Global Grammar Constraints

Claude-Guy Quimper[1] and Toby Walsh[2]

[1] School of Computer Science, University of Waterloo, Canada, cquimper@math.uwaterloo.ca
[2] NICTA and UNSW, Sydney, Australia, tw@cse.unsw.edu.au

Abstract. We consider global constraints over a sequence of variables which restrict the values assigned to be a string within a given language defined by a grammar or automaton. Such constraints are useful in a wide range of scheduling, rostering and sequencing problems. For regular languages, we gave a simple encoding into ternary constraints that can be used to enforce GAC in time linear in the number of variables. We study a number of extensions including regular languages specified by non-deterministic automata, and soft and cyclic versions of the global constraint. For context-free languages, we give two propagators which enforce GAC based on the CYK and Earley parsers.

1 Introduction

Global constraints are an important tool in the constraint toolkit. Unfortunately, whilst it is usually easy to specify when a global constraint holds, it is often difficult to build a good propagator. For example, it is easy to specify when an AllDifferent constraint holds as we just need to check $X_i \neq X_j$ for $i < j$. However, it is complex to implement a global propagator which is efficient, effective and incremental. For this reason, researchers have tried to build frameworks in which global constraints can be specified and efficient propagators automatically extracted. One direction is to specify global constraints using Boolean combinations of more primitive constraints [1–3]. The problem here is that either we lose the global nature of the propagator or propagation becomes intractable. Another direction is to specify global constraints via graph properties where the graph is a structured network of elementary constraints [4]. The problem here is to extract propagators from such specifications (but see [5, 6] for some first steps). Indeed, such specifications can easily specify constraints like NValues which are NP-hard to propagate completely.

Another direction is to specify global constraints via grammars or automata. The Regular constraint [7] permits us to specify a global constraint by means of a regular language, and to propagate this constraint specification efficiently and effectively. More precisely, the Regular constraint ensures that the values taken by a sequence of variables form a string accepted by the deterministic finite automaton (DFA). Regular languages are precisely those accepted by a DFA. We show here how to implement a propagator for the Regular constraint using nothing more than ternary constraints. We can therefore incorporate the

REGULAR constraint into constraint toolkits with relative ease. One limitation of this approach is that we cannot compactly specify everything we might like using just deterministic finite automaton. We therefore consider a number of extensions including regular languages specified by non-deterministic finite automata, and soft and cyclic versions of this global constraint. Finally, we consider global constraints specified by context-free and context-sensitive languages.

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 of values to variables satisfying the constraints. Finite domain variables take values which are taken from a finite set of integers. Set variables takes values which are sets of integers. A set variable S has a lower bound $lb(S)$ for its definite elements and an upper bound $ub(S)$ for its definite and potential elements. Constraint solvers typically explore partial assignments enforcing a local consistency property. Given a constraint C on finite domain variables, a *support* is assignment to each variable of a value in its domain which satisfies C. A constraint C on finite domain variables is *generalized arc consistent* (GAC) iff for each variable, every value in its domain belongs to a support. We will also consider constraints involving both finite domain and set variables. We therefore consider a local consistency property for such situations. Given a constraint C on finite domain and set variables, a *hybrid support* on C is an assignment of a value to each finite domain variable within its domain, and of a set to each set variable between its lower and upper bounds which satisfies C. A value for an integer or set variable is *hybrid consistent* with C iff there exists a hybrid support assigning this value to this variable. A constraint C is *hybrid consistent* (HC) iff for each integer variable, every value in its domain belongs to an hybrid support, and for each set variable S the values in $ub(S)$ belong to S in at least one hybrid support, and the values in $lb(S)$ belong to S in all hybrid supports.

We will consider global constraints which are specified in terms of a grammar or automaton which accepts just valid assignments for a sequence of variables. A deterministic finite automaton (DFA) Ω is given by $\langle Q, \Sigma, T, q_0, F\rangle$ where Q is a finite set of states, Σ is an alphabet, $T : \Sigma \times Q \mapsto Q$ is the transition function, $q_0 \in Q$ is the initial state and $F \subseteq Q$ is the set of final (or accepting) states. A non-deterministic finite automaton (NFA) is given by a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where δ is a transition function from $\Sigma \times Q \mapsto 2^Q$. The automaton starts in the initial state q_0. In state q, the automaton inputs the next symbol $s \in \Sigma$ and non-deterministically moves to one of the states in $\delta(s, q)$. The string is accepted iff there is a path that ends in a state $q \in F$. Both DFA and NFA accept precisely regular languages. Context-free languages are above regular languages in the Chomsky hierarchy. Context-free languages are exactly those accepted by non-deterministic pushdown automaton. A context-free language can be specified by a set of grammar rules in which the left-hand side has just one non-terminal, and the right-hand side may have a string of terminals and non-terminals. In fact, any context-free grammar can be written in Chomsky normal form in which each rule yields either just one terminal or two non-terminals.

3 Regular constraint

In many scheduling, rostering and sequencing problems, we may need to ensure certain patterns do (or do not) occur over time. For example, we may wish that no worker has three consecutive night shifts. The Regular constraint and related constraints like Stretch have been proposed to model such situations [7]. The constraint Regular($\mathcal{A}, [X_1, \ldots, X_n]$) holds iff the string defined by the sequence of variables, X_1 to X_n is part of the regular language recognized by the deterministic finite automaton (DFA), $\mathcal{A}$. This can be used to encode a wide variety of useful global constraints including the Stretch constraint [7], and the precedence constraint for breaking value symmetries [8]. Pesant gives a domain consistency algorithm for the Regular constraint based on dynamic programming that runs in $O(ndQ)$ time and $O(ndQ)$ space where d is the maximum domain size and Q is the number of states of the DFA.

We show here that dynamic programming is not needed. The Regular constraint can be encoded using a simple sequence of ternary constraints. Enforcing GAC on this decomposition achieves GAC on the original Regular constraint. In addition, enforcing GAC on the decomposition takes just $O(ndQ)$ time, as with Pesant's propagator. To encode the Regular constraint, we introduce a second sequence of variables, Q_0 to Q_n to represent the state of the automaton. We then post the sequence of transition constraints $C(X_{i+1}, Q_i, Q_{i+1})$ for $0 \leq i < n$ which hold iff $Q_{i+1} = T(X_{i+1}, Q_i)$ where T is the transition function of the DFA. In addition, we post the unary constraints $Q_0 = q_0$ and $Q_n \in F$.

Theorem 1 *Enforcing GAC on the ternary encoding of the* Regular *constraint enforces GAC on the original* Regular *constraint in $O(ndQ)$ time.*

Proof: Clearly an assignment which satisfies the encoding corresponds to a string accepted by the DFA. As the constraint graph of this encoding is Berge-acyclic, enforcing GAC on the individual ternary constraints achieves GAC on the original Regular constraint. Since the third argument of T is functional on the first two arguments, it takes $O(dQ)$ time to enforce GAC on each ternary constraint. As there are $O(n)$ ternary constraints, the total time complexity to enforce GAC on the decomposition is $O(ndQ)$. $\diamond$

To enforce GAC on the ternary constraints in the decomposition, we can use the table constraint available in many solvers, a generic propagator like GAC-schema, or primitives like implication. Alternatively we can use any efficient algorithm available to propagate the ternary constraints. Consider, for example, the constraint $Max(N, [X_1, \ldots, X_n])$ which ensures that N is the maximum value taken by X_1 to X_n. This can be implemented with a Regular constraint that uses a DFA with d possible states. A naive implementation using a Table constraint will take $O(nd^2)$ time to propagate. However, we can post the ternary transition constraints symbolically: $Q_{n+1} = \max(X_{n+1}, Q_n)$. Each ternary transition constraint can be propagated in constant time so the overall time complexity reduces from $O(nd^2)$ to just $O(n)$.

Another advantage of this encoding is that we have explicit access to the states of the automaton. Consider, for example, a rostering problem where workers are allowed to work for up to three consecutive shifts and then must take a break. This can be specified with a simple REGULAR language constraint. Suppose we want to minimize the number of times a worker has to work for three consecutive shifts. To model this, we can impose a global cardinality constraint on the state variables to count the number of times we visit the state representing three consecutive shifts, and minimize the value taken by this variable.

The states of the automaton need not be finite domain variables but can, for example, be set variables. For instance, we can model the open stacks problem from the IJCAI-05 constraint modelling and solving challenge using two automata. The first automaton has a state variable which is the set of customers who have products produced so far. The transition function is simply $Q_{i+1} = Q_i \cup customer(X_i)$ where *customer* returns the set of customers ordering a given product. The second automaton is identical but runs backwards from X_n to X_1. To restrict the number of open stacks, we simply post a constraint on the cardinality of the intersection of the two sets of state variables.

4 NFA constraint

A limitation of the REGULAR constraint is that the language needs to be specified by a DFA. Unfortunately, there are regular languages which can only be defined by a DFA with an exponential number of states. Consider, for example, a scheduling problem where we wish to ensure that a maintenance shift takes place k shifts before we close the factory for the winter. This might translate to a sequence of shift variables which take the values defined by the regular expression: $0^*(1|2)^*2(1|2)^{k-1}0^*$ where 0 represents the factory being closed, 1 represents a production shift and 2 represents a maintenance shift. Any DFA specifying this regular language has at least 2^k states.

One way around this problem is to use a non-deterministic finite automaton (NFA). Whist NFA still only recognize regular languages, they can do so with exponentially fewer states than the smallest DFA. Our maintenance scheduling problem can, for example, be represented by a NFA with just $O(k)$ states. We let NFA$(\mathcal{A}, [X_1, \ldots, X_n])$ be a global constraint which holds iff X_1 to X_n give a string recognized by the non-deterministic finite automaton $\mathcal{A}$. We can permit the NFA to use ϵ transitions (which do not consume any input symbol) since any NFA using such transitions can be converted into one that does not use them without increasing the number of states.

To propagate NFA$(\mathcal{A}, [X_1, \ldots, X_n])$ we can again encode it into a simple sequence of ternary constraints by introducing a sequence of state variables, Q_0 to Q_n to represent the state of the DFA. We then post the sequence of ternary constraints $D(X_{i+1}, Q_i, Q_{i+1})$ for $0 \leq i < n$ which hold iff $Q_{i+1} \in \delta(X_{i+1}, Q_i)$

where δ is the transition function of the NFA. We again let $Q_0 = q_0$ and $Q_n \in F$. As with the REGULAR constraint, the constraint hyper-graph of the encoding is Berge-acyclic [9]. Enforcing GAC on this decomposition therefore achieves GAC on NFA$(\mathcal{A}, [X_1, \ldots, X_n])$. This takes $O(ndQ^2)$ time in general. However, if the automaton $\mathcal{A}$ is deterministic, the ternary constraints are functional on their first two arguments, and we can achieve GAC in $O(ndQ)$ time (as with Pesant's algorithm for the REGULAR constraint specified by a DFA). This analysis may leave the impression that propagating the NFA constraint is more difficult than propagating the REGULAR constraint. This is not the case. The NFA constraint can give exponential savings. Enforcing GAC for a deterministic or non-deterministic automaton takes $O(nT)$ time, where T is the number of transitions in the automaton. This number can be exponentially smaller for a NFA compared to a DFA. The key to a fast propagation algorithm is to encode the regular language with the fewest number of transitions T regardless of whether the automaton is deterministic or non-deterministic.

5 Soft forms of the REGULAR constraint

If our problem is over-constrained, we might want to insist that we are "near" to a string in the regular language. van Hoeve, Pesant and Rousseau [10] proposed a generalization of the REGULAR constraint to deal with such situations. REGULAR$_{\text{soft}}([X_1, \ldots, X_n], N, \Omega)$ holds iff the values taken by X_1 to X_n form a string that is at most distance N from a string accepted by Ω. Distance is either Hamming distance (giving the usual variable-based costs) or edit distance (which may be more useful in certain circumstances). We can model Hamming distance using an additional sequence of variables, D_0 to D_n in which to count distance. More precisely, we post the constraints $E(X_{i+1}, Q_i, Q_{i+1}, D_i, D_{i+1})$ which hold iff either $T(X_{i+1}, Q_i) = Q_{i+1}$ and $D_{i+1} = D_i$, or $T(X_{i+1}, Q_i) \neq Q_{i+1}$ and $D_{i+1} = D_i + 1$. In addition, we need $Q_0 = q_0$, $Q_n \in F$, $D_0 = 0$ and $D_n \leq N$. We can again implement a constraint like E using a table constraint, a generic propagator like GAC-schema, or primitives like implication and equality. To model edit distance, we again count using a sequence of variables. We model substitutions (as above), and insertions by letting E also hold if $Q_{i+1} = Q_i$ and $D_{i+1} = D_i + 1$. Deletions require a little more work. Let $G_k(q)$ be the set of states that can be reached from state q with a word of size k. We then also let E hold if $q \in G_k(Q_i)$, $T(X_{i+1}, q) = Q_{i+1}$ and $D_{i+1} = D_i + k$. In either case, decomposition now hinders propagation as the constraint graph is not Berge-acyclic. We can either accept this, or achieve global consistency by combining together the cost and the state information into a single tuple variable. This increases the number of states by a factor of $O(n)$ so we achieve GAC in $O(n^2dQ)$. We can deal with soft forms of the NFA constraint in a similar way.

6 Cyclic forms of the REGULAR constraint

We may want to find a repeating sequence. We therefore introduce two cyclic forms of the REGULAR constraint. REGULAR$_+$ considers how the sequence wraps around. More precisely, $\text{REGULAR}_{+}(\mathcal{A}, [X_1, \ldots, X_n])$ holds iff the string defined by $X_1 \ldots X_n X_1$ is part of the regular language recognized by the deterministic finite automaton $\mathcal{A}$. For example, in a rostering problem where the shift pattern is repeated every four weeks, such a constraint can be used to ensure that shifts changes only according to a set of valid patterns (e.g. a night shift is only followed by another night shift or a rest day, and is not followed by a day shift). A stronger form of cyclicity is defined by the $\text{REGULAR}_{\text{o}}$ constraint which ensures that any rotation of the sequence gives a string in the regular language. More precisely, $\text{REGULAR}_{\text{o}}(\mathcal{A}, [X_1, \ldots, X_n])$ holds iff the strings defined by $X_i \ldots X_{1+(i+n-1 \bmod n)}$ for $1 \leq i \leq n$ are part of the regular language recognized by the deterministic finite automaton $\mathcal{A}$. For example, in a rostering problem where the shift pattern is repeated every four weeks, we might want to insist that no person works more than three night shifts in any seven day period.

6.1 REGULAR_{+} constraint

We can decompose REGULAR_{+} into a non-cyclic REGULAR constraint and an additional equality constraint. More precisely, $\text{REGULAR}_{+}(\mathcal{A}, [X_1, \ldots, X_n])$ decomposes into $\text{REGULAR}(\mathcal{A}, [X_1, \ldots, X_n, X_{n+1}])$ and $X_1 = X_{n+1}$. Not surprisingly, this decomposition hinders propagation. To enforce GAC on REGULAR_{+}, we can create a new automaton $\mathcal{A}' = \langle Q', \Sigma, T', q'_0, F' \rangle$. Let $C = \{c \in \Sigma \mid T(c, q_0) \text{ is defined}\}$ be the set of characters allowed to start a sequence. We construct the set of states Q' by duplicating each non-initial state of $\mathcal{A}$. We have $Q' = \{q^c \mid q \in Q - \{q_0\} \text{ and } c \in C\} \cup \{q'_0\}$. The set of final states is given by $F' = \{q^c \mid q \in F \text{ and } c \in C\}$. We have the following transitions T': $T'(t, q_i^c) = q_j^c$ for all $c \in C$, $q_i, q_j \in Q - (\{q_0\} \cup F)$, $t \in \Sigma$ such that $T(t, q_i) = q_j$; $T'(c, q'_0) = q_i^c$ for $T(c, q_0) = q_i$ and $T'(c, q^c) = q_f^c$ for all $q_f \in F$ such that $T(c, q) = q_f$. This new automaton accepts a subset of the sequences accepted by $\mathcal{A}$, those whose first and last character are equal. We can therefore use this new automaton and a normal REGULAR constraint to enforce GAC on REGULAR_{+} in $O(nd^2Q)$ time.

6.2 $\text{REGULAR}_{\text{o}}$ constraint

The cyclic $\text{REGULAR}_{\text{o}}$ constraint can also be decomposed into a sequence of REGULAR constraints. More precisely, $\text{REGULAR}_{\text{o}}(\mathcal{A}, [X_1, \ldots, X_n])$ decomposes into $\text{REGULAR}(\mathcal{A}, [X_i, \ldots, X_{1+(i+n-1 \bmod n)}])$ for $1 \leq i \leq n$. Not surprisingly, this decomposition hinders propagation. Consider the automaton $\mathcal{A}$ which accepts strings that alternate 0 and 1. Then the decomposition into three REGULAR constraints is GAC if $X_i \in \{0, 1\}$. On the other hand, enforcing GAC on $\text{REGULAR}_{\text{o}}(\mathcal{A}, [X_1, X_2, X_3])$ gives a domain wipeout. Nevertheless, the decomposition is a way to propagate $\text{REGULAR}_{\text{o}}$ as enforcing GAC is NP-hard.

Theorem 2 *Enforcing GAC on a* $\text{REGULAR}_{\text{o}}$ *constraint is NP-hard.*

Proof: Consider a graph $G = \langle V, E \rangle$ in which we want to find a Hamiltonian cycle. Let $\mathcal{A}$ be an automaton with alphabet Σ accepting any sequence such that the first character appears only once in the sequence. Such an automaton requires no more than $O(d)$ states and $O(d^2)$ transitions where $d = |\Sigma|$. Let $\mathcal{B}$ be an automaton whose alphabet is Σ and that accepts any walk in graph G, i.e. a character $a \in \Sigma$ can be followed by a character $b \in \Sigma$ iff $(a, b) \in E$. Automaton $\mathcal{B}$ can be constructed with one state per node in V and one transition per edge in E. We choose an arbitrary state as an initial state and all states are terminal states. Finally, we construct in polynomial time an automaton $\mathcal{C} = \mathcal{A} \cap \mathcal{B}$ that accepts the intersection of both languages. This automaton has no more than $O(d^2)$ states and $O(d^4)$ transitions. There is a Hamiltonian cycle in graph G if and only if $\textsc{Regular}_{\text{o}}(\mathcal{C}, [X_1, \ldots, X_n])$ is satisfiable for $\text{dom}(X_i) = \Sigma$. Hence checking for support for the $\textsc{Regular}_{\text{o}}$ constraint is NP-hard. $\diamond$

Similar results can be derived for cyclic forms of the NFA constraint.

7 Other extensions of Regular

We consider two more generalizations of the Regular constraint that are met in practice. The first is where we have some values which are forced to occur (e.g. a maintenance shift must occur somewhere in the schedule). The second is where variables are repeated (e.g. we want to get the same shift each weekend). Unfortunately, in both cases, it becomes NP-hard in general to enforce GAC. To force certain values to occur somewhere within a Regular constraint, we consider $\textsc{Regular}_{\text{fix}}(\mathcal{A}, [X_1, \ldots, X_n], [B_1, \ldots, B_m])$ which holds iff both $\textsc{Regular}(\mathcal{A}, [X_1, \ldots, X_n])$ holds and $B_i = 1$ iff $\exists j. X_j = i$ for all $1 \leq i \leq m$.

Theorem 3 *Enforcing GAC on a* $\textsc{Regular}_{\text{fix}}$ *constraint is NP-hard even if* B_i *are ground.*

Proof: A simple reduction from Hamiltonian Path. The result also quickly follows from the proof that enforcing GAC on the Forced Shift Stretch constraint is NP-hard [11]. $\diamond$

Theorem 4 *Enforcing GAC on a* Regular *constraint is NP-hard when variables are repeated, even if no variable is repeated more than three times.*

Proof: We use a reduction from a special case of 3SAT in which at most three clauses contain a variable or its negation. Each Boolean variable x in the 3SAT problem is represented by (at most three occurrences of) a CSP variable X in the Regular constraint. If the 3SAT problem has m clauses, then we construct a Regular constraint over $4m$ CSP variables. Each clause is represented by a block of 4 variables. Suppose the ith clause is $x \vee \neg y \vee z$. Then we have the sequence of variables: $UXYZ$ where U is a CSP variable introduced to ensure we satisfy the clause, and X, Y and Z are $0/1$ variables representing the Boolean variables. The domain of U are the 7 tuples satisfying the clause (e.g. for $x \vee \neg y \vee z$, the domain of the clause variable includes $\langle 1, 0, 0 \rangle$ but not $\langle 0, 1, 0 \rangle$). The states of the DFA are all possible tuples of truth values up to size 3 and an initial state q_0 which is also the final accepting state. The transition function satisfies $T(U, q_0) = U$ (that is, we transition from the initial state to the value in the first clause variable), $T(X, \langle X, Y, Z \rangle) = \langle Y, Z \rangle$, $T(Y, \langle Y, Z \rangle) = \langle Z \rangle$, $T(Z, \langle Z \rangle) = q_0$. A string accepted by this DFA corresponds to a satisfying truth assignment. Hence, enforcing GAC is NP-hard. $\diamond$

8 Cfg constraint

Another generalization is to context-free languages. Unlike the generalizations considered in the last section, this is tractable. However, the high cost of propagation means this generalization may be limited to small grammars. Context-free languages strictly contain regular languages. For example, consider a stacking constraint which ensures that we have a sequence of null characters, n followed by a sequence of objects and then its reverse, and finally another sequence of null characters. For simplicity, we consider just two types of object, a and b being stacked. Thus, the global constraint ensures that we have a string of the form $n^* w w^{rev} n^*$ where $w \in \{a, b\}^*$ and $|w| \geq 1$. This language is not regular but can be specified by a Cfg constraint with the following grammar, $\mathcal{G}_1$:

$$
\begin{aligned}
S &\to N\ P \mid P \mid P\ N \\
P &\to a\ P\ a \mid b\ P\ b \mid a\ a \mid b\ b \\
N &\to n \mid n\ N
\end{aligned}
$$

Where S is the start symbol.

We therefore introduce the global grammar constraint Cfg(G, $[X_1, \ldots, X_n]$) which ensures that X_1 to X_n form a string accepted by the context-free grammar $\mathcal{G}$. Such a constraint might be useful in a number of applications:

Rostering and car sequencing: we might wish to express constraints that are not expressible using a regular language (e.g. that there must be twice as many cars on the assembly line without the sunroof option as with);

Configuration: a product might be hierarchically specified using a context-free grammar (e.g. the computer consists of a motherboard, and input and output devices, the motherboard itself consists of a CPU, and memory, etc.);

Bioinformatics: patterns in genes and other types of sequences involving palindromes might be represented using a context-free grammar;

Natural language processing: in speech recognition, we may need to choose between different possible words which can be parsed with a context-free grammar;

To illustrate the Cfg constraint, we consider a small example. Suppose $X_1 \in \{n, a\}$, $X_2 \in \{b\}$, $X_3 \in \{a, b\}$ and $X_4 \in \{n, a\}$, Then enforcing GAC on Cfg($\mathcal{G}_1, [X_1, X_2, X_3, X_4]$) prunes a from X_3 as there are only two satisfying sequences: $nbbn$ and $abba$.

8.1 CYK-style propagator

To achieve GAC on a Cfg constraint, we give a propagator based on the CYK parser which requires the context-free grammar to be in Chomsky normal form.

We assume that S is the unique starting non-terminal. The propagator given in Algorithm 1 proceeds in two phases. In the first phase (lines 1 to 7), we use dynamic programming to construct a table $V[i, j]$ with the potential non-terminal symbols that can be parsed using values in the domains of X_i to X_{i+j-1}. $V[1, n]$ thus contains all the possible parsings of the sequence of n variables.

In the second phase of the algorithm (lines 9 to 19), we backtrack in the table V and mark each triplet (i, j, A) such that there exists a valid sequence of size n in which A generates the substring of size j starting at i. When the triplet $(i, 1, A)$ is marked, we conclude there is a support for every value $a \in \text{dom}(X_i)$ such that $A \to a \in G$.

Algorithm 1: CYK-prop(G, $[X_1, \ldots, X_n]$)

```
1  for i = 1 to n do
2    V[i, 1] ← {A | A → a ∈ G, a ∈ dom(X_i)}
3  for j = 2 to n do
4    for i = 1 to n − j + 1 do
5      V[i, j] ← ∅
6      for k = 1 to j − 1 do
7        V[i, j] ← V[i, j] ∪ {A | A → BC ∈ G, B ∈ V[i, k], C ∈ V[i + k, j − k]}
8  if S ∉ V[1, n] then return "Unsatisfiable"
9  mark (1, n, S)
10 for j = n downto 2 do
11   for i = 1 to n − j + 1 do
12     for A → BC ∈ G such that (i, j, A) is marked do
13       for k = 1 to j − 1 do
14         if B ∈ V[i, k], C ∈ V[k + k, j − k] then
15           mark (i, k, B)
16           mark (i + k, j − k, C)
17 for i = 1 to n do
18   dom(X_i) ← {a ∈ dom(X_i) | A → a ∈ G, (i, 1, A) is marked}
19 return "Satisfiable"
```

Theorem 5 `CYK-prop` *enforces GAC on* $\textsc{Cfg}(\mathcal{G}, [X_1, \ldots, X_n])$ *in* $\Theta(|G|n^3)$ *time and* $\Theta(|G|n^2)$ *space.*

Proof: (Correctness) Follows quickly from the correctness of the CYK parser as we simply trace back every support that can generate a valid sequence.

(Complexity) Lines 1 to 7 run in $\Theta(|G|n^3)$ steps and require $\Theta(|G|n^2)$ memory since it is essentially the CYK-algorithm. Lines 9 to 19 clearly run in $\Theta(|G|n^3)$ steps and require $\Theta(|G|n^2)$ bits, one for each element in the sets contained in table V. $\diamond$

When the propagator is used within search, we can perform the dynamic programming incrementally. More specifically, we can restrict computation to those array elements $V[i,j]$ which can possibly have changed. For instance, if only the domain of X_k changes, then we only need to re-compute $V[i,j]$ when $k \in [i, i+j-1]$. There are exactly $k(n-k+1)$ such entries. The running time of the first phase can be brought downto $O(|G|k(n-k)n)$. When $k = 1$ or $k = n$, we obtain a complexity of $O(|G|n^2)$. When $k = \frac{n}{2}$, we remain with a complexity of $O(|G|n^3)$. The second phase can also be implemented incrementally. For each element $A \in V[i,j]$, we count how many times the triplet (i,j,A) has been marked. When a terminal A is removed from a cell $V[i,j]$, we check for each rule $A \to BC \in G$ and each $1 \leq k < j$ if we have $B \in V[i,k]$ and $C \in V[i+k, j-k]$. If so, we decrement by one the number of times the triplets (i,k,B) and $(i+k, j-k, C)$ have been marked. If a counter reaches 0, we can remove the corresponding element form the list and propagate the change. Again, the time needed for this operation is bounded by $O(|G|n^3)$ but will most likely perform better in practice than recomputing the whole table V.

8.2 Earley-style propagator

Our second propagator is based on the popular Earley chart parser which also uses dynamic programming to parse a context-free language. Whilst this propagator is more complex than the CYK-based propagator, it has several practical and theoretical advantages. First, the Earley parser is not restricted to grammars in Chomsky normal form. Second, it is often much more efficient than CYK as it parses strings top-down. For context-free grammars with few acceptable strings, it will therefore do little work. It also performs particularly well when the productions are left-recursive. Third, although the Earley parser runs in $O(n^3)$ time in the worst case like the CYK algorithm, due to its top-down nature, it takes just $O(n^2)$ time on any unambiguous context-free grammar, and is linear on a wide range of useful grammars like LR(0).

The propagator again uses dynamic programming to build up possible support. Productions are annotated with a "dot" indicating position of the parser. For example, $S \to N \bullet P$ represents the fact that: the production $S \to NP$ has been used in the parsing, we have already parsed N and so are now expecting P. Without loss of generality, we assume that there is an unique starting production of the form $S \to U$. We therefore begin with $S \to \bullet U$. A successful parsing is thus when we generate $S \to U \bullet$.

Algorithm 2 is the Earley chart parser augmented with the sets S that keep track of the supports for each value in the domains. We use a special data structure to implement these sets S. We first build the basic sets $\{X_i = v\}$ for every potential support $v \in \mathrm{dom}(X_i)$. Once a set is computed, its value is never changed. To compute the union of two sets $A \cup B$, we create a set C with a pointer on A and a pointer on B. This allows to represent the union of two sets in constant time. The data structure form a directed graph where the sets are the nodes and the pointers are the edges. To enumerate the content of a set S,

Algorithm 2: Earley-Prop(G, $[X_1, \ldots, X_n]$)

for $i = 0$ **to** n **do** $C[i] \leftarrow \emptyset$
queue $\leftarrow \{(s \rightarrow \bullet u, 0, \emptyset)\}$
for $i = 0$ **to** $n + 1$ **do**
  for $state \in C[i]$ **do** push(state,queue)
  while *queue is not empty* **do**
    $(r, j, S) \leftarrow pop(queue)$
    $add((r, j, S), C[i])$
    if $r = (u \rightarrow v\bullet)$ **then**
      foreach $(w \rightarrow \ldots \bullet u \ldots, k, T) \in C[j]$ **do**
        $add((w \rightarrow \ldots u \bullet \ldots, k, S \cup T), queue)$
    else if $i \leq n$ *and* $r = (u \rightarrow \ldots \bullet v \ldots)$ *and* $v \in dom(X_i)$ **then**
      $add((u \rightarrow \ldots v\bullet \ldots, j, S \cup \{X_i = v\}), C[i+1])$
    else if $r = (u \rightarrow \ldots \bullet v \ldots)$ *and* $non\_terminal(v, G)$ **then**
      foreach $v \rightarrow w \in G$ *such that* $(v \rightarrow \bullet w, i, \emptyset) \notin C[i] \cup queue$ **do**
        $push((v \rightarrow \bullet w, i, \emptyset), queue)$
  if $C[i] = \emptyset$ **then**
    return *"Unsatisfiable"*
if $(s \rightarrow u\bullet, 0, S) \in C[n]$ **then**
  for $i = 1$ **to** n **do**
    $\mathrm{dom}(X_i) = \{a \mid X_i = a \in S\}$
else
  return *"Unsatisfiable"*

one can do depth-first search. The basic sets $\{X_i = v\}$ that are visited in the search are the elements of S.

Theorem 6 `Earley-prop` *enforces GAC on* $\textsc{Cfg}(\mathcal{G}, [X_1, \ldots, X_n])$ *in* $O(|G|n^3)$ *time for an arbitrary context-free grammar, and in* $O(|G|n^3)$ *space.*

Proof: (Correctness) Follows quickly from the correctness of the Earley parser, as we have basically just augmented the parsing information with value supports.

(Complexity) The total time complexity is identical to the Earley chart parser except for the additional work involved in calling the `add` procedure within the innermost loop to compute support information. Using the data structure ex-

Algorithm 3: add$((a, b, c), q)$

if $\exists\, (a, b, d) \in q$ **then**
  $q \leftarrow replace((a, b, d), (a, b, c \cup d), q)$
else
  $push((a, b, c), q)$

plained above, the `add` procedure takes $O(1)$ time. Lines 3 to 20 involve a depth-first-search in the data structure used to store the sets. Since this is been constructed in $O(|G|n^3)$ time, it has no more than $O(|G|n^3)$ sets and pointers. The depth-first-search computing S in line 20 therefore requires $O(|G|n^3)$ steps. $\diamond$

We might consider moving up the Chomsky hierarchy to context-sensitive grammars. However, this would make constraint propagation highly intractable. A context-sensitive grammar is one in which every production is of the form $u \rightarrow v$ where $|u| \leq |v|$. A global constraint specified by a context-sensitive grammar is outside of what we usually consider a global constraint. Determining if a string belongs to a context-sensitive grammar is PSPACE-complete, whilst we usually assume that it takes polynomial time to check if an assignment satisfies a constraint [12]. In fact, since it is undecidable to determine if a context-sensitive grammar is empty, detecting domain wipeout (and thus enforcing GAC) on a context-sensitive grammar constraint is undecidable.

9 Experimental results

We implemented the ternary encoding of the Regular constraint and compared it with Pesant's propagator [7]. Pesant presents two propagators, we implemented the one that keeps track of all supports for a given character $c \in \text{dom}(X_i)$. As in [7], we generated random automata with $|Q|$ states and an alphabet of size $|\Sigma|$. We selected 30% of all possible tuples $(c, q_i) \in \Sigma \times Q$ and randomly chose a state $q_j \in Q$ to form the transition $T(c, q_i) = q_j$. We obtained the set of final states F by randomly selecting 50% of the states in Q. Following Pesant, we used a random variable ordering and random value selection. All experiments were run on a 900 Mhz Pentium and times were averaged over 30 runs. Table 1 shows the results. The ternary encoding of the Regular constraint dominates on all but one instance. We believe that the propagator for the Table constraint provided in ILog Solver is highly optimized and contributed to the performance offered by our ternary encoding.

n	$\|\Sigma\|$	$\|Q\|$	Pesant's Regular	Ternary encoding of Regular	n	$\|\Sigma\|$	$\|Q\|$	Pesant's Regular	Ternary encoding of Regular
25	5	10	0.0032	**0.0031**	50	5	10	**0.0047**	0.0051
		20	0.0029	**0.0025**			20	0.0047	**0.0037**
		40	0.0052	**0.0046**			40	0.0101	**0.0086**
		80	0.0079	**0.0041**			80	0.0168	**0.0087**
25	10	10	0.0053	**0.0038**	50	10	10	0.0105	**0.0071**
		20	0.0099	**0.0063**			20	0.0207	**0.0129**
		40	0.0165	**0.0087**			40	0.0359	**0.0185**
		80	0.0284	**0.0136**			80	0.0631	**0.0301**
25	20	10	0.0113	**0.0057**	50	20	10	0.0232	**0.0119**
		20	0.0195	**0.0083**			20	0.0396	**0.0177**
		40	0.0399	**0.0140**			40	0.0814	**0.0289**
		80	0.0812	**0.0226**			80	0.1655	**0.0457**

Table 1. Time in seconds to find a sequence satisfying a randomly generated automaton either using Pesant's propagator for the Regular constraint or a ternary encoding using the Table constraint

We also ran experiments on a model for the Mystery Shopper problem due to Helmut Simonis that appears in CSPLib (prob004). This model contains a large number of AMONG constraints. We encoded the AMONG constraint either as a REGULAR constraint or by means of a Boolean decomposition. This decomposes $\text{AMONG}([X_1, \ldots, X_n], s, N)$ into $B_i = 1$ iff $X_i \in s$ for all i, and $\sum_i B_i = N$. For the REGULAR constraint, we either used Pesant's propagator, or our ternary encoding. For the ternary encoding, we either implemented the transition constraint, $C(X_{i+1}, Q_i, Q_{i+1})$ using ILog's TABLE constraint or by the simple implications: $X_i \in s \rightarrow Q_{i+1} = Q_i + 1$, $Q_{i+1} = Q_i + 1 \rightarrow X_i \in s$, $X_i \notin s \rightarrow Q_{i+1} = Q_i$, $Q_{i+1} = Q_i \rightarrow X_i \notin s$.

	AMONG using Booleans			REGULAR using simple implications			REGULAR using TABLE constraints			REGULAR using Pesant's propagator		
Size	#fails	cpu time	#solved	#fails	cpu time	#solved	#fails	cpu time	#solved	#fails	cpu time	#solved
10	6	0.00400	9/10	6	**0.00244**	9/10	6	0.00755	9/10	6	0.01022	9/10
15	8342	0.64075	32/52	8342	**0.42647**	32/52	8342	1.15954	32/52	8342	1.19897	32/52
20	12960	2.21400	21/35	12960	**1.74521**	21/35	12960	3.40063	21/35	12960	5.63347	21/35
25	6186	0.67040	4/20	6186	**0.52567**	**5/20**	6186	0.87862	4/20	6186	1.41279	4/20
30	1438	0.32695	3/10	1438	**0.28229**	3/10	1438	0.47626	3/10	1438	0.72189	3/10
35	6297	1.66825	**21/56**	6297	**1.49327**	**21/56**	6297	2.36849	20/56	6297	3.73623	20/56

Table 2. Mytery Shopper problem, #fails and cpu time are only averaged on instances solved by *all* methods

Results are given in Table 2. All instances solved in the experiments use a time limit of 5 minutes. All methods acheive GAC on the AMONG constraint, so the search trees are identical and it is only the efficiency of the propagator which differ. Our ternary encoding using simple implications dominates on all problem instances. This example demonstrates the benefits of encoding the ternary transition constraints using simple but efficient builtin propagators.

Finally, we implemented the two propagators for the CFG constraint. To compare them, we considered the problem of providing an editor that only accepts grammatical sentences. Such editors have been proposed for entering natural language specifications of ontologies, web services and software [13]. Within such an editor, we have to take a partial string, and enforce GAC on a CFG constraint, thereby limiting the next word to those that give sentences within the language. We used a simple English grammar with 36 rules available online [14] that generates sentences like "The robber knew Vincent shot Marsellus". We gave the CFG constraint 20 randomly generated sentences containing up to 30 words. We then added each word in sequence and enforced GAC. As the grammar contains 25 words, this is the initial size of each variable domain. Results are given in Table 3. The Earley propagator is faster when the number of instantiated variables increases.

n	k	Cfg using CYK-prop	Cfg using Earley-prop	n	k	Cfg using CYK-prop	Cfg using Earley-prop
20	0	**0.0095**	0.0413	30	0	**0.0286**	0.1800
20	4	**0.0064**	0.0188	30	6	**0.0202**	0.0720
20	8	**0.0046**	0.0066	30	12	**0.0145**	0.0245
20	12	0.0035	**0.0014**	30	18	0.0114	**0.0053**
20	16	0.0028	**0.0000**	30	24	0.0095	**0.0001**
20	20	0.0026	**0.0000**	30	30	0.0092	**0.0000**

Table 3. Time in seconds to enforce GAC on a Cfg constraint of n symbols in which first k are fixed. When $n = k$, the problem degenerates to parsing the sentence.

10 Related work

Carlsson and Beldiceanu derived a propagation algorithm for a chain of lexicographical ordering constraints based on a deterministic finite automaton [15]. For the Regular constraint, a propagation algorithm based on dynamic programming that enforces GAC was given in [7]. Coincidently Beldiceanu, Carlsson and Petit proposed specifying global constraints by means of deterministic finite automaton augmented with counters [16]. Propagators for such automaton are constructed automatically from the specification of the automaton by constructing a conjunction of signature and transition constraints. The ternary encodings used here are similar to those proposed in [16]. However, there are a number of differences. One is that we permit non-deterministic transitions. As argued before, non-determinism can reduce the size of the automaton significantly. In addition, the counters used by Beldiceanu, Carlsson and Petit introduce complexity. For example, they need to achieve pairwise consistency to guarantee global consistency. Pesant encodes a cyclic Stretch constraint into a Regular constraint in which the initial variables of the sequence are repeated at the end, and then dummy unconstrained variables are placed at the start and end [7]. Hellsten, Pesant and van Beek propose a domain consistency algorithm for the Stretch constraint based on dynamic programming similar to that for the Regular constraint [11]. They also showed how to extend it to deal with cyclic Stretch constraints. Finally, Golden and Pang propose the use of string variables which are specificed using regular expressions or automata and show how to enforce GAC on matching, containment, cardinality and other constraints [17].

11 Conclusions

We have studied a range of grammar constraints. These are global constraints over a sequence of variables which restrict the values assigned to be a string within a given language. Such constraints are useful in a wide range of scheduling, rostering and sequencing problems. For regular languages, we gave a simple encoding into ternary constraints that can be used to enforce GAC in linear time. Our experiments demonstrated that such encodings are efficient and effective in practice. This ternary encoding is therefore an easy means to incorporate this global constraint into constraint toolkits. We also considered a number of extensions including regular languages specified by non-deterministic finite automata, and soft and cyclic versions of the global constraint. For context-free languages, we gave two propagators which enforce GAC based on the CYK and Earley parsers. There are many directions for future work. One promising direction is to learn grammar constraints from examples. We can leverage on results and algorithms from grammar induction. For example, it is not possible to learn a REGULAR constraint from just positive examples.

References

1. Lhomme, O.: An efficient filtering algorithm for disjunction of constraints. In Proc. of 9th Int. Conf. on Principles and Practice of Constraint Programming (CP2003), (2003) 904–908
2. Lhomme, O.: Arc-consistency filtering algorithms for logical combinations of constraints. In Proc. of CP AI OR, (2004) 209–224
3. Bacchus, F., Walsh, T.: Propagating logical combinations of constraints. In Proc. of 19th IJCAI, (2005) 35–40
4. Beldiceanu, N.: Global constraints as graph properties on a structured network of elementary constraints of the same type. In Proc. of 6th Int. Conf. on Principles and Practice of Constraint Programming (CP2000), (2000) 52–66
5. Beldiceanu, N., Carlsson, M., Rampon, J.X., Truchet, C.: Graph invariants as necessary conditions for global constraints. In Proc. of 11th Int. Conf. on Principles and Practice of Constraint Programming (CP2005), (2005) 92–106
6. Beldiceanu, N., Petit, T., Rochart, G.: Bounds of graph characteristics. In Proc. of 11th Int. Conf. on Principles and Practice of Constraint Programming (CP2005), (2005) 742–746
7. Pesant, G.: A regular language membership constraint for finite sequences of variables. In Proc. of 10th Int. Conf. on Principles and Practice of Constraint Programming (CP2004), (2004) 482–295
8. Law, Y., Lee, J.: Global constraints for integer and set value precedepnce. In Proc. of 10th Int. Conf. on Principles and Practice of Constraint Programming (CP2004), (2004) 362–376
9. Janssen, M., Hentenryck, P.V., Deville, Y.: Constraint satisfaction approach to parametric differential equations. In Proc. of the 17th IJCAI, (2001)
10. van Hoeve, W.J., Pesant, G., Rousseau, L.M.: On global warming : Flow-based soft global constaints. Journal of Heuristics (2006) To appear.
11. Hellsten, L., Pesant, G., van Beek, P.: A domain consistency algorithm for the stratch constraint. In Proc. of 10th Int. Conf. on Principles and Practice of Constraint Programming (CP2004), (2004) 290–304
12. Bessiere, C., Hebrard, E., Hnich, B., Walsh, T.: The complexity of global constraints. In Proc. of the 19th AAAI (2004)
13. Schwitter, R.: A controlled natural language layer for the semantic web. In Advances in Artificial Intelligence (AI'2005), (2005) 425–434 LNCS 3809.
14. Blackburn, P., Striegntiz, K.: Natural language processing techniques in Prolog (2005) http://www.coli.uni-saarland.de/ kris/nlp-with-prolog/html/index.html.
15. Carlsson, M., Beldiceanu, N.: Arc-consistency for a chain of lexicographic ordering constraints. Tech report T2002-18, Swedish Institute of Computer Science (2002)
16. Beldiceanu, N., Carlsson, M., Petit, T.: Deriving filtering algorithms from constraint checkers. In Proc. of 10th Int. Conf. on Principles and Practice of Constraint Programming (CP2004), (2004) 107–122
17. Golden, K., Pang, W.: Constraint reasoning over strings. In Proc. of 9th Int. Conf. on Principles and Practice of Constraint Programming (CP2003), (2003) 377–391