NFA propagator for `regular` constraint with explanations

Julius Gvozdiovas[1]

Supervisor(s): Emir Demirović[1]**, Maarten Flippo**[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
January 26, 2025

Name of the student: Julius Gvozdiovas
Final project course: CSE3000 Research Project
Thesis committee: Emir Demirović, Maarten Flippo, Benedikt Ahrens

An electronic version of this thesis is available at http://repository.tudelft.nl/.

Abstract

The regular constraint offers good balance between expressiveness and cost. Despite potential exponential blow-up, existing approaches use deterministic automata. Furthermore, the area of combining conflict-driven learning with regular is unexplored. We combine learning with non-determinism, to produce an NFA-based propagator with explanations, and compare its performance against decomposition of the constraint. Experimental results on Nonogram instances indicate that our specialized propagator is significantly better than decomposing regular constraint.

1 Introduction

Constraint Programming is a powerful paradigm that can solve a wide variety of satisfaction and optimization problems. The process is split into two parts: modelling and solving. During modelling, a variety of (global) constraints are used to define the requirements of the problem, after which, those constraints help reduce the solution space for the solver.

It is prohibitively expensive to brute-force all possible solutions, and check whether each solution fits all given constraints. Instead, constraints have associated propagators, which allow pruning the search space and finding solutions more effectively. Implementing propagators in a solver involves leveraging properties specific to those constraints. Even so, some constraints might themselves be very difficult (i.e. NP-hard) to propagate optimally, or might be too specific to be of general use.

The global extensive constraint regular balances well between expressiveness and computational cost [13]. It restricts an array of variables to match a given regular expression. Many problems, or constraints within problems, can be expressed well using regular expressions: rostering [13], shape placement [8], puzzles such as Nonograms [19; 5], and even other constraints such as slide, stretch, pattern [2; 13]. For example, when creating a time table, it might be useful to specify that after activity A, activity B must always follow. This can be represented using the regular expression `([^A]|AB)*` (meaning: "repeated any number of times: anything but an A, or an A followed by a B").

Pesant first introduced a propagator for the regular constraint [13]. Since then, it has been optimized [9] or reimagined to use different finite automata [3; 19] for better performance. Furthermore, the field of Constraint Programming has advanced, unveiling the technique of Lazy Clause Generation (LCG) to combine expressiveness of global constraints with conflict learning capability of SAT solvers [4]. Existing propagators using LCG have achieved multiple orders of magnitude better results for the regular constraint [5].

However, state-of-the-art LCG-based approach for the regular constraint works by converting deterministic finite automaton (DFA) into a multi-valued decision diagrams (MDD) [5]. They have developed a generic propagator for MDDs, which allows for a variety of constraints to be implemented. It might be possible to construct a better LCG propagator, by leveraging specific properties unique to the regular constraint. For example, regular expressions might result in exponentially sized equivalent DFAs/MDDs [7], which suggests that using a non-deterministic finite automaton (NFA) representation could lead to improved performance [3]. Furthermore, a specific propagator could incorporate previous improvements to the original propagator for the regular constraint [13].

Feydy's and Stuckey's paper on lazy clause generation states that "learning for decomposed globals is stronger" [4], compared to using specialized propagators. They argue that decomposition into simpler constraints enables better learning, allowing to recoup the lost improvements of a more specialized approach of a specific propagator. While this has been demonstrated for some constraints [16], it remains untested in research for the regular constraint.

In this paper, we extend the DFA-based Pesant's algorithm [13] together with lazy clause generation, producing the first NFA propagator with *explanations* for conflict learning. We challenge the notion that learning is stronger for decomposed constraints, by comparing the performance of our specialized implementation against a decomposition of the regular constraint [6].

Our experiments show that our NFA-based propagator is an order of magnitude faster than decomposition, by both implementing best possible propagation (domain consistency), and by generating smaller explanations.

The rest of the paper is organized as follows: Section 2 explains and formalizes finite domain constraint programming in more detail, as well as the regular constraint and its propagator. Section 3 recounts and compares existing state-of-the-art approaches for the constraint. In Section 4 we define the NFA-based propagator and consider alternative approaches. In Section 5, we show experimental results of running our propagator, and comparing it against decomposition. Section 6 describes the steps taken in order to ensure our research is reproducible. Finally, in Section 7, we conclude and discuss remaining open areas for study.

2 Preliminaries

2.1 Constraint Programming

We aim to use Constraint Programming to solve *finite domain* (FD) Constraint Satisfaction Problems (CSPs), which consist of:

- Sequence of variables $\mathcal{X} = x_1, x_2, \ldots, x_N$;
- A *finite domain* of each variable $\mathcal{D}(x_i) \in \mathbb{Z}$;
- Set of constraints $\mathcal{C}$, where each constraint $C(X)$ takes a subsequence of variables $X \subseteq \mathcal{X}$.

CSPs can encode a large variety of problems, which can then be solved in a general manner, without building algorithms for those problems specifically. Each variable in a CSP represents a decision from a finite number of choices. The constraints $\mathcal{C}$ then represent the relations between variables. FD solver then either finds a value for each variable, such that all constraints are satisfied, or return "no result", indicating that solution does not exist.

Each constraint $C(X)$ has its respective propagator f, which restricts the domains of variables in X, such that inconsistent values are removed from domains. Propagator f never increases the size of any of the domains, as only inconsistent values are ever removed. Formally, a propagator f is defined as a monotonically decreasing function $f(\mathcal{D})$, that returns an updated domain, while still preserving solutions for the CSP. That is, if CSP $(\mathcal{C}, \mathcal{X}, \mathcal{D})$ has a solution, then $(\mathcal{C}, \mathcal{X}, f(\mathcal{D}))$ must also have one. For a given constraint, a propagator is considered *domain consistent* (or that it achieves *generalized arc consistency*) if after it maps the domains $\mathcal{D}(X)$ of variables X, there are no values in any of the domains that could be removed, yet there be an assignment to the variables that satisfies the constraint.

Algorithm 1 Depth-first search algorithm for solving Constraint Satisfaction Problems, taken from [15]

```
Require: CSP P = (C, X, D)
Ensure: true, iff (C, X, D) has a solution; false otherwise.
  P' := propagate(P)                      ▷ Apply all propagators f of each constraint C(X) ∈ C
  if ∃x ∈ X' : |D'(x)| = 0 then            ▷ Empty domain means a constradiction is reached
      return false
  else if ∃x ∈ X' : |D'(x)| > 1 then
      Q_1, Q_2, ..., Q_k := divide(P')
      for i ∈ 1, 2, ..., k do
          if solve(Q_i) then
              return true
          end if
      end for
      return false
  else
      return true
  end if
```

FD solvers generally use depth-first search (example shown in Algorithm 1). It applies propagators on variable domains until a fixed-point is reached, i.e. the domains are not shrunk any more. An empty domain means that no value can be assigned to a variable, thus the CSP is inconsistent. Otherwise, a divide-and-conquer approach is used, dividing the CSP into smaller sub-problems. This is usually accomplished by introducing a new constraint, limiting one of the variables' domain. For example, a variable x might have a domain $\{7, 11, 13\}$, which would be split into three sub-problems: $\mathcal{D}_1(x) = \{7\}, \mathcal{D}_2(x) = \{11\}, \mathcal{D}_3(x) = \{13\}$. These splits can be done by introducing new constraints, $[[x = 7]], [[x = 11]], [[x = 13]]$ respectively. A solution is found once all domains have been reduced to only a single value. The previous solutions can be excluded using additional constraints, to find more solutions by rerunning the algorithm.

During the search, solvers inevitably run into inconsistent sub-problems, and have to backtrack. However, such backtracks are prone to making the same general type of deductions and conflicts, especially if the decisions leading to the conflict have been made far up the call stack. *Conflict learning* can help prevent repeating the same deductions in the search tree. Whenever a conflict occurs, the implication graph of Boolean clauses can be used to infer which domain splits (or decisions) lead to the current conflict, and a negation of those decisions (called *nogood*) can be added as a constraint. This approach is currently only viable for SAT solvers, which solve Boolean satisfiability problem: given a set a Boolean clauses, find an assignment to the variables such that all clauses evaluate to `true`. Traditional FD solvers cannot use *conflict learning*, and expressing CSPs using Boolean clauses might result in very large encodings.

Lazy clause generation (LCG) [4] can be used to bridge the gap between SAT solvers and FD solvers. In addition to storing set representation of variable domains $\mathcal{D}$, clausal representation is also used. For each variable x_i and each value $v_j \in \mathcal{D}(x_i)$, Boolean variables $[[x_i = v_j]]$ and $[[x_i \leq v_j]]$ are stored, with negations of these variables meaning $[[x_i \neq v_j]]$ and $[[x_i > v_j]]$ respectively. Both set representation and clausal representation are kept consistent with each other, by using additional set of constraints. LCG generates these clauses lazily, as otherwise the clausal representation of the domains may be too large. By combining the two approaches, LCG can achieve the best of both worlds: expressive and small *finite domain* constraints; and optimizations for SAT solvers, such as *conflict learning*. In order to combine the two approaches, propagators for global constraints must be modified to also include *explanations* for their propagations.

Formally, an *explanation* can be defined as a 2-tuple (X, y), representing implication $x_1 \wedge x_2 \wedge \ldots \wedge x_k \rightarrow y$:

- Antecedent: a set of Boolean literals X;
- Consequent: Boolean literal y.

For example, *explanation*

$$[[x_1 = 4]] \wedge \neg[[x_5 \leq 7]] \wedge [[x_2 \leq 3]] \rightarrow \neg[[x_4 = 6]]$$

means that the propagator deduced the following: if $x_1 = 4$, $x_5 > 7$ and $x_2 \leq 3$, then $x_4 \neq 6$. *Explanations* can be compared in strength - in general, we look for the smallest possible *explanation*. For instance,

$$\neg[[x_5 \leq 7]] \wedge [[x_2 \leq 3]] \rightarrow \neg[[x_4 = 6]]$$

is a strictly smaller *explanation* - it implies the aforementioned *explanation*, and thus is considered stronger.

For any propagation, it is always possible to generate a trivial *explanation*, where the antecedent is the entire current domain $\mathcal{D}$ (appropriately encoded using Boolean literals), and the consequent is derived from the values removed from the domain. However, these explanations cannot be effectively reused, as the domain $\mathcal{D}$ will be different in a different part of the search tree. As such, the main goal when incorporating global constraints into the LCG approach is to generate small *explanations*, which can be reused many times, allowing the solver to more effectively skip over parts of the search tree which would eventually be discarded by the constraint anyway.

2.2 `regular` constraint

The `regular`$(X, \mathcal{R})$ constraint works over a fixed-length array of n variables $X = \langle x_1, x_2, \ldots, x_n \rangle$, and applies a regular expression $\mathcal{R}$. The regular expression is processed to create a propagator for the solver.

Regular expressions are formed from a set of characters (representing possible values in variable domains), and various operators: $R_1 \cup R_2, R^*, R_1 \parallel R_2$. The regular expression is first converted into a finite automaton.

Finite automaton M is defined as as a 5-tuple $(Q, \Sigma, \delta, q_0, F)$:

- Q is a finite set of states.
- Σ is an alphabet, composed of values ("letters") in the domains $\mathcal{D}(X)$.
- $\delta(q, \sigma)$ is the transition function, taking a state $q \in Q$ and a value $\sigma \in \Sigma$.
- q_0 is the initial state of the automaton.
- F is the set of accepting states.

The transition function can be either $\delta : Q \times \Sigma \rightarrow Q$ or $\delta : Q \times \Sigma \rightarrow \mathcal{P}(Q)$, mapping to either a single state, or to a set of states. The respective automaton is either a Deterministic Finite Automaton (DFA), or a Non-deterministic Finite Automaton (NFA). NFAs can be converted to DFAs using subset construction [14], possibly leading to exponentially more states in the resulting DFA compared to the input NFA. On the other hand, all DFAs are trivially also NFAs. Finite automata can be represented using a directed multigraph (a graph where there may be multiple edges between two vertices). Figure 1 shows an example of both an NFA and a DFA, represented as multigraphs. States are represented as nodes, and transition function δ as edges. The initial state is indicated using an arrow pointing to one of nodes, and accepting states are shown using double circles for nodes.

In common usage, NFAs may also permit ε (meaning no character) transitions. Such ε-NFAs can be converted to ε-free NFAs without creating any additional states, by adding transitions where needed. As such, in this work, NFAs refer to ε-free NFAs.

Example 1. Consider the regular expression $\mathcal{R} = \{0, 1\}^* \parallel ((0 \parallel 1) \cup (1 \parallel 0)) \parallel \{0, 1\}$. Using commonly used PCRE-style, it would be expressed as `[01]*(01|10)[01]`. It can be converted into an NFA, shown in Figure 1a. It reads any number of `0`s or 1s, then reads either `10`, or `01`, and finally reads a `0` or `1` again before terminating. Non-determinism is used to decide whether to go from q_0 to q_1/q_2, after reading a `0` or a `1` respectively. It can also be converted to a DFA, as shown in Figure 1b. Since non-determinism cannot be used, the DFA remembers the three last inputs it received, choosing `000` or `111` after the first input. This example also highlights how DFAs might be exponentially bigger than corresponding NFAs. Consider $\mathcal{R} \parallel \{0, 1\}^k$ (k repetitions of $\{0, 1\}$ concatenated on the right). The resulting NFA would have $k + 5$ states, while the DFA would have $2^{k+3} + 1$ states.

Pesant [13] first introduced a *domain consistent* propagation algorithm for `regular`$(X, \mathcal{R})$ constraint. It takes advantage of the restricted length of input string $|X|$, as inputs to global constraints are fixed-length arrays of variables. Given a DFA $M = (Q, \Sigma, \delta, q_0, F)$, it constructs a layered multigraph G, with $|V| = |Q| \cdot (|X| + 1)$ total number of nodes, arranging them into $|X| + 1$ layers, where i-th layer represents the states the string can be in after traversing i letters. Edges are formed exclusively from i-th layer to $(i + 1)$-th layer. For each $q_a, q_b \in Q, \sigma \in \Sigma$, such that $\delta(q_a, \sigma) = q_b$, there is an edge labelled with σ from node $q_{(a,i)}$ (node in layer i, for state q_a) to node $q_{(b,i+1)}$ (node in layer $i + 1$, for state q_b). It then performs a two-phase (forward and backward) filtering step, removing nodes or edges which cannot be reached from the starting state in layer 0, or cannot reach any of the accepting states in layer $|X| + 1$.

Example 2. Consider the following instance: `regular`$(\langle x_1, x_2, x_3, x_4, x_5 \rangle, \mathcal{R})$, where R is same regular expression as in Example 1. The regular expression is converted to a DFA (as show in Figure 1b). Then the resulting multigraph that Pesant's algorithm [13] produces is shown in Figure 2.

During propagation, the algorithm removes edges corresponding to eliminated values, then recursively updates the graph so that edges which are no longer on any path from initial state to an accepting state are "killed". If all edges corresponding to a value in a domain are removed, the propagation algorithm removes that value from the domain.

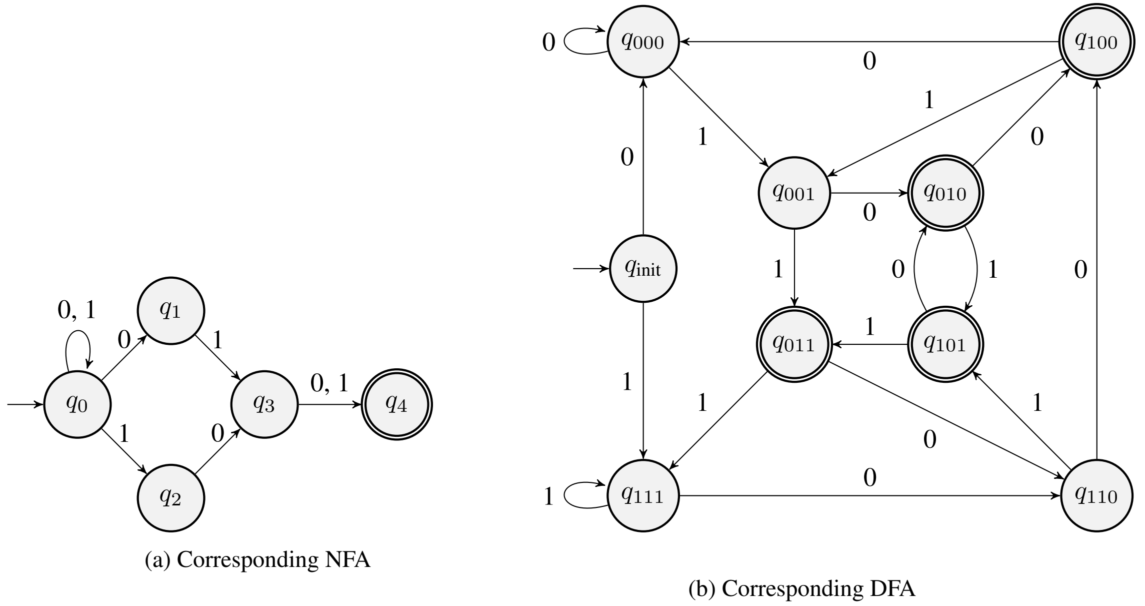


Figure 1: Finite automata, corresponding to regular expression $\{0, 1\}^* \parallel ((0 \parallel 1) \cup (1 \parallel 0)) \parallel \{0, 1\}$.

Example 3. Consider again the layered multigraph shown in Figure 2. Suppose that domain of x_2 was updated to $\{0\}$, and x_3 to $\{1\}$. Figure 3 shows the updated graph. Blue edges and nodes are removed by Pesant's propagation algorithm. The propagation would deduce that $x_4 = 0$. Pesant's algorithm does not create explanations for the deductions, however, we can explore what type of explanations are possible for this example. A trivial explanation would be:

$$[[x_2 = 0]] \wedge [[x_3 = 1]] \rightarrow [[x_4 = 0]]$$

However, a smaller (and indeed the smallest) explanation exists:

$$[[x_3 = 1]] \rightarrow [[x_4 = 0]]$$

3 Related Work

Lazy clause generation was initially introduced by Ohrimenko et al. [12], then later reengineered by Feydy and Stuckey [4]. It introduced a novel approach of using SAT solver techniques, such as using *nogoods*, conflict-driven search [10], and combining them with finite domain solvers. Thus, general purpose constraints (whose SAT encodings might be excessively large) could be used in conjunction with advanced SAT optimizations. The combination can be achieved by extending the constraints to also include *explanations* for their propagators.

Pesant [13] initially introduced a *domain consistency* algorithm for `regular` constraint exclusively for DFAs, using it to solve rostering problems. Multiple improvements and observations about it have been made since.

Makeeva and Szymanek [9] improve upon Pesant's algorithm by optimizing "characteristics such as incrementality, efficient backtracking, and memory usage". Zhen et al. [19] reworks Pesant's algorithm to use bit-vectors, by requiring input DFA to be a single-transition DFA, such that if there exists an edge from q_a to q_b using $\sigma \in \Sigma$, there must not be any other edges from q_a to q_b. Any DFA can be converted into polynomially-sized single-transition DFA, by duplicating nodes which have multiple edges coming into them.

Lagerkvist [7] raised the issue that converting a *regular expression* into a DFA might result in using exponentially more space, and suggested that Pesant's algorithm [13] can be adapted for NFAs instead. Cheng et al. [3] tests this hypothesis by implementing Pesant's algorithm for NFAs and multi-valued decision diagrams (MDDs). In their experiments, they observe that NFAs can perform better when the corresponding DFA/MDD would be significantly bigger.

Gange et al. [5] first incorporated *explanations* to the `regular` constraint. They extended Subbarayan's [17] algorithm (which creates *explanations* for binary decision diagrams) to explain propagations for MDDs as well. Their explanations are generated iteratively, sacrificing minimality for faster *explanation* generation. They convert the *regular expression* into a MDD, allowing using their propagator for `regular` constraint. Their algorithm is significantly faster than existing implementations. However, it is not clear how much their propagator helps *clause learning*, compared to decomposition (which Feydy and Stuckey suggest instead [4]).

Table 1 compares improvements to Pesant's original *domain consistency* algorithm [13] for the `regular` constraint. Each presented algorithm uses a different approach to improve upon Pesant's pioneering work, and do not heavily reference each

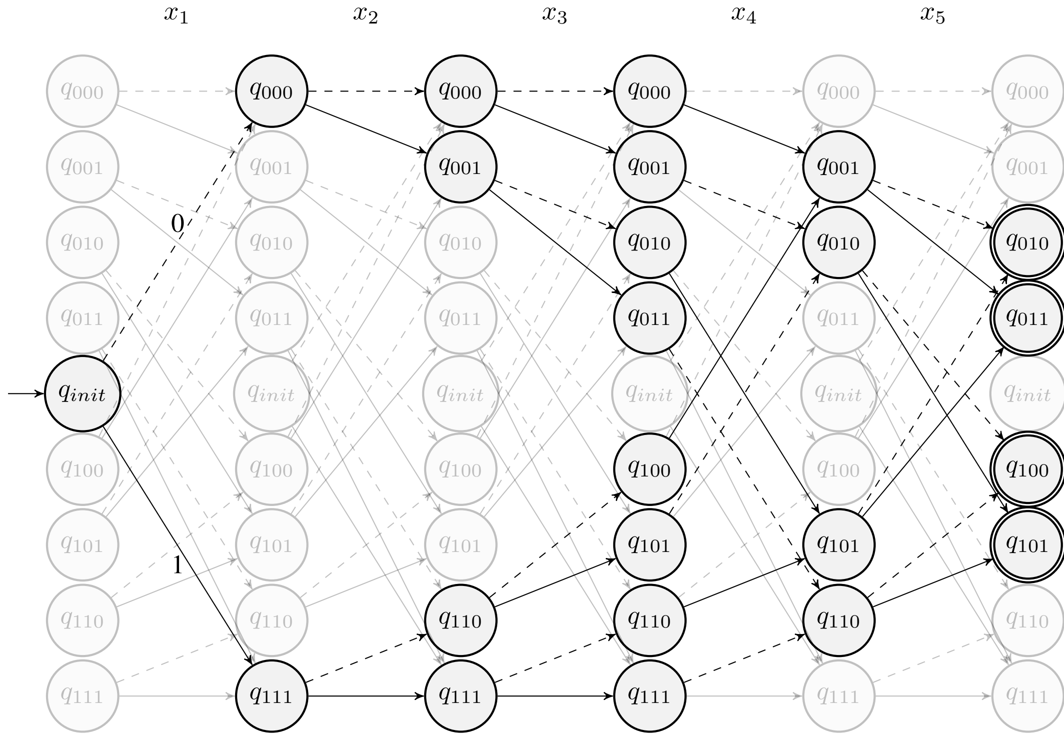


Figure 2: Layered multigraph resulting from Pesant's algorithm [13]. Transparent nodes/edges are removed by the two-phase process.

Table 1: Comparison between state-of-the-art algorithms for the `regular` constraint, indicating relative improvement for large instances, compared to Pesant's algorithm.

Author(s)	Time (rel.)	Space (rel.)	Approach
Pesant [13]	1	1	-
Makeeva and Szymanek [9]	$\times 2$	$\times 8$	Improved backtracking, optimized redundant data
Gange et al. [5]	$\times 1626$	-	Iterative MDD propagator with *explanations* and learning
Cheng et al. [3]	$\times 14$	$\times 1337$	Pesant's algorithm adapted for NFAs
Zhen et al. [19]	$\times 2.25$	-	Bit-vectors, single transition DFA

other. Overall, unsurprisingly, Gange et al. [5] learning-driven approach has the biggest improvement, as learning allows to prevent duplicate propagations and thus cuts down on the search space significantly [12]. Most works do not measure improvements to space usage, however, Gange et al. [5] shows how much improvement there can be in using NFAs instead of DFAs or MDDs.

Gange et al. [5] work has the greatest runtime improvement, however, it is not clear whether using their propagator with explanations is better compared to decomposing the `regular` constraint into simpler constraints. Lazy clause generation is already known to be wildly successful in decreasing runtime for many problems, and that it sometimes benefits from using decomposition of constraints [4; 16]. Thus, it is not clear whether how much of the runtime improvement can be attributed to the propagator, as opposed to using LCG. Further, their technique of using MDDs is very general and applicable to many constraints. It raises a question of whether using DFAs/NFAs, which represent the `regular` constraint more closely, would allow the propagator to be more efficient, at the cost of generality (i.e. not being applicable to more constraints). Our research objective seeks to answer these questions, by first constructing an NFA-based propagator with explanations, and then comparing it against decomposition.

All presented approaches tackle improving the propagator for `regular` in a different manner, however, none of them combine their efforts. Zhen et al. [19] work suggests that in some cases, it is sometimes possible to exploit the structure of a given `regular` constraint instance. For instance, our earlier example produces a single-transition DFA (shown in Figure 1b) that fits their model. Cheng et al. [3] also show that in some cases, the DFA might not be much larger than the corresponding NFA.

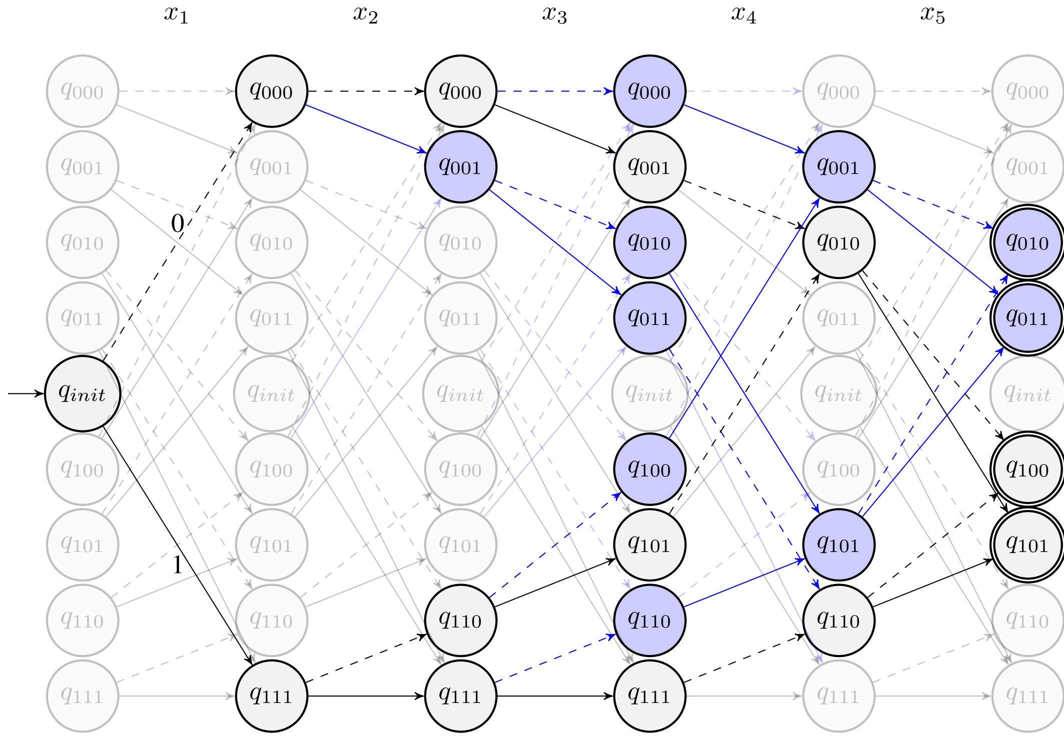


Figure 3: Layered multigraph after setting $x_2 = 0, x_3 = 1$. Blue nodes/edges indicate that they are not part of any solution with the current domain.

This suggests that it is possible to fine-tune the propagation strategy that the `regular` constraint employs, based on the given regular expression and the resulting automata. We aim to investigate how can these various improvements be combined, or used in different cases.

4 NFA-based propagator for the `regular` constraint with *explanations*

As mentioned in Section 3, Lagerkvist [7] shows that Pesant's propagation algorithm [13] can easily be extended to work with NFAs. We treat this extended algorithm as our baseline to build upon. Given a non-deterministic finite automaton[1] $M = (Q, \Sigma, \delta, q_0, F)$, and variables $X = \langle x_1, x_2, \ldots, x_n \rangle$ with domains $D(x_i) \subseteq \Sigma$, we construct a layered multigraph G with $|X| + 1$ layers $\mathcal{L} = \langle L_0, L_1, L_2, \ldots, L_n \rangle$. Each layer L_i consists of $|Q|$ nodes $v_1^i, v_2^i, \ldots v_{|Q|}^i$, with edges connecting adjacent layers corresponding to δ. That is, for each layer $L_i \in \{L_0, L_1, \ldots, L_{n-1}\}$ (i.e. all but the last layer), consider all transitions in the NFA: $q_a \in Q, \sigma \in \Sigma$, s.t. $\delta(q_a, \sigma) \subseteq Q$. Then for each $q_b \in \delta(q_a, \sigma)$, we connect an edge from v_a^i to v_b^{i+1}, labelled with σ. If the initial NFA contains $0 \leq T \leq |\Sigma| \cdot |Q|^2$ transitions, then the resulting layered multigraph $G = (\mathcal{L}, E)$ contains $T \cdot (|X| + 1)$ edges.

An important optimization to the above construction, as observed by Pesant [13], is to only create the relevant nodes: those that can reach the accepting nodes in the final layer, starting from the starting node. Thus, in practice, the construction of the layered multigraph follows the two-phase construction algorithm. In phase one, only nodes and edges reachable from the starting node, are created. In phase two, nodes and edges that cannot reach and accepting node in the final layer are deleted.

Explanations for the NFA propagator can be created using the same general principle as from Gange et al. [5]. Their (non-incremental) *explanation* algorithm works on multi-valued decision diagrams (MDDs), which we adopt to work on non-deterministic finite automata (NFAs). Three main changes must be made to their *explanation* algorithm:

1. Adopted to the layered multigraph (as opposed to traversing an MDD);
2. Support multiple final nodes, not just a single true node $\mathcal{T}$;
3. Support multiple edges labelled with the same value between two nodes.

[1] As discussed in Section 2, in this work we consider only ε-free NFAs. NFAs containing ε transitions can be converted to ε-free NFAs without adding new states.

Items 1 and 3 are already innately supported by Gange et al. [5] *explanation* algorithm. Item 2 requires a minor modification: instead of finding the set of nodes that can reach $\mathcal{T}$, we find the set of nodes that can reach any of the accepting nodes. This modification can be accomplished solely by changing the initial set and queue of the search.

It is important to note that this approach requires always storing the original layered multigraph in memory. A distinction must be drawn between deleted nodes/edges (which only happens during construction), and "killed" nodes/edges. In order to preserve the nodes/edges in memory, they are marked as "killed" when the solver removes a value from a variable's domain.

Our *explanation* algorithm is run each time the extended propagator removes a value v from domain $D(x)$ of variable x. To explain $[[x \neq v]]$, we assume the opposite ($[[x = v]]$) in the updated layered multigraph. This ensures that there are no more viable paths from the start node to any of the accepting nodes.[2] We then perform a breadth-first traversal, from the starting node. For each edge that is currently killed, but adding it back to the layered multigraph would create a valid path, is added to the explanation.

Algorithm 2 describes the procedure in detail. It follows the exact same idea as for MDD propagator: finding all nodes that each accepting states in the given state of the multigraph (except treating as if $[[x = v]]$ is true), then iterating through multigraph again, in breadth-first order, ignoring the state of whether edges are living, and adding any edges that would make a connection from the starting node to an accepting node, as explanations. The main difference from the `explain` and `mark_reacht` functions presented by Gange et al. [5], is that our equivalent of `mark_reacht` is initialized with all accepting nodes, not just one node $\mathcal{T}$.

In our implementation, at the beginning of solution search, the given regular expression is converted to an NFA, then the layered multigraph is constructed using the two-phase process. During search, it works in a non-incremental manner, starting from the initial state each time the solver calls the propagator. It iterates over all variables, killing appropriate edges, and recursively propagating the updates. After that, it iterates over all edges, and updates any variable values which no longer have any corresponding living edges. Explanations are constructed eagerly, using Algorithm 2. Whenever the propagator updates a variable x_i, to remove a value σ from it's domain $\mathcal{D}(x_i)$, the algorithm is called on the updated layered multigraph, passing in the propagated clause $[[x_i \neq \sigma]]$.

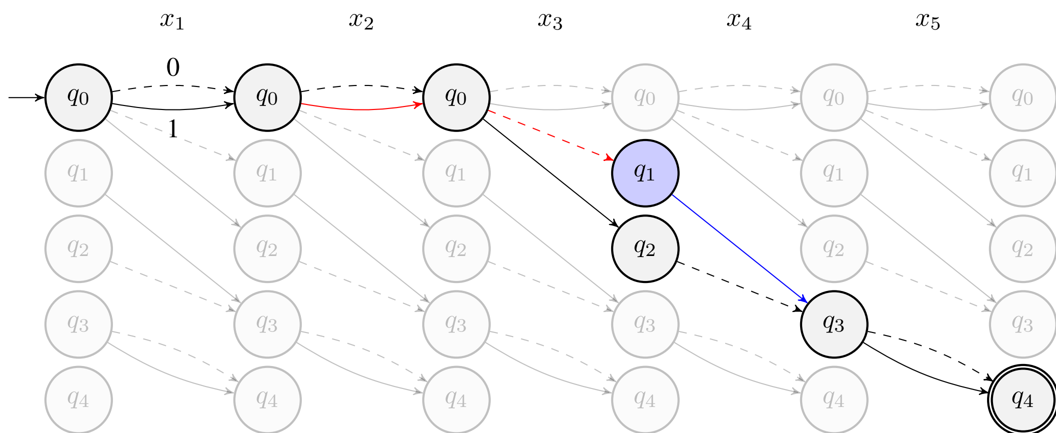


Figure 4: Layered multigraph for a NFA. Transparent edges/nodes are removed during initialization. Setting $x_2 = 0, x_3 = 1$ eliminates red edges, which is propagated to remove the blue node and corresponding blue edge.

Example 4. Consider, the instance from Example 2 again: $\texttt{regular}(\langle x_1, x_2, x_3, x_4, x_5\rangle, \mathcal{R})$, where $\mathcal{R} = \{0,1\}^* \parallel ((0 \parallel 1) \cup (1 \parallel 0)) \parallel \{0,1\}$. The regular expression $\mathcal{R}$ can then be converted to an NFA M, as depicted in Figure 1a. We then construct a layered multigraph G with six layers, where each layer has five nodes. We add edges between nodes of adjacent layers, corresponding to transitions in the NFA M. The resulting layered multigraph is depicted in Figure 4.

After setting $x_2 = 0, x_3 = 1$, the propagator deduces that $x_4 \neq 1$. Algorithm 2 is then called with $[[x_4 \neq 1]]$ clause. It assumes $[[x_4 = 1]]$ (while red edges are still removed), and finds that only the two right-most nodes (q_3 and q_4) can reach the final layer. It then performs a breadth-first traversal, starting from left-most q_0 node, checking if adding an edge would allow reaching accepting states. Edge from q_0 to q_0, with value $x_2 = 0$, is not added to the explanation, since adding it would not create a path to any accepting states. Edge from q_0 to q_1, with value $x_3 = 0$, is added, as adding it (and assuming $[[x_4 = 1]]$) would create a path to the accepting state.

[2]It might not be immediately obvious for why that is the case. The propagator only removes values when they cannot be part of any solution within the current domain $\mathcal{D}$. As such, adding an assumption that a variable has that removed value results in an inconsistent domain. Another way of viewing it is that the current domain is always a (trivial) valid explanation for any propagation: $\mathcal{D} \rightarrow [[x \neq v]]$. Adding a negation to the argument is the same as stating: $(\mathcal{D} \rightarrow [[x \neq v]]) \wedge [[x = v]]$, which immediately contradicts current domain $\mathcal{D}$.

Algorithm 2 Creating explanations from the layered multigraph for NFA. Code based on MDD explanations by Gange et al. [5]

Require: Propagated clause $[[x_k \neq \sigma]]$, Corresponding updated layered multigraph $G = (\mathcal{L}, E)$, as built from the NFA M.
Ensure: Set of variable and value pairs $\{(x_a, \sigma_a), (x_b, \sigma_b), \dots\}$, used to form explanation: $[[x_a \neq \sigma_a]] \wedge [[x_b \neq \sigma_b]] \wedge \dots \rightarrow [[x_k \neq \sigma]]$

```
reaches_accepting := {v_i^n | v_i^n ∈ L_n, q_i ∈ F}        ▷ Initialize reaches_accepting to nodes in the last layer,
                                                            ▷ corresponding to the accepting states F in the NFA
queue := {e | e ∈ G.inbound(v), v ∈ reaches_accepting}      ▷ Queue storing inbound edges
while queue ≠ ∅ do                                          ▷ Within current domain, find nodes which reach accepting
    e := queue.pop()
    if e.start ∈ reaches_accepting then
        continue
    end if
    if e.start_layer = k ∧ e.value = σ then                 ▷ Treat [[x_k = σ]] as true
        reaches_accepting ∪= {e.start}
        queue ∪= G.inbound(e.start)
    else if e.start_layer = k ∧ e.value ≠ σ then            ▷ Treat [[x_k ≠ σ]] as false
        continue
    else if e.value ∈ D(x_{e.start_layer}) ∧ e.start ∉ reaches_accepting then   ▷ Only consider the current domain
        reaches_accepting ∪= {estart}
        queue ∪= G.inbound(e.start)
    end if
end while
explanation := ∅
queue := {v_0^0}                                            ▷ Traverse again, from starting node
while queue ≠ ∅ do
    for v ∈ queue do
        for e ∈ G.outbound(v) do
            if e.start_layer ≠ k ∧ e.end ∈ reaches_accepting then
                explanation ∪= {(e.start_layer, e.value)}    ▷ [[x_k = σ]] ∧ e causes contradiction
            end if
        end for
    end for
    next_queue = ∅                                          ▷ BFS traversal
    for v ∈ queue do
        for e ∈ G.outbound(v) do
            if (e.start_layer = k ∧ e.value = σ) ∨ (e.start_layer ≠ k ∧ (e.start_layer, e.value) ∉ explanation) then
                next_queue ∪= {e.end}
            end if
        end for
    end for
    queue := next_queue
end while
return explanation
```

Examining Figure 3, we can observe that our algorithm would derive the same exact explanation for the layered multigraph derived from a DFA. NFA-based approach allows us to generate similar explanations, while consuming less space/time due to a smaller graph.

4.1 Other possible approaches to propagator for `regular` constraint

The explanations created by our approach are minimal (i.e., no single part of the explanation can be removed while remaining consistent), however, they are not minimally sized. Searching for a minimally sized explanation is NP-hard [17].

Algorithm 2 can be extended to allow for multiple initial nodes. This would allow us to reverse the entire multigraph, and work on its symmetric, but equivalent, counterpart. In principle, it is possible to search for explanations by traversing the graph from the opposite direction. While this could, in theory, find different explanations, we do not explore this approach, as the input regular expression can be equivalently reversed, to produce the same effect.

Algorithm 2 works on general layered multigraphs, as long as edges only point to the next layer. As such, it also works for

layered multigraphs produced by DFAs (in fact, since every DFA is also an NFA, no modifications need to be made). DFAs can be minimized [18] in polynomial time (unlike NFAs), and thus, for some problems, may even result in smaller sized layered multigraphs.

The minimization idea could also be applied to the layered multigraph. Due to its structure, nodes cannot be visited twice, allowing finding equivalent nodes more easily. However, additional considerations have to be made to account for the fact that the path to a node also matters.

5 Experimental Setup and Results

The NFA propagator for `regular` was implemented using Rust, in a modified version of finite-domain lazy clause generation constraint programming solver Pumpkin[3].

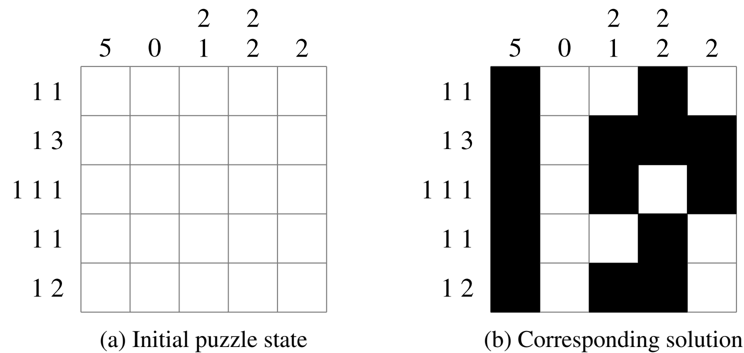
All experiments were run on 4.6GHz Intel i7-11800H CPU, with 16GB of RAM, running Manjaro Linux. Each experiment was run using MiniZinc [11] modelling language and the respective command-line tool. In our experiments, we compare three approaches:

- NFA-based propagator with explanations, as described in Section 4.
- DFA-based propagator with explanations, as described in Section 4.1. The DFA is constructed using powerset construction [14], then minimized [18].
- Decomposition of `regular` constraint into atomic constraints. MiniZinc decomposes the predicate, converting the regular expression into a DFA, which is then decomposed into atomic constraints.

For each instance and approach, the solver was given ten minutes, collecting relevant statistics using MiniZinc.

5.1 Nonograms

Nonograms, also called paint-by-numbers, are puzzles set in a $n \times m$ grid of squares, where each square may or may not be filled (painted in). Each row and column contains a sequence of numbers, called a hint for that line, indicating the runs of filled in squares, separated by non-zero empty squares. The puzzle is presented only as the empty grid and the hints, and the aim is to fill in the grid such that the hints are respected, i.e., to retrieve the original grid.



(a) Initial puzzle state

(b) Corresponding solution

Figure 5: Example 5×5 Nonogram puzzle

Figure 5 shows an example 5×5 Nonogram puzzle and its solution. Given the grid in sub-figure 5b, the hints are constructed such that each contiguous run of black squares is represented by a number. For example, bottom row has two runs of black squares, of length 1 and length 2, with a gap between them, so the hint is $[1, 2]$.

Nonogram puzzles can be converted into CSP made solely of `regular` constraints. Given a $n \times m$ Nonogram puzzle, with $n + m$ hints (for each row and column), the corresponding CSP instance can be made by creating $n \cdot m$ Boolean decision variables (i.e. with domains $\{0, 1\}$), and $n + m$ regular expressions over each line of variables. The regular expressions, when converted, result in small NFAs, with $\mathcal{O}(n + m)$ states and transitions. They can also be converted into small DFAs of similar size.

We generated random Nonogram puzzle instances, with each square having 50% probability of being filled in. This ratio of black-to-white squares was chosen to increase likelihood that the generated Nonogram puzzles are non-trivial to solve [1]. The performance of NFA propagator with explanations was compared against decomposition of `regular`[4]. Five instances were generated for sizes $20 \times 20, 20 \times 25, 25 \times 25, 25 \times 30, 30 \times 30$ and 30×35, resulting in a total of 30 instances.

[3]Available at https://github.com/ConSol-Lab/Pumpkin. The version used for experiments is uploaded to Zenodo at https://github.com/JulGvoz/Pumpkin/tree/f8edfbb55c5c8ebe83bb2356caaf2c6511f5844c.

[4]The script for generating Nonogram puzzle instances is available in Zenodo repository containing all code necessary for running the experiments: https://github.com/JulGvoz/nfa-propagator-explanations/tree/227196987c770e9fe682fd581ceed04f93b80b64.

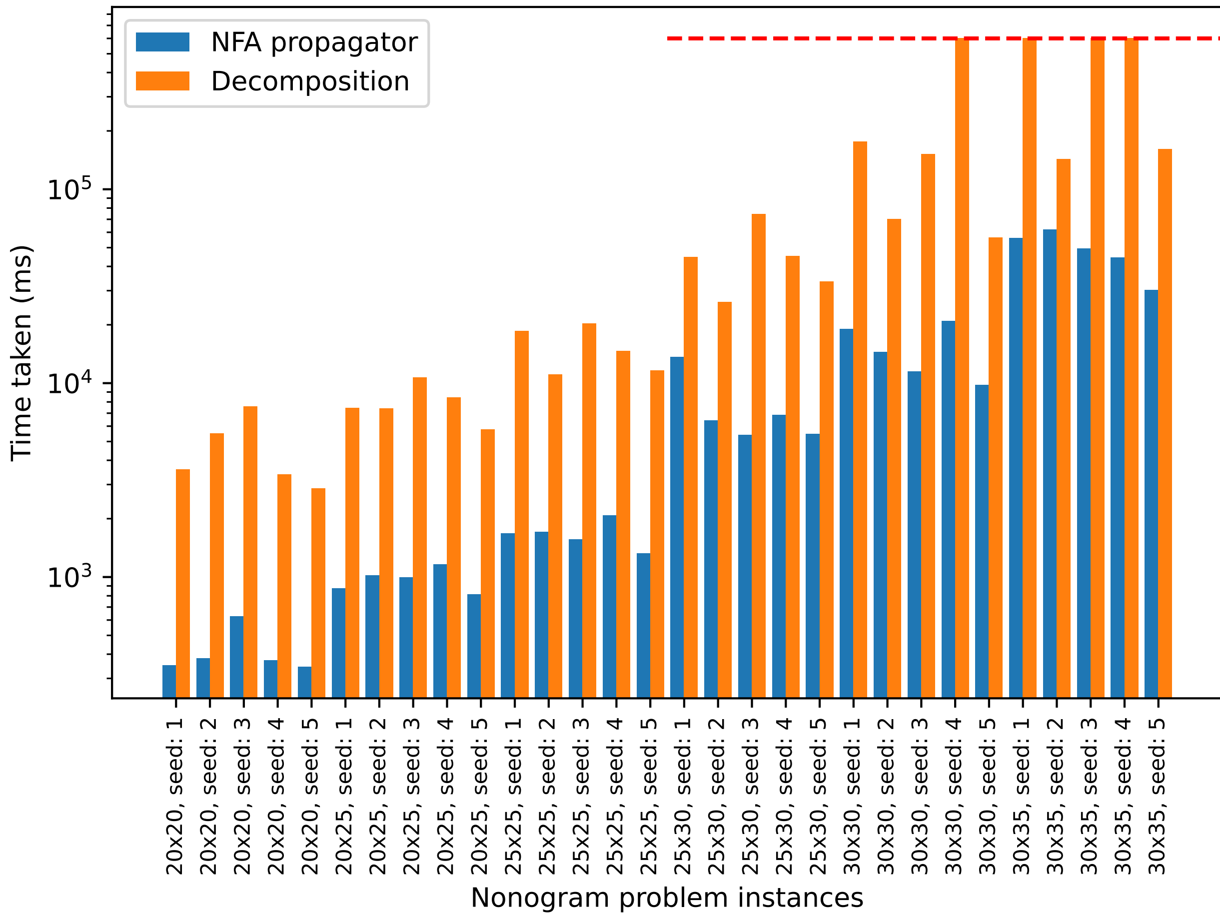


Figure 6: Logarithmic plot, comparing time-to-solve for NFA propagator vs decomposition for Nonograms. Red dashed threshold represents ten minute cut-off mark.

Figure 6 compares the time taken for NFA propagator against decomposition. They were both on all 30 instances, until they either found a solution, or ran out of time. For instances where both approaches finished in time (before ten minute cut-off), NFA propagator finds a solution an order of magnitude (geometric mean of $\times 8.09$) faster than decomposition of `regular`. As the grid size is increased (e.g. from 25×25 to 25×30), the time taken for both approaches increases exponentially, i.e. multiplied by a constant factor. DFA propagator is not visualized (results are shown in Table 2), as our implementation of conversion from NFA to DFA is slow, and the conversion time dominates the result.

Table 2 shows these results in more detail. In some cases, NFA propagator runs into fewer conflicts, and reliably backtracks less, compared to the decomposition. In majority of the instances, average learned clause length, i.e. number of variables in an explanation, is smaller for NFA propagator.

It is likely that decomposition runs into more conflicts due to exploring search space which a domain consistent propagator would have already discarded. Given the order-of-magnitude overall slowdown, this indicates that ”better” explanations (as Feydy and Stuckey suggest [4]) do not compensate for worse propagations, at least for the `regular` constraint.

The results show that NFA-based propagator with our explanations is consistently superior compared to decomposition of the `regular` constraint. Solvers with learning [5] are known to solve Nonogram puzzles quickly, however, it seems that it is not merely learning that allows for finding solutions quickly, but learning with effective explanations. Effective (i.e. domain consistent) propagation seems to be a stronger factor in Nonograms.

DFA propagator, given the exact same regular expression, results in exact same propagations and explanations as the NFA propagator. In our implementation, DFAs are minimized using table-filling algorithm [18], while NFAs are not minimized at all. For Nonograms, this results in smaller finite automata (NFA vs DFA), and smaller overall layered multigraphs. Overall, as both NFA and DFA propagators behave the same (from the perspective of the solver), the metrics of conflict learning and search, namely number of conflicts, average learned clause length, and average backtrack amount, all match. Our observations indicate that the slowdown occurs due to the overhead of converting a given NFA to DFA, and further minimization of the DFA.

Table 2: Results of experiments run on generated Nonogram instances. Missing entries indicate that the solver ran out of time on the instance.

Instance	NFA propagator							Decomposition				DFA propagator						
	Time (s)	Conflicts	Average Learned Clause Length	Average Backtrack Amount	Average RegExpr Size	Average Finite Automata Size	Average Layered Multigraph Size	Time (s)	Conflicts	Average Learned Clause Length	Average Backtrack Amount	Time (s)	Conflicts	Average Learned Clause Length	Average Backtrack Amount	Average RegExpr Size	Average Finite Automata Size	Average Layered Multigraph Size
20×20, seed: 1	0.35	4	1.50	1	44.20	37.80	101.55	3.59	9291	3.39	13.80	8.35	4	1.50	1	44.20	16.40	95.95
20×20, seed: 2	0.38	0	0	0	46	37.15	104.25	5.49	12122	8.17	15.57	8.55	0	0	0	46	16.07	98.33
20×20, seed: 3	0.63	29	9.86	1.48	45.60	37.25	104.30	7.56	17503	13.37	10.26	8.59	29	9.86	1.48	45.60	16.12	98.42
20×20, seed: 4	0.37	2	11	1.50	46	37.45	100.65	3.38	9532	4.10	16.11	8.78	2	11	1.50	46	16.23	94.88
20×20, seed: 5	0.34	0	0	0	44.40	37.05	103.70	2.86	8906	3.04	15.86	8.49	0	0	0	44.40	16.02	97.72
20×25, seed: 1	0.88	8	5.50	1.25	48.16	40.16	127.42	7.43	14009	9.37	19.15	140.07	8	5.50	1.25	48.16	17.58	120.78
20×25, seed: 2	1.02	0	0	0	51.18	41.36	120.60	7.40	15489	7.98	15.93	135.50	0	0	0	51.18	18.18	114.56
20×25, seed: 3	0.99	10	7.50	1.10	51.18	40.82	125.76	10.69	20013	13.28	13.77	135.93	10	7.50	1.10	51.18	17.91	119.44
20×25, seed: 4	1.16	39	8.28	1.38	49.04	39.58	126.31	8.43	18316	11.06	14.67	145.71	39	8.28	1.38	49.04	17.29	119.38
20×25, seed: 5	0.81	1	1	2	50.29	41.67	118.71	5.76	12884	5.42	16.15	134.29	1	1	2	50.29	18.33	112.82
25×25, seed: 1	1.68	37	24.92	1.11	53.56	43.64	157.64	18.55	23102	12.77	19.43	324.76	37	24.92	1.11	53.56	19.32	149.96
25×25, seed: 2	1.71	0	0	0	56.60	45.36	148.54	11.06	16434	5.54	21.68	322.82	0	0	0	56.60	20.18	141.72
25×25, seed: 3	1.56	5	1.80	1.40	56.92	45.68	147.66	20.27	29105	15.64	13.88	299.84	5	1.80	1.40	56.92	20.34	141
25×25, seed: 4	2.08	42	8.64	1.43	55.64	44.64	149.98	14.68	24227	13.11	14.93	327.66	42	8.64	1.43	55.64	19.82	142.80
25×25, seed: 5	1.32	1	1	2	54.84	46.20	144	11.59	18864	6.44	17.67	342.36	1	1	2	54.84	20.60	137.60
25×30, seed: 1	13.61	618	15.80	1.38	58.71	47.40	184.09	44.56	39874	26.82	17.20	–	–	–	–	–	–	–
25×30, seed: 2	6.43	9	6	1.25	60.31	48.75	175.60	26.13	28310	16.46	21.87	–	–	–	–	–	–	–
25×30, seed: 3	5.40	37	18.57	1.38	59.44	48.75	174.89	74.29	70109	27.93	14.86	–	–	–	–	–	–	–
25×30, seed: 4	6.86	71	19.49	1.49	59.87	48.20	178.87	45.28	45949	21.94	16.62	–	–	–	–	–	–	–
25×30, seed: 5	5.46	12	17.92	1.42	60.02	49.33	172.80	33.46	31846	17.00	21.61	–	–	–	–	–	–	–
30×30, seed: 1	18.98	719	76.60	1.40	62.73	50.13	223	176.52	106420	90.64	13.48	–	–	–	–	–	–	–
30×30, seed: 2	14.48	31	18	1.29	65.40	52.27	210.50	70.04	51006	32.03	19.97	–	–	–	–	–	–	–
30×30, seed: 3	11.48	53	30.43	1.43	64.47	52.77	207.10	151.55	93506	37.07	15.38	–	–	–	–	–	–	–
30×30, seed: 4	20.91	605	23.28	1.39	64.47	51.77	213.87	–	464850	65.71	7.54	–	–	–	–	–	–	–
30×30, seed: 5	9.81	4	14	1	65.67	53.73	202.07	56.48	40856	21.62	22.05	–	–	–	–	–	–	–
30×35, seed: 1	55.94	506	42.77	1.53	70.94	54.88	247.52	–	259860	115.88	8.40	–	–	–	–	–	–	–
30×35, seed: 2	61.98	160	31.40	1.26	68.35	55.22	247.45	142.84	102092	33.56	17.56	–	–	–	–	–	–	–
30×35, seed: 3	49.34	108	10.57	1.31	68.85	56.14	240.31	–	355322	73.05	9.00	–	–	–	–	–	–	–
30×35, seed: 4	44.32	631	112.33	1.56	68.11	55.22	245.18	–	289486	81.22	8.92	–	–	–	–	–	–	–
30×35, seed: 5	30.18	4	5.50	1	68.97	57.28	231.12	161.28	92555	38.26	19.32	–	–	–	–	–	–	–

6 Responsible Research

Scientific process fundamentally works by building upon prior research. Reproducibility lies at the heart of this process - in the goal of finding scientific truth, methods and results should be reproducible by other researchers.

In order to ensure full reproducibility of our experiments, we have published a reproducibility package of our work.[5] This includes: the source code of our propagator and all related software, as open source; the test set and results of all experiments; detailed instructions on how to reproduce the exact paper as closely as possible. Our reproducibility package has been version controlled from the beginning of our research.

Combined with the detailed explanation of our experimental setup in Section 5, this allows others to run the software on their own machines, and reproduce the experiments in this work.

Failures to reproduce results often stem from hidden variables that were not included in the original studies. Even in the event that our results cannot be reproduced from the description in the paper, additional details might allow to investigate the reasons for non-reproducibility. We have taken great care to describe not only strictly necessary information of our setup, but also details which *might have* impacted the experiments.

In light of transparency, we have also published the test set and results of all experiments that were run. Thanks to version control software, all discarded experiment results (i.e. due to code changes and improvements) is also present. Combined with the previous points, this allows our study to be reproduced near identically. Future research can built upon our results, and can compare them with their own, same as we did with prior research.

7 Conclusions and Future Work

In this work, we have developed an NFA propagator for `regular` constraint, which produces explanations in order to facilitate lazy clause generation and conflict learning. We combined existing concepts relating to the `regular` constraint: DFA-based domain consistent propagator [13], extended to work with NFAs, together with explanations created using the core idea from the MDD propagator [5]. We tested whether the overhead of domain consistent propagator for the constraint is better or worse than decomposing the constraint. Our experimental results, comparing our NFA-based propagator to a standard decomposition of the `regular` constraint, show that a specialized propagator is an order of magnitude faster. Our NFA-based propagator can achieve domain consistent propagations and good explanations (equivalent to what DFA-based equivalent propagator would create), without the overhead of DFA construction. Our approach to generating explanations is general, and works both on NFAs and DFAs as inputs.

Future work involves optimizing the propagator to work incrementally, so that the layered multigraph structure does not need to be updated each time from scratch in deeper layers. One key weakness of our explanation algorithm is that it requires the entire initial layered multigraph, while also referring to its current state of living/dead edges. We believe that it is possible to continue extending the ideas from the existing MDD propagator [5] to solve both these issues.

[5] Reproducibility package is available at https://github.com/JulGvoz/nfa-propagator-explanations/tree/227196987c770e9fe682fd581ceed04f93b80b64.

A possible area for more research is to determine whether it is possible to combine the ideas from both NFA and DFA approaches, in order to result in a better hybrid approach. For example, it may be possible to lessen the overhead of DFA construction by working directly on the layered multigraph (made from NFA) instead of first converting the finite automaton, then converting it into a layered multigraph.

Another area of implement is to find better decompositions of the `regular` constraint. If a specialized decomposition can be made, which would match the quality of domain consistent propagations, this could allow to achieve good explanations without the need of a dedicated propagator.

References

[1] K Joost Batenburg and Walter A Kosters. On the difficulty of nonograms. *ICGA journal*, 35(4):195–205, 2012.

[2] Christian Bessiere, Emmanuel Hebrard, Brahim Hnich, Zeynep Kiziltan, Claude-Guy Quimper, and Toby Walsh. Reformulating global constraints: The slide and regular constraints. In *Abstraction, Reformulation, and Approximation*, pages 80–92, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[3] Kenil CK Cheng, Wei Xia, and Roland HC Yap. Space-time tradeoffs for the regular constraint. In *International Conference on Principles and Practice of Constraint Programming*, pages 223–237. Springer, 2012.

[4] Thibaut Feydy and Peter J Stuckey. Lazy clause generation reengineered. In *International Conference on Principles and Practice of Constraint Programming*, pages 352–366. Springer, 2009.

[5] Graeme Gange, Peter J Stuckey, and Radoslaw Szymanek. MDD propagators with explanation. *Constraints*, 16(4):407–429, 2011.

[6] George Katsirelos, Nina Narodytska, and Toby Walsh. Reformulating global grammar constraints. In *International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 132–147. Springer, 2009.

[7] Mikael Zayenz Lagerkvist. *Techniques for efficient constraint propagation*. PhD thesis, KTH, 2008.

[8] Mikael Zayenz Lagerkvist and Gilles Pesant. Modeling irregular shape placement problems with regular constraints. In *First workshop on bin packing and placement constraints BPPC'08*, 2008.

[9] Polina Makeeva and Radoslaw Szymanek. Revisiting the regular constraint. 2009.

[10] Matthew W Moskewicz, Conor F Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient sat solver. In *Proceedings of the 38th annual Design Automation Conference*, pages 530–535, 2001.

[11] Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. Minizinc: Towards a standard CP modelling language. In Christian Bessière, editor, *Principles and Practice of Constraint Programming – CP 2007*, pages 529–543, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[12] Olga Ohrimenko, Peter J Stuckey, and Michael Codish. Propagation = lazy clause generation. In *Principles and Practice of Constraint Programming–CP 2007: 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007. Proceedings 13*, pages 544–558. Springer, 2007.

[13] Gilles Pesant. A regular language membership constraint for finite sequences of variables. In *International conference on principles and practice of constraint programming*, pages 482–495. Springer, 2004.

[14] Michael O Rabin and Dana Scott. Finite automata and their decision problems. *IBM journal of research and development*, 3(2):114–125, 1959.

[15] Andreas Schutt. *Improving scheduling by learning*. University of Melbourne, Department of Computer Science and Software Engineering, 2011.

[16] Andreas Schutt, Thibaut Feydy, Peter J Stuckey, and Mark G Wallace. Why cumulative decomposition is not as bad as it sounds. In *Principles and Practice of Constraint Programming-CP 2009: 15th International Conference, CP 2009 Lisbon, Portugal, September 20-24, 2009 Proceedings 15*, pages 746–761. Springer, 2009.

[17] Sathiamoorthy Subbarayan. Efficient reasoning for nogoods in constraint solvers with bdds. In *Practical Aspects of Declarative Languages: 10th International Symposium, PADL 2008, San Francisco, CA, USA, January 7-8, 2008. Proceedings 10*, pages 53–67. Springer, 2008.

[18] Bruce W Watson. A taxonomy of finite automata minimization algorithms. 1993.

[19] Luhan Zhen, Yonggang Zhang, Jingyao Li, Yanzhi Li, and Zhanshan Li. Improved bit-based filtering algorithm for regular constraint. *Expert Systems with Applications*, 235:121259, 2024.