Propagating Regular Counting with Lazy Clause Generation

DFA vs counter-DFA for finite domain satisfaction problems

Daniël Ravensbergen[1]
Supervisors: 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: Daniël Ravensbergen
Final project course: CSE3000 Research Project
Thesis committee: Emir Demirović, Imko Marijnissen, Andreea Costea

Abstract

cDFAs offer a more natural encoding of counting regular patterns—a prevalent problem in timetabling and sequencing—than widely-used DFAs. A cDFA-based propagator for finite domain constraint solving has been shown to solve satisfaction problems faster and make more propagations than the decomposition of a DFA-based regular constraint. This paper extends that algorithm with explanations for lazy clause generation and shows that it often solves satisfaction problems faster than a DFA-based regular constraint both decomposed and propagated, even though the explanations are simple and nogood quality is worse. Additionally, this paper showcases a generator for cDFAs and DFAs that are equivalent to each other to aid comparison of the constraints.

1 Introduction

Constraint programming (CP) is a paradigm for solving finite domain combinatorial satisfaction and optimization problems. These problems are often NP-Hard. As such, these problems tend to have sizable search spaces. CP allows for the pruning of unvisited branches of the search space. It does this by propagating constraints a solution needs to satisfy.

One of these constraints is the regular constraint. It is satisfied when its input is in a given regular language. The regular constraint is often based on deterministic finite automaton (DFA), because they accept inputs that are in a given regular language and reject ones that aren't.

Regular counting is a feature of regular languages that specifies how many times a pattern is allowed to occur. It is well known as the regular expression operation $\{n\}$. This operation is quite common when working with regular languages. According to Chapman and Stolee[1] regular counting using regex is used in 20% of 1204 python projects. In the field of CP, Regular counting is useful for constraining a variety of problems such as sequencing and timetabling[2] where you want to specify the number of occurrences of patterns.

Though regular counting can be modeled for CP with the aforementioned DFA-based regular constraints, other finite automata that describe regular languages can also be considered[3]. One class of these are counter-DFAs (cDFA), which have been used to implement 56 global constraints, as seen in the global constraint catalog[1] by Beldiceanu et al.[4]. The counter of a cDFA increases after specific regular patterns and the cDFA accepts only when it counts to specific numbers. Because of this, repeating regular patterns are able to be represented with higher numbers in stead of repeating structures as would be the case with DFAs. This can result in more compact finite automata.

Beldiceanu et al.[2] show that a dedicated constraint based on a cDFA is not only more straightforward to model regular counting with, but also more efficient, as it prunes more branches, providing significantly faster solve times. Even though they provide an algorithm for exact regular counting that is only bounds consistent[2], they show that it prunes more than earlier propagators.

Up to now, this algorithm had not been implemented with explanations for Lazy Clause Generation (LCG). LCG is a newer technology in the field of CP that generates explanations for propagations and uses those to learn from conflicts.

While Beldiceanu et al.[2] analyze the performance of their propagator, it is not clear what the relation is between counts, amounts of states and sizes of alphabet of a cDFA and performance, or if it is even beneficial to use this constraint over the readily available DFA-based regular constraints when extended with explanations for LCG. Additionally, comparing cDFAs and DFAs can be difficult, as no algorithm to convert between the two has been proposed.

This paper presents the first cDFA based regular counting propagator with explanations. To this end we showcase simple explanations for propagating the count and variables in the input sequence. In addition, we provide a generator for cDFAs and equivalent DFAs and use it to analyze the performance and nogood quality of our propagator compared to two DFA-based regular constraints respectively

[1]See https://sofdem.github.io/gccat/

decomposed into global constraints and propagated with Gvosdiovas et al.'s[5] LCG extension of Pesant's[6] DFA-based propagator. We show that it is beneficial to use our propagator for satisfaction problems, though our explanations are simple and have room for improvement.

We find that generally, our propagator has a lower solve time, but propagates more and runs into more conflicts than the compared methods. The quality of the nogoods of our propagator are generally worse than those of the compared methods.

Section 2 provides the background knowledge our research was based on, explaining CP and LCG, defining the used finite automata and explaining Beldiceanu et al.'s algorithm. Section 3 discusses the history of the field of CP and how it relates to our contributions. Section 4 presents how we provide explanations for LCG and our generator for cDFAs and equivalent DFAs. Section 5 elaborates on how our experiments were run, elaborates on the results and mentions their limitations. Section 6 discusses considerations with regards to ethics and the reproducibility of our research. Section 7 summarizes our contributions, highlighting unexplored avenues of research.

2 Preliminaries

This section explains the following topics relevant to our research: constraint programming in Section 2.1, lazy clause generation in Section 2.2, DFA and cDFA in Section 2.3 and the cDFA-based constraint in Section 2.4.

2.1 Constraint Programming

Constraint programming (CP) is a paradigm in which combinatorial problems are modeled with decision variables and constraints that need to be satisfied.

In this field, finite domain (FD) CP is one of the biggest subdomains. FD constraint solvers are programs that can solve FD combinatorial satisfaction and optimization problems, which respectively ask for *any* solution and a solution that *optimizes* a given metric. FD constraint solvers are able to prune branches of the search space without solutions, without entering them. They do this by deducing which assignments of the variables prevent the constraints from being satisfied, or in other terms results in a conflict. This can often lead to a sizable decrease in the search space compared to a brute-force approach.

The algorithms that make such deductions are called *propagators*, pruning based on a propagator's deduction *propagating* a constraint, and a single results of propagating a *propagation*. A proper propagator should never prune solutions.

Example 1: Consider the constraint $a = b$ with variables $a \in \{1, 2\}, b \in \{2, 3\}$. The constraint holds iff a is exactly equal to b. To propagate this constraint, we can update both the domains of a and b to $a \cap b$, as any value that is not present in both can not satisfy the constraint. We end up with $a = 2$ and $b = 2$. □

There are two approaches to propagating a constraint. One is to use an algorithm designed for that specific constraint. The other is to decompose the constraint into many simpler constraints with simpler propagators. When deciding which algorithm or decomposition to use, it is worth considering the trade-off between strength—how many propagations are applied and how useful are they—and speed—how fast are propagations computed.

Two relevant concepts that apply to propagators are *bounds consistency* and *domain consistency*. A propagator is bounds consistent when, if you apply it sequentially, it doesn't propagate on the bounds of the domain more than once; it has made all the deductions it can regarding the bounds. A propagator is domain consistent when, if you apply it sequentially, it doesn't propagate on the domain more than once; it has made all the deductions it can.

2.2 Lazy Clause Generation

Lazy clause generation (LCG) is an extension to FD constraint solvers based on solvers from the subdomain of Boolean CP: satisfiability-solvers (SAT-solvers). SAT-solvers are able to solve Boolean satisfiability problems. Given a set of variables and a set of Boolean clauses, they find an assignments of the variables that makes all clauses true. SAT-solvers are able to make use of *conflict learning*, something FD constraint solvers can not do. When a conflict occurs, they can deduce which decisions resulted in that conflict from the clauses in an implication graph. The negation of the clause that represents that decision—also referred to as a *nogood*—is then added as a new constraint. This is especially useful for searching large search spaces.

LCG adds advantages of SAT-solvers—like conflict learning—to FD constraint solvers. It does this by making propagations provide Boolean clauses for both the reason for propagation and the propagation itself. This combination of reason and propagation is called an *explanation*.

Example 2: Consider the constraint $a = b$ with $a \in \{1, 2\}, b \in \{2, 3\}$ again. By propagating on the bounds we would get the propagation clauses $[a \geq 2]$ and $[b \leq 2]$. The explanations would then be $[b \geq 2] \rightarrow [a \geq 2]$ and $[a \leq 2] \rightarrow [b \leq 2]$ □

Generally, the shorter the explanation, the more useful it is, as the more clauses you add to the reason, the less generalizable it is to the rest of the search space. Because of this, LCG is known to often learn more from decomposition into smaller constraints, compared to more complex propagators[7].

To ensure the correctness of explanations, it is good practice to implement an *inference checker* for a propagation. When solving a problem with an inference checker enabled, it will verify each explanation. It does this by applying each clause of an explanation's reason and the negation of its propagation, after which the constraint should no longer be satisfied if the explanation was correct. Because this introduces overhead, inference checkers are generally not enabled by default.

2.3 Finite Automata

2.3.1 DFA

Recall that a deterministic finite automaton (DFA) is a finite automaton that recognizes regular languages. It is defined by the five-tuple[8]:

$$\langle Q, \Sigma, \delta, q_0, F \rangle$$

Where:

1. Q is a finite set called the *states*;
2. Σ is a finite set called the *alphabet*;
3. $\delta : Q \times \Sigma \rightarrow Q$ is the *transition function*;
4. $q_0 \in Q$ is the *initial state*;
5. $F \subseteq Q$ is the *set of accept states*.

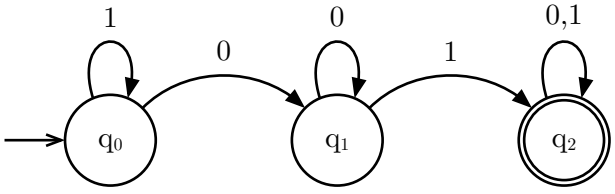


Figure 1: DFA that accepts the sequence "01"

Given a finite sequence X, where for each $x_i \in X : x_i \in \Sigma$, a DFA accepts X iff it is in an accepting state after consuming X. That is: starting in q_0 ($q \leftarrow q_0$), for each $x_i \in X$ in order, go from one state to the next, according to δ;

$$q' = \delta(q, x_i) : q \leftarrow q'$$

Example 3: Consider the DFA in Figure 1. When given the string "01", it will visit the following states in order: $q_0 \rightarrow q_1 \rightarrow q_2$. The final state it ends up in is q_2, which is an accept state; the DFA accepts. □

2.3.2 cDFA

Beldiceanu et al.[2] describes a subclass of counter-DFAs (cDFA). When referring to cDFAs in this paper, we mean this specific subclass. It is similar to a DFA, but differs in two aspects: It only has accepting states; it has a singular counter that is initialized to zero and increments on certain transitions. In this paper it is defined by the five-tuple:

$$\langle Q, \Sigma, \delta, q_0, K \rangle$$

Where:

1. Q is a finite set called the *states*;
2. Σ is a finite set called the *alphabet*;
3. $\delta : Q \times \Sigma \rightarrow Q \times \mathbb{N}$ is the *transition function*;
4. $q_0 \in Q$ is the *initial state*;
5. $K \subseteq \mathbb{N}$ is the *set of accepted counts.*

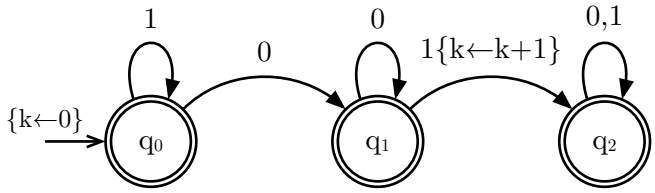


Figure 2: cDFA with $K = \{1\}$ that accepts the sequence "01"

Given a finite sequence X, where for each $x_i \in X$ s.t. $x_i \in \Sigma$, a DFA accepts X iff its counter has an accepting count after consuming X. That is: starting in q_0 ($q \leftarrow q_0$), with counter k initialized to zero ($k \leftarrow 0$), for each $x_i \in X$ in order, go from one state to the next and increment k with the count of the transition, according to δ;

$$\langle q', inc \rangle = \delta(q, x_i) : q \leftarrow q', k \leftarrow k + inc$$

For ease of notation, when $\delta(q, \ell) = \langle q', inc \rangle : \delta_Q(q, \ell) = q'$ and $\delta_{\mathbb{N}}(q, \ell) = inc$

Example 4: Consider the cDFA in Figure 2. When given the string "01", it will visit the following states in order: $q_0 \rightarrow q_1 \rightarrow q_2$. It increments by one when going from q_1 to q_2. The final count is one, which is an accepted count; the cDFA accepts. □

2.4 cDFA based constraint

2.4.1 Definition

The cDFA-based regular counting constraint is defined as:

$$\text{regular\_cDFA}(X, A, count)$$

where:

- $dom(var)$ is the domain of variable var;
- X is a sequence of variables $x_1 ... x_n$, $dom(x_i) \subseteq \Sigma$;
- A is a cDFA;
- $count$ is a variable, with $dom(count)$ as the set of accepting counts of A;

The constraint is satisfied when A counts to $count$ when consuming sequence X.

2.4.2 Propagating the cDFA based constraint

The propagation proposed by[2] works as follows: For each variable in the sequence, it computes the minimum and maximum possible counter increases before and after consuming that variable when

consuming the sequence. It then prunes what values of *count* fall outside of the possible range of counter increases after consuming the entire sequence. For each variable in the sequence, it prunes any value of that variable that results in a range of possible counts that does not overlap with the accepting counts. Mathematical descriptions with examples follow.

We define the following functions:

- QCF(i): The set of pairs $\langle q, c\rangle$, where c is the set of possible counter increases when the cDFA consumes a sequence of length i, starting in the starting state and ending in state q—the possible counter increases c to reach state q in i transitions.
- QCB(i): The set of pairs $\langle q, c\rangle$, where c is the set of possible counter increases after visiting state q after exactly $n - i$ transitions when the cDFA consumes a sequence of length n, starting in the starting state and ending in some other state—the possible counter increases c to reach a final state from state q in i transitions.
- $\underline{\text{QCF}}(i)$ (and $\overline{\text{QCF}}(i)$): The set of pairs $\langle q, c\rangle$, where c is the minimum (maximum) counter increase to reach state q.
- $\underline{\text{QCB}}(i)$ (and $\overline{\text{QCB}}(i)$): The set of pairs $\langle q, c\rangle$, where c is the minimum (maximum) counter increase to reach a final state from state q.

The behaviour of QCF and QCB are described in Appendix A. For polynomial time algorithms for $\underline{\text{QCF}}(i)$, $\overline{\text{QCF}}(i)$, $\underline{\text{QCB}}(i)$ and $\overline{\text{QCB}}(i)$, see Appendix B.

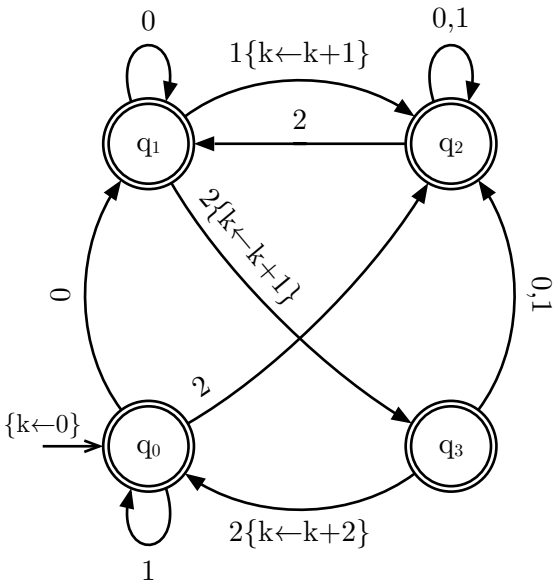


Figure 3: cDFA with $\Sigma = \{0, 1, 2\}$ and $K = \{2, 3, 4\}$

Propagation on the bounds of *count* is implemented as:

$$count \geq min(c) \mid \langle q, c\rangle \in \underline{\text{QCF}}(n)$$

$$count \leq max(c) \mid \langle q, c\rangle \in \overline{\text{QCF}}(n)$$

Example 5: We will show a propagation on the upper bound of *count*. Consider the cDFA with $K = \{2, 3, 4\}$ in Figure 3. Given an arbitrary sequence X of $n = 4$ variables with Σ as domain, we have:

1. $\overline{\text{QCF}}(4) = \{\langle q_0, \{3\}\rangle, \langle q_1, \{3\}\rangle, \langle q_2, \{3\}\rangle, \langle q_3, \{2\}\rangle\}$
2. $max(c) = 3$
3. $count \leq 3$
4. $dom(count) = \{2, 3\}$

□

To implement propagation for variables in X, a value ℓ is removed from the domain of x_i iff for all states, taking transition ℓ results in a range of counts that does not overlap with the domain of *count*:

$$\forall q \left| \begin{bmatrix} \langle q,\underline{c}\rangle \in \underline{\mathrm{QCF}}(i-1) \\ \langle q,\overline{c}\rangle \in \overline{\mathrm{QCF}}(i-1) \\ \langle q',\, inc\rangle = \delta(q,\ell) \\ \langle q',\underline{c'}\rangle \in \underline{\mathrm{QCB}}(i+1) \\ \langle q',\overline{c'}\rangle \in \overline{\mathrm{QCB}}(i+1) \end{bmatrix} : \left[\underline{c} + inc + \underline{c'}, \overline{c} + inc + \overline{c'}\right] \cap dom(count) = \emptyset \right.$$

Example 6: We will show how to derive the propagation $x_3 \neq 2$. Consider the cDFA in Figure 3, but now with with $K = \{4, 5, 6\}$. For each state we have listed the ranges $\left[\underline{c} + inc + \underline{c'}, \overline{c} + inc + \overline{c'}\right]$:

$$q_0 : [0 + 0 + 0, 0 + 0 + 0] = [0, 0]$$
$$q_1 : [0 + 1 + 0, 0 + 1 + 2] = [1, 3]$$
$$q_2 : [0 + 0 + 0, 1 + 0 + 1] = [0, 2]$$
$$q_3 : [1 + 2 + 0, 1 + 2 + 0] = [3, 3]$$

The intersection of each range with $dom(count)$ is $\emptyset$, thus $x_3 \neq 2$. □

3 Related work

The DFA based regular constraint and an accompanying domain consistent propagator were originally proposed by Pesant[6]. Gvosdiovas et al.[5] recently created a version of Pesant's propagator with explanations for LCG and extended it to work with non-deterministic finite automata (NFA). As we also extend a propagator with explanations for LCG, this research was influential in how we approached our research. We compare our propagator with their DFA-based propagator. However, a comparison between their NFA-based constraint and our cDFA-based constraint does not seem worthwhile to us. Regular counting is mostly sequential and as such we can't think of ways that non-determinism would provide significant advantages for regular counting.

Using cDFAs for regular counting was originally proposed by Beldiceanu et al.[3], as the use of a counter allows for a simpler representation to count regular patterns opposed to normal DFAs. Later, Beldiceanu et al.[2] proposed propagators for "at least", "at most" and "exact" counting of regular patterns using cDFAs, that are domain consistent on the input sequence. Their "at most" and "at least" propagators were shown to also be domain consistent on the count, whereas their "exact" propagator only provides bounds consistency. They further proved that computing satisfiability for their "exact" constraint is NP-Hard.

Although they show that their propagators prune more than decomposition of a DFA-based regular constraint, they only show this for FD constraint solving *without* LCG. This poses the question if that is still the case when implemented *with* LCG, which can benefit from decomposition. Another question left unanswered is for what configurations of cDFA it offers an significant advantage to model problems with them. Our research answers these questions.

Martin and Pearson[9] provided an algorithm that decides when bounds consistency implies domain consistency for a given cDFA. This also analyzes when cDFAs are more efficient, but does so by analyzing their structure in stead of their parameters and provides no comparison with DFAs.

The cost-regular constraint proposed by Demassey et al.[10] has many similarities—both accumulate values when taking transitions in finite automata and compare the accumulation to a variable —to the regular counting constraint proposed by Beldiceanu et al.[2], but propagates less and has asymptotically worse space complexity. No comparison of these two propagators extended with explanations for LCG has been done, but due to the limited scope of our research, we left this untouched.

4 Propagating regular_cDFA with explanations

This section showcases our contributions. Section 4.1 showcases our explanations for propagating, and Section 4.2 describes our generator for random cDFA and DFA that are equivalent.

4.1 Explanations

These explanations are directly based on the propagations from Beldiceanu et al.[2]. As such, they are simple, but lay the groundwork for future improvements. We implemented an inference checker to verify the validity of our explanations.

4.1.1 The count

The propagation clause for *count* is always an update to the bounds of the domain:

$$[count \geq value] \text{ or } [count \leq value]$$

The clauses making up the reason for propagating on *count* represent the values in the domains of the variables in X. For example, $dom(x_i) = \{1, 3\} \rightarrow [x_i \geq 1] \wedge [x_i \neq 2] \wedge [x_i \leq 3]$. The state of the domain of *count* is not included in the reason, as other possible values of *count* have no effect on what counts are possible.

4.1.2 The input sequence

The propagation clauses for the variables in X are unit clauses for each value that is impossible:

$$[x_i \neq value]$$

The clauses making up the reason for propagating on x_i not only consist of the values in the domains of the variables in X. They also include clauses for the upper and lower bound of *count* and unit clauses for each hole in *count* that falls in the range of possible counts of any state.

$$[count \geq \text{LB}] \wedge [count \leq \text{UB}] \wedge$$

$$\bigwedge \left\{ [count \neq value] \middle| value \in \left(\bigcup \left\{ \left[\underline{c} + inc + \underline{c'}, \overline{c} + inc + \overline{c'} \right] \cap [\text{LB}, \text{UB}] - dom(count) \middle| \forall q \middle| \begin{bmatrix} \langle q, \underline{c} \rangle \in \underline{\text{QCF}}(i-1) \\ \langle q, \overline{c} \rangle \in \overline{\text{QCF}}(i-1) \\ \langle q', inc \rangle = \delta(q, \ell') \\ \langle q', \underline{c'} \rangle \in \underline{\text{QCB}}(i+1) \\ \langle q', \overline{c'} \rangle \in \overline{\text{QCB}}(i+1) \end{bmatrix} \right\} \right) \right\}$$

4.2 Random equivalent finite automata generator

We created a generator for random cDFA and DFA that are equivalent to be able to compare equivalent constraints. Generation consists of the following three steps:

1. Construct a random intermediate FA.
2. Convert the intermediate FA to a cDFA.
3. Using the intermediate FA, construct a DFA that is equivalent to the generated cDFA, that is then minimized.

To work with the DFA and to minimize the equivalent DFA, the python library FAdo[2] was used.

4.2.1 Constructing the intermediate finite automaton

An FA with $n + 1$ states and only one accepting state—where n is the amount of states our resulting cDFA will end up with—is generated randomly. See Figure 4. This is done in a way that a random Hamiltonian path—marked with red for illustration—from the starting state to the accept state exists, to guarantee that the resulting automata accept some sequences. The accept state is incomplete; it does not have any outgoing edges. A set of accepting counts K is defined.

[2]https://fado.dcc.fc.up.pt/

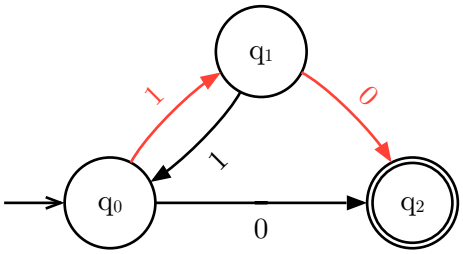


Figure 4: Intermediate finite automaton, $K = \{1, 3\}$

4.2.2 Constructing the cDFA

The ingoing edges to the accept state of the intermediate FA are marked as incrementing by one. The accept state is merged with the starting state. All states are made accepting.

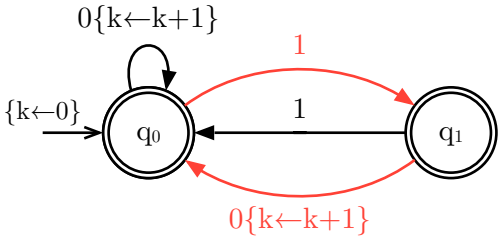


Figure 5: Constructed cDFA, $K = \{1, 3\}$

4.2.3 Constructing the equivalent DFA

From the intermediate FA $max(K) + 1$ copies are made.

$$\{copy_i \mid i \in 0, 1, ..., max(K)\}$$

Each accepting state of $copy_i$ is merged with the starting state of $copy_{i+1}$, essentially concatenating the pattern $max(K) + 1$ times. The final incomplete state is made into a rejecting trap-state. States corresponding to the rejecting states of $copy_i$ are accepting iff $i \in K$. See Figure 6. The resulting DFA is then minimized with Hopcroft's method[11] using FAdo[2].

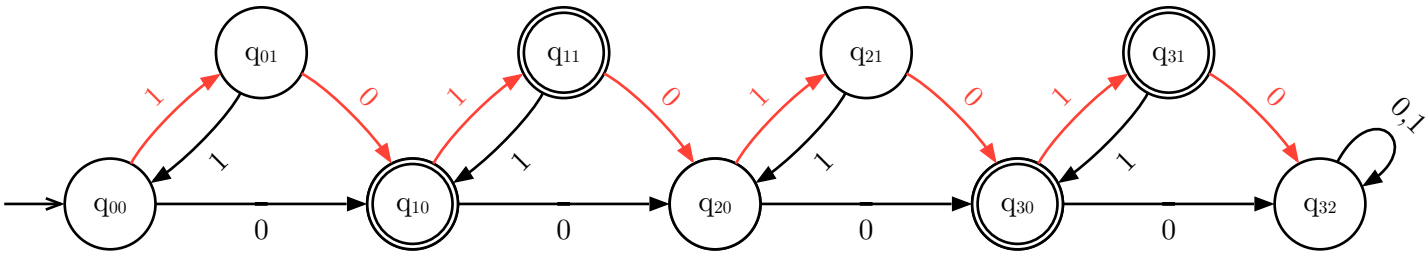


Figure 6: Equivalent DFA

5 Experimental Setup and Results

Our results show that our propagator generally solves faster than both decomposition and Gvosdiovas et al.'s[5] propagator. It generally propagates more often and runs into more conflicts than Gvosdiovas et al.'s propagator, but does so significantly less than decomposition. The quality of its nogoods is respectively slightly and significantly worse than for the nogoods of Gvosdiovas et al.'s propagator and decomposition.

This section explains our setup in Section 5.1, examines our results in depth in Section 5.2 and then highlights limitations of our approach in Section 5.3.

5.1 Setup

The regular_cDFA constraint and the DFA-based regular constraint were implemented in Pumpkin[12] using Rust. The DFA-based regular constraint was adapted from the code of Gvosdiovas et al.[5]. The decomposition of the DFA-based regular constraint uses the decomposition built into

Pumpkin. Minizinc[13] was used to model our benchmarks using the constraints. Those models were then provided with json files describing individual instances to run, which were generated by Python scripts. Shell scripts were used to generate many instances at once. A python script was used to run all instances of a benchmark and store the results. A python script was used to aggregate, analyse and graph the data.

This approach was inspired by Gvosdivas et al.. An important change is that we opted to not parallelize the running of the instances of our benchmarks, to make sure that their solve speed was impacted as little as possible by random variables.

The benchmarks were run on a laptop with an Intel Core i7-10750H running at 5.00GHz with 15.40GiB RAM running NixOS 26.05 without a graphical user interface, to minimize background processes.

Parameter	**Values**
Alphabet size ($\|\Sigma\|$)	3, 6
Amount of states of the cDFA ($\|Q\|$)	3, 5, 7, 9
Accepting counts (K)	{1, 2}, {1, 4}, {1, 8}, {1, 16}

Table 1: The different values of the parameters used to generate random cDFA and equivalent DFA

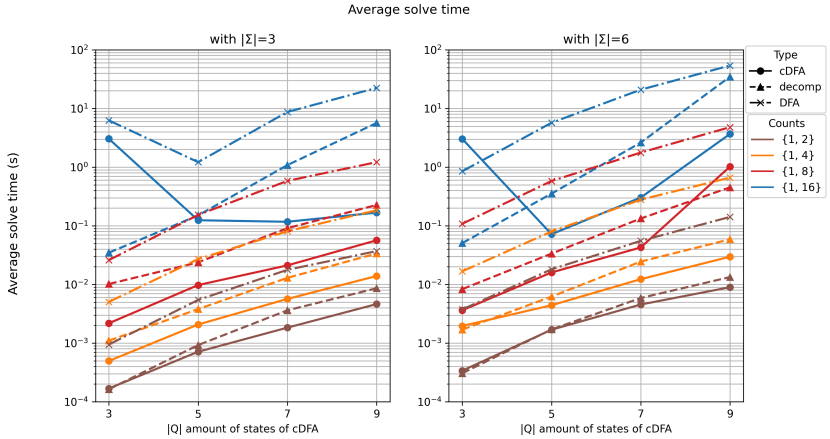
For each unique combination of cDFA qualities found in Table 1, 200 instances were generated by our random generator. The length of the sequence of an instance equals $|Q| \cdot (max(K) + 1)$. This way, there exists a sequence which counts past the maximum count in stead of only counting up to the maximum count. Each instance was then used to generate satisfaction problem instances for the three propagators such that the respective finite automaton accepts the sequence both front to back and back to front. This symmetric approach was used to increase the complexity and reduce the number of solutions. Each problem instance was then run with a cut-off time of five minutes.

To compare performance and quality of the nogoods, for each configuration the following metrics were compared based on the average of the problem instances of each propagator:

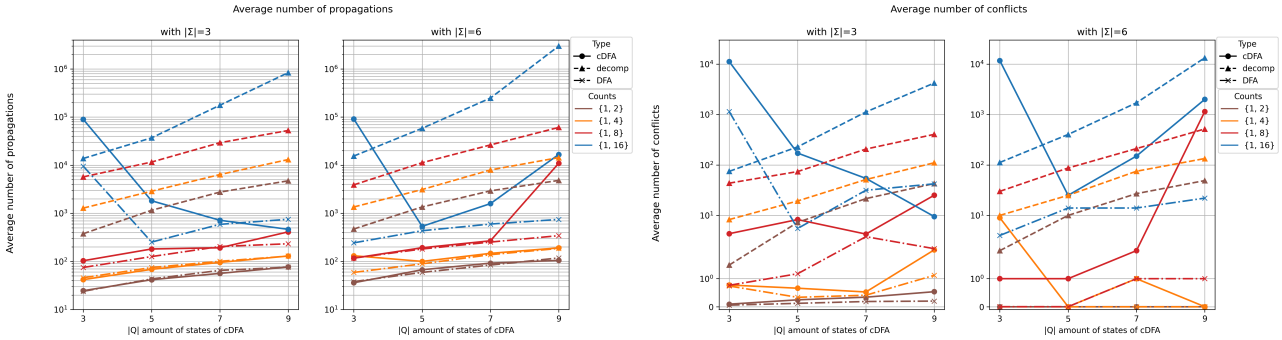
- Solve time;
- Number of propagations;
- Number of conflicts;
- Average learned nogood length;
- Average LBD.

Because the symmetric approach makes some problem instances unsatisfiable, any problem instance found to be unsatisfiable was pruned for all three propagators. Solve time and number of propagations and conflicts of unsolved problem instances were multiplied by two before being used to calculate the average, to roughly estimate these metrics. Not including these problem instances or including them as is, would result in underestimating them, whereas this approach may result in an overestimate or smaller underestimate. For average learned nogood length and average LBD, problem instances that were not solved or that did not have any conflicts were pruned.

5.2 Results



(a) Solve time



(b) Number of propagations

(c) Number of conflicts

Figure 7: Average performance of the propagators, with varying alphabet sizes ($|\Sigma|$), number of states of the cDFA ($|Q|$) and accepting counts (K).

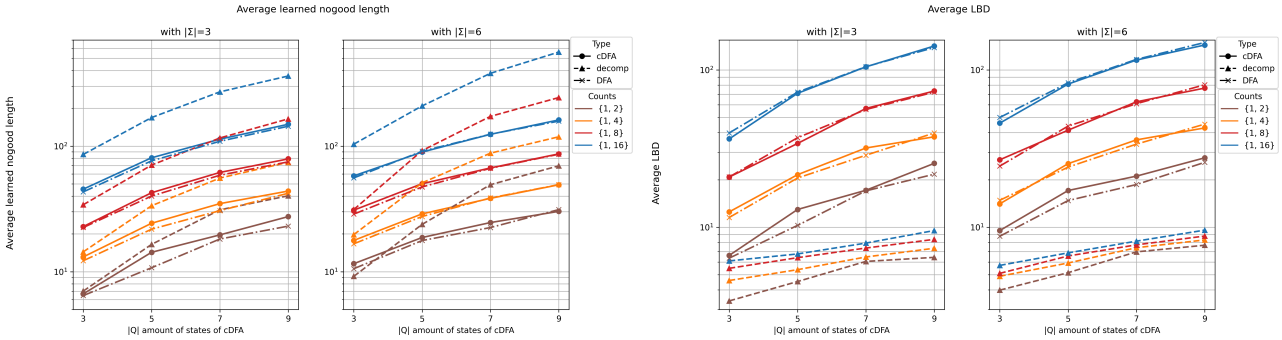
After running the benchmark, four problem instances were found to be unsatisfiable. Furthermore, our propagator, decomposition and Gvosdiovas et al.'s[5] propagator respectively did not solve three, two and nine problem instances.

As shown by Figure 7.a, our propagator solves in a similar amount of time or faster compared to both decomposition and Gvosdiovas et al.'s propagator. The exceptions are instances with $|Q| = 3, K = \{1, 16\}$, were our propagator is significantly slower than decomposition for both alphabet sizes and significantly slower than Gvosdiovas et al.'s propagator when $|\Sigma| = 6$. Our propagator is also slower than decomposition for $|\Sigma| = 6, |Q| = 9, K = \{1, 8\}$.

As shown by Figure 7.b, our propagator has significantly less propagations than decomposition—a decrease of factor 20 for $|\Sigma| = 3, |Q| = 3, K = \{1, 2\}$, up to factor 1000 for larger parameters. As this factor keeps increasing it is likely that this difference keeps increasing for even larger parameters. For configurations with $K \in \{\{1, 2\}, \{1, 4\}\}$, our propagator has a similar number of propagations as Gvosdiovas et al.'s propagator. For larger parameters, our propagator propagates more.

As shown by Figure 7.b, the amount of conflicts our propagator runs into are upper-bounded by decomposition and lower-bounded by Gvosdiovas et al.'s[5] propagator. There are outliers however. For $|Q| = 3, K = \{1, 16\}$ and for $|\Sigma| = 6, |Q| = 9, K = \{1, 8\}$, our propagator runs into more conflicts than decomposition. Additionally, $|\Sigma| = 3, K = \{1, 16\}$ has a downward trend with respect to the number of states, contrary to the results for other configurations. For $|\Sigma| = 3, |Q| = 9, K = \{1, 16\}$, our propagator even runs into less conflicts than Gvosdiovas et al.'s propagator. Important to note is that for some configurations, both our propagator and Gvosdiovas et al.'s propagator run into zero conflicts.

Our propagator propagating less and having less conflicts than decomposition can be explained by decomposition having many smaller propagators. Our propagator propagating more and having more conflicts than Gvosdiovas et al.'s propagator could be explained by the fact that our propagator is not domain consistent.



(a) Average learned nogood length (b) Average LBD

Figure 8: Average quality of the learned nogoods, with varying alphabet sizes ($|\Sigma|$), number of states of the cDFA ($|Q|$) and accepting counts (K).

As shown by Figure 8.a, our propagator has similar, but often slightly worse (longer) nogood length compared to Gvosdiovas et al.'s propagator. Both these propagators have shorter nogood lengths compared to decomposition. Note that decomposition has a similar—or even slightly shorter—length for $|\Sigma| = 3, |Q| = 3, K \in \{\{1, 2\}, \{1, 4\}\}$, $|\Sigma| = 3, |Q| = 5, K = \{1, 2\}$ and $|\Sigma| = 3, |Q| = 3, K \in \{\{1, 2\}, \{1, 4\}, \{1, 8\}\}$.

As shown by Figure 8.b, our propagator has similar, but often worse (higher) LBDs compared to Gvosdiovas et al.'s propagator. Both these propagators have significantly worse LBDs than decomposition.

The quality of the nogoods of our propagator is slightly worse compared toGvosdiovas et al.'s propagator, while significantly worse than those of decomposition. Even though the average length of the learned nogoods of decomposition is worse, LBD is a better indicator of nogood quality.

We suspect that our propagator solves faster, even though nogood quality is worse, because of the inherently smaller datastructure used. This is supported by the fact that our propagator solves comparatively faster as the maximum accepting count—and thus the size of the equivalent DFA—grows.

5.3 Limitations

There are five limitations of our research we want to highlight. First of all, the explanations of our propagator are simple. This means that there is potential for improvement. However, we have shown that even with these simple explanations, our propagator is able to solve satisfaction problems with certain configurations faster than other propagators.

Secondly, only 32 configurations are compared. Though our research examines configurations with a similar or slightly larger amount of states as the cDFAs from the global constraint catalog[4], it is unknown if findings generalize to other configurations.

Thirdly, Our benchmark examines the performance of the constraints in general; not for specific problems. It could be the case that for specific problems, a given propagator performs significantly better or worse. We chose to use this benchmark specifically to test the general performance.

Fourthly, Some configurations of problem instances not running into conflicts suggests that the problems are under-constrained. More constrained problems could show different relationships between the configurations and performance.

Finally, as optimization problems were not benchmarked, how the performance of the propagators compares in that context is unknown.

6 Responsible Research

6.1 Reproducibility

All our experiments and code have been version controlled since the beginning using git, and are available in our experiment suite[3]. This was done not only to allow others to check our work, but also in an effort to make any material that might be of interest for future research readily available. Our experiment suite includes the following:

- A fork of Pumpkin, extended with the constraints used;
- Scripts to generate and run the benchmarks;
- Results of our benchmark, including boxplots comparing all individual configurations;
- Discarded benchmarks that were run;
- A script to process and visualise the results.

In addition to this, all random generated instances were seeded and labeled with the used seed.

By using devenv[4] we have tried to prevent additional setup and dependency conflicts that might otherwise have occurred when running the experiments on other machines.

6.2 Data generation

The data we used for benchmarking was all synthetically generated by scripts. As such, its usage is unrestricted by ethical considerations. One important detail to mention however, is that FAdo[2]—a python library for manipulating finite automata—falls under the GNU General Public License. As such, the python script using it to generate random DFAs and cDFAs that are equivalent also falls under this license. The rest of our code falls under the MIT license.

6.3 Reporting failure

Research tends to less often report on failures, which can result in knowledge gaps and failing approaches being repeated. We have thus decided to include details of a failed approach to benchmark using nonograms in Appendix C.

6.4 AI usage

LLM's—and other generative AI—have become prevalent not only in everyday life, but also in academia. However, we have opted to not use any generative AI during the course of this research, as it makes it easy to outsource thinking, which can lead to research that is less critical and stagnation of personal development. This paper and our contributions have been human generated.

7 Conclusion

In this paper we have presented the first ever extension with explanations for lazy clause generation (LCG) of Beldiceanu et al.'s[2] propagator for the exact regular counting constraint based on a counter-DFA (cDFA). We have also presented a generator for cDFAs and DFAs that are equivalent.

We have shown that for cDFAs between three and nine states and alphabets of size three and six, generally, our propagator solves satisfaction problems faster than both an equivalent DFA based regular constraint decomposed into global constraints and an equivalent DFA based regular constraint propagated with Gvosdiovas et al.'s[5] extension with explanations for LCG of Pesant's[6] propagator. We have also shown that our propagator propagates and runs into conflicts slightly more than Gvosdiovas et al.'s propagator, but significantly less than decomposition.

The quality of our explanations is significantly worse than those of decomposition, but only slightly worse than those of Gvosdiovas et al.. As our explanations are trivially based on the propagator,

[3]https://github.com/ICorvidusI/experiment-suite-RP-2026/tree/submission-21-06-2026

[4]https://devenv.sh/, a tool for declarative development environments that isolates and version locks packages, based on the nix package manager[5]

[5]https://nixos.org/

there is potential room for improvement. We have provided a basis for future research to improve these explanations.

7.1 Future Work

There are multiple opportunities for future research that we would like to see explored:

- As LCG is known to learn more from simpler propagation[7], how does decomposing exact regular counting into at-most and at-least regular counting[2]—extended with explanations for LCG—compare to our propagator?[6]
- Similarly, how can cDFA based regular counting be decomposed into global constraints? How does this affect solve time and nogood quality?
- how does our propagator compare to the similar, but more widely used cost_regular constraint —decomposed or propagated with an extension to Demassey et al.'s[10] propagator—with explanations for LCG?
- As our explanations for LCG are trivially based on the propagation, can they be shortened to provide better conflict-learning? One avenue worth exploring would be Martin and Pearson's[9] algorithm to compute when bounds consistency implies domain consistency for cDFA-based regular counting.

[6] A non-verified propagator for the at-most regular counting constraint can be found—commented out—in the methods of our regular_cdfa propagator.[3]

Appendix

A Behaviour of QCF and QCB

Example A.1: We will demonstrate the behaviour of $\mathrm{QCF}(i)$. Consider the cDFA in Figure 3. Given an arbitrary sequence X of $n = 4$ variables with Σ as domain, we have:

$$\begin{aligned}
\mathrm{QCF}(0) &= \{\langle q_0, \{0\}\rangle\} \\
\mathrm{QCF}(1) &= \{\langle q_0, \{0\}\rangle, \langle q_1, \{0\}\rangle, \langle q_2, \{0\}\rangle\} \\
\mathrm{QCF}(2) &= \{\langle q_0, \{0\}\rangle, \langle q_1, \{0\}\rangle, \langle q_2, \{0,1\}\rangle, \langle q_3, \{1\}\rangle\} \\
\mathrm{QCF}(3) &= \{\langle q_0, \{0,3\}\rangle, \langle q_1, \{0,1\}\rangle, \langle q_2, \{0,1\}\rangle, \langle q_3, \{1\}\rangle\} \\
\mathrm{QCF}(4) &= \{\langle q_0, \{0,3\}\rangle, \langle q_1, \{0,1,3\}\rangle, \langle q_2, \{0,1,2,3\}\rangle, \langle q_3, \{1,2\}\rangle\}
\end{aligned}$$

□

Example A.2: We will demonstrate the behaviour of $\mathrm{QCB}(i)$. Consider the cDFA in Figure 3. Given an arbitrary sequence X of $n = 4$ variables with Σ as domain, we have:

$$\begin{aligned}
\mathrm{QCB}(5) &= \{\langle q_0, \{0\}\rangle, \langle q_1, \{0\}\rangle, \langle q_2, \{0\}\rangle, \langle q_3, \{0\}\rangle\} \\
\mathrm{QCB}(4) &= \{\langle q_0, \{0\}\rangle, \langle q_1, \{0,1\}\rangle, \langle q_2, \{0\}\rangle, \langle q_3, \{0,2\}\rangle\} \\
\mathrm{QCB}(3) &= \{\langle q_0, \{0,1\}\rangle, \langle q_1, \{0,1,3\}\rangle, \langle q_2, \{0,1\}\rangle, \langle q_3, \{0,2\}\rangle\} \\
\mathrm{QCB}(2) &= \{\langle q_0, \{0,1,3\}\rangle, \langle q_1, \{0,1,2,3\}\rangle, \langle q_2, \{0,1,3\}\rangle, \langle q_3, \{0,1,2,3\}\rangle\} \\
\mathrm{QCB}(1) &= \{\langle q_0, \{0,1,2,3\}\rangle, \langle q_1, \{0,1,2,3,4\}\rangle, \langle q_2, \{0,1,2,3\}\rangle, \langle q_3, \{0,1,2,3,5\}\rangle\}
\end{aligned}$$

□

B Polynomial time algorithms

Using the following two functions to respectively keep only the tuples with the minimum and maximum cost for a given state:

$$\begin{aligned}
\mathrm{trimMin}(S) &= \{\langle q, c\rangle \in S \mid \nexists \langle q, c'\rangle \in S : c' < c\} \\
\mathrm{trimMax}(S) &= \{\langle q, c\rangle \in S \mid \nexists \langle q, c'\rangle \in S : c' > c\}
\end{aligned}$$

$\underline{\mathrm{QCF}}(i)$, $\overline{\mathrm{QCF}}(i)$, $\underline{\mathrm{QCB}}(i)$ and $\overline{\mathrm{QCB}}(i)$ can be implemented inductively:

$$\underline{\mathrm{QCF}}(i) = \begin{cases} \{\langle q_0, 0\rangle\} & \text{if } i = 0 \\ \mathrm{trimMin}\Big(\Big\{\langle \delta_Q(q,\ell), c + \delta_{\mathbb{N}}(q,\ell)\rangle \mid \langle q, c\rangle \in \underline{\mathrm{QCF}}(i-1), \ell \in dom(x_i)\Big\}\Big) & \text{if } i \in [1,n] \end{cases}$$

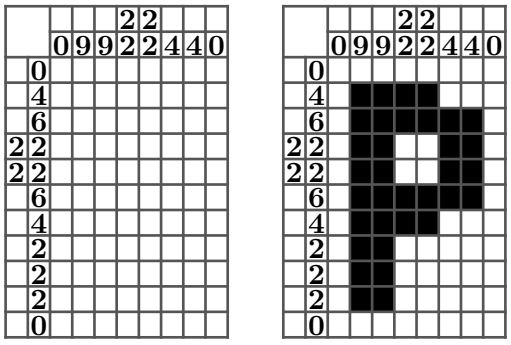
$$\overline{\mathrm{QCF}}(i) = \begin{cases} \{\langle q_0, 0\rangle\} & \text{if } i = 0 \\ \mathrm{trimMax}\Big(\Big\{\langle \delta_Q(q,\ell), c + \delta_{\mathbb{N}}(q,\ell)\rangle \mid \langle q, c\rangle \in \overline{\mathrm{QCF}}(i-1), \ell \in dom(x_i)\Big\}\Big) & \text{if } i \in [1,n] \end{cases}$$

$$\underline{\mathrm{QCB}}(i) = \begin{cases} \Big\{\langle q, 0\rangle \mid \exists c \in \mathbb{N} : \langle q, c\rangle \in \underline{\mathrm{QCB}}(i)\Big\} & \text{if } i = n+1 \\ \mathrm{trimMin}\Big(\Big\{\langle q, c' + inc\rangle \mid \langle q', c'\rangle \in \underline{\mathrm{QCB}}(i+1), \ell \in dom(x_i), \delta(q,\ell) = \langle q', inc\rangle\Big\}\Big) & \text{if } i \in [1,n] \end{cases}$$

$$\overline{\mathrm{QCB}}(i) = \begin{cases} \Big\{\langle q, 0\rangle \mid \exists c \in \mathbb{N} : \langle q, c\rangle \in \overline{\mathrm{QCB}}(i)\Big\} & \text{if } i = n+1 \\ \mathrm{trimMax}\Big(\Big\{\langle q, c' + inc\rangle \mid \langle q', c'\rangle \in \overline{\mathrm{QCB}}(i+1), \ell \in dom(x_i), \delta(q,\ell) = \langle q', inc\rangle\Big\}\Big) & \text{if } i \in [1,n] \end{cases}$$

C Nonograms

Nonograms are well-known puzzles. See Figure C.1 for an example. They consist of an empty grid, where each row and column has a "hint"—a list of numbers—that corresponds to what squares in the row or column can be filled. Each number n in the hint corresponds to a subsequence of n filled in squares; the subsequences occur in order; subsequences are separated by at least one empty square.



(a) Empty nonogram (b) Solved nonogram

Figure C.1: An example of a nonogram

As this problem relies on patterns that count, it seems like a good fit to model with cDFAs. And yes, it is possible to model the hints for the rows and columns with regular counting; a given hint $[k, m, n]$ can be modeled by the regular expression "$0^*1\{k\}0^+1\{m\}0^+1\{n\}0^*$". There are two problems when modeling with cDFAs however.

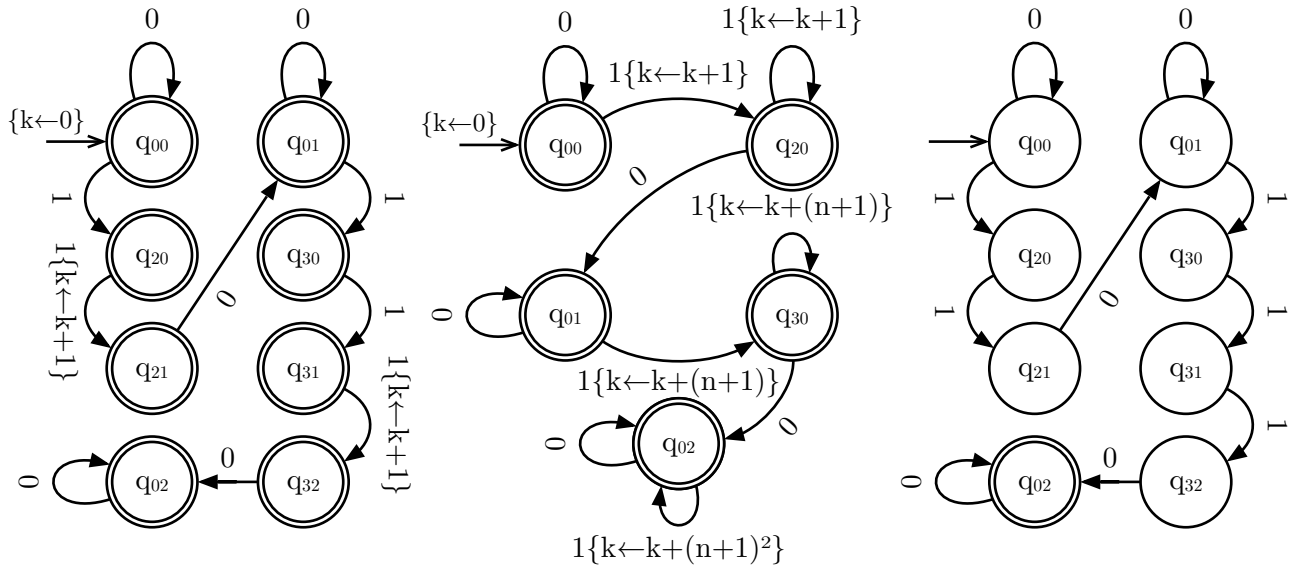
First of all, the count of a subsequence is always a fixed value, whereas regular_cDFA is able to propagate on the count. This means modeling nonograms does not exploit an advantage of cDFAs.

Second of all, there are multiple patterns that are counted. This poses the biggest problem. cDFAs only have one counter. As such, nonograms can not be trivially modeled using cDFAs in a way that results in a smaller encoding than when modeled with a DFA.

There is one approach however. If a cDFA increments by at most one, the maximum count after consuming a sequence of n values is n. To implement a second counter, whose count can be distinguished from the first, edges incrementing by $n + 1$ can thus be used. An accepting count can then be encoded as $1 \cdot count_1 + (n + 1) \cdot count_2$. This can be generalized for k counters as:

$$\sum_{i=1}^{k} (n+1)^{i-1} \cdot count_i$$

Note that this essentially represents each count as a digit in a base $n + 1$ number. See Figure C.2 for examples of each approach.



(a) Trivial cDFA, K={2} Transitions to the trapstate increment with {k←k+3}.

(b) Encoded cDFA, K={2 · 1 + 3 · (n+1)}

(c) DFA

Figure C.2: Three ways to represent the hint [2, 3]. For brevity the trap-state and its ingoing transitions are left out of (a) and (c).

This introduces a new problem however: The accepting count grows exponentially proportional to the amount of encoded counts. For small nonograms, this is no issue, as they have small sequences and consequently a small amount of numbers per hint. As nonograms grow in size however, so do their sequences and consequently the amount of numbers per hint. For a 30x30 nonogram, that thus has sequences of length 30, it is not uncommon to see hints with seven or more numbers. As $31^7 > 2^{32}$, the accepting counts used to model larger nonograms are unable to be represented by either signed or unsigned 32-bit integers.

These two factors—having a single count per sequence and exponential blowup of the accepted count—make it impractical to pursue this problem as a benchmark.

References

[1] C. Chapman and K. T. Stolee, "Exploring regular expression usage and context in Python," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, in ISSTA 2016. New York, NY, USA: Association for Computing Machinery, July 2016, pp. 282–293. doi: 10.1145/2931037.2931073.

[2] N. Beldiceanu, P. Flener, J. Pearson, and P. V. Hentenryck, "Propagating Regular Counting Constraints." Accessed: Apr. 29, 2026. [Online]. Available: http://arxiv.org/abs/1309.7145

[3] N. Beldiceanu, M. Carlsson, and T. Petit, "Deriving Filtering Algorithms from Constraint Checkers," in *Principles and Practice of Constraint Programming – CP 2004*, M. Wallace, Ed., Berlin, Heidelberg: Springer, 2004, pp. 107–122. doi: 10.1007/978-3-540-30201-8_11.

[4] N. Beldiceanu, M. Carlsson, S. Demassey, and T. Petit, "Global Constraint Catalogue: Past, Present and Future," *Constraints*, vol. 12, no. 1, pp. 21–62, Mar. 2007, doi: 10.1007/s10601-006-9010-8.

[5] J. Gvozdiovas, D. Emir, and F. Maarten, "NFA propagator for regular constraint with explanations," Jan. 2025, Accessed: Apr. 21, 2026. [Online]. Available: https://repository.tudelft.nl/record/uuid:16153696-2b90-46f7-8b48-03455d427ba1

[6] G. Pesant, "A Regular Language Membership Constraint for Finite Sequences of Variables," in *Principles and Practice of Constraint Programming – CP 2004*, M. Wallace, Ed., Berlin, Heidelberg: Springer, 2004, pp. 482–495. doi: 10.1007/978-3-540-30201-8_36.

[7] T. Feydy and P. J. Stuckey, "Lazy Clause Generation Reengineered," in *Principles and Practice of Constraint Programming - CP 2009*, I. P. Gent, Ed., Berlin, Heidelberg: Springer, 2009, pp. 352–366. doi: 10.1007/978-3-642-04244-7_29.

[8] M. Sipser, "Formal definition of a finite automaton," *Introduction to the Theory of Computation*. Cengage Learning, pp. 35–40, 2013.

[9] B. Martin and J. Pearson, "When bounds consistency implies domain consistency for regular counting constraints," *Constraints*, vol. 27, no. 3, pp. 161–167, July 2022, doi: 10.1007/s10601-022-09333-0.

[10] S. Demassey, G. Pesant, and L.-M. Rousseau, "A cost-regular based hybrid column generation approach," in *Constraints*, 2006, pp. 315–333. doi: 10.1007/s10601-006-9003-7.

[11] J. Hopcroft, "AN n log n ALGORITHM FOR MINIMIZING STATES IN A FINITE AUTOMATON," *Theory of Machines and Computations*. Academic Press, pp. 189–196, Jan. 1971. doi: 10.1016/B978-0-12-417750-5.50022-1.

[12] M. Flippo, K. Sidorov, I. Marijnissen, J. Smits, and E. Demirović, "A Multi-Stage Proof Logging Framework to Certify the Correctness of CP Solvers," in *30th International Conference on Principles and Practice of Constraint Programming (CP 2024)*, P. Shaw, Ed., in Leibniz International Proceedings in Informatics (LIPIcs), vol. 307. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024, pp. 11:1–11:20. doi: 10.4230/LIPIcs.CP.2024.11.

[13] N. Nethercote, P. J. Stuckey, R. Becket, S. Brand, G. J. Duck, and G. Tack, "MiniZinc: Towards a Standard CP Modelling Language," in *Principles and Practice of Constraint*

Programming – CP 2007, C. Bessière, Ed., Berlin, Heidelberg: Springer, 2007, pp. 529–543. doi: 10.1007/978-3-540-74970-7_38.