Constraints (2010) 15:117–144
DOI 10.1007/s10601-009-9073-4

Grammar constraints

Serdar Kadioglu · Meinolf Sellmann

Published online: 10 July 2009

Abstract With the introduction of the Regular Membership Constraint, a new line of research has opened where constraints are based on formal languages. This paper is taking the next step, namely to investigate constraints based on grammars higher up in the Chomsky hierarchy. We devise a time- and space-efficient incremental arc-consistency algorithm for context-free grammars, investigate when logic combinations of grammar constraints are tractable, show how to exploit non-constant size grammars and reorderings of languages, and study where the boundaries run between regular, context-free, and context-sensitive grammar filtering.

Keywords Constraint filtering · Global constraints · Grammar constraints

1 Introduction

With the introduction of the regular language membership constraint [1, 13, 14], a new field of study for filtering algorithms has opened. Given the great expressiveness of formal grammars and their (at least for someone with a background in computer science) intuitive usage, grammar constraints are extremely attractive modeling entities that subsume many existing definitions of specialized global constraints. Moreover, Pesant's implementation [19] of the regular grammar constraint has shown that this type of filtering can also be performed incrementally and generally so

S. Kadioglu · M. Sellmann (✉)
Department of Computer Science, Brown University,
115 Waterman St, Providence, RI 02912, USA
e-mail: sello@cs.brown.edu

S. Kadioglu
e-mail: serdark@cs.brown.edu

efficiently that it even rivals custom filtering algorithms for special regular grammar constraints like Stretch and Pattern [4, 14].

In this paper, we theoretically investigate filtering problems that arise from grammar constraints. We answer questions like: Can we efficiently filter context-free grammar constraints? How can we achieve arc-consistency for conjunctions of regular grammar constraints? Given that we can allow non-constant grammars and reordered languages for the purposes of constraint filtering, what languages are suited for filtering based on regular and context-free grammar constraints? Are there languages that are suited for context-free, but not for regular grammar filtering?

This paper is largely based on our [21] in which we introduced context-free grammar constraints and [10] in which we presented a space-efficient incremental filtering algorithm. After recalling some essential basic concepts from the theory of formal languages in the next section, we devise an arc-consistency algorithm that propagates context-free grammar constraints in Section 3. In the following Section 4, we modify our basic algorithm to make it both more space- and time-efficient. In Section 5, we give a filtering algorithm for grammar constraints with costs. In Section 6, we study how logic combinations of grammar constraints can be propagated efficiently. Finally, we investigate non-constant size grammars and reorderings of languages in Section 7.

2 Basic concepts

We start our work by reviewing some well-known definitions from the theory of formal languages. For a full introduction, we refer the interested reader to [9]. All proofs that are omitted in this paper can also be found there.

Definition 1 (Alphabet and Words) Given sets Z, Z_1, and Z_2, with $Z_1 Z_2$ we denote the set of all *sequences* or *strings* $z = z_1 z_2$ with $z_1 \in Z_1$ and $z_2 \in Z_2$, and we call $Z_1 Z_2$ the *concatenation* of Z_1 and Z_2. Then, for all $n \in \mathbb{N}$ we denote with Z^n the set of all sequences $z = z_1 z_2 \dots z_n$ with $z_i \in Z$ for all $1 \leq i \leq n$. We call z a *word* of length n, and Z is called an *alphabet* or set of *letters*. The empty word has length 0 and is denoted by ε. It is the only member of Z^0. We denote the set of all words over the alphabet Z by $Z^* := \bigcup_{n \in \mathbb{N}} Z^n$. In case that we wish to exclude the empty word, we write $Z^+ := \bigcup_{n \geq 1} Z^n$.

Definition 2 (Grammar) A *grammar* is a tuple $G = (\Sigma, N, P, S_0)$ where Σ is the alphabet, N is a finite set of *non-terminals*, $P \subseteq (N \cup \Sigma)^* N (N \cup \Sigma)^* \times (N \cup \Sigma)^*$ is the set of *productions*, and $S_0 \in N$ is the start non-terminal. We will always assume that $N \cap \Sigma = \emptyset$.

Remark 1 We will use the following convention: Capital letters A, B, C, D, and E denote non-terminals, lower case letters a, b, c, d, and e denote letters in Σ, Y and Z denote symbols that can either be letters or non-terminals, u, v, w, x, y, and z denote strings of letters, and α, β, and γ denote strings of letters and non-terminals. Moreover, productions (α, β) in P can also be written as $\alpha \to \beta$.

Definition 3 (Derivation and Language)

- Given a grammar $G = (\Sigma, N, P, S_0)$ we write $\alpha\beta_1\gamma \underset{G}{\Rightarrow} \alpha\beta_2\gamma$ if and only if there exists a production $(\beta_1 \to \beta_2) \in P$. We write $\alpha_1 \overset{*}{\underset{G}{\Rightarrow}} \alpha_m$ if and only if there exists a sequence of strings $\alpha_2, \ldots, \alpha_{m-1}$ such that $\alpha_i \underset{G}{\Rightarrow} \alpha_{i+1}$ for all $1 \leq i < m$. Then, we say that α_m can be *derived* from α_1.
- The *language given by* G is $L_G := \left\{w \in \Sigma^* \mid S_0 \overset{*}{\underset{G}{\Rightarrow}} w\right\}$.

Definition 2 gives a very general form of grammars which is known to be Turing machine equivalent. Consequently, reasoning about languages given by general grammars is infeasible. For example, the word problem for grammars as defined above is undecidable.

Definition 4 (Word Problem) Given a grammar $G = (\Sigma, N, P, S_0)$ and a word $w \in \Sigma^*$, the *word problem* consists in answering the question whether $w \in L_G$.

Therefore, in the theory of formal languages, more restricted forms of grammars have been defined. Chomsky introduced a hierarchy of decreasingly complex sets of languages [5]. In this hierarchy, the grammars given in Definition 2 are called Type-0 grammars. In the following, we define the Chomsky hierarchy of formal languages.

Definition 5 (Type 1–Type 3 Grammars)

- Given a grammar $G = (\Sigma, N, P, S_0)$ such that for all productions $(\alpha \to \beta) \in P$ we have that β is at least as long as α, then we say that the grammar G and the language L_G are *context-sensitive*. In Chomsky's hierarchy, these grammars are known as Type-1 grammars.
- Given a grammar $G = (\Sigma, N, P, S_0)$ such that $P \subseteq N \times (N \cup \Sigma)^*$, we say that the grammar G and the language L_G are *context-free*. In Chomsky's hierarchy, these grammars are known as Type-2 grammars.
- Given a grammar $G = (\Sigma, N, P, S_0)$ such that $P \subseteq N \times (\Sigma^* N \cup \Sigma^*)$, we say that G and the language L_G are *right-linear* or *regular*. In Chomsky's hierarchy, these grammars are known as Type-3 grammars.

Remark 2 The word problem becomes easier as the grammars become more and more restricted: For context-sensitive grammars, the problem is already decidable, but unfortunately PSPACE-complete. For context-free languages, the word problem can be answered in polynomial time. For Type-3 languages, the word problem can even be decided in time linear in the length of the given word.

For all grammars mentioned above there exists an equivalent definition based on some sort of automaton that accepts the respective language. As mentioned earlier, for Type-0 grammars, that automaton is the Turing machine. For context-sensitive languages it is a Turing machine with a linearly space-bounded tape. For context-free languages, it is the so-called push-down automaton (in essence a Turing machine with a stack rather than a tape). And for right-linear languages, it is the finite automaton (which can be viewed as a Turing machine with only one read-only

input tape on which it cannot move backwards). Depending on what one tries to prove about a certain class of languages, it is convenient to be able to switch back and forth between different representations (i.e. grammars or automata). In this work, when reasoning about context-free languages, it will be most convenient to use the grammar representation. For right-linear languages, however, it is often more convenient to use the representation based on finite automata:

Definition 6 (Finite Automaton) Given a finite set Σ, a *finite automaton* A is defined as a tuple $A = (Q, \Sigma, \delta, q_0, F)$, where Q is a set of *states*, Σ denotes the *alphabet* of our language, $\delta \subseteq Q \times \Sigma \times Q$ defines the *transition function*, q_0 is the *start state*, and F is the set of *final states*. A finite automaton is called *deterministic* if and only if $(q, a, p_1), (q, a, p_2) \in \delta$ implies that $p_1 = p_2$.

Definition 7 (Accepted Language) The language defined by a finite automaton A is the set $L_A := \{w = (w_1, \ldots w_n) \in \Sigma^* \mid \exists\, (p_0, \ldots,\ p_n) \in Q^n\ \forall\ 1 \leq i \leq n : (p_{i-1}, w_i, p_i) \in \delta$ and $p_0 = q_0, p_n \in F\}$. L_A is called a *regular language*.

Lemma 1 *For every right-linear grammar G there exists a finite automaton A such that $L_A = L_G$, and vice versa.*

3 Context-free grammar constraints

Within constraint programming it would be convenient to use formal languages to describe certain features that we would like our solutions to exhibit. It is worth noting here that any constraint and conjunction of constraints really defines a formal language by itself when we view the instantiations of variables $X_1, \ldots, X_n$ with domains $D_1, \ldots, D_n$ as forming a word in $D_1 D_2 \ldots D_n$. Conversely, if we want a solution to belong to a certain formal language in this view, then we need appropriate constraints and constraint filtering algorithms that will allow us to express and solve such constraint programs efficiently. We formalize the idea by defining grammar constraints.

Definition 8 (Grammar Constraint) For a given grammar $G = (\Sigma, N, P, S_0)$ and variables $X_1, \ldots, X_n$ with domains $D_1 := D(X_1), \ldots, D_n := D(X_n) \subseteq \Sigma$, we say that $Grammar_G(X_1, \ldots, X_n)$ is true for an instantiation $X_1 \leftarrow w_1, \ldots, X_n \leftarrow w_n$ if and only if it holds that $w = w_1 \ldots w_n \in L_G \cap D_1 D_2 \ldots D_n$.

Pesant pioneered the idea to exploit formal grammars for constraint programming by considering regular languages [13, 14]. Based on the review of our knowledge of formal languages in the previous section, we can now ask whether we can also develop efficient filtering algorithms for grammar constraints of higher-orders. Clearly, for Type-0 grammars, this is not possible, since the word problem is already undecidable. For context-sensitive languages, the word problem is PSPACE complete, which means that even checking the corresponding grammar constraint is computationally intractable.

However, for context-free languages deciding whether a given word belongs to the language can be done in polynomial time. Since their recent introduction,

context-free grammar constraints have already been used successfully to model real-world problems. For instance, in [16] a shift-scheduling problem was modeled and solved efficiently by means of grammar constraints. Context-free grammars come in particularly handy when we need to look for a recursive sequence of nested objects. Consider for instance the puzzle of forming a mathematical term based on two occurrences of the numbers 3 and 8, operators $+, -, *, /$, and brackets $(,)$, such that the term evaluates to 24. The generalized problem is NP-hard, but when formulating the problem as a constraint program, with the help of a context-free grammar constraint we can easily express the syntactic correctness of the term formed. Or, again closer to the real-world, consider the task of organizing a group of workers into a number of teams of unspecified size, each team with one team leader and one project manager who is the head of all team leaders. This organizational structure can be captured easily by a combination of an all-different and a context-free grammar constraint. Therefore, in this section we will develop an algorithm that propagates context-free grammar constraints.

3.1 Parsing context-free grammars

One of the most famous algorithms for parsing context-free grammars is the algorithm by Cocke, Younger, and Kasami (CYK). It takes as input a word $w \in \Sigma^n$ and a context-free grammar $G = (\Sigma, N, P, S_0)$ in some special form and decides in time $O(n^3|G|)$ whether it holds that $w \in L_G$. The algorithm is based on the dynamic programming principle. In order to keep the recursion equation under control, the algorithm needs to assume that all productions are length-bounded on the right-hand side.

Definition 9 (Chomsky Normal Form) A Type-2 or context-free grammar $G = (\Sigma, N, P, S_0)$ is said to be in *Chomsky Normal Form* if and only if for all productions $(A \to \alpha) \in P$ we have that $\alpha \in \Sigma^1 \cup N^2$. Without loss of generality, we will then assume that each literal $a \in \Sigma$ is associated with exactly one unique non-literal $A_a \in N$ such that $(B \to a) \in P$ implies that $B = A_a$ and $(A_a \to b) \in P$ implies that $a = b$.

Lemma 2 *Every context free grammar G such that $\varepsilon \notin L_G$ can be transformed into a grammar H such that $L_G = L_H$ and H is in Chomsky Normal Form.*

The proof of this lemma is given in [9]. It is important to note that the proof is constructive but that the resulting grammar H may be exponential in size of G, which is really due to the necessity to remove all productions $A \to \varepsilon$. When we view the grammar size as constant (i.e. if the size of the grammar is independent of the word-length as it is commonly assumed in the theory of formal languages), then this is not an issue. As a matter of fact, in most references one will simply read that CYK could solve the word problem for any context-free language in cubic time. For now, let us assume that indeed all grammars given can be treated as having constant-size, and that our asymptotic analysis only takes into account the increasing word lengths. We will come back to this point later in Section 6 when we discuss logic combinations of grammar constraints, and in Section 7 when we discuss the possibility of non-constant grammars and reorderings.

Now, given a word $w \in \Sigma^n$, let us denote the sub-sequence of letters starting at position i with length j (that is, $w_i w_{i+1} \dots w_{i+j-1}$) by w_{ij}. Based on a grammar $G = (\Sigma, N, P, S_0)$ in Chomsky Normal Form, CYK determines iteratively the set of all non-terminals from where we can derive w_{ij}, i.e. $S_{ij} := \left\{ A \in N \mid A \stackrel{*}{\underset{G}{\Rightarrow}} w_{ij} \right\}$ for all $1 \leq i \leq n$ and $1 \leq j \leq n - i$. It is easy to initialize the sets S_{i1} just based on w_i and all productions $(A \rightarrow w_i) \in P$. Then, for j from 2 to n and i from 1 to $n - j + 1$, we have that

$$S_{ij} = \bigcup_{k=1}^{j-1} \left\{ A \mid (A \rightarrow BC) \in P \text{ with } B \in S_{ik} \text{ and } C \in S_{i+k,j-k} \right\}. \tag{1}$$

Then, $w \in L_G$ if and only if $S_0 \in S_{1n}$. From the recursion equation it is simple to derive that CYK can be implemented to run in time $O(n^3|G|) = O(n^3)$ when we treat the size of the grammar as a constant.

3.2 Example

Assume we are given the following context-free, normal-form grammar $G = (\{], [\}, \{A, B, C, S_0\}, \{S_0 \rightarrow AC, S_0 \rightarrow S_0 S_0, S_0 \rightarrow BC, B \rightarrow AS_0, A \rightarrow [\ , C \rightarrow]\ \}, S_0)$ that gives the language L_G of all correctly bracketed expressions (like, for example, "[[][]]" or "[][[]]"). Given the word "[][[]]", CYK first sets $S_{11} = S_{31} = S_{41} = \{A\}$, and $S_{21} = S_{51} = S_{61} = \{C\}$. Then it determines the non-terminals from which we can derive sub-sequences of length 2: $S_{12} = S_{42} = \{S_0\}$ and $S_{22} = S_{32} = S_{52} = \emptyset$. The only other non-empty sets that CYK finds in iterations regarding longer sub-sequences are $S_{34} = \{S_0\}$ and $S_{16} = \{S_0\}$. Consequently, since $S_0 \in S_{16}$, CYK decides (correctly) that [][[]] $\in L_G$.

3.3 Context-free grammar filtering

We denote a given grammar constraint $Grammar_G(X_1, \dots, X_n)$ over a context-free grammar G in Chomsky Normal Form by $CFGC_G(X_1, \dots, X_n)$. Obviously, we can use CYK to determine whether $CFGC_G(X_1, \dots, X_n)$ is satisfied for a full instantiation of the variables, i.e. we could use the parser for generate-and-test purposes. In the following, we show how we can augment CYK to a filtering algorithm that achieves generalized arc-consistency for $CFGC$. An alternative filtering algorithm based on the Earley parser is presented in [17].

First, we observe that we can check the satisfiability of the constraint by making just a very minor adjustment to CYK. Given the domains of the variables, we can decide whether there exists a word $w \in D_1 \dots D_n$ such that $w \in L_G$ simply by adding all non-terminals A to S_{i1} for which there exists a production $(A \rightarrow v) \in P$ with $v \in D_i$. From the correctness of CYK it follows trivially that the constraint is satisfiable if and only if $S_0 \in S_{1n}$. The runtime of this algorithm is the same as that for CYK.

As usual, whenever we have a polynomial-time algorithm that can decide the satisfiability of a constraint, we know already that achieving arc-consistency is also computationally tractable. A brute force approach could simply probe values by setting $D_i := \{v\}$, for every $1 \leq i \leq n$ and every $v \in D_i$, and checking whether the constraint is still satisfiable or not. This method would result in a runtime in $O(n^4 D|G|)$, where $D \leq |\Sigma|$ is the size of the largest domain D_i.

We will now show that we can achieve a much improved filtering time. The core idea is once more to exploit Trick's method of filtering dynamic programs [22]. Roughly speaking, when applied to our CYK-constraint checker, Trick's method simply reverses the recursion process after it has assured that the constraint is satisfiable so as to see which non-terminals in the sets S_{i1} can actually be used in the derivation of any word $w \in L_G \cap (D_1 \dots D_n)$. The methodology is formalized in Algorithm 1.

Algorithm 1 CFGC Filtering Algorithm

1. We run the dynamic program based on recursion equation (1) with initial sets $S_{i1} := \{A \mid (A \to v) \in P, v \in D_i\}$.
2. We define the directed graph $Q = (V, E)$ with node set $V := \{v_{ijA} \mid A \in S_{ij}\}$ and arc set $E := E_1 \cup E_2$ with $E_1 := \{(v_{ijA}, v_{ikB}) \mid \exists\, C \in S_{i+k,j-k} : (A \to BC) \in P\}$ and $E_2 := \{(v_{ijA}, v_{i+k,j-k,C}) \mid \exists\, B \in S_{ik} : (A \to BC) \in P\}$ (see Fig. 1).
3. Now, we remove all nodes and arcs from Q that cannot be reached from v_{1nS_0} and denote the resulting graph by $Q' := (V', E')$.
4. We define $S'_{ij} := \{A \mid v_{ijA} \in V'\} \subseteq S_{ij}$, and set $D'_i := \{v \mid \exists\, A \in S'_{i1} : (A \to v) \in P\}$.

Lemma 3 *In Algorithm 1:*

1. *It holds that $A \in S_{ij}$ if and only if there exists a word $w_i \dots w_{i+j-1} \in D_i \dots D_{i+j-1}$ such that $A \overset{*}{\underset{G}{\Rightarrow}} w_i \dots w_{i+j-1}$.*
2. *It holds that $B \in S'_{ik}$ if and only if there exists a word $w \in L_G \cap (D_1 \dots D_n)$ such that $S_0 \overset{*}{\underset{G}{\Rightarrow}} w_1 \dots w_{i-1}\ B\ w_{i+k} \dots w_n$.*

Proof

1. We induce over j. For $j = 1$, the claim holds by definition of S_{i1}. Now assume $j > 1$ and that the claim is true for all S_{ik} with $1 \le k < j$. Now, by definition of S_{ij}, $A \in S_{ij}$ if and only if there exists a $1 \le k < j$ and a production $(A \to BC) \in P$ such that $B \in S_{ik}$ and $C \in S_{i+k,j-k}$. Thus, $A \in S_{ij}$ if and only if there exist $w_{ik} \in D_i \dots D_{i+k-1}$ and $w_{i+k,j-k} \in D_{i+k} \dots D_{i+j-1}$ such that $A \overset{*}{\underset{G}{\Rightarrow}} w_{ik} w_{i+k,j-k}$.
2. We induce over k, starting with $k = n$ and decreasing to $k = 1$. For $k = n$, $S'_{1k} = S'_{1n} \subseteq \{S_0\}$, and it is trivially true that $S_0 \overset{*}{\underset{G}{\Rightarrow}} S_0$. Now let us assume the claim holds for all S'_{ij} with $k < j \le n$. Choose any $B \in S'_{ik}$. According to the definition of S'_{ik} there exists a path from v_{1nS_0} to v_{ikB}. Let $(v_{ijA}, v_{ikB}) \in E_1$ be the last arc on any one such path (the case when the last arc is in E_2 follows analogously). By the definition of E_1 there exists a production $(A \to BC) \in P$ with $C \in S_{i+k,j-k}$. By induction hypothesis, we know that there exists a word $w \in L_G \cap (D_1 \dots D_n)$ such that $S_0 \overset{*}{\underset{G}{\Rightarrow}} w_1 \dots w_{i-1}\ A\ w_{i+j} \dots w_n$. Thus, $S_0 \overset{*}{\underset{G}{\Rightarrow}} w_1 \dots w_{i-1}\ BC\ w_{i+j} \dots w_n$. And therefore, with the readily proven fact (1) and $C \in S_{i+k,j-k}$, there exists a word $w_{i+k} \dots w_{i+j-1} \in D_{i+k} \dots D_{i+j-1}$ such that $S_0 \overset{*}{\underset{G}{\Rightarrow}} w_1 \dots w_{i-1}\ B\ w_{i+k} \dots w_{i+j-1}\ w_{i+j} \dots w_n$. Since we can also apply (1) to non-terminal B, we have proven the claim. □

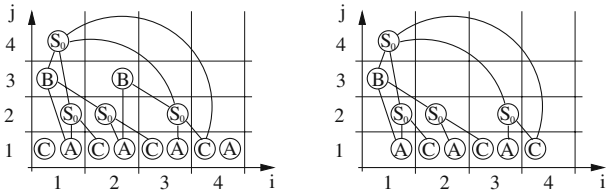

Fig. 1 Context-free filtering: a rectangle with coordinates (i, j) contains nodes v_{ijA} for each non-terminal A in the set S_{ij}. All arcs are considered to be directed from top to bottom. The *left picture* shows the situation after step (2). S_0 is in S_{14}, therefore the constraint is satisfiable. The *right picture* illustrates the shrunken graph with sets S'_{ij} after all parts have been removed that cannot be reached from node v_{14S_0}. We see that the value ']' will be removed from D_1 and '[' from D_4

Theorem 1 *Algorithm 1 achieves generalized arc-consistency for the CFGC.*

Proof We show that $v \notin D'_i$ if and only if for all words $w = w_1 \dots w_n \in L_G \cap (D_1 \dots D_n)$ it holds that $v \neq w_i$.

$\Rightarrow$ (Soundness) Let $v \notin D'_i$ and $w = w_1 \dots w_n \in L_G \cap (D_1 \dots D_n)$. Due to the assumption that $w \in L_G$ there must exist a derivation $S_0 \overset{*}{\underset{G}{\Rightarrow}} w_1 \dots w_{i-1}\ A\ w_{i+1} \dots w_n \overset{}{\underset{G}{\Rightarrow}} w_1 \dots w_{i-1} w_i w_{i+1} \dots w_n$ for some $A \in N$ with $(A \to w_i) \in P$. According to Lemma 3, $A \in S'_{i1}$, and thus $w_i \in D'_i$, which implies $v \neq w_i$ as $v \notin D'_i$.

$\Leftarrow$ (Completeness) Now let $v \in D'_i \subseteq D_i$. According to the definition of D'_i, there exists some $A \in S'_{i1}$ with $(A \to v) \in P$. With Lemma 3 we know that then there exists a word $w \in L_G \cap (D_1 \dots D_n)$ such that $S_0 \overset{*}{\underset{G}{\Rightarrow}} w_1 \dots w_{i-1}\ A\ w_{i+1} \dots w_n$. Thus, it holds that $S_0 \overset{*}{\underset{G}{\Rightarrow}} w_1 \dots w_{i-1}\ v\ w_{i+1} \dots w_n \in L_G \cap (D_1 \dots D_n)$. □

3.4 Example

Assume we are given the context-free grammar from Section 3.2 again. In Fig. 1, we illustrate how Algorithm 1 works on the problem: First, we work bottom-up, adding non-terminals to the sets S_{ij} if they allow to generate a word in $D_i \times \dots \times D_{i+j-1}$. Then, in the second step, we work top down and remove all non-terminals that cannot be reached from $S_0 \in S_{1n}$. In Fig. 2, we show how this latter step can affect the lowest level, where the removal of non-terminals results in the pruning of the domains of the variables.

3.5 Runtime analysis

We now have a filtering algorithm that achieves generalized arc-consistency for context-free grammar constraints. Since the computational effort is dominated by carrying out the recursion equation, Algorithm 1 runs asymptotically in the same time as CYK. In essence, this implies that checking one complete assignment via CYK is as costly as performing full arc-consistency filtering for *CFGC*. Clearly, achieving arc-consistency for a grammar constraint is at least as hard as parsing. Now,

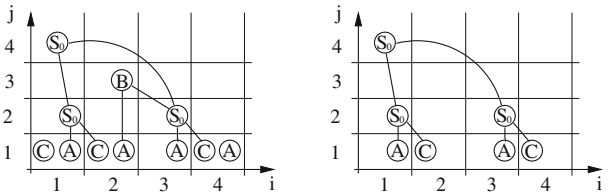

Fig. 2 We show how the algorithm works when the initial domain of X_3 is $D_3 = \{[\}$. The *left picture* shows sets S_{ij} and the *right* the sets S'_{ij}. We see that the constraint filtering algorithm determines the only word in $L_G \cap D_1 \ldots D_4$ is "[][]"

there exist faster parsing algorithms for context-free grammars. For example, the fastest known algorithm was developed by Valiant and parses context-free grammars in time $O(n^{2.8})$. While this is only moderately faster than the $O(n^3)$ that CYK requires, there also exist special purpose parsers for non-ambiguous context-free grammars (i.e. grammars where each word in the language has exactly one parse tree) that run in $O(n^2)$. It is known that there exist inherently ambiguous context-free languages, so these parsers lack some generality. However, in case that a user specifies a grammar that is non-ambiguous it would actually be nice to have a filtering algorithm that runs in quadratic rather than cubic time. It is a matter of further research to find out whether grammar constraint propagation can be done faster for non-ambiguous context-free grammars.

4 Efficient context-free grammar filtering

Given the fact that context-free grammar filtering entails the word problem, there is little hope that, for general context-free grammar constraints, we can devise significantly faster filtering algorithms. However, with respect to filtering performance it is important to realize that a filtering algorithm should work quickly within a constraint propagation engine. Typically, constraint filtering needs to be conducted on a sequence of problems that differ only slightly from the last. When branching decisions and other constraints tighten a given problem, state-of-the-art systems like Ilog Solver provide a filtering routine with information regarding which values were removed from which variable domains since the last call to the routine. By exploiting such condensed information, incremental filtering routines can be devised that work faster than starting each time from scratch. To analyze such incremental algorithms, it has become the custom to provide an upper bound on the total work performed by a filtering algorithm over one entire branch of the search tree (see, e.g., [11]).

Naturally, given a sequence of s monotonically tightening problems (that is, when in each successive problem the variable domains are subsets of the previous problem), context-free grammar constraint filtering for the entire sequence takes at most $O(sn^3|G|)$ steps. Using existing ideas how to perform incremental graph updates in DAGs efficiently (see for instance [7]), it is trivial to modify Algorithm 1 so that this time is reduced to $O(n^3|G|)$: Roughly, when storing additional information which productions support which arcs in the graph (whereby each production can support at most $2n$ arcs for each set S_{ij}), we can propagate the effects of domain values being removed at the lowest level of the graph to adjacent nodes without ever touching

parts of the graph that do not change. The total workload for the entire problem sequence can then be distributed over all $O(|G|n)$ production supports in each of $O(n^2)$ sets, which results in a time bound of $O(n^2|G|n) = O(n^3|G|)$.

The second efficiency aspect regards the memory requirements. In Algorithm 1, they are in $\Theta(n^3|G|)$. It is again trivial to reduce these costs to $O(n^2|G|)$ simply by recomputing the sets of incident arcs rather than storing them in step 2 for step 3 of Algorithm 1. However, when we follow this simplistic approach we only trade time for space. The incremental version of our algorithm as sketched above is based on the fact that we do not need to recompute arcs incident to a node which is achieved by storing them. So while it is straight-forward and easy to achieve a space-efficient version that requires time in $O(sn^3|G|)$ and space $O(n^2|G|)$ or a time-efficient incremental variant that requires time and space in $O(n^3|G|)$, the challenge is to devise an algorithm that combines low space requirements with good incremental performance.

In the following, we will thus modify our algorithm such that the total workload of a sequence of s monotonically tightening filtering problems is reduced to $O(n^3|G|)$, which implies that, asymptotically, an entire sequence of more and more restricted problems can be filtered with respect to context-free grammars in the same time as it takes to filter just one problem from scratch. At the same time, we will ensure that our modified algorithm will require space in $O(n^2|G|)$.

4.1 Space-efficient incremental filtering

In Algorithm 1, we observe that it first works bottom-up, determining from which nodes (associated with non-terminals of the grammar) we can derive a legal word. Then, it works top-down determining which non-terminal nodes can be used in a derivation that begins with the start non-terminal $S_0 \in S_{1n}$. In order to save both space and time, we will modify these two steps in that every non-terminal in each set S_{ij} will perform just enough work to determine whether its respective node will remain in the shrunken graph Q' or not.

To this end, in the first step that works bottom-up we will need a routine that determines whether there exists, what we call, a *support from below*: That is, this routine determines whether a node v_{ijA} has any *outgoing* arcs in $E_1 \cup E_2$. To save space, the routine must perform this check without ever storing sets E_1 and E_2 explicitly, as this would require space in $\Theta(n^3|G|)$.

Analogously, in the second step that works top-down we will rely on a procedure that checks whether there exists a *support from above*: Formally, this procedure determines whether a node v_{ijA} has any *incoming* arcs in $E'_1 \cup E'_2$, again without ever storing these sets which would require too much memory.

The challenge here is to avoid having to pay with time what we save in space. To this end, we need a methodology which prevents us from searching for supports (from above or below) that have been checked unsuccessfully before. Very much like the well-known arc-consistency algorithm AC-6 for binary constraint problems [2], we achieve this goal by ordering potential supports so that, when a support is lost, the search for a new support can start right after the last support, in the respective ordering.

According to the definition of E_1, E_2, E'_1, E'_2, supports of v_{ijA} (from above or below) are directly associated with productions in the given grammar G and a

Algorithm 2 Functions that incrementally (re-)compute the support from below and above for a given node v_{ijA}

```
void findOutSupport(i, j, A)
p_ijA^Out ← p_ijA^Out + 1
while k_ijA^Out < j do
    while p_ijA^Out ≤ |Out_A| do
        (A → B1 B2) ← Out_A[p_ijA^Out]
        if (B1 ∈ S_{i k_ijA^Out}) and (B2 ∈ S_{i+k_ijA^Out, j−k_ijA^Out}) then
            return
        end if
        p_ijA^Out ← p_ijA^Out + 1
    end while
    p_ijA^Out ← 1, k_ijA^Out ← k_ijA^Out + 1
end while

bool findInSupport(i, j, A)
// returns true iff afterwards In_A[p_ijA^In] = (B1 → A B3) for some B1, B3 ∈ N
p_ijA^In ← p_ijA^In + 1
while k_ijA^In > j do
    while p_ijA^In ≤ |In_A| do
        (B1 → B2 B3) ← In_A[p_ijA^In]
        if (A = B2) then
            if (B1 ∈ S_{i k_ijA^In}) and (B3 ∈ S_{i+j, k_ijA^In − j}) then
                return true
            end if
        end if
        if (A = B3) then
            if (B1 ∈ S_{i+j−k_ijA^In, k_ijA^In}) and (B2 ∈ S_{i+j−k_ijA^In, k_ijA^In − j}) then
                return false
            end if
        end if
        p_ijA^In ← p_ijA^In + 1
    end while
    p_ijA^In ← 1, k_ijA^In ← k_ijA^In − 1
end while
return false
```

splitting index k. To order these supports, we cover and order the productions in G that involve non-terminal A in two lists:

- In list $Out_A := [(A \to B_1 B_2) \in P]$ we store and implicitly fix an ordering on all productions with non-terminal A on the left-hand side.
- In list $In_A := [(B_1 \to B_2 B_3) \in P \mid B_2 = A \ \vee \ B_3 = A]$ we store and implicitly fix an ordering on all productions where non-terminal A appears as non-terminal on the right-hand side.

Now, for each node v_{ijA} we store two production indices p_{ijA}^{Out} and p_{ijA}^{In}, two splitting indices $k_{ijA}^{Out} \in \{1, \ldots, j\}$ and $k_{ijA}^{In} \in \{j, \ldots, n\}$, and a flag l_{ijA}. The intended meaning of these indices is that node v_{ijA} is currently supported from below by production $(A \to B_1 B_2) = Out_A[p_{ijA}^{Out}]$ such that $B_1 \in S_{i,k_{ijA}^{Out}}$ and $B_2 \in$

Algorithm 3 A method that performs context-free grammar filtering in time $O(n^3|G|)$ and space $O(n^2|G|)$

```
bool filterFromScratch(D_1, ..., D_n)
// return true iff constraint can be satisfied
for r = 1 to n do
    S_r1 ← ∅
    for all a ∈ D_r do
        S_r1 ← S_r1 ∪ {A_a}
    end for
end for
for j = 2 to n do
    for i = 1 to n − j + 1 do
        for all A ∈ N do
            p^Out_ijA ← 0, k^Out_ijA ← 1
            findOutSupport(i,j,A)
            if (k^Out_ijA < j) then
                S_ij ← S_ij ∪ {A}
                lost^Out_ijA ← false
                informOutSupport(i, j, A, Add)
            end if
        end for
    end for
end for
if (S_0 ∉ S_1n) then
    return false
end if
S_1n ← {S_0}
for j = n − 1 downto 1 do
    for i = 1 to n − j + 1 do
        for all A ∈ S_ij do
            p^In_ijA ← 0, k^In_ijA ← 1
            l_ijA ← findInSupport(i, j, A)
            if (k^In_ijA > j) then
                lost^In_ijA ← false
                informInSupport(i, j, A, Add)
            else
                S_ij ← S_ij \ {A}
                lost^Out_ijA ← true
                informOutSupport(i, j, A, Remove)
            end if
        end for
    end for
end for
for r = 1 to n do
    D_r ← {a | A_a ∈ S_r1}
end for
return true
```

$S_{i+k^{Out}_{ijA}, j-k^{Out}_{ijA}}$. Analogously, the current support from above is production $(B_1 \rightarrow B_2 B_3) = In_A[p^{In}_{ijA}]$ such that $B_1 \in S'_{i,k^{In}_{ijA}}$ and $B_3 \in S'_{i+j,k^{In}_{ijA}-j}$ if $B_2 = A$, in which case l_{ijA} equals to true, or $B_1 \in S'_{i+j-k^{In}_{ijA},k^{In}_{ijA}}$ and $B_2 \in S'_{i+j-k^{In}_{ijA},k^{In}_{ijA}-j}$ if $B_3 = A$, in which case l_{ijA} equals to false. When node v_{ijA} has no support from below (or above), we will have $k^{Out}_{ijA} = j\,(k^{In}_{ijA} = j)$.

Algorithm 4 Functions that inform nodes which other nodes they support from below or above

```
void informOutSupport(i, j, A, State)
(A → B1 B2) ← Out_A[p^Out_ijA]
if State == Add then
    left^Out_ijA ← listOfOutSupported_{i,k^Out_ijA,B1}.add(i, j, A)
    right^Out_ijA ← listOfOutSupported_{i+k^Out_ijA,j−k^Out_ijA,B2}.add(i, j, A)
else
    listOfOutSupported_{i,k^Out_ijA,B1}.remove(left^Out_ijA)
    listOfOutSupported_{i+k^Out_ijA,j−k^Out_ijA,B2}.remove(right^Out_ijA)
end if

void informInSupport(i, j, A, State)
(B1 → B2 B3) ← In_A[p^In_ijA]
if State == Add then
    if l_ijA then
        left^In_ijA ← listOfInSupported_{i,k^In_ijA,B1}.add(i, j, A)
        right^In_ijA ← listOfInSupported_{i+j,k^In_ijA−j,B3}.add(i, j, A)
    else
        left^In_ijA ← listOfInSupported_{i+j−k^In_ijA,k^In_ijA,B1}.add(i, j, A)
        right^In_ijA ← listOfInSupported_{i+j−k^In_ijA,k^In_ijA−j,B2}.add(i, j, A)
    end if
else
    if l_ijA then
        listOfInSupported_{i,k^In_ijA,B1}.remove(left^In_ijA)
        listOfInSupported_{i+j,k^In_ijA−j,B3}.remove(right^In_ijA)
    else
        listOfInSupported_{i+j−k^In_ijA,k^In_ijA,B1}.remove(left^In_ijA)
        listOfInSupported_{i+j−k^In_ijA,k^In_ijA−j,B2}.remove(right^In_ijA)
    end if
end if
```

In Algorithm 2, we show two functions that (re-)compute the support from below and above for a given node v_{ijA}, whereby we assume that variables p^{Out}_{ijA}, p^{In}_{ijA}, k^{Out}_{ijA}, k^{In}_{ijA}, S_{ij}, Out_A, and In_A are global and initialized outside these functions. We see that both routines start their search for a new support right after the last. This is correct as within a sequence of monotonically tightening problems we will never add edges to the graphs Q and Q'. Therefore, replacement supports can only be found later in the respective ordering of supports. Note that the second function which computes a new support from above is slightly more complicated as it needs to make the distinction whether the target non-terminal A appears as first or second non-terminal on the right-hand side of its support production. This information is returned by the second function to facilitate the handling of these two cases.

In Algorithm 3, we illustrate the use of the two support computing functions. The algorithm given here performs context-free grammar filtering from scratch when provided with the current domains $D_1, \ldots, D_n$. We assume again that all global variables are allocated outside of this function. Again, we observe the main two phases in which the algorithm proceeds: After initializing the sets S_{i1}, the algorithm first

Algorithm 5 A method that performs context-free grammar filtering incrementally in space $O(n^2|G|)$ and amortized total time $O(n^3|G|)$ for any sequence of monotonically tightening problems

```
void filterFromUpdate(varSet,Δ_1, ..., Δ_n)
for r = 1 to n do
    nodeListOut[r] ← ∅, nodeListIn[r] ← ∅
end for
for all X_i ∈ varSet do
    for all a ∈ Δ_i do
        if (A_a ∈ S_i1) then
            S_i1 ← S_i1 \ {A_a}
            lost^In_ijA ← true
            informInSupport(i, j, A, Remove)
            for all (p, q, B) ∈ listOfOutSupported_i,j,A_a do
                if (lost^Out_pqB) then
                    continue
                end if
                lost^Out_pqB ← true
                nodeListOut[q].add(p, q, B)
            end for
            for all (p, q, B) ∈ listOfInSupported_i,j,A_a do
                if (lost^In_pqB) then
                    continue
                end if
                lost^In_pqB ← true
                nodeListIn[q].add(p, q, B)
            end for
        end if
    end for
end for
updateOutSupports()
if (S_0 ∉ S_1n) then
    return false
end if
S_1n ← {S_0}
updateInSupports()
return true
```

computes supports from below for nodes on higher levels. In contrast to Algorithm 1, the modified method computes just *one* support from below rather than all of them. Of course, we use the previously devised function findOutSupport for this purpose. After checking whether the constraint can be satisfied at all (test of $S_0 \in S_{1n}$), in the second phase, the algorithm then performs the analogous computation of supports from above with the help of the previously devised function findInSupport. Note that we do not introduce sets S'_{ij} anymore as, for potentially following incremental updates, we would otherwise need to set $S_{ij} = S'_{ij}$ anyway.

In both phases, we note calls to routines informOutSupport and informInSupport. These functions are given in Algorithm 4 and are there to inform the nodes on which the support of the current node relies about this fact. This would not be necessary if we only wanted to filter the constraint once. However, we later want to propagate incrementally the effects of removed nodes, and to do this efficiently, we need a fast way of determining which other nodes currently rely on their existence. At the

Algorithm 6 Function computing new supports from below by proceeding bottom-up

```
void updateOutSupports(void)
for r = 2 to n do
  for all (i, j, A) ∈ nodeListOut[r] do
    informOutSupport(i, j, A, Remove)
    findOutSupport(i,j,A)
    if (k^Out_ijA < j) then
      lost^Out_ijA ← false
      informOutSupport(i, j, A, Add)
      continue
    end if
    S_ij ← S_ij \ {A}
    lost^In_ijA ← true
    informInSupport(i, j, A, Remove)
    for all (p, q, B) ∈ listOfOutSupported_i,j,A do
      if (lost^Out_pqB) then
        continue
      end if
      lost^Out_pqB ← true
      nodeListOut[q].add(p, q, B)
    end for
    for all (p, q, B) ∈ listOfInSupported_i,j,A do
      if (lost^In_pqB) then
        continue
      end if
      lost^In_pqB ← true
      nodeListIn[q].add(p, q, B)
    end for
  end for
end for
```

same time, in pointers left and right we store where the information is located at the supporting nodes so that it is easy to update it when a support should have to be replaced later (because one of the two supporting nodes is removed). Given that both functions work in constant time, the overhead of providing the infrastructure for incremental constraint filtering is minimal and invisible in the O-calculus.

Finally, in Algorithm 5 we present our function filterFromUpdate that re-establishes arc-consistency for the context-free grammar constraint based on the information which variables were affected and which values were removed from their domains. The function starts by iterating through the domain changes, whereby each node on the lowest level adds those nodes whose current support relies on its existence to a list of affected nodes (nodesListOut and nodesListIn). This is a simple task since we have stored this information before. Furthermore, by organizing the affected nodes according to the level to which they belong, we make it easy to perform the two phases (one working bottom-up, the other top-down) later, whereby a simple flag (lost) ensures that no node is added twice.

In Algorithm 6, we show how the phase that recomputes the supports from below proceeds: We iterate through the affected nodes bottom-up. First, for each node that has lost its support from below, because one of its supporting nodes was lost, we inform the other supporting node that it is no longer supporting the current node (for the sake of simplicity, we simply inform both supporting nodes, even though at

Algorithm 7 Function computing new supports from above by proceeding top-down

```
void updateInSupports(void)
for r = n − 1 downto 1 do
  for all (i, j, A) ∈ nodeListIn[r] do
    informInSupport(i, j, A, Remove)
    l_ijA ← findInSupport(i,j,A)
    if (k^In_ijA > j) then
      lost^In_ijA ← false
      informInSupport(i, j, A, Add)
      continue
    end if
    if (j = 1) then
      D_i ← D_i \ {a | A = A_a}
    end if
    S_ij ← S_ij \ {A}
    lost^Out_ijA ← true
    informOutSupport(i, j, A, Remove)
    for all (p, q, B) ∈ listOfInSupported_i,j,A do
      if (lost^In_pqB) then
        continue
      end if
      lost^In_pqB ← true
      nodeListIn[q].add(p, q, B)
    end for
  end for
end for
```

least one of them will never be looked at again anyway). Then, we try to replace the lost support from below by calling findOutSupport. Recall that the function seeks a new support starting at the old, so that no two potential supports are investigated more than just once. Now, if we were successful in providing a new support from below (test $k^{Out}_{ijA} < j$), we inform the new supporting nodes that the support of the current node relies on them. Otherwise, the current node is removed and the nodes that it supports are being added to the lists of affected nodes. The second phase, presented in Algorithm 7, works analogously. The one interesting difference regards the fact that, in function updateInSupport, nodes that are removed because they lost their support from above do not inform the nodes that they support from below. This is obviously not necessary as the supported nodes must have been removed before as they could otherwise provide a valid support from above. This is also the reason why, after having conducted one bottom-up phase and one top-down in filterFromUpdate, all remaining nodes must have an active support from below and above.

With the complete method as outlined in Algorithms 2–7, we can now show:

Theorem 2 *For a sequence of s monotonically tightening context-free grammar constraint filtering problems, based on the grammar G in Chomsky Normal Form filtering for the entire sequence can be performed in time* $O(n^3|G|)$ *and space* $O(n^2|G|)$.

Proof Our algorithm is complete since, as we just mentioned, upon termination all remaining nodes have a valid support from above and below. With the completeness proof of Algorithm 1, this implies that we filter enough. On the other hand, we also

never remove a node if there still exist a support from above and below: we know that, if a support is lost, a replacement can only be found later in the chosen ordering of supports. Therefore, if functions findOutSupport or findInSupport fail to find a replacement support, then none exists. Consequently, our filtering algorithm is also sound.

With respect to the space requirements, we note that the total space needed to store, for each of the $O(n^2|G|)$ nodes, those nodes that are supported by them is not larger then the number of nodes supporting each node (which is equal to four) times the number of all nodes. Therefore, while an individual node may support many other nodes, for all nodes together the space required to store this information is bounded by $O(4n^2|G|) = O(n^2|G|)$. All other global arrays (like left, right, p, k, S, lost, and so on) also only require space in $O(n^2|G|)$.

Finally, it remains to analyze the total effort for a sequence of s monotonically tightening filtering problems. Given that, in each new iteration, at least one assignment is lost, we know that $s \leq |G|n$. For each filtering problem, we need to update the lowest level of nodes and then iterate twice (once bottom-up and once top-down) through all levels (even if levels should turn out to contain no affected nodes), which imposes a total workload in $O(|G|n + 2n|G|n) = O(n^2|G|)$. All other work is dominated by the total work done in all calls to functions findInSupport and findOutSupport. Since these functions are never called for any node for which it has been assessed before that it has no more support from either above or below, each time that one of these functions is called, the support pointer of some node is increased by at least one. Again we find that the total number of potential supports for an individual node could be as large as $\Theta(|G|n)$, while the number of potential supports for all nodes together is asymptotically not larger. Consequently, the total work performed is bounded by the number of sets S_{ij} times the number of potential supports for all nodes in each of them. Thus, the entire sequence of filtering problems can be handled in $O(n^2|G|n) = O(n^3|G|)$. □

Note that the above theorem states time- and space complexity for monotonic reductions without restorations only! Within a backtrack search, we need to also store lost supports for each node leading to the current choice point so that we can backtrack quickly. However, as we will see in the next section, in practice the amount of such additional information needed is very low.

4.2 Numerical results

We implemented the previously outlined incremental context-free grammar constraint propagation algorithm and compared it against its non-incremental counterpart on real-world instances of the shift-scheduling problem introduced in [6].[1] The problem is that of a retail store manager who needs to staff his employees such that the expected demands of workers for various activities that must be performed at all times of the day are met. The demands are given as upper and lower bounds of workers performing an activity a_i at each 15-min time period of the day.

Labor requirements govern the feasibility of a shift for each worker: 1. A work-shift covers between 3 and 8 h of actual work activities. 2. If a work-activity is started,

[1] Many thanks to L.-M. Rousseau for providing the benchmark!

it must be performed for at least one consecutive hour. 3. When switching from one work-activity to another, a break or lunch is required in between. 4. Breaks, lunches, and off-shifts do not follow each other directly. 5. Off-shifts must be assigned at the beginning and at the end of each work-shift. 6. If the actual work-time of a shift is at least 6 h, there must be two 15-min breaks and one 1-h lunch break. 7. If the actual work-time of a shift is less than 6 h, then there must be exactly one 15-min break and no lunch-break. We implemented these constraints by means of one context-free grammar constraint per worker and several global-cardinality constraints ("vertical" gcc's over each worker's shift to enforce the last two constraints, and "horizontal" gcc's for each time period and activity to meet the workforce demands) in Ilog Solver 6.4. To break symmetries between the indistinguishable workers, we introduced constraints that force the *i*th worker to work at most as much as the *i*+first worker.

Table 1 summarizes the results of our experiments. We see that the incremental propagation algorithm vastly outperforms its non-incremental counterpart, resulting in speed-ups of two orders of magnitude while exploring *identical* search-trees! It is quite rare to find that the efficiency of filtering techniques leads to such dramatic improvements with unchanged filtering effectiveness. These results confirm the speed-ups reported in [18]. We also tried to use the decomposition approach from [18], but, due to the method's excessive memory requirements, on our machine with 2 GByte main memory we were only able to solve benchmarks with one activity

Table 1 Shift-scheduling: We report running times on an AMD Athlon 64 X2 Dual Core Processor 3800+ for benchmarks with one and two activity types

Benchmark			Search-tree			Non-incremental		Incremental		Time-speedup
ID	#Act.	#Workers	#Fails	#Prop's	#Nodes	Time [sec]	Mem [MB]	Time [sec]	Mem [MB]	
1_1	1	1	2	136	19	79	12	1.75	24	45
1_2	1	3	144	577	290	293	38	5.7	80	51
1_3	1	4	409	988	584	455	50	8.12	106	56
1_4	1	5	6	749	191	443	60	9.08	124	46
1_5	1	4	6	598	137	399	50	7.15	104	55
1_6	1	5	6	748	161	487	60	9.1	132	53
1_7	1	6	5311	6282	5471	1948	72	16.13	154	120
1_8	1	2	6	299	99	193	26	3.57	40	54
1_9	1	1	2	144	35	80	16	1.71	18	47
1_10	1	7	17447	19319	1769	4813	82	25.57	176	188
2_1	2	2	10	430	65	359	44	7.34	88	49
2_5	2	4	24	412	100	355	44	7.37	88	48
2_6	2	5	30	603	171	496	58	9.89	106	50
2_7	2	6	44	850	158	713	84	15.14	178	47
2_8	2	2	24	363	114	331	44	3.57	84	92
2_9	2	1	16	252	56	220	32	4.93	52	44
2_10	2	7	346	1155	562	900	132	17.97	160	50

For each worker, the corresponding grammar in CNF has 30 non-terminals and 36 productions. Column #Propagations shows how often the propagation of grammar constraints is called for. Note that this value is different from the number of choice points as constraints are usually propagated more than just once per choice point

and one worker only (1_1 and 1_9). On these instances, the decomposition approach implemented in Ilog Solver 6.4 runs about ten times slower than our approach (potentially because of swapped memory) and uses about 1.8 GBytes memory. Our method, on the other hand, requires only 24 MBytes. In general, when comparing the memory requirements of the non-incremental and the incremental variants, we find that the additional memory needed to store restoration data is very limited in practice.

5 Cost-based filtering for context-free grammar constraints

We next consider problems where context-free grammar constraints appear in conjunction with a linear objective function that we are trying to maximize. Assume that each potential variable assignment $X_i \leftarrow w_i$ is associated with a profit $p^i_{w_i}$, and that our objective is to find a complete assignment $X_1 \leftarrow w_1 \in D_1, \ldots, X_n \leftarrow w_n \in D_n$ such that $CFGC_G(X_1, \ldots, X_n)$ is true for that instantiation and $p(w_1 \ldots w_n) := \sum_{i=1}^{n} p^i_{w_i}$ is maximized. Once we have found a feasible instantiation that achieves profit T, we are only interested in improving solutions. Therefore, we consider the conjunction of the context-free grammar constraint $CFGC_G$ with the requirement that solutions ought to have profit greater than T.

This conjunction of a structured constraint (in our case the grammar constraint) with an algebraic constraint that guides our search towards improving solutions is commonly referred to as an *optimization constraint*. The task of achieving generalized arc-consistency is then often called *cost-based filtering* [8]. Optimization constraints and cost-based filtering play an essential role in constrained optimization and hybrid problem decomposition methods such as CP-based Lagrangian Relaxation [20]. In Algorithm 8, we give an efficient algorithm that performs cost-based filtering for context-free grammar constraints. We will prove the correctness of our algorithm by using the following lemma.

Lemma 4 *In Algorithm 8:*

1. *It holds that* $f_A^{ij} = \max\{p(w_{ij}) \mid A \overset{*}{\underset{G}{\Rightarrow}} w_{ij} \in D_i \times \cdots \times D_{i+j-1}\}$, *and* $S_{ij} = \{A \mid \exists\, w_{ij} \in D_i \times \cdots \times D_{i+j-1} : A \overset{*}{\underset{G}{\Rightarrow}} w_{ij}\}$.
2. *It holds that* $g_B^{ik} = \max\{p(w) \mid w \in L_G \cap D_1 \times \cdots \times D_n,\ B \overset{*}{\underset{G}{\Rightarrow}} w_{ik}, S_0 \overset{*}{\underset{G}{\Rightarrow}} w_1 \ldots w_{i-1} B w_{i+k} \ldots w_n\}$.

Proof

1. The lemma claims that the sets S_{ij} contains all non-terminals that can derive a word supported by the domains of variables $X_i, \ldots, X_{i+j-1}$, and that f_A^{ij} reflects the value of the highest profit word $w_{ij} \in D_i \times \cdots \times D_{i+j-1}$ that can be derived from non-terminal A. To prove this claim, we induce over j. For $j = 1$, the claim holds by definition of S_{i1} and f_A^{i1} in step 1. Now assume $j > 1$

Algorithm 8 CFGC Cost-Based Filtering Algorithm

1. For all $1 \leq i \leq n$, initialize $S_{i1} := \emptyset$. For all $1 \leq i \leq n$ and productions $(A_a \to a) \in P$ with $a \in D_i$, set $f_{A_a}^{i1} := p(a)$, and add all such A_a to S_{i1}.
2. For all $j > 1$ in increasing order, $1 \leq i \leq n$, and $A \in N$, set $f_A^{ij} := \max\{f_B^{ik} + f_C^{i+k,j-k} \mid A \underset{G}{\Rightarrow} BC, B \in S_{ik}, C \in S_{i+k,j-k}\}$, and $S_{ij} := \{A \mid f_A^{ij} > -\infty\}$.
3. If $f_{S_0}^{1n} \leq T$, then the optimization constraint is not satisfiable, and we stop.
4. Initialize $g_{S_0}^{1n} := f_{S_0}^{1n}$.
5. For all $k < n$ in decreasing order, $1 \leq i \leq n$, and $B \in N$ set $g_B^{ik} := \max\{g_A^{ij} - f_A^{ij} + f_B^{ik} + f_C^{i+k,j-k} \mid (A \to BC) \in P, A \in S_{ij}, C \in S_{i+k,j-k}\} \cup \{g_A^{i-j,j+k} - f_A^{i-j,j+k} + f_C^{i-j,j} + f_B^{i,k} \mid (A \to CB) \in P, A \in S_{i-j,j+k}, C \in S_{i-j,j}\}$.
6. For all $1 \leq i \leq n$ and $a \in D_i$ with $(A_a \to a) \in P$ and $g_{A_a}^{i1} \leq T$, remove a from D_i.

and that the claim is true for all $1 \leq k < j$. Then $\max\left\{p(w_{ij}) \mid A \underset{G}{\overset{*}{\Rightarrow}} w_{ij} \in D_i \times \cdots \times D_{i+j-1}\right\} = \max\left\{p(w_{ij}) \mid w_{ij} \in D_i \times \cdots \times D_{i+j-1}, A \underset{G}{\Rightarrow} BC, B \underset{G}{\overset{*}{\Rightarrow}} w_{ik}, C \underset{G}{\overset{*}{\Rightarrow}} w_{i+k,j-k}\right\} = \max\left\{p(w_{ik}) + p(w_{i+k,j-k}) \mid w_{ij} \in D_i \times \cdots \times D_{i+j-1}, A \underset{G}{\Rightarrow} BC, B \underset{G}{\overset{*}{\Rightarrow}} w_{ik}, C \underset{G}{\overset{*}{\Rightarrow}} w_{i+k,j-k}\right\} = \max_{(A \to BC) \in P} \max\left\{p(w_{ik}) \mid B \underset{G}{\overset{*}{\Rightarrow}} w_{ik} \in D_i \times \cdots \times D_{i+k-1}\right\} + \max\left\{p(w_{i+k,j-k}) \mid C \underset{G}{\overset{*}{\Rightarrow}} w_{i+k,j-k} \in D_{i+k} \times \cdots \times D_{i+j-1}\right\} = \max\left\{f_B^{ik} + f_C^{i+k,j-k} \mid A \underset{G}{\Rightarrow} BC, B \in S_{ik}, C \in S_{i+k,j-k}\right\} = f_A^{ij}$. Then, f_A^{ij} marks the maximum over the empty set if and only if no word in accordance with the domains of $X_i, \ldots, X_{i+j-1}$ can be derived. This proves the second claim that $S_{ij} = \{A \mid f_A^{ij} > -\infty\}$ contains exactly all those non-terminals from where a word in $D_i \times \cdots \times D_{i+j-1}$ can be derived.

2. The lemma claims that the value g_A^{ij} reflects the maximum value of any word $w \in L_G \cap D_1 \times \cdots \times D_n$ in whose derivation non-terminal A can be used to produce w_{ij}. We prove this claim by induction over k, starting with $k = n$ and decreasing to $k = 1$. We only ever get past step 3 if there exists a word $w \in L_G \cap D_1 \times \cdots \times D_n$ at all. Then, for $k = n$, with the previously proven part 1 of this lemma, $\max\{p(w) \mid w \in L_G \cap D_1 \times \cdots \times D_n, S_0 \underset{G}{\overset{*}{\Rightarrow}} w_{1n}\} = f_{S_0}^{1n} = g_{S_0}^{1n}$. Now let $k < n$ and assume the claim is proven for all $k < j \leq n$. For any given $B \in N$ and $1 \leq i \leq n$, denote with $w \in L_G \cap D_1 \times \cdots \times D_n$ the word that achieves the maximum profit such that $B \underset{G}{\overset{*}{\Rightarrow}} w_{ik}$ and $S_0 \underset{G}{\overset{*}{\Rightarrow}} w_1..w_{i-1}Bw_{i+k}..w_n$. Let us assume there exist non-terminals $A, C \in N$ such that $S_0 \underset{G}{\overset{*}{\Rightarrow}} w_1..w_{i-1}Aw_{i+j}..w_n \underset{G}{\Rightarrow} w_1..w_{i-1}BCw_{i+j}..w_n \underset{G}{\overset{*}{\Rightarrow}} w_1..w_{i-1}Bw_{i+k}..w_n$ (the case where non-terminal B is introduced in the derivation by application of a production $(A \to CB) \in P$ follows analogously). Due to the fact that w achieves maximum profit for $B \in S_{ij}$, we know that $g_A^{ij} = p(w_{1,i-1}) + f_A^{ij} + p(w_{i+j,n-i-j})$. Moreover, it must hold that $f_B^{ik} = p(w_{ik})$ and $f_C^{i+k,j-k} = p(w_{i+k,j-k})$. Then, $p(w) = (p(w_{1,i-1}) + p(w_{i+j,n-i-j})) + p(w_{ik}) + p(w_{i+k,j-k}) = (g_A^{ij} - f_A^{ij}) + f_B^{ik} + f_C^{i+k,j-k} = g_B^{ik}$. □

Now, we can show:

Theorem 3 *Algorithm* 8 *achieves generalized arc-consistency on the conjunction of a CFGC and a linear objective function constraint. The algorithm requires cubic time and quadratic space in the number of variables.*

Proof We show that value a is removed from D_i if and only if for all words $w = w_1 \dots w_n \in L_G \cap (D_1 \dots D_n)$ with $a = w_i$ it holds that $p(w) \leq T$.

$\Rightarrow$ (Soundness) Assume that value a is removed from D_i. Let $w \in L_G \cap (D_1 \dots D_n)$ with $w_i = a$. Due to the assumption that $w \in L_G$ there must exist a derivation $S_0 \overset{*}{\underset{G}{\Rightarrow}} w_1 \dots w_{i-1}\ A_a\ w_{i+1} \dots w_n \underset{G}{\Rightarrow} w_1 \dots w_{i-1} w_i w_{i+1} \dots w_n$ for some $A_a \in N$ with $(A_a \to a) \in P$. Since a is being removed from D_i, we know that $g^{i1}_{A_a} \leq T$. According to Lemma 4, $p(w) \leq g^{i1}_{A_{w_i}} \leq T$.

$\Leftarrow$ (Completeness) Assume that for all $w = w_1 \dots w_n \in L_G \cap (D_1 \dots D_n)$ with $w_i = a$ it holds that $p(w) \leq T$. According to Lemma 4, this implies $g^{i1}_{A_a} \leq T$. Then, a is removed from the domain of X_i in step 6.

Regarding the time complexity, it is easy to verify that the workload is dominated by steps 2 and 5, both of which require time in $O(n^3|G|)$. The space complexity is dominated by the memorization of values f^{ij}_A and g^{ij}_A, and it is thus limited by $O(n^2|G|)$. □

As it turns out, not only can cost-based filtering be performed efficiently. Grammar constraints can also be linearized so that the continuous relaxation is tight, i.e., it does not cause an objective gap between linear and integer programming solution [15].

6 Logic combinations of grammar constraints

We define regular grammar constraints analogously to *CFGC*, but as in [13] we base it on automata rather than right-linear grammars:

Definition 10 (Regular Grammar Constraint) Given a finite automaton A and a right-linear grammar G with $L_A = L_G$, we set

$$RGC_A(X_1, \dots, X_n) := Grammar_G(X_1, \dots, X_n).$$

Efficient arc-consistency algorithms for *RGC*s have been developed in [13, 14]. Now equipped with efficient filtering algorithms for regular and context-free grammar constraints, in the spirit of [3, 12] we focus on certain questions that arise when a problem is modeled by logic combinations of these constraints. An important aspect when investigating logical combinations of grammar constraints is under what operations the given class of languages is closed. For example, when given a conjunction of regular grammar constraints, the question arises whether the conjunction of the constraints could not be expressed as one global *RGC*. This question can be answered affirmatively since the class of regular languages is known to be closed

under intersection. In the following we summarize some relevant, well-known results for formal languages (see for instance [9]).

Lemma 5 *For every regular language L_{A^1} based on the finite automaton A^1 there exists a deterministic finite automaton A^2 such that $L_{A^1} = L_{A^2}$.*

Proof When $Q^1 = \{q_0, \ldots, q_{n-1}\}$, we set $A^2 := (Q^2, \Sigma, \delta^2, q_0^2, F^2)$ with $Q^2 := 2^{Q^1}$, $q_0^2 = \{q_0^1\}$, $\delta^2 := \{(P, a, R) \mid R = \{r \in Q^1 \mid \exists\ p \in P :\ (p, a, r) \in \delta^1\}\}$, and $F^2 := \{P \subseteq Q^1 \mid \exists\ p \in P \cap F^1\}$. With this construction, it is easy to see that $L_{A^1} = L_{A^2}$. □

We note that the proof above gives a construction that can change the properties of the language representation, just like we had noted it earlier for context-free grammars that we had transformed into Chomsky Normal Form first before we could apply CYK for parsing and filtering. And just like we were faced with an exponential blow-up of the representation when bringing context-free grammars into normal-form, we see the same again when transforming a non-deterministic finite automaton of a regular language into a deterministic one.

Theorem 4 *Regular languages are closed under the following operations: Union, Intersection, and Complement.*

Proof Given two regular languages L_{A^1} and L_{A^2} with respective finite automata $A^1 = (Q^1, \Sigma, \delta^1, q_0^1, F^1)$ and $A^2 = (Q^2, \Sigma, \delta^2, q_0^2, F^2)$, without loss of generality, we may assume that the sets Q^1 and Q^2 are disjoint and do not contain symbol q_0^3.

- We define $Q^3 := Q^1 \cup Q^2 \cup \{q_0^3\}$, $\delta^3 := \delta^1 \cup \delta^2 \cup \{(q_0^3, a, q) \mid (q_0^1, a, q) \in \delta^1$ or $(q_0^2, a, q) \in \delta^2)\}$, and $F^3 := F^1 \cup F^2$. Then, it is straight-forward to see that the automaton $A^3 := (Q^3, \Sigma, \delta^3, q_0^3, F^3)$ defines $L_{A^1} \cup L_{A^2}$.
- We define $Q^3 := Q^1 \times Q^2$, $\delta^3 := \{((q^1, q^2), a, (p^1, p^2) \mid \exists (q^1, a, p^1) \in \delta^1, (q^2, a, p^2) \in \delta^2\}$, and $F^3 := F^1 \times F^2$. The automaton $A^3 := (Q^3, \Sigma, \delta^3, (q_0^1, q_0^2), F^3)$ defines $L_{A^1} \cap L_{A^2}$.
- According to Lemma 5, we may assume that A^1 is a deterministic automaton. Then, $(Q^1, \Sigma, \delta^1, q_0^1, Q^1 \setminus F^1)$ defines $L_{A^1}^C$. □

The results above suggest that any logic combination (disjunction, conjunction, and negation) of *RGC*s can be expressed as one global *RGC*. While this is true in principle, from a computational point of view, the size of the resulting automaton needs to be taken into account. In terms of disjunctions of *RGC*s, all that we need to observe is that the algorithm developed in [14] actually works with non-deterministic automata as well. In the following, denote by m an upper bound on the number of states in all automata involved, and denote the size of the alphabet Σ by D. We obtain our first result for disjunctions of regular grammar constraints:

Lemma 6 *Given RGCs $R_1, \ldots, R_k$, all over variables $X_1, \ldots, X_n$ in that order, we can achieve arc-consistency for the global constraint $\bigvee_i R_i$ in time $O((km + k)nD) = O(nDk)$ for automata with constant state-size m.*

If all that we need to consider are disjunctions of *RGC*s, then the result above is subsumed by the well known technique of achieving arc-consistency for disjunctive

constraints which simply consists in removing, for each variable domain, the intersection of all values removed by the individual constraints. However, when considering conjunctions over disjunctions the result above is interesting as it allows us to treat a disjunctive constraint over *RGC*s as one new *RGC* of slightly larger size.

Now, regarding conjunctions of *RGC*s, we find the following result:

Lemma 7 *Given RGCs $R_1, \ldots, R_k$, all over variables $X_1, \ldots, X_n$ in that order, we can achieve arc-consistency for the global constraint $\bigwedge_i R_i$ in time $O(nDm^k)$.*

Finally, for the complement of a regular constraint, we have:

Lemma 8 *Given an RGC R* based on a deterministic automaton*, we can achieve arc-consistency for the constraint $\neg R$ in time $O(nDm) = O(nD)$ for an automaton with constant state-size.*

Proof Lemmas 6–8 are an immediate consequence of the results in [14] and the constructive proof of Theorem 4. □

Note that the lemma above only covers *RGC*s for which we know a deterministic finite automaton. However, when negating a disjunction of regular grammar constraints, the automaton to be negated is non-deterministic. Fortunately, this problem can be entirely avoided: When the initial automata associated with the elementary constraints of a logic combination of regular grammar constraints are deterministic, we can apply the rule of DeMorgan so as to only have to apply negations to the original constraints rather than the non-deterministic disjunctions or conjunctions thereof. With this method, we have:

Corollary 1 *For any logic combination (disjunction, conjunction, and negation) of deterministic RGCs $R_1, \ldots, R_k$, all over variables $X_1, \ldots, X_n$ in that order, we can achieve generalized arc-consistency in time $O(nDm^k)$.*

Regarding logic combinations of context-free grammar constraints, unfortunately we find that this class of languages is not closed under intersection and complement, and the mere disjunction of context-free grammar constraints is not interesting given the standard methods for handling disjunctions. We do know, however, that context-free languages are closed under intersection with regular languages. Consequently, these conjunctions are tractable as well.

7 Limits of the expressiveness of grammar constraints

So far we have been very careful to mention explicitly how the size of the state-space of a given automaton or how the size of the set of non-terminals of a grammar influences the running time of our filtering algorithms. From the theory of formal languages' viewpoint, this is rather unusual, since here the interest lies purely in the asymptotic runtime with respect to the word-length. For the purposes of constraint programming, however, a grammar may very well be generated on the fly and may depend on the word-length, whenever this can be done efficiently. This fact makes

grammar constraints even more expressive and powerful tools from the modeling perspective. Consider for instance the context-free language $L = \{a^n b^n\}$ that is well-known not to be regular. Note that, within a constraint program, the length of the word is known—simply by considering the number of variables that define the scope of the grammar constraint. Now, by allowing the automaton to have $2n + 1$ states, we can express that the first n variables shall take the value a and the second n variables shall take the value b by means of a regular grammar constraint. Of course, larger automata also result in more time that is needed for propagation. However, as long as the grammar is polynomially bounded in the word-length, we can still guarantee a polynomial filtering time.

The second modification that we can safely allow is the reordering of variables. In the example above, assume the first n variables are $X_1, \ldots, X_n$ and the second n variables are $Y_1, \ldots, Y_n$. Then, instead of building an automaton with $2n + 1$ states that is linked to $(X_1, \ldots, X_n, Y_1, \ldots, Y_n)$, we could also build an automaton with just two states and link it to $(X_1, Y_1, X_2, Y_2, \ldots, X_n, Y_n)$ (see Fig. 3). The same ideas can also be applied to $\{a^n b^n c^n\}$ which is not even context-free but context-sensitive. The one thing that we really need to be careful about is that, when we want to exploit our earlier results on the combination of grammar constraints, we need to make sure that the ordering requirements specified in the respective theorems are met (see for instance Lemmas 6 and 7).

While these ideas can be exploited to model some required properties of solutions by means of grammar constraints, they make the theoretical analysis of which properties can or cannot be modeled by those constraints rather difficult. Where do the boundaries run between languages that are suited for regular or context-free grammar filtering? The introductory example, as uninteresting as it is from a filtering point of view, showed already that the theoretical tools that have been developed to assess that a certain language cannot be expressed by a grammar on a lower level in the Chomsky hierarchy fail. The well-known pumping lemmas for regular and context-free grammars for instance rely on the fact that grammars be constant in size. As soon as we allow reordering and/or non-constant size grammars, they do not apply anymore.

To be more formal: what we really need to consider for propagation purposes is not an entire infinite set of words that form a language, but just a slice of words of a given length. I.e., given a language L what we need to consider is just $L_{|n} := L \cap \Sigma^n$. Since $L_{|n}$ is a finite set, it really is a regular language. In that regard, our previous finding that $\{a^n b^n\}$ for fixed n can be modeled as regular language is not surprising. The interesting aspect is that we can model $\{a^n b^n\}$ by a regular grammar of size *linear in n*, or even of *constant size* when reordering the variables appropriately.

Definition 11 (Suitedness for Grammar Filtering) Given a language L over the alphabet Σ, we say that L is *suited for regular (or context-free) grammar filtering*

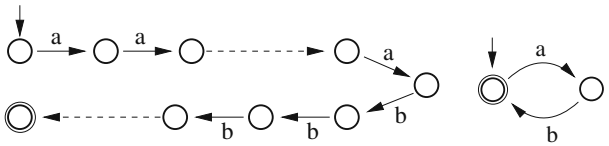

Fig. 3 Regular grammar filtering for $\{a^n b^n\}$. The *left figure* shows a linear-size automaton, the *right* an automaton that accepts a reordering of the language

if and only if there exist constants $k, n \in \mathbb{N}$ such that there exists a permutation $\sigma : \{1, \ldots, n\} \to \{1, \ldots, n\}$ and a finite automaton A (or normal-form context-free grammar G) such that both σ and A (G) can be constructed in time $O(n^k)$ with $\sigma(L_{|n}) = \sigma(L \cap \Sigma^n) := \{w_{\sigma(1)} \ldots w_{\sigma(n)} \mid w_1 \ldots w_n \in L\} = L_A$ ($\sigma(L_{|n}) = L_G$).

Remark 3 Note that the previous definition implies that the size of the automaton (grammar) constructed is in $O(n^k)$. Note further that, if the given language is regular (context-free), then it is also suited for regular (context-free) grammar filtering.

Now, we have the terminology at hand to express that some properties cannot be modeled efficiently by regular or context-free grammar constraints. We start out by proving the following useful Lemma:

Lemma 9 *Denote with $N = \{S_0, \ldots, S_r\}$ a set of non-terminal symbols and $G = (\Sigma, N, P, S_0)$ a context-free grammar in Chomsky-Normal-Form. Then, for every word $w \in L_G$ of length n, there must exist $t, u, v \in \Sigma^*$ and a non-terminal symbol $S_i \in N$ such that $S_0 \overset{*}{\underset{G}{\Rightarrow}} tS_iv$, $S_i \overset{*}{\underset{G}{\Rightarrow}} u$, $w = tuv$, and $n/4 \leq |u| \leq n/2$.*

Proof Since $w \in L_G$, there exists a derivation $S_0 \overset{*}{\underset{G}{\Rightarrow}} w$ in G. We set $h_1 := 0$. Assume the first production used in the derivation of w is $S_{h_1} \to S_{k_1} S_{k_2}$ for some $0 \leq k_1, k_2 \leq r$. Then, there exist words $u_1, u_2 \in \Sigma^*$ such that $w = u_1 u_2$, $S_{k_1} \overset{*}{\underset{G}{\Rightarrow}} u_1$, and $S_{k_2} \overset{*}{\underset{G}{\Rightarrow}} u_2$. Now, either u_1 or u_2 fall into the length interval claimed by the lemma, or one of them is longer than $n/2$. In the first case, we are done, the respective non-terminal has the claimed properties. Otherwise, if $|u_1| < |u_2|$ we set $h_2 := k_2$, else $h_2 := k_1$. Now, we repeat the argument that we just made for S_{h_1} for the non-terminal S_{h_2} that derives to the longer subsequence of w. At some point, we are bound to hit a production $S_{h_m} \to S_{k_m} S_{k_{m+1}}$ where S_{h_m} still derives to a subsequence of length greater than $n/2$, but both $S_{k_m}, S_{k_{m+1}}$ derive to subsequences that are at most $n/2$ letters long. The longer of the two is bound to have length greater than $n/4$, and the respective non-terminal has the desired properties. □

Now consider the language

$$L_{\text{AllDiff}} := \{w \in \mathbb{N}^* \mid \forall\, 1 \leq k \leq |w| :\ \exists\, 1 \leq i \leq |w| :\ w_i = k\}.$$

Since the word problem for L_{AllDiff} can be decided in linear space, L_{AllDiff} is (at most) context-sensitive.

Theorem 5 *L_{AllDiff} is not suited for context-free grammar filtering.*

Proof We observe that reordering the variables linked to the constraint has no effect on the language itself, i.e. we have that $\sigma(L_{\text{AllDiff}|n}) = L_{\text{AllDiff}|n}$ for all permutations σ. Now assume that, for all $n \in \mathbb{N}$, we could actually construct a minimal normal-form context-free grammar $G = (\{1, \ldots, n\}, \{S_0, \ldots, S_r\}, P, S_0)$ that generates $L_{\text{AllDiff}|n}$. We will show that the minimum size for G is exponential in n. Due to Lemma 9, for every word $w \in L_{\text{AllDiff}|n}$ there exist $t, u, v \in \{1, \ldots, n\}^*$ and a non-terminal symbol S_i such that $S_0 \overset{*}{\underset{G}{\Rightarrow}} tS_iv$, $S_i \overset{*}{\underset{G}{\Rightarrow}} u$, $w = tuv$, and $n/4 \leq |u| \leq n/2$. Now, let us count

for how many words non-terminal S_i can be used in the derivation. Since from S_i we can derive u, all terminal symbols that are in u must appear in one block in any word that can use S_i for its derivation. This means that there can be at most $(n-|u|)(n-|u|)!(|u|)! \leq \frac{3n}{4}(\frac{n}{2}!)^2$ such words. Consequently, since there exist $n!$ many words in the language, the number of non-terminals is bounded from below by

$$r \quad \geq \quad \frac{n!}{\frac{3n}{4}\left(\frac{n}{2}!\right)^2} \quad = \quad \frac{4(n-1)!}{3\left(\frac{n}{2}!\right)^2} \quad \approx \quad \frac{4\sqrt{2}}{3\sqrt{\pi}}\frac{2^n}{n^{3/2}} \quad \in \quad \omega(1.5^n).$$

□

Now, the interesting question arises whether there exist languages at all that are fit for context-free, but not for regular grammar filtering? If this wasn't the case, then the algorithms developed in Sections 3 and 4 would be utterly useless. What makes the analysis of suitedness so complicated is the fact that the modeler has the freedom to change the ordering of variables that are linked to the grammar constraint—which essentially allows him or her to change the language almost ad gusto. We have seen an example for this earlier where we proposed that $a^n b^n$ could be modeled as $(ab)^n$.

Theorem 6 *The set of languages that are suited for context-free grammar filtering is a strict superset of the set of languages that are suited for regular grammar filtering.*

Proof Consider the language $L = \{ww^R\#vv^R \mid v, w \in \{0,1\}^*\} \subseteq \{0,1,\#\}^*$ (where x^R denotes the reverse of a word x). Obviously, L is context-free with the grammar $(\{0,1,\#\}, \{S_0, S_1\}, \{S_0 \to S_1\#S_1,\ S_1 \to 0S_10,\ S_1 \to 1S_11,\ S_1 \to \varepsilon\}, S_0)$. Consequently L is suited for context-free grammar filtering.

Note that, when the position $2k+1$ of the sole occurrence of the letter # is fixed, for every position i containing a letter 0 or 1, there exists a *partner position* $p^k(i)$ so that both corresponding variables are forced to take the same value. Crucial to our following analysis is the fact that, in every word $x \in L$ of length $|x| = n = 2l+1$, every even (odd) position is linked in this way exactly with *every* odd (even) position for some placement of #. Formally, we have that $\{p^k(i) \mid 0 \leq k \leq l\} = \{1,3,5,\ldots,n\}$ ($\{p^k(i) \mid 0 \leq k \leq l\} = \{2,4,6,\ldots,2l\}$) when i is even (odd).

Now, assume that, for every odd $n = 2l+1$, there exists a finite automaton that accepts some reordering of $L \cap \{0,1,\#\}^n$ under variable permutation $\sigma : \{1,\ldots,n\} \to \{1,\ldots,n\}$. For a given position $2k+1$ of # (in the original ordering), by $dist^k_\sigma := \sum_{i=1,3,\ldots,2l+1} |\sigma(i) - \sigma(p^k(i))|$ we denote the total distance of the pairs after the reordering through σ. Then, the average total distance after reordering through σ is

$$\begin{aligned}\frac{1}{l+1}\sum\nolimits_{0\leq k\leq l} dist^k_\sigma &= \frac{1}{l+1}\sum\nolimits_{0\leq k\leq l}\sum\nolimits_{i=1,3,\ldots,2l+1}\left|\sigma(i) - \sigma\left(p^k(i)\right)\right| \\ &= \frac{1}{l+1}\sum\nolimits_{i=1,3,\ldots,2l+1}\sum\nolimits_{0\leq k\leq l}\left|\sigma(i) - \sigma\left(p^k(i)\right)\right|.\end{aligned}$$

Now, since we know that every odd i has l even partners, even for an ordering σ that places all partner positions in the immediate neighborhood of i, we have that

$$\sum_{0\leq k\leq l} |\sigma(i) - \sigma\left(p^k(i)\right)| \geq 2 \sum_{s=1,\ldots,\lfloor l/2\rfloor} s = (\lfloor l/2\rfloor + 1)\lfloor l/2\rfloor.$$

Thus, for sufficiently large l, the average total distance under σ is

$$\begin{aligned}\frac{1}{l+1}\sum\nolimits_{0\leq k\leq l} dist_\sigma^k &\geq \frac{1}{l+1}\sum\nolimits_{i=1,3,\ldots,2l+1}(\lfloor l/2\rfloor + 1)\lfloor l/2\rfloor \\ &\geq \frac{l}{l+1}(\lfloor l/2\rfloor + 1)\lfloor l/2\rfloor \\ &\geq l^2/8.\end{aligned}$$

Consequently, for any given reordering σ, there must exist a position $2k+1$ for the letter # such that the total distance of all pairs of linked positions is at least the average, which in turn is greater or equal to $l^2/8$. Therefore, since the maximum distance is $2l$, there must exist at least $l/16$ pairs that are at least $l/8$ positions apart after reordering through σ. It follows that there exists an $1 \leq r \leq n$ such that there are at least $l/128$ positions $i \leq r$ such that $p^k(i) > r$. Consequently, after reading r inputs, the finite automaton that accepts the reordering of $L \cap \{0, 1, \#\}^n$ needs to be able to reach at least $2^{l/128}$ different states. It is therefore not polynomial in size. It follows: L is not suited for regular grammar filtering. □

8 Conclusions

We investigated the idea of basing constraints on formal languages. Particularly, we devised an incremental space- and time-efficient arc-consistency algorithm for grammar constraints based on context-free grammars in Chomsky Normal Form. We studied logic combinations of grammar constraints and showed where the boundaries run between regular, context-free, and context-sensitive grammar constraints when allowing non-constant grammars and reorderings of variables. Our hope is that grammar constraints can serve as powerful, highly expressive modeling entities for constraint programming in the future, and that our theory can help to better understand and tackle the computational problems that arise in the context of grammar constraint filtering.

Acknowledgements This work was supported by the National Science Foundation through the Career: Cornflower Project (award number 0644113).

References

1. Beldiceanu, N., Carlsson, M., & Petit, T. (2004). Deriving filtering algorithms from constraint checkers. In *Principles and practice of constraint programming (CP). LNCS* (Vol. 3258, pp. 107–122).
2. Bessiere, C., & Cordier, M. O. (1993). Arc-consistency and arc-consistency again. In *AAAI*.
3. Bessière, C., & Régin, J.-C. (1998). Local consistency on conjunctions of constraints. In *Workshop on non-binary constraints*.

4. Bourdais, S., Galinier, P., & Pesant, G. (2003). HIBISCUS: A constraint programming application to staff scheduling in health care. In *Principles and practice of constraint programming (CP). LNCS* (Vol. 2833, pp. 153–167).
5. Chomsky, N. (1956). Three models for the description of language. *IRE Transactions on Information Theory, 2*, 113–124.
6. Demassey, S., Pesant, G., & Rousseau, L.-M. (2006). A cost-regular based hybrid column generation approach. *Constraints, 11*(4), 315–333.
7. Fahle, T., Junker, U., Karisch, S. E., Kohl, N., Sellmann, M., & Vaaben, B. (2002). Constraint programming based column generation for crew assignment. *Journal of Heuristics, 8*(1), 59–81.
8. Focacci, F., Lodi, A., & Milano, M. (1999). Cost-based domain filtering. In *Principles and practice of constraint programming (CP). LNCS* (Vol. 1713, pp. 189–203).
9. Hopcroft, J. E., & Ullman, J. D. (1979). *Introduction to automata theory, languages and computation*. Reading: Addison Wesley.
10. Kadioglu, S., & Sellmann, M. (2008). Efficient context-free grammar constraints. In *AAAI* (pp. 310–316).
11. Katriel, I., Sellmann, M., Upfal, E., & Van Hentenryck, P. (2007). Propagating knapsack constraints in sublinear time. In *AAAI* (pp. 231–236).
12. Lhomme, O. (2004). Arc-consistency filtering algorithms for logical combinations of constraints. In *Integration of AI and OR techniques in constraint programming (CPAIOR). LNCS* (Vol. 3011, pp. 209–224).
13. Pesant, G. (2003). A regular language membership constraint for sequences of variables In *Workshop on modelling and reformulating constraint satisfaction problems* (pp. 110–119).
14. Pesant, G. (2004). A regular language membership constraint for finite sequences of variables. In *Principles and practice of constraint programming (CP). LNCS* (Vol. 3258, pp. 482–495).
15. Pesant, G., Quimper, C.-G., Rousseau, L.-M., & Sellmann, M. (2009). The polytope of context-free grammar constraints. In *Integration of AI and OR techniques in constraint programming (CPAIOR). LNCS* (Vol. 5547, pp. 223–232).
16. Quimper, C.-G., & Rousseau, L.-M. (2007). Language based operators for solving shift scheduling problems. In *Metaheuristics conference*.
17. Quimper, C.-G., & Walsh, T. (2006). Global grammar constraints. In *Principles and practice of constraint programming (CP). LNCS* (Vol. 3258, pp. 751–755).
18. Quimper, C.-G., & Walsh, T. (2007). Decomposing global grammar constraints. In *Principles and practice of constraint programming (CP). LNCS* (Vol. 4741, pp. 590–604).
19. Rousseau, L.-M., Pesant, G., & van Hoeve, W.-J. (2005). On global warming: Flow based soft global constraint. *Informs, 27*.
20. Sellmann, M. (2004). Theoretical foundations of CP-based lagrangian relaxation. In *Principles and practice of constraint programming (CP). LNCS* (Vol. 3258, pp. 634–647).
21. Sellmann, M. (2006). The theory of grammar constraints. In *Principles and practice of constraint programming (CP). LNCS* (Vol. 4204, pp. 530–544).
22. Trick, M. (2001). A dynamic programming approach for consistency and propagation for knapsack constraints. In *Integration of AI and OR techniques in constraint programming (CPAIOR)* (pp. 113–124).