Reformulating Global Grammar Constraints[*]

George Katsirelos[1], Nina Narodytska[2], and Toby Walsh[2]

[1] NICTA, Sydney, Australia
george.katsirelos@nicta.com.au
[2] NICTA and University of NSW, Sydney, Australia
ninan@cse.unsw.edu.au, toby.walsh@nicta.com.au

Abstract. An attractive mechanism to specify global constraints in rostering and other domains is via formal languages. For instance, the REGULAR and GRAMMAR constraints specify constraints in terms of the languages accepted by an automaton and a context-free grammar respectively. Taking advantage of the fixed length of the constraint, we give an algorithm to transform a context-free grammar into an automaton. We then study the use of minimization techniques to reduce the size of such automata and speed up propagation. We show that minimizing such automata after they have been unfolded and domains initially reduced can give automata that are more compact than minimizing before unfolding and reducing. Experimental results show that such transformations can improve the size of rostering problems that we can "model and run".

1 Introduction

Constraint programming provides a wide range of tools for modelling and efficiently solving real world problems. However, modelling remains a challenge even for experts. Some recent attempts to simplify the modelling process have focused on specifying constraints using formal language theory. For example the REGULAR [1] and GRAMMAR constraints [2,3] permit constraints to be expressed in terms of automata and grammars. In this paper, we make two contributions. First, we investigate the relationship between REGULAR and GRAMMAR. In particular, we show that it is often beneficial to reformulate a GRAMMAR constraint as a REGULAR constraint. Second, we explore the effect of minimizing the automaton specifying a REGULAR constraint. We prove that by minimizing this automaton *after* unfolding and initial constraint propagation, we can get an exponentially smaller and thus more efficient representation. We show that these transformations can improve runtimes by over an order of magnitude.

2 Background

A constraint satisfaction problem consists of a set of variables, each with a domain of values, and a set of constraints specifying allowed combinations of values for given subsets of variables. A solution is an assignment to the variables satisfying the constraints. A constraint is *domain consistent* iff for each variable, every value in its domain can be extended to an assignment that satisfies the constraint. We will consider

constraints specified by automata and grammars. An automaton $A = \langle \Sigma, Q, q_0, F, \delta \rangle$ consists of an alphabet Σ, a set of states Q, an initial state q_0, a set of accepting states F, and a transition relation δ defining the possible next states given a starting state and symbol. The automaton is deterministic (DFA) is there is only one possible next state, non-deterministic (NFA) otherwise. A string s is *recognized* by $\mathcal{A}$ iff starting from the state q_0 we can reach one of the accepting states using the transition relation δ. Both DFAs and NFAs recognize precisely regular languages. The constraint REGULAR$(\mathcal{A}, [X_1, \ldots, X_n])$ is satisfied iff X_1 to X_n is a string accepted by $\mathcal{A}$ [1]. Pesant has given a domain consistency propagator for REGULAR based on unfolding the DFA to give a n-layer automaton which only accepts strings of length n [1].

Given an automaton $\mathcal{A}$, we write $unfold_n(\mathcal{A})$ for the unfolded and layered form of $\mathcal{A}$ that just accepts words of length n which are in the regular language, $\min(\mathcal{A})$ for the canonical form of $\mathcal{A}$ with minimal number of states, $simplify(\mathcal{A})$ for the simplified form of $\mathcal{A}$ constructed by deleting transitions and states that are no longer reachable after domains have been reduced. We write $f_{\mathcal{A}}(n) \ll g_{\mathcal{A}}(n)$ iff $f_{\mathcal{A}}(n) \leq g_{\mathcal{A}}(n)$ for all n, and there exist $\mathcal{A}$ such that $\log \frac{g_{\mathcal{A}}(n)}{f_{\mathcal{A}}(n)} = \Omega(n)$. That is, $g_{\mathcal{A}}(n)$ is never smaller than $f_{\mathcal{A}}(n)$ and there are cases where it is exponentially larger.

A context-free grammar is a tuple $G = \langle T, H, P, S \rangle$, where T is a set of *terminal* symbols called the *alphabet* of G, H is a set of *non-terminal* symbols, P is a set of productions and S is a unique starting symbol. A production is a rule $A \to \alpha$ where A is a non-terminal and α is a sequence of terminals and non-terminals. A string in Σ^* is *generated* by G if we start with the sequence $\alpha = \langle S \rangle$ and non deterministically generate α' by replacing any non-terminal A in α by the right hand side of any production $A \to \alpha$ until α' contains only terminals. A context free language $\mathcal{L}(G)$ is the language of strings generated by the context free grammar G. A context free grammar is in Chomsky normal form if all productions are of the form $A \to BC$ where B and C are non terminals or $A \to a$ where a is a terminal. Any context free grammar can be converted to one that is in Chomsky normal form with at most a linear increase in its size. A grammar G_a is *acyclic* iff there exists a partial order $\prec$ of the non-terminals, such that for every production $A_1 \to A_2 A_3$, $A_1 \prec A_2$ and $A_1 \prec A_3$. The constraint GRAMMAR$([X_1, \ldots, X_n], G)$ is satisfied iff X_1 to X_n is a string accepted by G [2,3].

Example 1. As the running example we use the GRAMMAR$([X_1, X_2, X_3], G)$ constraint with domains $D(X_1) = \{a\}$, $D(X_2) = \{a, b\}$, $D(X_3) = \{b\}$ and the grammar G in Chomsky normal form [3] $\{S \to AB, A \to AA \mid a, B \to BB \mid b\}$.

Since we only accept strings of a fixed length, we can convert any context free grammar to a regular grammar. However, this may increase the size of the grammar exponentially. Similarly, any NFA can be converted to a DFA, but this may increase the size of the automaton exponentially.

3   GRAMMAR **Constraint**

We briefly describe the domain consistency propagator for the GRAMMAR constraint proposed in [2,3]. This propagator is based on the CYK parser for context-free grammars. It constructs a dynamic programing table V where an element A of $V[i, j]$ is a

non-terminal that generates a substring from the domains of variables $X_i, \ldots, X_{i+j-1}$ that can be extended to a solution of the constraint using the domains of the other variables. The table V produced by the propagator for Example 1 is given in Figure 1.
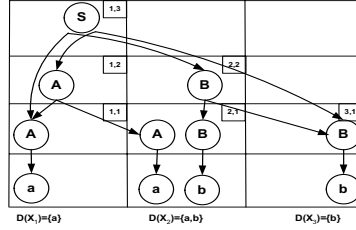

Fig. 1. Dynamic programming table produced by the propagator of the GRAMMAR constraint. Pointers correspond to possible derivations.

An alternative view of the dynamic programming table produced by this propagator is as an AND/OR graph [4]. This is a layered DAG, with layers alternating between AND-NODES or OR-NODES. Each OR-NODE in the AND/OR graph corresponds to an entry $A \in V[i, j]$. An OR-NODE has a child AND-NODE for each production $A \rightarrow BC$ so that $A \in V[i, j]$, $B \in V[i, k]$ and $C \in V[i + k, j - k]$. The children of this AND-NODE are the OR-NODES that correspond to the entries $B \in V[i, k]$ and $C \in V[i + k, j - k]$. Note that the AND/OR graph constructed in this manner is equivalent to the table V [4], so we use them interchangeably in this paper.

Every derivation of a string $s \in \mathcal{L}(G)$ can be represented as a tree that is a subgraph of the AND/OR graph and therefore can be represented as a trace in V. Since every possible derivation can be represented this way, both the table V and the corresponding AND/OR graph are a compilation of all solutions of the GRAMMAR constraint.

4   Reformulation into an Automaton

The time complexity of propagating a GRAMMAR constraint is $O(n^3|G|)$, as opposed to $O(n|\delta|)$ for a REGULAR constraint. Therefore, reformulating a GRAMMAR constraint as a REGULAR constraint may improve propagation speed if it does not require a
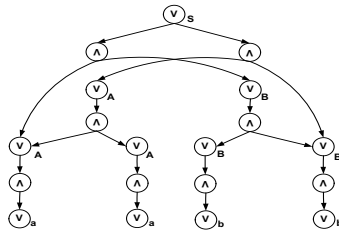
Fig. 2. AND/OR graph.

large transition relation. In addition, we can perform optimizations such as minimizing the automaton. In this section, we argue that reformulation is practical in many cases (sections 4.1-4.3), and there is a polynomial test to determine the size of the resulting NFA (section 4.4). In the worst case, the resulting NFA is exponentially larger then the original GRAMMAR constraint as the following example shows. Therefore, performing the transformation itself is not a suitable test of the feasibility of the approach.

Example 2. Consider GRAMMAR$([X_1, \ldots, X_n], G)$ where G generates $L = \{ww^R | w \in \{0,1\}^{n/2}\}$. Solutions of GRAMMAR can be compiled into the dynamic programming table of size $O(n^3)$, while an equivalent NFA that accepts the same language has exponential size. Note that an exponential separation does not immediately follow from that between regular and context-free grammars, because solutions of the GRAMMAR constraint are the strict subset of $\mathcal{L}(G)$ which have length n.

In the rest of this section we describe the reformulation in three steps. First, we convert into an acyclic grammar (section 4.1), then into a pushdown automaton (section 4.2), and finally we encode this as a NFA (section 4.3). The first two steps are well known in formal language theory but we briefly describe them for clarity.

4.1 Transformation into an Acyclic Grammar

We first construct an acyclic grammar, G_a such that the language $\mathcal{L}(G_a)$ coincides with solutions of the GRAMMAR constraint. Given the table V produced by the GRAMMAR propagator (section 3), we construct an acyclic grammar in the following way. For each possible derivation of a nonterminal A, $A \rightarrow BC$, such that $A \in V[i,j]$, $B \in V[i,k]$ and $C \in V[i+k, j-k]$ we introduce a production $A_{i,j} \rightarrow B_{i,k}C_{i+k,j-k}$ in G_a (lines 11- 17 of algorithm 1). The start symbol of G_a is $S_{1,n}$. By construction, the obtained grammar G_a is acyclic. Every production in G_a is of the form $A_{i,j} \rightarrow B_{i,k}C_{i+k,j-k}$ and nonterminals $B_{i,k}$, $C_{i+k,j-k}$ occur in rows below jth row in V. Example 3 shows the grammar G_a obtained by Algorithm 1 on our running example.

Example 3. The acyclic grammar G_a constructed from our running example.

$$S_{1,3} \rightarrow A_{1,2}B_{3,1} \mid A_{1,1}B_{2,2} \qquad A_{1,2} \rightarrow A_{1,1}A_{2,1} \qquad B_{2,2} \rightarrow B_{2,1}B_{3,1}$$
$$A_{i,1} \rightarrow a_i \qquad\qquad B_{i,1} \rightarrow b_i \qquad\qquad \forall i \in \{1,2,3\}$$

To prove equivalence, we recall that traces of the table V represent all possible derivations of GRAMMAR solutions. Therefore, every derivation of a solution can be simulated by productions from G_A. For instance, consider the solution (a, a, b) of GRAMMAR from Example 1. A possible derivation of this string is $S|_{S \in V[1,3]} \rightarrow AB|_{A \in V[1,2], B \in V[3,1]} \rightarrow AAB|_{A \in V[1,1], A \in V[2,1], B \in V[3,1]} \rightarrow aAB|_{\ldots} \rightarrow aaB|_{\ldots} \rightarrow aab|_{\ldots}$. We can simulate this derivation using productions in G_a: $S_{1,3} \rightarrow A_{1,2}B_{3,1} \rightarrow A_{1,1}A_{2,1}B_{3,1} \rightarrow a_1A_{2,1}B_{3,1} \rightarrow a_1a_2B_{3,1} \rightarrow a_1a_2b_3$.

Observe that, the acyclic grammar G_a is essentially a labelling of the AND/OR graph, with non-terminals corresponding to OR-NODES and productions corresponding to AND-NODES. Thus, we use the notation G_a to refer to both the AND/OR graph and the corresponding acyclic grammar.

Algorithm 1. Transformation to an Acyclic Grammar

1: **procedure** CONSTRUCTACYCLICGRAMMAR($in : X, G, V; out : G_a$)
2:    $T = \emptyset$ ▷ T is the set of terminals in G_a
3:    $H = \emptyset$ ▷ H is the set of nonterminals in G_a
4:    $P = \emptyset$ ▷ P is the set of productions in G_a
5:    **for** $i = 1$ **to** n **do**
6:        $V[i, 1] = \{A | A \to a \in G, a \in D(X_i)\}$
7:        **for** $A \in V[i, 1]$ s.t $A \to a \in G, a \in D(X_i)$ **do**
8:            $T = T \cup \{a_i\}$
9:            $H = H \cup \{A_{i,1}\}$
10:            $P = P \cup \{A_{i,1} \to a_i\}$
11:    **for** $j = 2$ **to** n **do**
12:        **for** $i = 1$ **to** $n - j + 1$ **do**
13:            **for each** $A \in V[i, j]$ **do**
14:                **for** $k = 1$ **to** $j - 1$ **do**
15:                    **for each** $A \to BC \in G$ s.t. $B \in V[i, k], C \in V[i + k, j - k]$ **do**
16:                        $H = H \cup \{A_{i,j}, B_{i,k}, C_{i+k,j-k}\}$
17:                        $P = P \cup \{A_{i,j} \to B_{i,k}C_{i+k,j-k}\}$

4.2 Transformation into a Pushdown Automaton

Given an acyclic grammar $G_a = (T, H, P, S_{1,n})$ from the previous section, we now construct a pushdown automaton $P_a(\langle S_{1,n} \rangle, T, T \cup H, \delta, Q_P, F_P)$, where $\langle S_{1,n} \rangle$ is the initial stack of P_a, T is the alphabet, $T \cup H$ is the set of stack symbols, δ is the transition function, $Q_P = F_P = \{q_P\}$ is the single initial and accepting state. We use an algorithm that encodes a context free grammar into a pushdown automaton (PDA) that computes the leftmost derivation of a string[5]. The stack maintains the sequence of symbols that are expanded in this derivation. At every step, the PDA non-deterministically uses a production to expand the top symbol of the stack if it is a non-terminal, or consumes a symbol of the input string if it matches the terminal at the top of the stack.

We now describe this reformulation in detail. There exists a single state q_P which is both the starting and an accepting state. For each non-terminal $A_{i,j}$ in G_a we introduce the set of transitions $\delta(q_P, \varepsilon, A_{i,j}) = \{(q_P, \beta) | \forall A_{i,j} \to \beta \in G_a\}$. For each terminal $a_i \in G_a$, we introduce a transition $\delta(q_P, a_i, a_i) = \{(q_P, \varepsilon)\}$. The automaton P_a accepts on the empty stack. This constructs a pushdown automaton accepting $\mathcal{L}(G_a)$.

Example 4. The pushdown automaton P_a constructed for the running example.

$$\delta(q_P, \varepsilon, S_{1,3}) = \delta(q_P, A_{1,2}B_{3,1}) \qquad \delta(q_P, \varepsilon, S_{1,3}) = \delta(q_P, A_{1,1}B_{2,2})$$
$$\delta(q_P, \varepsilon, A_{1,2}) = \delta(q_P, A_{1,1}A_{2,1}) \qquad \delta(q_P, \varepsilon, B_{2,2}) = \delta(q_P, B_{2,1}B_{3,1})$$
$$\delta(q_P, \varepsilon, A_{i,1}) = \delta(q_P, a_i) \qquad \delta(q_P, \varepsilon, B_{i,1}) = \delta(q_P, b_i) \forall i \in \{1, 2, 3\}$$
$$\delta(q_P, a_i, a_i) = \delta(q_P, \varepsilon) \qquad \delta(q_P, b_i, b_i) = \delta(q_P, \varepsilon) \forall i \in \{1, 2, 3\}$$

4.3 Transformation into a NFA

Finally, we construct an NFA$(\Sigma, Q, Q_0, F_0, \sigma)$, denoted N_a, using the PDA from the last section. States of this NFA encode all possible configurations of the stack of the PDA that can appear in parsing a string from G_a. To reflect that a state of the NFA represents a stack, we write states as sequences of symbols $\langle \alpha \rangle$, where α is a possibly empty sequence of symbols and $\alpha[0]$ is the top of the stack. For example, the initial

---

Algorithm 2. Transformation to NFA

1: **procedure** PDA TO NFA($in : P_a, out : N_a$)
2:     $Q_u = \{\langle S_{1,n}\rangle\}$ ▷ Q_u is the set of unprocessed states
3:     $Q = \emptyset$ ▷ Q is the set of states in N_a
4:     $\sigma = \emptyset$ ▷ σ is the set of transitions in N_a
5:     $Q_0 = \{\langle S_{1,n}\rangle\}$ ▷ Q_0 is the initial state in N_a
6:     $F_0 = \{\langle\rangle\}$ ▷ F_0 is the set of final states in N_a
7:     **while** Q_u is not empty **do**
8:         **if** $q \equiv \langle A_{i,j}, \alpha\rangle$ **then**
9:             **for** each transition $\delta(q_P, \varepsilon, A_{i,j}) = (q_P, \beta) \in \delta$ **do**
10:                 $\sigma = \sigma \cup \{\sigma(\langle A_{i,j}, \alpha\rangle, \varepsilon) = \langle\beta, \alpha\rangle\}$
11:                 **if** $\langle\beta, \alpha\rangle \notin Q$ **then**
12:                     $Q_u = Q_u \cup \{\langle\beta, \alpha\rangle\}$
13:             $Q = Q \cup \{\langle A_{i,j}, \alpha\rangle\}$
14:         **else if** $q \equiv \langle a_i, \alpha\rangle$ **then**
15:             **for** each transition $\delta(q_P, a_i, a_i) = (q_P, \varepsilon) \in \delta$ **do**
16:                 $\sigma = \sigma \cup \{\sigma(\langle a_i, \alpha\rangle, a_i) = \langle\alpha\rangle\}$
17:                 **if** $\langle\alpha\rangle \notin Q$ **then**
18:                     $Q_u = Q_u \cup \{\langle\alpha\rangle\}$
19:             $Q = Q \cup \{\langle a_i, \alpha\rangle\}$
20:         $Q_u = Q_u \setminus \{q\}$
21:     $N_a(\Sigma, Q, Q_0, F_0, \sigma) = \varepsilon - Closure(N_a(\Sigma, Q, Q_0, F_0\sigma))$.

---

state is $\langle S_{1,n}\rangle$ corresponding to the initial stack $\langle S_{1,n}\rangle$ of P_a. Algorithm 2 unfolds the PDA in a similar way to unfolding the DFA. Note that the NFA accepts only strings of length n and has the initial state $Q_0 = \langle S_{1,n}\rangle$ and the single final state $F_0 = \langle\rangle$.

We start from the initial stack $\langle S_{1,n}\rangle$ and find all distinct stack configurations that are reachable from this stack using transitions from P_a. For each reachable stack configuration we create a state in the NFA and add the corresponding transitions. If the new stack configurations are the result of expansion of a production in the original grammar, these transitions are $\varepsilon-$transitions, otherwise they consume a symbol from the input string. Note that if a non-terminal appears on top of the stack and gets replaced, then it cannot appear in any future stack configuration due to the acyclicity of G_a. Therefore $|\alpha|$ is bounded by $O(n)$ and Algorithm 2 terminates. The size of N_a is $O(|G_a|^n)$ in the worst case. The automaton N_a that we obtain before line 21 is an acyclic NFA with ε transitions. It accepts the same language as the PDA P_a since every path between the starting and the final state of N_A is a trace of the stack configurations of P_a. Figure 3(a) shows the automaton N_a with ε-transitions constructed from the running example. After applying the ε-closure operation, we obtain a layered NFA that does not have ε transitions (line 21) (Figure 3(b)).

4.4   Computing the Size of the NFA

As the NFA may be exponential in size, we provide a polynomial method of computing its size in advance. We can use this to decide if it is practical to transform it in this way. Observe first that the transformation of a PDA to an NFA maintains a queue of states that correspond to stack configurations. Each state corresponds to an OR-NODE in the AND/OR graph and each state of an OR-NODE v is generated from the states of the parent OR-NODES of v. This suggests a relationship between paths in the AND/OR graph of the CYK algorithm and states in N_a. We use this relationship to compute
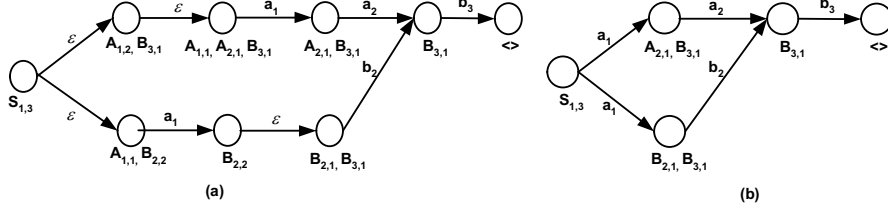
Fig. 3. N_a produced by Algorithm 2

a loose upper bound for the number of states in N_a in time linear in the size of the AND/OR graph by counting the number of paths in that graph. Alternatively, we compute the exact number of states in N_a in time quadratic in the size of the AND/OR graph.

Theorem 1. *There exists a surjection between paths in G_a from the root to* OR-NODES *and stack configurations in the PDA P_a.*

Proof. Consider a path p from the root of the AND/OR graph to an OR-NODE labelled with $A_{i,j}$. We construct a stack configuration $\Gamma(p)$ that corresponds to p. We start with the empty stack $\Gamma = \langle\rangle$. We traverse the path from the root to $A_{i,j}$. For every AND-NODE $v_1 \in p$, with left child v_l and right child v_r, if the successor of v_1 in p is v_l, then we push v_r on Γ, otherwise do nothing. When we reach $A_{i,j}$, we push it on Γ. The final configuration Γ is unique for p and corresponds to the stack of the PDA after having parsed the substring $1 \ldots i-1$ and having non-deterministically chosen to parse the substring $i \ldots i+j-1$ using a production with $A_{i,j}$ on the LHS.

We now show that all stack configurations can be generated by the procedure above. Every stack configuration corresponds to at least one partial left most derivation of a string. We say a stack configuration $\langle\alpha\rangle$ corresponds to a derivation $dv = \langle a_1, \ldots, a_{k-1}, A_{k,j}, \alpha\rangle$ if α is the context of the stack after parsing the prefix of the string of length $k + j$. Therefore, it is enough to show that all partial left most derivation (we omit the prefix of terminals) can be generated by the procedure above. We prove by a contradiction. Suppose that $\langle a_1, \ldots, a_{i-1}, B_{i,j}, \beta\rangle$ is the partial left most derivation such that $\Gamma(p(root, B_{i,j})) \neq \beta$, where $p(root, B_{i,j})$ is the path from the root to the OR-NODE $B_{i,j}$ and for any partial derivation $\langle a_1, \ldots, a_{k-1}, A_{k,j}, \alpha\rangle$, such that $k < i$ $A_{k,j} \in G_a$ $\Gamma(p(root, A_{k,j})) = \alpha$. Consider the production rule that introduces the nonterminal $B_{i,j}$ to the partial derivation. If the production rule is $D \rightarrow C, B_{i,j}$, then the partial derivation is $\langle a_1, \ldots, a_f, D, \beta\rangle \Rightarrow|_{D \rightarrow C,B_{i,j}} \langle a_1, \ldots, a_f, C, B_{i,j}, \beta\rangle$. The path from the root to the node $B_{i,j}$ is a concatenation of the paths from D to $B_{i,j}$ and from the root to D. Therefore, $\Gamma(p(root, B_{i,j}))$ is constructed as a concatenation of $\Gamma(p(D, B_{i,j}))$ and $\Gamma(p(root, D))$. $\Gamma(p(D, B_{i,j}))$ is empty because the node $B_{i,j}$ is the right child of AND-NODE that corresponds to the production $D \rightarrow C, B_{i,j}$ and $\Gamma(p(root, D)) = \beta$ because $f < i$. Therefore, $\Gamma(p(root, B_{i,j})) = \beta$. If the production rule is $D \rightarrow B_{i,j}, C$, then the partial derivation is $\langle a_1, \ldots, a_{i-1}, D, \gamma\rangle \Rightarrow|_{D \rightarrow B_{i,j},C} \langle a_1, \ldots, a_{i-1}, B_{i,j}, C, \gamma\rangle = \langle a_1, \ldots, a_{i-1}, B_{i,j}, \beta\rangle$. Then, $\Gamma(p(root, D)) = \gamma$, because $i - 1 < i$ and $\Gamma(p(D, B_{i,j})) = \langle C\rangle$, because the node $B_{i,j}$ is the left

child of AND-NODE that corresponds to the production $D \rightarrow C, B_{i,j}$. Therefore, $\Gamma(p(root, B_{i,j})) = \langle C, \gamma \rangle = \beta$. This leads to a contradiction.   □

Example 5. An example of the mapping described in the last proof is in Figure 4(a) for the grammar of our running example. Consider the OR-NODE $A_{1,1}$. There are 2 paths from $S_{1,3}$ to $A_{1,1}$. One is direct and uses only OR-NODES $\langle S_{1,3}, A_{1,1} \rangle$ and the other uses OR-NODES $\langle S_{1,3}, A_{1,2}, A_{1,1} \rangle$. The 2 paths are mapped to 2 different stack configurations $\langle A_{1,1}, B_{2,2} \rangle$ and $\langle A_{1,1}, A_{2,1}, B_{3,1} \rangle$ respectively. We highlight edges that are incident to AND-NODES on each path and lead to the right children of these AND-NODES. There is exactly one such edge for each element of a stack configuration.   □

Note that theorem 1 only specifies a surjection from paths to stack configurations, not a bijection. Indeed, different paths may produce the same configuration Γ.

Example 6. Consider the grammar $G = \{S \rightarrow AA, A \rightarrow a|AA|BC, B \rightarrow b|BB, C \rightarrow c|CC\}$ and the AND/OR graph of this grammar for a string of length 5. The path $\langle S_{1,5}, A_{2,4}, B_{2,2} \rangle$ uses the productions $S_{1,5} \rightarrow A_{1,1}A_{2,4}$ and $A_{2,4} \rightarrow B_{2,2}C_{4,2}$, while the path $\langle S_{1,5}, A_{3,3}, B_{3,1} \rangle$ uses the productions $S_{1,5} \rightarrow A_{1,2}A_{3,3}$ and $A_{3,3} \rightarrow B_{3,1}C_{4,2}$. Both paths map to the same stack configuration $\langle C_{4,2} \rangle$.   □

By construction, the resulting NFA has one state for each stack configuration of the PDA in parsing a string. Since each path corresponds to a stack configuration, the number of states of the NFA before applying ε-closure is bounded by the number of paths from the root to any OR-NODE in the AND/OR graph. This is cheap to compute using the following recursive algorithm [6]:

$$PD(v) = \begin{cases} 1 & \text{If } v \text{ has no incoming edges} \\ \sum_p PD(p) & \text{where } p \text{ is a parent of } v \end{cases} \qquad (1)$$

Therefore, the number of states of the NFA N_a is at most $\sum_v PD(v)$, where v is an OR-NODE of G_a (Figure 4).

We can compute the exact number of paths in N_a before ε-closure without constructing the NFA by counting paths in the *stack graph* G_v for each OR-NODE v. The stack graph captures the observation that each element of a stack configuration generated
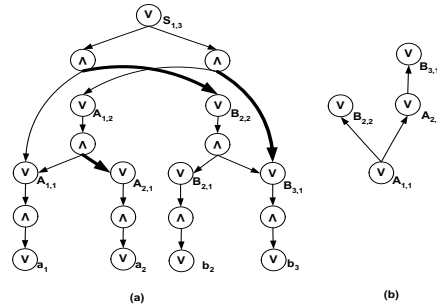
Fig. 4. Computing the size of N_a. (a) AND/OR graph G_a. (b) Stack graph $G_{A_{1,1}}$

from a path p is associated with exactly one edge e that is incident on p and leads to the right child of an AND-NODE. G_v contains one path for each sequence of such edges, so that if two paths p and p' in G_a are mapped to the same stack configuration, they are also mapped to the same path in G_v. Formally, the stack graph of an OR-NODE $v \in V(G_a)$ is a DAG G_v, such that for every stack configuration Γ of P_a with k elements, there is exactly one path p in G_v of length k and v' is the ith vertex of p if and only if v' is the ith element from the top of Γ.

Example 7. Consider the grammar of the running example and the OR-NODE $A_{1,1}$ in the AND/OR graph. The stack graph $G_{A_{1,1}}$ for this OR-NODE is shown in figure 4(b). Along the path $\langle S_{1,3} A_{1,1} \rangle$, only the edge that leads to $B_{2,2}$ generates a stack element. This edge is mapped to the edge $(A_{1,1}, B_{2,2})$ in $G_{A_{1,1}}$. Similarly, the edges that lead to $A_{2,1}$ and $B_{3,1}$ are mapped to the edges $(A_{1,1}, A_{2,1})$ and $(A_{2,1}, B_{3,1})$ respectively.   □

Since G_v is a DAG, we can efficiently count the number of paths in it. We construct G_v using algorithm 3. The graph G_v computed in algorithm 3 for an OR-NODE v has as many paths as there are unique stack configurations in P_a with v at the top.

Algorithm 3. Computing the stack DAG G_v of an OR-NODE v

```
 1: procedure STACKGRAPH((in : G_a, v, out : G_v))
 2:     V(G_v) = {v}
 3:     label(v) = {v}
 4:     Q = {(v, v_p)|v_p ∈ parents(v)}                        ▷ queue of edges
 5:     while Q not empty do
 6:         (v_c, v_p) = pop(Q)
 7:         if v_p is an AND-NODE v_c is left child of v_p then
 8:             v_r = children_r(v_p)
 9:             V(G_v) = V(G_v) ∪ {v_r}
10:             E(G_v) = E(G_v) ∪ {(v_l, v_r)|v_l ∈ label(v_c)}
11:             label(v_p) = label(v_p) ∪ {v_r}
12:         else
13:             label(v_p) = label(v_p) ∪ label(v_c)
14:         Q = Q ∪ {(v_p, v'_p)|v'_p ∈ parents(v_p)}          ▷
```

Theorem 2. *There exists a bijection between paths in G_v and states in the NFA N_a which correspond to stacks with v at the top.*

Proof. Let p be a path from the root to v in G_a. First, we show that every path p' in G_v corresponds to a stack configuration, by mapping p to p'. Therefore p' corresponds to $\Gamma(p)$. We then show that p' is unique for $\Gamma(p)$. This establishes a bijection between paths in G_v and stack configurations.

We traverse the inverse of p, denoted $inv(p)$ and construct p' incrementally. Note that every vertex in $inv(p)$ is examined by algorithm 3 in the construction of G_v. If $inv(p)$ visits the left child of an AND-NODE, we append the right child of that AND-NODE to p'. This vertex is in G_v by line 7. By the construction of $\Gamma(p)$ in the proof of theorem 1, a symbol is placed on the stack if and only if it is the right child of an AND-NODE, hence if and only if it appears in p'. Moreover, if a vertex is the ith vertex in a path, it corresponds to the ith element from the top of $\Gamma(p)$. We now see that p' is unique for $\Gamma(p)$. Two distinct paths of length k cannot map to the same stack configuration,

because they must differ in at least one position i, therefore they correspond to stacks with different symbols at position i. Therefore, there exists a bijection between paths in G_v and stack configurations with v at the top. $\square$

Hence $|Q(Ng)| = \sum_v \#paths(G_v)$, where v is an OR-NODE of G_a. Computing the stack graph G_v of every OR-NODE v takes $O(|G_a|)$ time, as does counting paths in G_v. Therefore, computing the number of states in N_a takes $O(|G_a|^2)$ time. We can also compute the number of states in the ε-closure of N_a by observing that if none of the OR-NODES that are reachable by paths of length 2 from an OR-NODE v correspond to terminals, then any state that corresponds to a stack configuration with v at the top will only have outgoing $\varepsilon-$transitions and will be removed by the $\varepsilon-$closure. Thus, to compute the number of states in N_a after $\varepsilon-$closure, we sum the number of paths in G_v for all OR-NODES v such that a terminal OR-NODE can be reached from v by a path of length 2.

4.5   Transformation into a DFA

Finally, we convert the NFA into a DFA using the standard subset construction. This is optional as Pesant's propagator for the REGULAR constraints works just as well with NFAs as DFAs. Indeed, removing non-determinism may increase the size of the automaton and slow down propagation. However, converting into a DFA opens up the possibility of further optimizations. In particular, as we describe in the next section, there are efficient methods to minimize the size of a DFA. By comparison, minimization of a NFA is PSPACE-hard in general [7]. Even when we consider just the acyclic NFA constructed by unfolding a NFA, minimization remains NP-hard [8].

5   Automaton Minimization

The DFA constructed by this or other methods may contain redundant states and transitions. We can speed up propagation of the REGULAR constraint by minimizing the size of this automaton. Minimization can be either offline (i.e. before we have the problem data and have unfolded the automaton) or online (i.e. once we have the problem data and have unfolded the automaton). There are several reasons why we might prefer an online approach where we unfold before minimizing. First, although minimizing after unfolding may be more expensive than minimizing before unfolding, both are cheap to perform. Minimizing a DFA takes $O(Q \log Q)$ time using Hopcroft's algorithm and $O(nQ)$ time for the unfolded DFA where Q is the number of states [9]. Second, thanks to Myhill-Nerode's theorem, minimization does not change the layered nature of the unfolded DFA. Third, and perhaps most importantly, minimizing a DFA after unfolding can give an exponentially smaller automaton than minimizing the DFA and then unfolding. To put it another way, unfolding may destroy the minimality of the DFA.

Theorem 3. *Given any DFA $\mathcal{A}$,* $|\min(unfold_n(\mathcal{A}))| \ll |unfold_n(\min(\mathcal{A}))|$.

Proof: To show $|\min(unfold_n(\mathcal{A}))| \leq |unfold_n(\min(\mathcal{A}))|$, we observe that both $\min(unfold_n(\mathcal{A}))$ and $unfold_n(\min(\mathcal{A}))$ are automata that recognize the same language. By definition, minimization returns the smallest DFA accepting this language. Hence $\min(unfold_n(\mathcal{A}))$ cannot be larger than $unfold_n(\min(\mathcal{A}))$.

To show unfolding then minimizing can give an exponentially smaller sized DFA, consider the following language L. A string of length k belongs to L iff it contains the symbol j, $j = k \bmod n$, where n is a given constant. The alphabet of the language L is $\{0, \ldots, n-1\}$. The minimal DFA for this language has $\Omega(n2^n)$ states as each state needs to record which symbols from 0 to $n-1$ have been seen so far, as well as the current length of the string mod n. Unfolding this minimal DFA and restricting it to strings of length n gives an acyclic DFA with $\Omega(n2^n)$ states. Note that all strings are of length n and the equation $j = n \bmod n$ has the single solution $j = 0$. Therefore, the language L consists of the strings of length n that contain the symbol 0. On the other hand, if we unfold and then minimize, we get an acyclic DFA with just $2n$ states. Each layer of the DFA has two states which record whether 0 has been seen.   □

Further, if we make our initial problem domain consistent, domains might be pruned which give rise to possible simplifications of the DFA. We show here that we should also perform such simplification before minimizing.

Theorem 4. *Given any DFA* $\mathcal{A}$, $|\min(simplify(unfold_n(\mathcal{A})))| \ll |simplify(\min(unfold_n(\mathcal{A})))|$.

Proof: Both $\min(simplify(unfold_n(\mathcal{A})))$ and $simplify(\min(unfold_n(\mathcal{A})))$ are DFAs that recognize the same language of strings of length n. By definition, minimization must return the smallest DFA accepting this language. Hence $\min(simplify(unfold_n(\mathcal{A})))$ is no larger than $simplify(\min(unfold_n(\mathcal{A})))$.

To show that minimization after simplification may give an exponentially smaller sized automaton, consider the language which contains sequences of integers from 1 to n in which at least one integer is repeated and in which the last two integers are different. The alphabet of the language L is $\{1, \ldots, n\}$. The minimal unfolded DFA for strings of length n from this language has $\Omega(2^n)$ states as each state needs to record which integers have been seen. Suppose the integer n is removed from the domain of each variable. The simplified DFA still has $\Omega(2^n)$ states to record which integers 1 to $n-1$ have been seen. On the other hand, suppose we simplify before we minimize. By a pigeonhole argument, we can ignore the constraint that an integer is repeated. Hence we just need to ensure that the string is of length n and that the last two integers are different. The minimal DFA accepting this language requires just $O(n)$ states.   □

6   Empirical Results

We empirically evaluated the results of our method on a set of shift-scheduling benchmarks [11,14] [1]. Experiments were run with the Minisat+ solver for pseudo-Boolean instances and Gecode 2.2.0 for constraint problems, on an Intel Xeon 4 CPU, 2.0 Ghz, 4G RAM. We use a timeout of 3600 sec in all experiments. The problem is to schedule employees to activities subject to various rules, e.g. a full-time employee has one hour for lunch. This rules are specified by a context-free grammar augmented with restrictions on productions [4]. A schedule for an employee has $n = 96$ slots of 15 minutes

---

[1] We would like to thank Louis-Martin Rousseau and Claude-Guy Quimper for providing us with the benchmark data.

represented by n variables. In each slot, an employee can work on an activity (a_i), take a break (b), lunch (l) or rest (r). These rules are specified by the following grammar:

$$S \rightarrow RPR, f_P(i,j) \equiv 13 \le j \le 24, \quad P \rightarrow WbW, \; L \rightarrow lL|l, f_L(i,j) \equiv j = 4$$
$$S \rightarrow RFR, f_F(i,j) \equiv 30 \le j \le 38, \quad R \rightarrow rR|r, \; W \rightarrow A_i, f_W(i,j) \equiv j \ge 4$$
$$A_i \rightarrow a_i A_i | a_i, f_A(i,j) \equiv open(i), \quad F \rightarrow PLP$$

where functions $f(i,j)$ are predicates that restrict the start and length of any string matched by a specific production, and $open(i)$ is a function that returns 1 if the business is open at i^{th} slot and 0 otherwise. In addition, the business requires a certain number of employees working in each activity at given times during the day. We minimize the number of slots in which employees work such that the demand is satisfied.

As shown in [4], this problem can be converted into a pseudo-Boolean (PB) model. The GRAMMAR constraint is converted into a SAT formula in conjunctive normal form using the AND/OR graph. To model labour demand for a slot we introduce Boolean variables $b(i,j,a_k)$, equal to 1 if j^{th} employee performs activity a_k at i^{th} time slot. For each time slot i and activity a_k we post a pseudo-Boolean constraint $\sum_{j=1}^{m} b(i,j,a_k) > d(i,a_k)$, where m is the number of employees. The objective is modelled using the function $\sum_{i=1}^{n} \sum_{j=1}^{m} \sum_{k=1}^{a} b_{i,j,a_k}$. Additionally, the problem can be formulated as an optimization problem in a constraint solver, using a matrix model with one row for each employee. We post a GRAMMAR constraint on each row, AMONG constraints on each column for labour demand and LEX constraints between adjacent rows to break symmetry. We use the static variable and value ordering used in [4].

We compare this with reformulating the GRAMMAR constraint as a REGULAR constraint. Using algorithm 3, we computed the size of an equivalent NFA. Surprisingly, this is not too big, so we converted the GRAMMAR constraint to a DFA then minimized. In order to reduce the blow-up that may occur converting a NFA to a DFA, we heuristically minimized the NFA using the following simple observation: two states are equivalent if they have identical outgoing transitions. We traverse the NFA from the last to the first layer and merge equivalent states and then apply the same procedure to the reversed NFA. We repeat until we cannot find a pair of equivalent states. We also simplified the original CYK table, taking into account whether the business is open or closed at each slot. Theorem 4 suggests such simplification can significantly reduce the size both of the CYK table and of the resulting automata. In practice we also observe a significant reduction in size. The resulting minimized automaton obtained before simplification is about ten times larger compared to the minimised DFA obtained after simplification. Table 1 gives the sizes of representations at each step. We see from this that the minimized DFA is always smaller than the original CYK table. Interestingly, the subset construction generates the minimum DFA from the NFA, even in the case of two activities, and heuristic minimization of the NFA achieves a notable reduction.

For each instance, we used the resulting DFA in place of the GRAMMAR constraint in both the CP model and the PB model using the encoding of the REGULAR constraint (DFA or NFA) into CNF [10]. We compare the model that uses the PB encoding of the GRAMMAR constraint (GR$_1$) with two models that use the PB encoding of the

Table 1. Shift Scheduling Problems. G_a is the acyclic grammar, N_a^ε is NFA with ε-transitions, N_a is NFA without ε-transitions, $\min(N_a)$ is minimized NFA, $\mathcal{A}$ is DFA obtained from $\min(N_a)$, $\min(\mathcal{A})$ is minimized $\mathcal{A}$, a is the number of activities, $\#$ is the benchmark number.

$\#act$	$\#$	G_a		NFA_a^ε		NFA_a		$\min(NFA_a)$		DFA		$\min(DFA)$	
		terms	prods	states	trans	states	trans	states	trans	states	trans	states	trans
1	2/3/8	4678	/ 9302	69050	/ 80975	29003	/ 42274	**3556**	**/ 4505**	3683	/ 4617	3681	/ 4615
1	4/7/10	3140	/ 5541	26737	/ 30855	11526	/ 16078	**1773**	**/ 2296**	1883	/ 2399	1881	/ 2397
1	5/6	2598	/ 4209	13742	/ 15753	5975	/ 8104	**1129**	**/ 1470**	1215	/ 1553	1213	/ 1551
2	1/2/4	3777	/ 6550	42993	/ 52137	19654	/ 29722	**3157**	**/ 4532**	3306	/ 4683	3303	/ 4679
2	3/5/6	**5407**	**/ 10547**	111302	/ 137441	50129	/ 79112	5975	**/ 8499**	6321	/ 8846	6318	/ 8842
2	8/10	**6087**	**/ 12425**	145698	/ 180513	65445	/ 104064	7659	**/ 10865**	8127	/ 11334	8124	/ 11330
2	9	4473	/ 8405	76234	/ 93697	34477	/ 53824	**4451**	**/ 6373**	4691	/ 6614	4688	/ 6610

REGULAR constraint (REGULAR$_1$, REGULAR$_2$), a CP model that uses the GRAMMAR constraint (GR$_1^{CP}$) and a CP model that uses a REGULAR constraint (REGULAR$_1^{CP}$). REGULAR$_1$ and REGULAR$_1^{CP}$ use the DFA, whilst REGULAR$_2$ uses the NFA constructed after simplification by when the business is closed.

The performance of a SAT solver can be sensitive to the ordering of the clauses in the formula. To test robustness of the models, we randomly shuffled each of PB instances to generate 10 equivalent problems and averaged the results over 11 instances. Also, the GRAMMAR and REGULAR constraints were encoded into a PB formula in two different ways. The first encoding ensures that unit propagation enforces domain consistency on the constraint. The second encoding ensures that UP detects disentailment of the constraint, but does not always enforce domain consistency. For the GRAMMAR constraint we omit the same set of clauses as in [4] to obtain the weaker PB encoding. For the REGULAR constraint we omit the set of clauses that performs the backward propagation of the REGULAR constraint. Note that Table 2 shows the median time and the number of backtracks to prove optimality over 11 instances. For each model we show the median time and the corresponding number of backtracks for the best PB encoding between the one that achieves domain consistency and the weaker one.

Table 2 shows the results of our experiments using these 5 models. The model REGULAR$_2$ outperforms GR$_1$ in all benchmarks, whilst model REGULAR$_1$ outperforms GR$_1$ in most of the benchmarks. The model REGULAR$_2$ also proves optimality in several instances of hard benchmarks. It should be noted that performing simplification before minimization is essential. It significantly reduces the size of the encoding and speeds up MiniSat+ by factor of 5[2]. Finally, we note that the PB models consistently outperformed the CP models, in agreement with the observations of [4]. Between the two CP models, REGULAR$_1^{CP}$ is significantly better than GR$_1^{CP}$, finding a better solution in many instances and proving optimality in two instances. In addition, although we do not show it in the table, Gecode is approximately three orders of magnitude faster per branch with the REGULAR$_1^{CP}$ model. For instance, in benchmark number 2 with 1 activity and 4 workers, it explores approximately 80 million branches with the REGULAR$_1^{CP}$ and 24000 branches with the GR$_1^{CP}$ model within the 1 hour timeout.

---

[2] Due to lack of space we do not show these results.

Table 2. Shift Scheduling Problems. GR_1 is the PB model with GRAMMAR, $REGULAR_1$ is the PB model with $\min(simplify(DFA))$, $REGULAR_2$ is the PB model with $\min(simplify(NFA))$, GR_1^{CP} is the CSP model with GRAMMAR, $REGULAR_1^{CP}$ is the CSP model with $\min(simplify(DFA))$. We show **t**ime and number of **b**acktracks to prove optimality (the median time and the median number of backtracks for the PB encoding over **s**olved shuffled instances), number of **a**ctivities, the number of **w**orkers and the benchmark number **#**.

| | | | PB/Minisat+ | | | | | | | | | CSP/Gecode | | | |
| | | | GR_1 | | | $REGULAR_1$ | | | $REGULAR_2$ | | | GR_1^{CP} | | $REGULAR_1^{CP}$ | |
a	$\#$	w	cost	s	$t\,/\,b$	cost	s	$t\,/\,b$	cost	s	$t\,/\,b$	cost	$t\,/\,b$	cost	$t\,/\,b$
1	2	4	**26.00**	11	27 / 8070	**26.00**	11	9 / 11053	**26.00**	11	4 / 7433	26.75	- / -	**26.00**	- / -
1	3	6	**36.75**	11	530 / 101560	**36.75**	11	94 / 71405	**36.75**	11	39 / 58914	37.00	- / -	37.00	- / -
1	4	6	**38.00**	11	31 / 16251	**38.00**	11	12 / 10265	**38.00**	11	6 / 7842	38.00	- / -	38.00	- / -
1	5	5	**24.00**	11	5 / 3871	**24.00**	11	2 / 4052	**24.00**	11	2 / 2598	24.00	- / -	24.00	- / -
1	6	6	**33.00**	11	9 / 5044	**33.00**	11	4 / 4817	**33.00**	11	3 / 4045	-	- / -	33.00	- / -
1	7	8	**49.00**	11	22 / 7536	**49.00**	11	9 / 7450	**49.00**	11	7 / 8000	49.00	- / -	49.00	- / -
1	8	3	**20.50**	11	13 / 4075	**20.50**	11	4 / 5532	**20.50**	11	2 / **1901**	21.00	- / -	**20.50**	92 / 2205751
1	10	9	**54.00**	11	242 / 106167	**54.00**	11	111 / **91804**	**54.00**	11	110 / 109123	-	- / -	-	- / -
2	1	5	**25.00**	11	92 / 35120	**25.00**	11	96 / 55354	**25.00**	11	32 / 28520	25.00	- / -	25.00	90 / 1289554
2	2	10	**58.00**	1	3161 / **555249**	**58.00**	0	- / -	**58.00**	4	2249 / 701490	-	- / -	58.00	- / -
2	3	6	**37.75**	0	- / -	**37.75**	1	3489 / 590649	**37.75**	9	2342 / 570863	42.00	- / -	40.00	- / -
2	4	11	**70.75**	0	- / -	71.25	0	- / -	71.25	0	- / -	-	- / -	-	- / -
2	5	4	**22.75**	11	739 / 113159	**22.75**	11	823 / 146068	**22.75**	11	308 / 69168	23.00	- / -	23.00	- / -
2	6	5	**26.75**	11	86 / 25249	**26.75**	11	153 / 52952	**26.75**	11	28 / 21463	26.75	- / -	26.75	- / -
2	8	5	**31.25**	11	1167 / 135983	**31.25**	11	383 / 123612	**31.25**	11	74 / 47627	32.00	- / -	31.50	- / -
2	9	3	**19.00**	11	1873 / 333299	**19.00**	11	629 / 166908	**19.00**	11	160 / 131069	19.25	- / -	**19.00**	- / -
2	10	8	**55.00**	0	- / -	**55.00**	0	- / -	**55.00**	0	- / -	-	- / -	-	- / -

7   Other Related Work

Beldiceanu *et al* [12] and Pesant [1] proposed specifying constraints using automata and provided filtering algorithms for such specifications. Quimper and Walsh [3] and Sellmann [2] then independently proposed the GRAMMAR constraint. Both gave a monolithic propagator based on the CYK parser. Quimper and Walsh [4] proposed a CNF decomposition of the GRAMMAR constraint, while Bacchus [10] proposed a CNF decomposition of the REGULAR constraint. Kadioglu and Sellmann [13] improved the space efficiency of the propagator for the GRAMMAR constraint by a factor of n. Their propagator was evaluated on the same shift scheduling benchmarks as here. However, as they only found feasible solutions and did not prove optimality, their results are not directly comparable. Côté, Gendron, Quimper and Rousseau proposed a mixed-integer programming (MIP) encoding of the GRAMMAR constraint [14], Experiments on the same shift scheduling problem used here show that such encodings are competitive.

There is a body of work on other methods to reduce the size of constraint representations. Closest to this work is Lagerkvist who observed that a REGULAR constraint represented as a multi-value decision diagram (MDD) is no larger than that represented by a DFA that is minimized and then unfolded [15]. A MDD is similar to an unfolded and then minimized DFA except a MDD can have long edges which skip over layers. We extend this observation by proving an exponential separation in size between such representations. As a second example, Katsirelos and Walsh compressed table constraints representing allowed or disallowed tuples using decision tree methods [16]. They also used a compressed representation for tuples that can provide exponentially savings in space. As a third example, Carlsson proposed the CASE constraint which can be

represented by a DAG where each node represents a range of values for a variable, and a path from the root to a leaf represents a set of satisfying assignments [17].

8   Conclusions

We have shown how to transform a GRAMMAR constraint into a REGULAR constraint specified. In the worst case, the transformation may increase the space required to represent the constraint. However, in practice, we observed that such transformation reduces the space required to represent the constraint and speeds up propagation. We argued that transformation also permits us to compress the representation using standard techniques for automaton minimization. We proved that minimizing such automata after they have been unfolded and domains initially reduced can give automata that are exponentially more compact than those obtained by minimizing before unfolding and reducing. Experimental results demonstrated that such transformations can improve the size of rostering problems that can be solved.

References

1. Pesant, G.: A regular language membership constraint for finite sequences of variables. In: Wallace, M. (ed.) CP 2004. LNCS, vol. 3258, pp. 482–495. Springer, Heidelberg (2004)
2. Sellmann, M.: The theory of grammar constraints. In: Benhamou, F. (ed.) CP 2006. LNCS, vol. 4204, pp. 530–544. Springer, Heidelberg (2006)
3. Quimper, C.G., Walsh, T.: Global grammar constraints. In: Benhamou, F. (ed.) CP 2006. LNCS, vol. 4204, pp. 751–755. Springer, Heidelberg (2006)
4. Quimper, C.G., Walsh, T.: Decomposing global grammar constraints. In: Bessière, C. (ed.) CP 2007. LNCS, vol. 4741, pp. 590–604. Springer, Heidelberg (2007)
5. Hopcroft, J., Ullman, J.: Introduction to Automata Theory, Languages, and Computation. Addison Wesley Publishing Company, Reading (1979)
6. Darwiche, A.: On the tractable counting of theory models and its application to truth maintenance and belief revision. J. of Applied Non-Classical Logics 11, 11–34 (2001)
7. Meyer, A., Stockmeyer, L.: The equivalence problem for regular expressions with squaring requires exponential space. In: 13th Annual Symposium on Switching and Automata Theory, pp. 125–129. IEEE, Los Alamitos (1972)
8. Amilhastre, J., Janssen, P., Vilarem, M.C.: FA minimisation heuristics for a class of finite languages. In: Boldt, O., Jürgensen, H. (eds.) WIA 1999. LNCS, vol. 2214, pp. 1–12. Springer, Heidelberg (2001)
9. Revuz, D.: Minimization of ayclic deterministic automata in linear time. TCS 92, 181–189 (1992)
10. Bacchus, F.: GAC via unit propagation. In: Bessière, C. (ed.) CP 2007. LNCS, vol. 4741, pp. 133–147. Springer, Heidelberg (2007)
11. Demassey, S., Pesant, G., Rousseau, L.M.: Constraint programming based column generation for employee timetabling. In: Barták, R., Milano, M. (eds.) CPAIOR 2005. LNCS, vol. 3524, pp. 140–154. Springer, Heidelberg (2005)
12. Beldiceanu, N., Carlsson, M., Petit, T.: Deriving filtering algorithms from constraint checkers. In: Wallace, M. (ed.) CP 2004. LNCS, vol. 3258, pp. 107–122. Springer, Heidelberg (2004)

13. Kadioglu, S., Sellmann, M.: Efficient context-free grammar constraints. In: AAAI 2008, pp. 310–316 (2008)
14. Cote, M.C., Bernard, G., Claude-Guy, Q., Louis-Martin, R.: Formal languages for integer programming modeling of shift scheduling problems. TR (2007)
15. Lagerkvist, M.: Techniques for Efficient Constraint Propagation. PhD thesis, KTH, Sweden, Licentiate thesis (2008)
16. Katsirelos, G., Walsh, T.: A compression algorithm for large arity extensional constraints. In: Bessière, C. (ed.) CP 2007. LNCS, vol. 4741, pp. 379–393. Springer, Heidelberg (2007)
17. Carlsson, M.: Filtering for the case constraint, Talk given at Advanced School on Global Constraints, Samos, Greece (2006)