one successor, so each state contributes at most $|\Sigma|$ outgoing arcs per layer, giving $|E_{\text{DFA}}| = O(n|Q||\Sigma|)$. For the NFA, each state-symbol pair has at most s successors, giving $|E_{\text{NFA}}| = O(n|Q||\Sigma|s)$.

These bounds expose a trade-off between the two representations' worst cases. In the worst case, every state-symbol pair of the NFA transitions to every state, so $s = |Q_{\text{NFA}}|$ and the arc count $|E_{\text{NFA}}| = O(n|Q_{\text{NFA}}|^2|\Sigma|)$ is quadratic in its state count. The DFA has $s = 1$ by definition, so $|E_{\text{DFA}}| = O(n|Q_{\text{DFA}}||\Sigma|)$ is only linear in its state count, but $|Q_{\text{DFA}}|$ may itself be exponentially larger than $|Q_{\text{NFA}}|$, up to $|Q_{\text{DFA}}| = 2^{|Q_{\text{NFA}}|}$. The two representations therefore trade a quadratic dependence on a small state count against a linear dependence on a potentially exponentially larger one, and Section 5 investigates how this theoretical trade-off plays out on real problem instances.

4.2 - Theoretical Explanation Differences: Propagation efficiency is not the only factor that determines solver performance in LCG. Every removal must also carry an explanation, and explanation quality directly feeds into learned clause strength. When a value v is removed from D_i, the forward traversal described in Section 3.5 greedily collects the prior removals whose absence collectively blocks every accepting path through the edge labelled v at layer i. An explanation's length is the number of such removals the traversal records.

This length depends on the shape of the graph being traversed. Under a DFA, $s = 1$, so each value leads to exactly one successor, and the routes the traversal explores do not branch into parallel copies of themselves. Each candidate route towards an accepting state is either alive or blocked at a single identifiable point, and when blocked, one recorded literal accounts for it before the traversal continues to the next route. Under an NFA, a single value at a single node can lead to several successors at once. Each successor opens its own family of onward routes, and the removal of v at layer i is justified only once every one of them is blocked. The traversal must therefore chase down each parallel family, and since each may be blocked at a different point, each may contribute its own literal. More branching means more places a blocking removal must be recorded, potentially lengthening NFA explanations.

The NFA's compactness pulls in the other direction. With one NFA state potentially representing many DFA states, each layer of its layered graph may contain fewer nodes. Routes that pass through distinct nodes in the DFA's layered graph are funnelled through a single shared node in the NFA's. Routes that overlap in this way are often blocked at the same point by the same removal, so one

literal can do the work that several would do in the DFA. Branching and compactness therefore pull NFA explanation length in opposite directions, and theory alone does not resolve which effect dominates. Section 5.4 settles this empirically by comparing nogood length and Literal Block Distance across both representations.

A second question concerns the quality of the explanations themselves, independent of which representation produces them. The greedy pass favours speed over minimality [11]. As a result, it may record a literal guarding a route that literals it adds later have already cut off, and the NFA's wider branching makes such redundancy more likely. The shortest valid explanation corresponds to a minimum-cardinality labelled cut on the layered graph, which is the smallest set of removed labels whose deletion leaves no accepting path through the target edge. Computing this is NP-hard in general [18], so how far the greedy explanation sits from the optimum, and whether closing that gap is worthwhile, is an open question.

We answer this with an exact branch-and-bound algorithm that searches over these labelled cuts directly. Each candidate is a subset of the previously removed labels, and the algorithm branches on whether to include each label in the cut, using path search to test whether a candidate cut is feasible, that is, whether it blocks every accepting path through the target edge. The greedy explanation serves as the initial best-known solution, so the result is never larger, and because this solution is replaced only on strict improvement, the procedure returns exactly the greedy set whenever that set is already minimal. The two therefore agree in literals, nogoods, and search except where the minimum is strictly smaller, so running the solver with this algorithm substituted for the greedy method serves purely to measure the greedy-to-optimal gap, rather than as a practical replacement for it. The algorithm, proof of minimality, and a demonstrative example are given in Appendix B, and Section 5.5 reports the resulting gap and whether exact minimisation is worth pursuing.

4.3 - Propagator Implementations: The NFA and DFA propagators are based on the implementations of Gvozdiovas et al. [13], adapted for this work. This adaptation did not change the propagation procedures, but consisted of reimplementing them in the Pumpkin solver [19] and altering their interface to accept a formal automaton tuple $(Q, \Sigma, \delta, q_0, F)$, as defined in Section 3.3, rather than a regular expression. Automata are constructed prior to solving, so that each propagator receives an automaton for the same language in its respective representation. This helps isolate automata representation as the primary variable, so any performance difference is attributable to it rather than construction costs. The decomposition-based propagator is MiniZinc's default decomposition[1], used unmodified as a third baseline, decomposing the `regular` constraint into primitive constraints and serving as a reference point.

4.4 - Problem Selection and Instance Generation: The central axis of the evaluation is the *blowup ratio*, the number of states in the minimal DFA divided by that in the reduced NFA, summed across each instance's constraints and expressed as a ratio of these totals. It directly captures the compactness difference between the representations, and since a more compact automaton yields a sparser layered graph (Section 3.4), it is the natural axis for studying how compactness affects propagation efficiency and clause strength. Instances are stratified into three bins: low ($< 2\times$), medium ($2\times$–$10\times$), and high ($> 10\times$). As existing benchmarks were not designed with blowup ratio as a control variable, instances are generated rather than drawn from the literature, ensuring balanced coverage across all three.

[1] `https://github.com/MiniZinc/libminizinc/blob/master/share/minizinc/std/fzn_regular.mzn`

The three problem types are nonograms, polyominoes, and regex optimisation, each modelled exclusively with `regular` constraints. Nonograms are a well-established benchmark for the constraint [20] and express naturally as NFAs, their row and column constraints being non-deterministic patterns; instances follow the logic of Gvozdiovas et al. [13]. Polyominoes share this property, with tile constraints expressible as NFAs [21], and are generated using a generator by Lagerkvist[2]. For both problem types, DFA minimisation exploits structural symmetries, often yielding low blowup ratios and sometimes a DFA smaller than the NFA, so both populate the low bin and cannot cover the medium and high regimes. The regex type was introduced for that coverage, where the structural difference between representations is most pronounced.

The regex type is based on a classic construction causing exponential DFA blowup relative to the NFA [9], for which we developed a dedicated generator. Each instance encodes two `regular` constraints over a finite alphabet, a pattern-matching constraint whose NFA grows linearly while its DFA grows exponentially, and a cardinality constraint. Instances are solved as COPs rather than CSPs, since without an objective a propagator finds a valid solution immediately, generating too little backtracking to differentiate the propagators. The objective maximises a random weighted sum of assignments, with weights drawn from a fixed seed and shared across the NFA and DFA files for each instance. Being designed to provoke DFA state explosion, the construction is artificial and may not represent practical instances, as discussed in Section 5.6.

Instances are generated from a single fixed random seed, with duplicates, defined as instances sharing identical parameters, skipped to ensure all problem instances are unique. For each constraint, the NFA is built as a Glushkov NFA [22], chosen for its one state per symbol occurrence, then reduced via right-bisimulation [23], merging states with equivalent future behaviour. The minimal DFA is built via subset construction [24] and minimisation [25]. Since exact NFA minimisation is PSPACE-complete [26], Glushkov NFA construction followed by right-bisimulation serves as a tractable approximation. Both automata therefore have redundancy removed as far as tractable, rather than comparing a minimised DFA against an unreduced NFA, which would understate the blowup. Each bin is then sub-sampled to the size of the smallest bin by a seeded draw, so every bin contributes equally, yielding 141 instances split evenly across the three bins (47 each). Each retained instance is run against all three configurations under identical solver conditions with a 600-second per-instance limit.

Appendix C provides further detail on problem selection and instance generation procedures, including examples of each problem type.

5 Experimental Results

With the NFA, DFA, and decomposition-based propagators evaluated under identical conditions, the results consistently favour NFA across blowup regimes. NFA propagation is fastest at medium and high blowup, reaching roughly 22× faster than DFA at high blowup, while at low blowup the three are closely matched. As no regime clearly favours DFA over NFA, this supports NFAs as the default representation for the `regular` constraint in LCG. The greedy explanations used throughout are also close to optimal, within one percent of the exact minimum overall and within a few percent for any individual problem type measured. The following sections examine propagation efficiency,

[2] `https://github.com/zayenz/minizinc-pentominoes-generator`

including per-call rebuild cost and automaton construction overhead (RQ1), clause quality (RQ2), and the gap between greedy and minimum-cardinality explanations (RQ3) in detail.

Experiments are run on an HP ZBook Power G10 (Windows 11, Intel Core i7-13700H, 16 GB RAM). The solver is Pumpkin [19], invoked via MiniZinc with input-order, smallest-value-first branching and a 600-second limit, ensuring identical branching across configurations so differences are attributable to the propagator. All aggregates are geometric means, suited to runtimes spanning orders of magnitude. Solve-time tables are restricted to the commonly-solved set (solved by all three propagators), while clause-quality metrics exclude only instances on which the NFA or DFA propagator timed out.

5.1 - Solve Time vs Blowup Ratio: At high blowup NFA reaches a geometric mean of 0.54s against 12.12s for DFA, a factor of roughly 22×, with decomposition close behind at 11.15s. NFA also leads at medium blowup by a factor of roughly 3×, while at low blowup NFA and DFA are within ten percent of one another. Full results are in Table 1 and the left panel of Figure 3.

Blowup Bin	**Solved** (n)	**Timeout** (n)	**NFA (s)**	**DFA (s)**	**Decomp (s)**
Low	35	12	2.99	2.73	3.63
Medium	44	3	0.49	1.69	2.92
High	40	7	0.54	12.12	11.15
Total	119	22	0.86	3.77	4.88

Table 1: Geometric mean solve time (s) by blowup bin and propagator, over the commonly-solved instances, where blowup denotes the ratio of DFA to NFA state counts, binned as low ($< 2\times$), medium ($2\times$–$10\times$), and high ($> 10\times$).

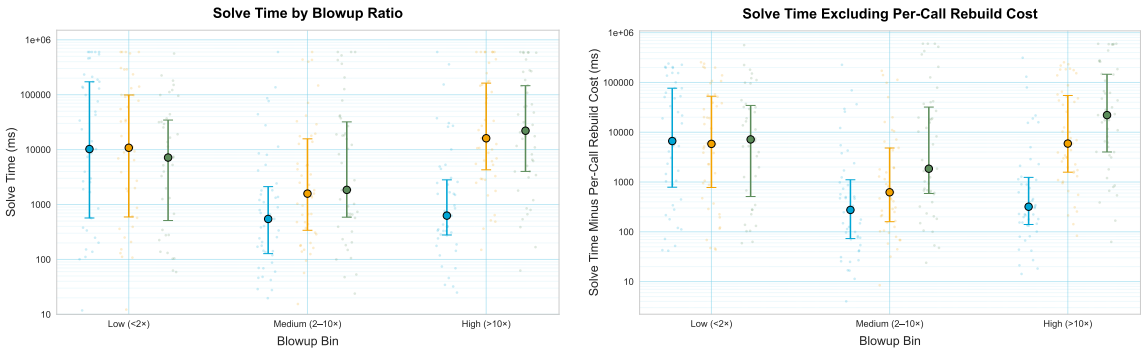


Figure 3: Solve time by blowup bin, over the commonly-solved instances (NFA: Blue, DFA: Orange, Decomposition: Green). The left panel shows full solve time, the right removes the per-call rebuild cost. Points show individual instances, while large markers show the geometric mean and interquartile range.

These results align with the propagation analysis of Section 4.1. At medium and high blowup the DFA traverses a far larger layered graph per call, an overhead that compounds, while at low blowup the state-count difference is too small to give either a consistent advantage. Across all considered instances the NFA's maximum successor count s stays far below its worst case of $|Q_{\text{NFA}}|$, so the NFA never approaches its quadratic upper bound and remains close to linear in its small state count.

The comparison therefore reduces to the difference in state count, the exponentially larger $|Q_{\mathrm{DFA}}|$ captured by the blowup axis, which drives the observed gap. Decomposition degrades sharply with blowup, winning no regime.

Problem Type	**Solved** (n)	**Timeout** (n)	**NFA (s)**	**DFA (s)**	**Decomp (s)**
Nonogram	24	0	0.62	0.61	1.18
Polyominoes	10	12	130.55	87.45	46.20
Regex	85	10	0.52	4.35	5.60
Total	119	22	0.86	3.77	4.88

Table 2: Geometric mean solve time (s) by problem type, over the commonly-solved instances.

Table 2 breaks down solve time by problem type, confirming a strong problem-type effect. Nonograms are fast and closely matched across all three configurations, while the regex class drives the NFA advantage, dominating the medium and high bins. Polyominoes run against the trend, with DFA faster than NFA (87.45s versus 130.55s). For polyominoes instances, DFA minimisation exploits the structural symmetries of the tile constraints (Section 4.4), yielding fewer states than the reduced NFA and a smaller, sparser layered graph (Section 4.1). Since every DFA is simply an NFA with $s = 1$, the NFA-based propagator accepts either representation unmodified, so this result argues only for supplying the smaller automaton, in this case the DFA, not for retaining DFA as a separate representation.

5.2 - Per-Call Rebuild Cost: Both automaton propagators rebuild their layered graph on every call, neither being incremental, whereas decomposition does not. The right panel of Figure 3 and Table 5 (Appendix D.1) report solve time with this cost removed, separating the cost of representation from non-incremental reconstruction overhead. Timing the rebuild on every call inflates these absolute figures, so they are best read as a comparison rather than standalone times. Both representations are instrumented identically, so the relative ordering is unaffected.

The per-call rebuild cost explains much but not all of the DFA disadvantage. At high blowup the DFA's solve time falls from 12.12s to 4.49s and the NFA's from 0.54s to 0.28s, so even with rebuild cost removed from both the DFA remains about sixteen times slower, meaning the penalty is part rebuild and part intrinsic traversal over the larger graph. An incremental scheme [7] would avoid the rebuild, saving more on the larger DFA graph, but the DFA would still traverse more nodes and arcs per layer than the NFA at high blowup (Section 4.1), so that intrinsic cost would remain. Decomposition, which never rebuilds, is slowest at high blowup on equal footing (11.15s versus 4.49s), confirming its poor scaling comes from propagation count, not per-call overhead.

5.3 - Automata Construction and Minimisation Cost: For each problem the automata are constructed and minimised before solving, with construction time per pipeline step reported in Table 3. At low blowup NFA and DFA are comparable (1829.5ms versus 1730.9ms), both dominated by their respective reduction steps. At medium blowup both are fast, under 40ms in total. At high blowup they diverge sharply, with the NFA totalling 17.1ms against the DFA's 4138.2ms, almost all of it minimisation (4019.1ms), consistent with the potential exponential state growth between representations [9]. At high blowup, the DFA thus pays roughly four seconds to build a representation that is also slower to solve, offering no compensating benefit. More generally, when deploying either propagator in practice, construction cost forms part of the total pipeline cost and should be accounted for alongside pure solve time.

	NFA		DFA	
Bin	**Glushkov (ms)**	**Bisimulation (ms)**	**Subset Construction (ms)**	**Minimisation (ms)**
Low	96.0	1733.5	80.4	1650.5
Medium	4.0	9.4	3.6	29.5
High	4.5	12.6	119.1	4019.1
Total	11.9	58.8	32.6	580.6

Table 3: Geometric mean automaton construction time (ms) by blowup bin and pipeline step, over all instances. NFA construction consists of Glushkov construction followed by right-bisimulation reduction. DFA construction consists of subset construction followed by minimisation.

5.4 - Clause Quality: While GAC equivalence guarantees identical inference strength, it does not guarantee identical explanations. Table 6 (Appendix D.2) reports nogood statistics by blowup bin, over instances on which neither NFA nor DFA timed out. In every bin the two are indistinguishable to the reported precision on all three metrics, nogood count included, so they learn identical clauses and traverse the same search tree, leaving per-call speed as their sole difference. This is consistent with Section 4.2, since divergence in learned clause quality would require the two automata to encode the language through sufficiently different supporting paths. The problem types evaluated here do not exhibit this, though this may be a property of the problem types themselves rather than the representations, and may not hold in general.

The cross-bin variation therefore reflects problem structure, not representation. Table 4 shows a problem-type effect, with nonograms giving longer nogoods (length 14.21) compared to polyominoes (7.60) and regex (5.56). As the low bin is dominated by nonograms and polyominoes and the higher bins by regex, the apparent fall in length with blowup merely tracks the shift in problem mix, with LBD likewise showing a problem-type effect rather than tracking blowup bin directly.

			NFA			DFA		
Problem Type	**Solved (n)**	**Timeout (n)**	**Nogoods**	**Avg. Length**	**Avg. LBD**	**Nogoods**	**Avg. Length**	**Avg. LBD**
Nonogram	24	0	78	14.21	4.80	78	14.21	4.80
Polyominoes	17	5	111	7.60	6.38	111	7.60	6.38
Regex	87	8	1013	5.56	5.56	1013	5.56	5.56
Total	128	13	481	6.83	5.52	481	6.83	5.52

Table 4: Geometric mean nogood statistics by problem type, over instances where neither NFA nor DFA timed out (Solved n). NFA and DFA are identical to the reported precision in every problem type, indicating they learn the same clauses and explore the same search tree.

5.5 - Greedy Versus Minimal Explanations: Section 4.2 left open how far the greedy explanation sits from the minimum-cardinality optimum. We measure this by substituting the exact branch-and-bound algorithm of Appendix B.1 for the greedy method, comparing per instance over the 85 instances both runs solved without an NFA or DFA timeout. As the minimum-cardinality explanation corresponds to an NP-hard labelled-cut problem, the exact search is far costlier than the greedy method's single forward pass, and so uses a 1200-second limit. As the goal is to measure explanation size rather than solver performance, solve times are not reported.

Table 7 (Appendix D.3) shows that greedy, despite costing only a single forward pass against an

NP-hard exact search, lands within one percent of the minimum-cardinality optimum in aggregate, with LBD essentially unchanged, and the small difference concentrated in one class. On regex it is exactly zero, so greedy is already minimal there, though as a synthetic class designed to maximise DFA state growth, this result alone is insufficient to draw broader conclusions. On nonograms, minimal explanations are about 3.7% shorter with around 3% fewer nogoods. Polyominoes are uninterpretable, with only two instances completing, their more complex layered graphs making explanation minimisation exponentially slower so that it times out even at 1200 seconds. The same intractability that makes exact explanation minimisation costly is thus visible in the missing data.

The greedy explanation is close to optimal wherever it can be measured, though instances where measurement is infeasible, such as polyominoes, are also those where the gap is most likely to be non-trivial. These figures are specific to the problem types evaluated and may not generalise, as the gap should widen only when the layered graph leaves the greedy approach room to record redundant literals, most likely for automata whose wider branching funnels many overlapping accepting paths through shared nodes, as described in Section 4.2. Even so, greedy sits at or near the optimum wherever it can be measured, suggesting it is a strong default that balances speed against minimality, though broader conclusions would require evaluation across a wider range of problem types.

5.6 - Limitations: Several limitations qualify these findings. Runtime optimisation improvements such as bit-vector representations [5] or incremental propagation [7] would lower absolute times, but since they apply equally to both representations they are unlikely to change the ordering. The relative performance is also sensitive to problem-type composition, as the NFA advantage is driven largely by the regex class dominating the higher bins. More broadly, the evaluation isolates representation across instances constructed to span the blowup axis, characterising the structural difference between representations rather than their behaviour on naturally occurring models, limiting how directly the results transfer to practice. The fixed input-order, smallest-value-first heuristic holds branching identical across configurations, but results may not carry over to activity-based search, restarts, or other heuristics that interact differently with learned clauses. Finally, blowup ratio is the only stratification axis, leaving other automaton properties such as alphabet size, transition density, and accepting-state count uncontrolled and possibly influential.

6 Responsible Research

Transparent and reproducible experimental work is fundamental to the scientific process. To support this, we have released a reproducibility package[3] containing all implementations, problem instances, experimental configurations, and evaluation scripts, maintained under version control throughout development. Instances were randomly generated using fixed random seeds and predefined parameter ranges to ensure reproducibility and limit selection bias.

We adhered to established academic codes of conduct, including honesty in reporting, proper attribution of prior work, and fairness in evaluation. Large language models were used for editing, rephrasing, and structural suggestions, with all core content, methodology, and conclusions written by the author. All experiments were conducted on generated instances, so no human participants or personal data were involved. The work is primarily methodological and has no direct societal impact, though improvements in constraint propagation may indirectly benefit optimisation tools in real-world applications.

[3]Reproducibility Package: `https://github.com/RossMackay365/regular-lcg-nfa-dfa-eval/tree/main`

7 Conclusions and Future Work

In this paper, we compared NFA, DFA, and decomposition-based propagators for the `regular` constraint in LCG, stratifying instances by the blowup ratio between DFA and NFA state counts. The central finding is that NFA propagation is substantially faster than DFA propagation at high blowup, achieving a geometric mean speedup of approximately 22×, driven by the smaller graph it traverses per call. At medium blowup NFA remains fastest by a factor of approximately three, while at low blowup the two representations are closely matched. Decomposition is similarly competitive at low blowup but degrades sharply as blowup grows, and is the slowest of the three across all blowup levels once per-call rebuild overhead is accounted for. Although NFA and DFA compute the same filtering under generalised arc consistency, they can justify their prunings via different supporting states, and thus are not guaranteed to learn the same clauses or explore the same search tree. In principle this could produce a difference in clause quality, but only where the two automata encode the language through sufficiently different supporting paths. This condition does not arise for the problem types evaluated here, as nogood length and LBD are identical to the reported precision across both representations in every blowup bin and problem type, leaving per-call propagation speed as their only practical difference. The greedy explanations are themselves close to optimal, with an exact minimum-cardinality search changing nogood length by under one percent overall and by only a few percent for any individual problem type measured, so the speed-over-minimality trade-off built into both propagators costs little in practice. There is no blowup regime in which DFA is clearly preferable to NFA on solve time, and at high blowup it is both far slower to solve and far more expensive to construct. The one exception by problem type is polyominoes, where DFA minimisation exploits structural symmetries to yield a smaller automaton than the reduced NFA. Since every DFA is simply an NFA with $s = 1$, the NFA propagator accepts either representation without modification. Whichever automaton happens to be smaller, DFA or NFA, can therefore be supplied to it directly and processed correctly, so this advantage is captured automatically, with no separate DFA-specific code path required. This supports NFA as the default representation for the `regular` constraint in CP with LCG.

These findings are subject to limitations detailed in Section 5. To summarise, the propagator implementations are unoptimised, the comparison is sensitive to problem-type composition, and the evaluation uses synthetic instances under a single fixed branching heuristic, with blowup ratio being the only controlled axis. None of these should overturn the relative ordering between representations, but each bounds how far the reported magnitudes generalise.

Several directions for future work remain open. First, current NFA propagators adapt DFA propagation structures without leveraging non-determinism or set-based transitions, leaving open whether this branching structure could improve propagation efficiency or yield more informative explanations, with effects on both performance and clause quality. Second, optimisation techniques developed for DFA-based propagators, such as bit-vector representations and incremental propagation, have not been explored for NFAs, and their effectiveness under non-deterministic transitions is unknown. Third, blowup ratio was the only stratification axis considered here, leaving open how other automaton properties, such as alphabet size, transition density, and accepting-state count, influence the trade-offs observed. Finally, the greedy-to-minimum gap could only be measured for nonograms and regex, since exact minimisation timed out on most polyominoes instances. Broadening this comparison would clarify whether the near-optimality observed here generalises, or whether problem classes with more intricate layered-graph structure exhibit a substantially larger gap.

References

[1] Francesca Rossi, Peter van Beek, and Toby Walsh, editors. *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*. Elsevier, 2006. ISBN 978-0-444-52726-4. URL https://www.sciencedirect.com/science/bookseries/15746526/2.

[2] Gilles Pesant. A regular language membership constraint for finite sequences of variables. In Mark Wallace, editor, *Principles and Practice of Constraint Programming - CP 2004, 10th International Conference, CP 2004, Toronto, Canada, September 27 - October 1, 2004, Proceedings*, Lecture Notes in Computer Science, pages 482–495. Springer, 2004. doi: 10.1007/978-3-540-30201-8_36. URL https://doi.org/10.1007/978-3-540-30201-8_36.

[3] Sven Löffler, Ke Liu, and Petra Hofstedt. The power of regular constraints in csps. In Maximilian Eibl and Martin Gaedke, editors, *47. Jahrestagung der Gesellschaft für Informatik, Digitale Kulturen, INFORMATIK 2017, Chemnitz, Germany, September 25-29, 2017*, volume P-275 of *LNI*, pages 603–614. GI, 2017. doi: 10.18420/IN2017_57. URL https://doi.org/10.18420/in2017_57.

[4] Sophie Demassey, Gilles Pesant, and Louis-Martin Rousseau. Constraint programming based column generation for employee timetabling. In Roman Barták and Michela Milano, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, Second International Conference, CPAIOR 2005, Prague, Czech Republic, May 30 - June 1, 2005, Proceedings*, volume 3524 of *Lecture Notes in Computer Science*, pages 140–154. Springer, 2005. doi: 10.1007/11493853_12. URL https://doi.org/10.1007/11493853_12.

[5] Luhan Zhen, Yonggang Zhang, Jingyao Li, Yanzhi Li, and Zhanshan Li. Improved bit-based filtering algorithm for regular constraint. *Expert Syst. Appl.*, 235:121259, 2024. doi: 10.1016/J.ESWA.2023.121259. URL https://doi.org/10.1016/j.eswa.2023.121259.

[6] Christian Bessiere, Emmanuel Hebrard, Brahim Hnich, Zeynep Kiziltan, Claude-Guy Quimper, and Toby Walsh. Reformulating global constraints: The slide and regular constraints. In Ian Miguel and Wheeler Ruml, editors, *Abstraction, Reformulation, and Approximation, 7th International Symposium, SARA 2007, Whistler, Canada, July 18-21, 2007, Proceedings*, volume 4612 of *Lecture Notes in Computer Science*, pages 80–92. Springer, 2007. doi: 10.1007/978-3-540-73580-9_9. URL https://doi.org/10.1007/978-3-540-73580-9_9.

[7] Polina Makeeva and Radoslaw Szymanek. Revisiting the Regular Constraint, 2009. URL https://www.researchgate.net/publication/268339951_Revisiting_the_Regular_Constraint.

[8] Kenil C. K. Cheng, Wei Xia, and Roland H. C. Yap. Space-time tradeoffs for the regular constraint. In Michela Milano, editor, *Principles and Practice of Constraint Programming - 18th International Conference, CP 2012, Québec City, QC, Canada, October 8-12, 2012. Proceedings*, volume 7514 of *Lecture Notes in Computer Science*, pages 223–237. Springer, 2012. doi: 10.1007/978-3-642-33558-7_18. URL https://doi.org/10.1007/978-3-642-33558-7_18.

[9] Mikael Zayenz Lagerkvist. *Techniques for Efficient Constraint Propagation*. PhD thesis, KTH, 2008.

[10] Olga Ohrimenko, Peter J. Stuckey, and Michael Codish. Propagation via lazy clause generation.

Constraints An Int. J., 14(3):357–391, 2009. doi: 10.1007/S10601-008-9064-X. URL https://doi.org/10.1007/s10601-008-9064-x.

[11] Graeme Gange, Peter J. Stuckey, and Radoslaw Szymanek. MDD propagators with explanation. *Constraints An Int. J.*, 16(4):407–429, 2011. doi: 10.1007/S10601-011-9111-X. URL https://doi.org/10.1007/s10601-011-9111-x.

[12] Thibaut Feydy and Peter J. Stuckey. Lazy clause generation reengineered. In Ian P. Gent, editor, *Principles and Practice of Constraint Programming - CP 2009, 15th International Conference, CP 2009, Lisbon, Portugal, September 20-24, 2009, Proceedings*, volume 5732 of *Lecture Notes in Computer Science*, pages 352–366. Springer, 2009. doi: 10.1007/978-3-642-04244-7_29. URL https://doi.org/10.1007/978-3-642-04244-7_29.

[13] Julius Gvozdiovas, Emir Demirović, and Maarten Filipo. Nfa propagator for regular constraint with explanations. https://repository.tudelft.nl/record/uuid:16153696-2b90-46f7-8b48-03455d427ba1, 2025.

[14] Christian Bessière and Jean-Charles Régin. Arc consistency for general constraint networks: Preliminary results. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence, IJCAI 97, Nagoya, Japan, August 23-29, 1997, 2 Volumes*, pages 398–404. Morgan Kaufmann, 1997.

[15] Nicholas Downing, Thibaut Feydy, and Peter J. Stuckey. Explaining alldifferent. In Mark Reynolds and Bruce H. Thomas, editors, *Thirty-Fifth Australasian Computer Science Conference, ACSC 2012, Melbourne, Australia, January 2012*, volume 122 of *CRPIT*, pages 115–124. Australian Computer Society, 2012. URL http://crpit.scem.westernsydney.edu.au/abstracts/CRPITV122Downing.html.

[16] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, June 18-22, 2001*, pages 530–535. ACM, 2001. doi: 10.1145/378239.379017. URL https://doi.org/10.1145/378239.379017.

[17] Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern SAT solvers. In Craig Boutilier, editor, *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009*, pages 399–404, 2009. URL http://ijcai.org/Proceedings/09/Papers/074.pdf.

[18] Sathiamoorthy Subbarayan. Efficient reasoning for nogoods in constraint solvers with bdds. In Paul Hudak and David Scott Warren, editors, *Practical Aspects of Declarative Languages, 10th International Symposium, PADL 2008, San Francisco, CA, USA, January 7-8, 2008*, volume 4902 of *Lecture Notes in Computer Science*, pages 53–67. Springer, 2008. doi: 10.1007/978-3-540-77442-6_5. URL https://doi.org/10.1007/978-3-540-77442-6_5.

[19] Maarten Flippo, Konstantin Sidorov, Imko Marijnissen, Jeff Smits, and Emir Demirović. A Multi-Stage Proof Logging Framework to Certify the Correctness of CP Solvers. In Paul Shaw, editor, *30th International Conference on Principles and Practice of Constraint Programming (CP 2024)*, volume 307 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 11:1–11:20, Dagstuhl, Germany, 2024. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-95977-336-2. doi: 10.4230/LIPIcs.CP.2024.11. URL https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.CP.2024.11.

[20] Kees Joost Batenburg and Walter A. Kosters. On the difficulty of nonograms. *ICGA J.*, 35(4):195–205, 2012. doi: 10.3233/ICG-2012-35402. URL https://doi.org/10.3233/ICG-2012-35402.

[21] Mikael Z. Lagerkvist and Gilles Pesant. Modeling irregular shape placement problems with regular constraints. In *First workshop on bin packing and placement constraints BPPC'08*, 2008. URL http://contraintes.inria.fr/CPAIOR08/BPPC/bppc08_submission_5.pdf.

[22] V M Glushkov. The abstract theory of automata. *Russian Mathematical Surveys*, 16(5): 1, oct 1961. doi: 10.1070/RM1961v016n05ABEH004112. URL https://doi.org/10.1070/RM1961v016n05ABEH004112.

[23] Jean-Marc Champarnaud and Fabien Coulon. NFA reduction algorithms by means of regular inequalities. *Theor. Comput. Sci.*, 327(3):241–253, 2004. doi: 10.1016/J.TCS.2004.02.048. URL https://doi.org/10.1016/j.tcs.2004.02.048.

[24] Michael O. Rabin and Dana S. Scott. Finite automata and their decision problems. *IBM J. Res. Dev.*, 3(2):114–125, 1959. doi: 10.1147/RD.32.0114. URL https://doi.org/10.1147/rd.32.0114.

[25] John Hopcroft. An n log n algorithm for minimizing states in a finite automaton. In Zvi Kohavi and Azaria Paz, editors, *Theory of Machines and Computations*, pages 189–196. Academic Press, 1971. ISBN 978-0-12-417750-5. doi: https://doi.org/10.1016/B978-0-12-417750-5.50022-1. URL https://www.sciencedirect.com/science/article/pii/B9780124177505500221.

[26] Tao Jiang and Bala Ravikumar. Minimal NFA problems are hard. *SIAM J. Comput.*, 22(6): 1117–1141, 1993. doi: 10.1137/0222067. URL https://doi.org/10.1137/0222067.

A DFA Layered Graph

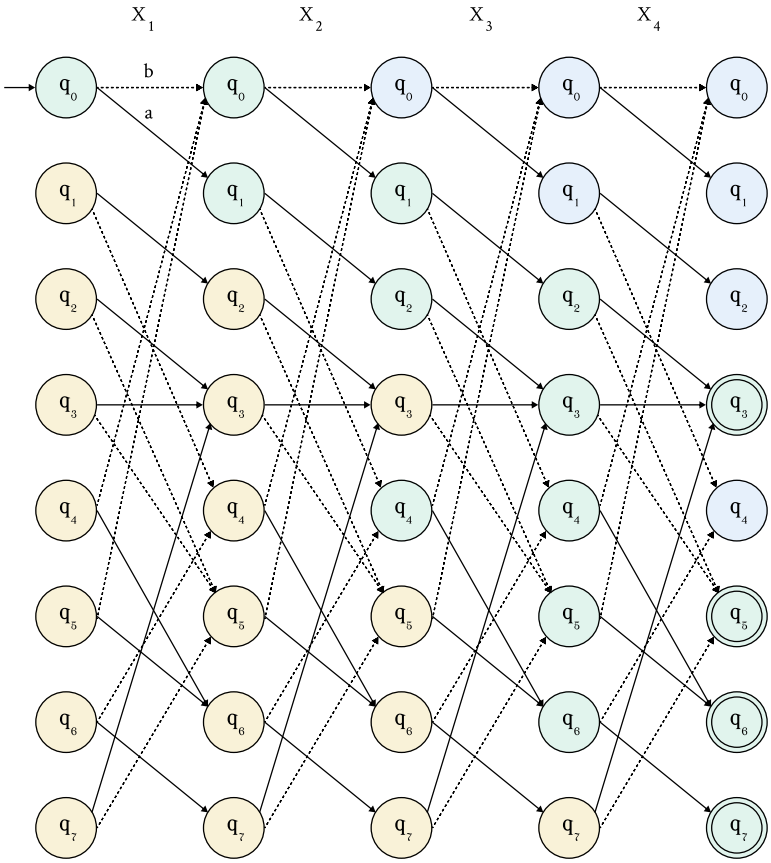


Figure 4: Layered graph for the DFA in Figure 1, coloured by reachability and co-reachability. Green is both, blue is reachable only, yellow is co-reachable only.

B Minimal-Sized Explanations

B.1 Algorithm

A minimum-cardinality explanation for removing ℓ from X_i is the smallest set of prior removals (j, v), representing literals $X_j \neq v$ that combine with the hypothesis $X_i = \ell$ to disconnect $(0, q_0)$ from every accepting node in the layered graph. This set corresponds to a minimum labelled cut, since we need to block every path through the graph that the hypothesis would otherwise permit, using as few previously-removed values as possible. Computing this exactly is NP-hard in general [18], so the algorithm uses branch-and-bound, seeded with the greedy explanation of Section 4.2 as an initial upper bound on cut size.

The two procedures are given in Algorithms 1 and 2 respectively. BranchAndBound searches over sets of prior removals (cuts), pruning branches that cannot beat the current best, and tracking visited sets to avoid redundant work. The key subroutine is FindEscapingPath, which determines whether a given partial cut is already sufficient by checking whether any path from $(0, q_0)$ to an accepting node remains open. If none does, the cut is valid. If one does, it is returned so the algorithm can branch on it.

Algorithm 1 BranchAndBound$(G, i, \ell, \mathit{fixed})$, called initially with $\mathit{fixed} = \emptyset$, $\mathit{best} =$ ExplainGreedy(G, i, ℓ), and $\mathit{visited} = \emptyset$. Returns the minimum-cardinality cut in *best*.

```
1: if |fixed| ≥ |best| then return
2: end if                                          ▷ Cannot improve on current best
3: key ← Sort(fixed)
4: if key ∈ visited then return
5: end if
6: visited += {key}
7: path ← FindEscapingPath(G, i, ℓ, fixed)          ▷ Is fixed already a valid cut?
8: if path = None then
9:     best ← fixed                                 ▷ fixed disconnects all accepting nodes
10: else
11:    for λ ∈ path do          ▷ Branch: add each cuttable arc of the escaping path
12:        BranchAndBound(G, i, ℓ, fixed ∪ {λ})
13:    end for
14: end if
```

Algorithm 2 FINDESCAPINGPATH$(G, i, \ell, \mathit{fixed})$. Returns the cuttable arcs on the cheapest open path to an accepting node, or None if no such path exists.

$\mathit{dist}[(0, q_0)] \leftarrow 0$; all other nodes $\leftarrow \infty$
$Q \leftarrow$ 0–1 priority queue initialised with $(0, (0, q_0))$
while Q is not empty **do**
    $(d, (k, q)) \leftarrow$ POPMIN(Q)
    if $d > \mathit{dist}[(k, q)]$ **then continue**
    end if ▷ Stale entry
    for each arc $((k, q) \xrightarrow{\sigma} (k{+}1, q'))$ in G **do**
        if $(k{+}1, q', \sigma) \in \mathit{fixed}$ **then continue**
        end if ▷ Arc is already cut
        if $k = i$ **and** $\sigma \neq \ell$ **then continue**
        end if ▷ Inconsistent with hypothesis $X_i = \ell$
        $c \leftarrow 1$ if $(k{+}1, q', \sigma)$ is cuttable (removed from domain, $k \neq i$), else 0
        if $\mathit{dist}[(k, q)] + c < \mathit{dist}[(k{+}1, q')]$ **then**
            $\mathit{dist}[(k{+}1, q')] \leftarrow \mathit{dist}[(k, q)] + c$
            $\mathit{prev}[(k{+}1, q')] \leftarrow (k, q, \sigma)$ ▷ Record predecessor for path reconstruction
            Push $(\mathit{dist}[(k{+}1, q')], (k{+}1, q'))$ to Q
        end if
    end for
end while
if no accepting node (n, q_{acc}) was reached **then**
    return None ▷ *fixed* is already a valid cut
end if
$q^* \leftarrow \arg\min_{q_{acc}} \mathit{dist}[(n, q_{acc})]$ ▷ Cheapest accepting node
return cuttable arcs on the path to (n, q^*) reconstructed via *prev*

B.2 Proof of Minimality

Validity. Any cut returned is a valid explanation. FINDESCAPINGPATH returning None certifies that *fixed* disconnects $(0, q_0)$ from every accepting node under the hypothesis $X_i = \ell$, so its literals jointly imply $X_i \neq \ell$.

Minimality. The bounding condition $|\mathit{fixed}| \geq |\mathit{best}|$ only prunes branches that cannot improve on the current best, so no valid cut smaller than $|\mathit{best}|$ is ever discarded. Any valid cut must contain at least one cuttable arc from every escaping path, and the algorithm branches on every cuttable arc of the current escaping path, so every minimal hitting set is eventually reached. The greedy explanation gives a finite initial $|\mathit{best}|$, which is tightened whenever a strictly smaller valid cut is found. Since the number of cuttable arcs is finite and each set is visited at most once, the search terminates with a minimum-cardinality cut.

Completeness. Because the algorithm branches on every cuttable arc of one escaping path, and any valid cut must contain at least one arc from every escaping path, the branching is sufficient to guarantee that every minimal valid cut is eventually considered.

B.3 Example: Greedy Versus Minimal

We show a small instance where ExplainMinimum finds a strictly shorter explanation than the greedy traversal. Let $\mathcal{A}$ be the DFA over $\Sigma = \{a, b\}$ accepting strings that contain aa as a substring, with states q_0 (initial, no a seen), q_1 (one a seen), and q_2 (accepting, absorbing), shown in Figure 5a. Figure 5b shows the corresponding layered graph for the variable sequence $X_1X_2X_3X_4$. The propagator has already removed a from D_1 (literal $X_1 \neq a$) and a from D_2 (literal $X_2 \neq a$). We wish to explain the removal of b from D_3, justifying $X_3 \neq b$ under the hypothesis $X_3 = b$.

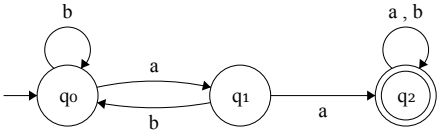
With a removed from D_1 and D_2, the only available words through the third-layer arc labelled b are $bbba$ and $bbbb$, neither of which contains aa, so no accepting path runs through b at D_3, and it is removed. Under the hypothesis, positions 2–3 and 3–4 cannot hold the aa, so it must fall at positions 1–2, forcing $X_1 = a$ and $X_2 = a$. This reaches the absorbing state q_2 by the second layer, the hypothesis arc $q_2 \xrightarrow{b} q_2$ holds the path there, and $X_4 \in \{a, b\}$ keeps it accepting. The single escaping path is therefore

$$(0, q_0) \xrightarrow{a} (1, q_1) \xrightarrow{a} (2, q_2) \xrightarrow{X_3=b} (3, q_2) \xrightarrow{a} (4, q_2),$$

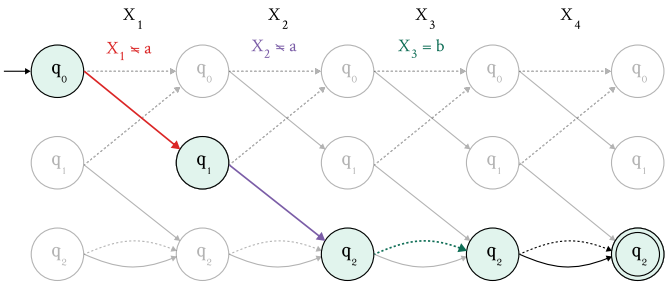
with cuttable arcs at the first and second layers (the removed values $X_1 \neq a$ and $X_2 \neq a$), shown in Figure 5b.

Greedy. The forward traversal (Section 3.5) records every cuttable arc along the path, producing $\{X_1 \neq a,\ X_2 \neq a\}$ of length two.

Minimum. FindEscapingPath returns both cuttable arcs as candidates. BranchAndBound first extends with $X_1 \neq a$: with *fixed* $= \{X_1 \neq a\}$, the 0–1 BFS finds no escaping path, since the only route to aa at positions 1–2 is now cut. So *fixed* is valid, *best* becomes $\{X_1 \neq a\}$ of length 1, and the branch on $X_2 \neq a$ is pruned ($|\mathit{fixed}| = 1 \geq |\mathit{best}| = 1$). ExplainMinimum returns $\{X_1 \neq a\}$ of length one, strictly shorter than greedy. Once $X_1 \neq a$ severs the path, $X_2 \neq a$ guards a route already gone, but greedy's single forward pass records it regardless.



(a) The DFA $\mathcal{A}$.



(b) The layered graph under hypothesis $X_3 = b$.

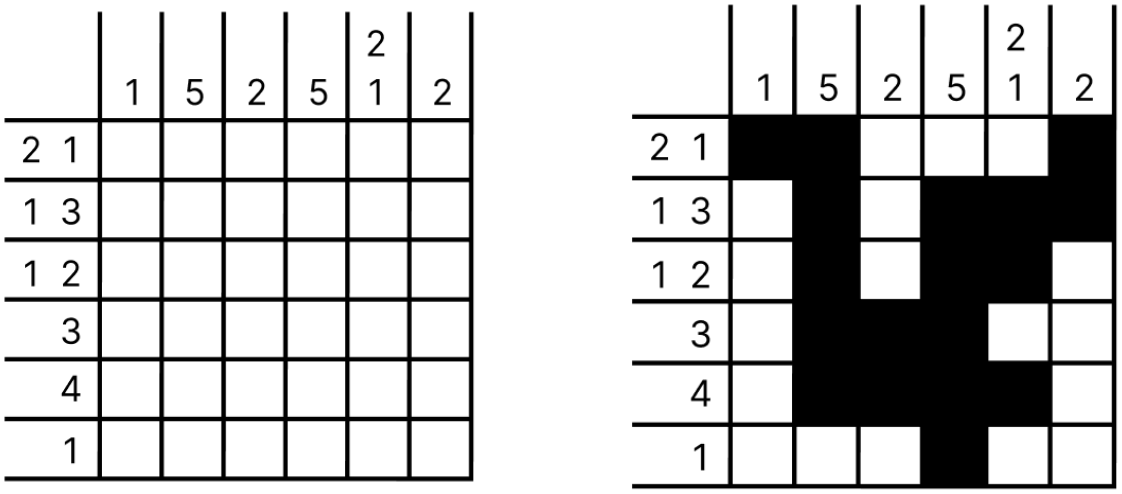
Figure 5: Left: The DFA $\mathcal{A}$ over $\{a, b\}$ accepting strings containing aa. Right: the layered graph for $X_1X_2X_3X_4$ under the hypothesis $X_3 = b$ (dark green dotted arc, layer 3), where a-transitions are drawn solid and b-transitions dotted. The two cuttable a-arcs are coloured: $X_1 \neq a$ at layer 1 (red) and $X_2 \neq a$ at layer 2 (purple). The red arc is used by the minimum explanation, the purple by greedy only. The shaded nodes trace the unique escaping path.

C Problem Selection

C.1 Nonograms

A nonogram is a grid-colouring puzzle over a *width* × *height* binary grid. Each row and column carries a hint, which is an ordered list of positive integers specifying the lengths of the consecutive runs of filled cells in that line, separated by at least one empty cell. The solver must find a colouring consistent with every hint. Each hint $(b_1, \ldots, b_p)$ is encoded as a regular constraint over {1=empty, 2=filled} via the regular expression $1^*\ 2^{b_1}\ 1^+\ \cdots\ 1^+\ 2^{b_p}\ 1^*$, compiled to NFA and DFA representations as described in Section 3. One constraint is applied per row and column, giving *height* + *width* constraints per instance. Per-instance blowup is the ratio of total DFA to total NFA states summed across all constraints.

Instances are generated by sampling a random grid at density 0.5, deriving row and column hints, and compiling to automata. Grid dimensions are drawn uniformly from {25×25, 25×30, 30×30, 30×35}, a range selected through preliminary experiments to produce instances with solve times within a tractable range for both propagators. Duplicates (identical dimensions and hint sets) are discarded. Generation uses master seed 71 with a target of 100 unique instances.



(a) Nonogram Instance

(b) Corresponding Solved Instance

Figure 6: An example nonogram puzzle. Row and column hints specify the lengths of consecutive filled-cell runs.

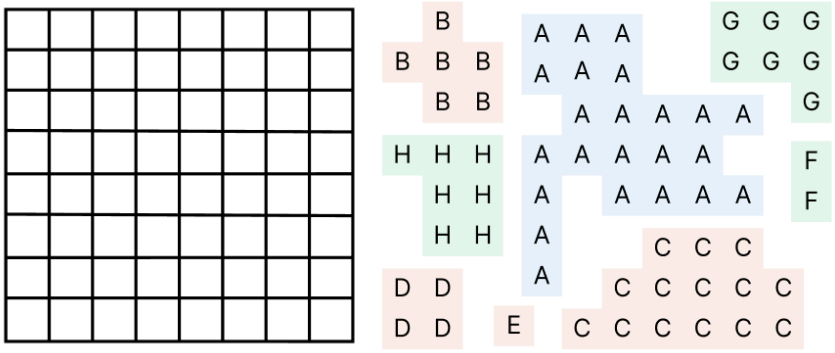
C.2 Polyominoes

A polyominoes tiling puzzle places *tiles* polyomino pieces on a *size* × *size* board such that every cell is covered exactly once. The board is encoded as a flattened sequence of tile labels with a marker appended after each row. The generator of Lagerkvist[4] emits one regular expression per tile

[4] `https://github.com/zayenz/minizinc-pentominoes-generator`

encoding its legal placements over $\{1, \ldots, \mathit{tiles}, \mathit{marker}\}$, compiled to NFA and DFA representations (Section 3), one constraint per tile over the board-and-markers sequence. Per-instance blowup is the ratio of summed DFA to NFA states across all tile constraints.

Instances are generated by invoking the `minizinc-pentominoes-generator` CLI with a seeded draw, parsing its emitted regexes, and discarding duplicates. All instances use an 8×8 board with the `source` placement strategy, a configuration identified in preliminary experiments to produce instances with solve times within a tractable range for both propagators, without introducing trivial problems. Generation uses master seed 71 with a target of 100 unique instances.



(a) Polyominoes Instance

A	A	A	D	D	G	G	G
A	A	A	D	D	G	G	G
E	A	A	A	A	A	B	G
A	A	A	A	A	B	B	B
A	F	A	A	A	A	B	B
A	F	C	C	C	H	H	H
A	C	C	C	C	C	H	H
C	C	C	C	C	C	H	H

(b) Corresponding Solved Instance

C.3 Regex

The regex type is a synthetic constraint optimisation problem designed to maximise DFA state growth relative to the NFA. Each instance encodes $K = 2$ regular constraints over a shared sequence of $\mathit{varCount} \in [18, 23]$ variables.

The first constraint is an anchor constraint $(\Sigma)^* \, a_1 \cdots a_m \, (\Sigma)^n$ over alphabet $\Sigma = \{1, \ldots, k\}$ with $k \in \{2, 3\}$, anchor length $m \in \{2, 3\}$ drawn uniformly from Σ^m, and suffix length $n \in \{4, \ldots, 13\}$. The resulting NFA has $O(n + m)$ states while the minimal DFA can have up to $2^{n+1} \cdot k^{m-1}$ states.

The second constraint enforces that a chosen symbol $c \in \Sigma$ appears exactly t times, with t drawn uniformly from $[\lfloor varCount/4 \rfloor, \lfloor (3 \times varCount)/4 \rfloor]$.

The objective is a random weighted sum of variable assignments, maximised under a smallest-value-first branching heuristic. This combination is deliberate, as the heuristic guides search toward small values first, while the objective rewards large values, creating a structural conflict that forces the solver to explore and backtrack extensively before it can certify optimality. This backtracking pressure stresses the propagators in a controlled and reproducible way.

The parameter ranges were chosen through preliminary experiments to place blowup ratios and solve times in a regime where differences between NFA and DFA propagation are observable, without instances being so large that construction or solving becomes intractable. Instances predicted to exceed 150,000 minimal DFA states are discarded before construction. This threshold is necessary as instances beyond it would rarely solve within the time limit, and the subset-construction cost itself grows with the number of DFA states, making their construction prohibitively expensive. Instances are deduplicated on the canonical relabelling of their anchor pattern and all other parameters. Random weights are shared across NFA and DFA files so both propagators face an identical objective. Generation uses master seed 71 with a target of 100 unique instances.

D Supplementary Experimental Results

D.1 Per-Call Rebuild Cost

Table 5 reports solve time with the per-call layered-graph rebuild cost removed, over the same commonly-solved set as Table 1. As Section 5 discusses, removing the rebuild explains much but not all of the DFA disadvantage, with the DFA remaining 16× slower than the NFA at high blowup.

Blowup Bin	Solved (n)	Timeout (n)	NFA (s)	DFA (s)	Decomp (s)
Low	35	12	2.95	2.68	3.63
Medium	44	3	0.25	0.68	2.92
High	40	7	0.28	4.49	11.15
Total	119	22	0.51	1.90	4.88

Table 5: Geometric mean solve time excluding the per-call rebuild cost (s), over the same commonly-solved set as Table 1. DFA remains about 16× slower than NFA at high blowup.

D.2 Additional Clause-Quality Results

Table 6 reports nogood statistics by blowup bin, over instances on which neither the NFA or DFA propagator timed out. As Section 5 discusses, NFA and DFA are identical to the reported precision in every bin, so the cross-bin variation reflects problem structure rather than representation.

			NFA			DFA		
Blowup Bin	Solved (n)	Timeout (n)	Nogoods	Avg. Length	Avg. LBD	Nogoods	Avg. Length	Avg. LBD
Low	42	5	101	10.77	5.51	101	10.77	5.51
Medium	46	1	1053	5.44	5.44	1053	5.44	5.44
High	40	7	928	5.63	5.63	928	5.63	5.63
Total	128	13	481	6.83	5.52	481	6.83	5.52

Table 6: Geometric mean nogood statistics by blowup bin, over instances where neither NFA nor DFA timed out (Solved n).

D.3 Greedy Versus Minimal Explanation Gap

Table 7 reports the per-instance change in NFA nogood metrics when greedy explanations are replaced by minimal ones, over the 85 instances both runs solved. As Section 5 notes, the gap is under 1% overall, zero on regex, and 3–4% on nonograms, while polyominoes are not interpretable.

Problem Type	Solved (n)	Timeout (n)	Nogoods %Δ	Length %Δ	LBD %Δ
Nonogram	22	2	−3.1	−3.7	−0.5
Polyominoes	2	20	0.0	0.0	0.0
Regex	61	34	0.0	0.0	0.0
Overall	85	56	−0.8	−0.9	−0.1

Table 7: Geometric-mean per-instance change in NFA nogood metrics when greedy explanations are replaced by minimal ones. Negative values mean the minimum is smaller.