GAC via Unit Propagation

Fahiem Bacchus

Department of Computer Science,University of Toronto, Canada.
fbacchus@cs.toronto.edu

Abstract. In this paper we argue that an attractive and potentially very general way of achieving generalized arc consistency (GAC) on a constraint is by using unit propagation (UP) over a CNF encoding of the constraint. This approach to GAC offers a number of advantages over traditional constraint specific algorithms (propagators): it is easier to implement, it automatically provides incrementality and decrementality in a backtracking context, and it can provide clausal reasons to support learning and non-chronological backtracking. Although UP on standard CNF encodings of a constraint fails to achieve GAC, we show here that alternate CNF encodings can be used on which UP does achieve GAC. We provide a generic encoding applicable to any constraint. We also give structure specific encodings for the **regular**, **among**, and **gen-sequence** constraints on which UP can achieve GAC with the same run time bounds as previously presented propagators. Finally, we explain how a UP engine can be added to a CSP solver to achieve a seamless integration of constraints encoded in CNF and propagated via UP and those propagated via traditional constraint specific propagators.

1 Introduction

Unit propagation (UP) is a local propagation mechanism for propositional formulas expressed in conjunctive normal form (CNF). Any constraint over finite domain variables can be converted to CNF using the direct encoding of [17]. However, UP on that encoding (or on the other encodings presented in [17]) does not achieve GAC—it has only the power of forward checking. In this paper we demonstrate that alternate CNF encodings can be constructed that allow UP to efficiently achieve GAC.

Using UP on a CNF encoding to achieve GAC has a number of advantages over the specialized constraint specific algorithms (propagators) that are typically used. Intuitively, these advantages arise from the fact that propagators are procedural where as the CNF encoding utilized by UP is declarative. In particular, the sequencing of operations defining a propagator is typically quite rigid. Altering these operations so as to support features like interleaving with the propagators for other constraints or incrementality and decrementality in a backtracking context often requires non-trivial modifications.

In contrast UP can be run on a CNF encoding in flexible ways. In particular, UP can easily interleave the propagation of multiple constraints; it is always incremental; decrementality can be achieved almost without cost; and it readily supports the derivation of new clauses (generalized nogoods [12]) that can be used to improve search performance via learning and non-chronological backtracking. First, once encoded in CNF a constraint's clauses need not be distinguished from the clauses encoding other constraints. Thus, UP can interleave its GAC propagation across different constraints in arbitrary ways. For example, if the constraints C_1 and C_2 are both encoded in CNF then UP can detect that a value for a variable has become inconsistent for constraint C_1 while it is in the middle of propagating the clauses for computing GAC on C_2. Furthermore, this pruned value will then automatically be taken into account during the rest of UP's computation of GAC on C_2. Second, UP works by forcing the truth value of propositions; the order in which these truth values are set is irrelevant to the final result. Hence, it is irrelevant to UP if some of these propositions already have their truth value set: i.e., UP is always incremental. Third, the modern technique for implementing UP employs a lazy data structure called watch literals. One of the salient features of watch literals is that they need not be updated on backtrack. Thus, with UP decrementality can be achieved essentially without cost: only the forced truth assignments need to be undone. Fourth, if UP detects a contradiction, existent techniques can be used to learn powerful nogoods that can improve search performance. These techniques learn generalized nogoods (clauses) which are more powerful than standard nogoods [12].

Previous work has investigated the connections between SAT propagation mechanisms (including UP) and various CSP propagation mechanisms (e.g., [6, 3, 9, 17]). Most of this work, however, has been confined to binary constraints and thus has addressed AC rather than GAC. However, Hebrard et al. [11] building on the work of [9] present an CNF encoding for an arbitrary constraint that allows UP to achieve relational k-arc-consistency. Although that paper did not directly address achieving GAC via UP, their encoding can easily be adapted accomplish this. Furthermore, UP on this encoding achieves GAC in the same time complexity as a generic GAC algorithm like GAC-Schema [4] (GAC-Schema can however have a lower space complexity). Here one of our contributions is to elaborate the connection between GAC and UP on generic constraints, showing how Hebrard et al.'s encoding idea can be adapted to achieve GAC and how a more direct CNF encoding of the constraint can also allow UP to achieve GAC.

The main benefit of GAC in Constraint Programming, however, lies not so much with generic constraints, but rather in the fact that for a number of constraints specialized algorithms exist, called propagators, that can compute GAC in time that is typically polynomial in the arity of the constraint. This is a significant speed up over generic GAC algorithms which require time exponential in the arity of the constraint.

The main contribution of this paper is to demonstrate that just as some constraints admit specialized algorithms for computing GAC, some constraints also admit specialized CNF encodings on which UP can compute GAC much more efficiently. In this paper, we demonstrate such encodings for three different constraints: **regular**, **among** and **gen-sequence**. However, UP on our encoding for **gen-sequence** is not quite sufficient to achieve GAC. Nevertheless, a simple extension of UP called the **failed literal test** is able to achieve GAC on this encoding. The failed literal test retains the advantages of UP mentioned above.

An additional contribution of the paper is to explain how a CSP solver can easily be modified to integrate a state of the art UP engine that can then be used to achieve GAC for some constraints while the other constraints are propagated using traditional mechanisms. It is worth noting that a UP engine does not need to be implemented from scratch. Rather, *very* efficient state of the art UP engines are publically available. For example, the open source MiniSat SAT solver [7] (a consistent winner of recent SAT solver competitions) contains a clearly coded UP engine that can be extracted and utilized for this purpose—the code can be freely used even for commercial purposes.

In the sequel we first present the background properties of GAC and UP needed for the rest of the paper, and explain what is already known about the connection between GAC and UP. We then give a new result that elaborates on this connection. These results illustrate how generic GAC can be accomplished by UP. We then illustrate how a UP engine can be seamlessly integrated into a CSP solver. Turning to global constraints, we then present encodings for the afore mentioned constraints that allow UP to achieve GAC in polynomial time. We close with some final conclusions and ideas for future work.

2 Background

Let $\mathcal{V} = \{V_1, \ldots, V_n\}$ be a sequence of variables each of which has a finite domain of possible values $dom[V_i]$. Corresponding to the variable sequence $\mathcal{V}$ is the Cartesian product of the variable domains. We extend dom to apply to sequences of variables: $dom[\mathcal{V}] = dom[V_1] \times \cdots \times dom[V_n]$. A constraint C is a function over a sequence of variables $scope(C)$. The size of $scope(C)$ is called the **arity** of C. C maps tuples from $dom[scope(C)]$ to **true**/**false**. That is, if $scope(C) = \langle X_1, \ldots, X_k \rangle$ and $\tau \in dom[X_1] \times \cdots \times dom[X_k]$, then $C(\tau) =$ **true**/**false**. If $C(\tau) =$ **true** we say that τ **satisfies** C, else it **falsifies** it. We also view τ as being a set of assignments and say that $V = d \in \tau$ if τ's V'th dimension is equal to d.

A constraint C is said to be GAC if for every variable $X \in scope(C)$ and every value $d \in dom[X]$ there exists a $\tau \in dom[scope(C)]$ with $X = d \in \tau$ and $C(\tau) =$ **true**. A satisfying tuple τ containing $X = d$ is called a **support** for $X = d$. A constraint can be made GAC by simply removing every unsupported value from the domains of its variables. If the value $d \in dom[V]$ is unsupported by C we say that the assignment $V = d$ is **GAC-inconsistent** for C.

GAC propagation is the process of making some or all of the constraints of the problem GAC. If more than one constraint is to be made GAC, then GAC propagation will be an iterative process as making one constraint GAC might cause another constraint to no longer be GAC. However, since enforcing GAC (making a constraint GAC) is a monotonic process of removing unsupported values from the domains of variables, propagation must converge in at most a polynomial number of steps.

Unit propagation (UP) works on propositional formulas expressed in conjunctive normal form (CNF). A CNF formula is a conjunction of clauses, each of which is a disjunction of literals, each of which is a variable of the formula valued either positively or negatively. Given a CNF formula F and a literal ℓ we denote the reduction of F by ℓ as $F\big|_\ell$. The reduction $F\big|_\ell$ is a new CNF formula obtained from F by removing all clauses of F containing ℓ and then removing $\neg\ell$ (the negation of ℓ) from all remaining clauses. UP works by iteratively identifying all unit clauses of F (i.e., clauses containing only a single literal). Each unit clause $(\ell) \in F$ entails that ℓ must be true. Hence if (ℓ) is a unit clause of F then F is equivalent to $F\big|_\ell$. UP forces ℓ and transforms F to $F\big|_\ell$

whenever it finds a unit clause (ℓ). Furthermore, since $F\big|_\ell$ might contain additional unit clauses UP iteratively identifies unit clauses and continues to reduce the formula until no more units exist.

It is not hard to demonstrate that the final formula produced by UP does not depend on the order in which UP removes the unit clauses. Similarly, if GAC is being achieved for a collection of constraints, the order in which these constraints are processed (and reprocessed) has no effect on the final set of reduced variable domains. It can also be seen that UP runs in time linear in the total size of the input formula F (the sum of the lengths of the clauses of F): each clause of length k need only be visited at most k times to remove literals or force the last literal during UP. GAC also runs in time linear in the size of a tabular representation of the constraint: by processing each satisfying tuple of the constraint GAC can identify all supported variable values.[1] However, the tabular representation of a constraint has size exponential in the constraint's arity. It can be shown that when taking the arity of the constraint as the complexity parameter achieving GAC is NP-Hard. Both UP and GAC are sound rules of inference. That is, if d is removed from the domain of V by GAC, then no tuple of assignments satisfying C can contain $V = d$. Similarly, if UP forces the literal ℓ then ℓ must be true in all truth assignments satisfying F. Finally, Both GAC and UP can detect contradictions. If UP produces an empty clause during its propagation, the initial formula is UNSAT, and if GAC produces an empty variable domain the constraint cannot be satisfied.

UP can also be utilized for look-ahead. In particular the **failed literal rule** [8] involves adding the unit clause (ℓ) and then doing UP. If this yields a contradiction we can conclude that $\neg\ell$ is implied by the original formula.

2.1 Achieving generic GAC with UP

In [11] a CNF encoding for a constraint was given on which UP is able to achieve relational k-arc consistency. Although GAC was not mentioned in that paper, their idea can be easily be adapted to allow UP to compute GAC on an arbitrary constraint.

The required encoding contains clauses for representing the variables and their domains of possible values. We call these clauses the **Variable Domain Encoding** (**VDom**). **VDom** consists of the following propositional variables and clauses. For every multi-valued variable V with $dom[V] = \{d_1, \ldots, d_m\}$:

1. m **assignment variables** $A_{V=d_j}$ one for each value $d_j \in dom[V]$. This variable is true when $V = d_j$, it is false when V cannot be assigned the value d_j, i.e., V has been assigned another value, or d_j has been pruned from V's domain.
2. The $O(m^2)$ binary clauses $(\neg A_{V=d}, \neg A_{V=d'})$ for every $d, d' \in dom[V]$ such that $d \neq d'$, capturing the condition that V cannot be assigned two different values.
3. The single clause $(A_{V=d_1}, \ldots, A_{V=d_m})$ which captures the condition that V must be assigned some value.

One important point about the **VDom** clauses is that although there are $O(km^2)$ clauses, UP can be performed on these clauses in time $O(km)$. In particular, it is easy

[1] Practical GAC algorithms employ support data structures that make re-establishing GAC more efficient and that save space when the constraint is intensionally represented.

to modify a UP engine so that the information contained in the **VDom** clauses is given a more compact $O(km)$ representation and propagated in $O(km)$ time (e.g., the e-clauses described in [5]). In the sequel we will assume that this optimization to the UP engine is used, so as to avoid an unnecessary m^2 factor in our complexity results.

In addition to the **VDom** clauses, the encoding contains additional clauses used to capture the constraint C. These constraint clauses utilize additional propositional variables $t_1, \ldots, t_m$, each one representing one of the m different tuples over $dom[scope(C)]$ that satisfies C. Let τ_i be the satisfying tuple represented by the propositional variable t_i. Using the t_i variables the clauses capturing C are as follows:

1. For the variable $V \in scope(C)$ and value $d \in dom[V]$, let $\{s_1, \ldots, s_i\}$ be the subset of $\{t_1, \ldots, t_m\}$ such that the satisfying tuples represented by the s_i are precisely the set of tuples containing $V = d$. That is, the s_i represent all of the supports for $V = d$ in the constraint C. Then for each variable and value $V = d$ we have the clause $(s_1, \ldots, s_i, \neg A_{V=d})$, which captures the condition that $V = d$ cannot be true if it has no support.
2. For each satisfying tuple of C, τ_i, and assignment $V = d \in \tau_i$ we have the clause $(A_{V=d}, \neg t_i)$, which captures the condition that the tuple of assignments τ_i cannot hold if $V = d$ cannot be true.

When building this encoding, any value $d \in dom[V]$ that is unsupported in C will yield a unit clause $\neg A_{V=d}$ in item 1. Thus for every value $d \in dom[V]$ pruned by GAC, UP will force $\neg A_{V=d}$. More interestingly, in a dynamic context if d is pruned (say by GAC on another constraint), then the clauses of item 2 will allow UP to negate all tuple variables t_i such that $V = d \in \tau_i$. Then the clauses of item 1 will allow UP to delete any newly unsupported domain values. This propagation will continue until all unsupported domain values have been removed and GAC has been re-established.

It can also be observed that the size of this encoding (i.e., the sum of the lengths of the clauses) is linear in the size of the constraint's tabular representation: $O(mk)$ where m is the number of satisfying tuples, and k is the constraint's arity. In particular, t_i appears in only k clauses from item 1 (which bounds to total length of these clauses), and k clauses from item 2 (which are all of length 2). Hence UP on this encoding will operate in time linear in the size of the constraint's tabular representation, i.e., the same time complexity as a generic GAC algorithm.

3 UP for generic GAC revisited

We now present an new method for achieving GAC on a generic constraint via UP. Our method does not introduce any new propositional variables (only assignment variables are used).

In the new encoding we have the **VDom** clauses encoding the multi-valued variables in $scope(C)$, as before. Then for each **falsifying** tuple of assignments $\tau = \langle V_1 = d_1, V_2 = d_2, \ldots, V_k = d_k \rangle$ to the variables in $scope(C)$ we add the clause $(\neg A_{V_1=d_1}, \ldots, \neg A_{V_k=d_k})$ which blocks this tuple of assignments. Call this set of clauses $direct(C)$ [17]. Let the total set of clauses encoding C be $T =$ **VDom** $\cup\ direct(C)$. Now we replace T with its set of prime-implicates.

Definition 1 *If F is a CNF formula then the clause c is a **prime implicate** of F if $F \models c$ and for any c' that is a sub-clause of c ($c' \subsetneq c$), then $F \not\models c'$.*

Letting $PI(T)$ be the set of prime implicates of T we have the following result.

Theorem 1. *If the constraint C is satisfied by some tuple of assignments, UP on the CNF theory $PI(T)$ achieves GAC on C. That is, the assignment $V = d$ is GAC-inconsistent for C iff UP on $PI(T)$ forces $\neg A_{V=d}$. If C has no satisfying tuple, UP on $PI(T)$ will generate the empty clause (and GAC will generate an empty domain).*

Proof: First we observe that the set of satisfying tuples of C and the set of models of T are in 1-1 correspondence. From any model of T we must have one and only one assignment variable $A_{V=d}$ being true for each variable $V \in scope(C)$. Furthermore, this set of true assignments do not correspond to any falsifying tuple, else they would falsify one of the clauses encoding C. Similarly, from a satisfying tuple for C we can make the corresponding assignment variables of T true, this will falsify all other assignment variables and thus satisfy all the clauses encoding C.

Hence, if GAC prunes $V = d$, $A_{V=d}$ cannot be true in any model of T, thus $T \models (\neg A_{V=d})$ and this unit clause must be part of $PI(T)$. On the other hand if UP on $PI(T)$ forces $\neg A_{V=d}$ then $A_{V=d}$ must be false in every model of $PI(T)$, hence false in every model of T (since $PI(T)$ is logically equivalent to T), hence $V = d$ cannot be part of any satisfying tuple, and hence it will be pruned by GAC. Finally, if C is unsatisfiable, T will be UNSAT, and $PI(T)$ will contain only the empty clause. ■

Corollary 1 *If after establishing GAC by running UP on $PI(T)$ we additionally prune the assignment $V = d$ (say by GAC on another constraint), then UP on $PI(T) \cup (\neg A_{V=d})$ will re-establish GAC.*

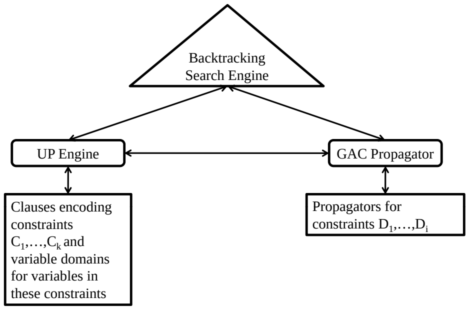
This corollary can be seen to be true by observing that UP on $PT(T) \cup (\neg A_{\,V=d})$ is the same as UP on $PT(T)\big|_{\neg A_{V=d}}$ and that this is the set of prime implicates of $T\big|_{\neg A_{V=d}}$.

Computing the set of prime implicates of a theory can be expensive; in the worst case it is exponential in the sum of the domain sizes of the variables in $scope(C)$. Nevertheless, we can make the following observations. First, the prime implicate encoding can be computed prior to search. Furthermore, once a constraint has been converted to its prime implicate encoding that encoding can be reused in any CSP containing the constraint. Second, some constraints might have a special structure that permit a more efficient computation of their prime implicates. And third even partial computation of some of the implicates of T could allow UP to derive more consequences even if it fails to achieve GAC.

4 Using a UP engine in a CSP solver

The two encodings presented above facilitate achieving GAC on an arbitrary constraint, but the complexity remains exponential in the constraint's arity. In the next section we investigate encodings that exploit constraint specific structure to move from exponential to polynomial complexity. But before presenting those results we first discuss how an UP engine could be utilized in a CSP solver.

The architecture would be as illustrated in the figure to the right. The UP engine operates on a set of clauses encoding some of the constraints $C_1, \ldots, C_k$. These clauses would also include the **VDom** clauses encoding variable domains. The other constraints of the problem, $D_1, \ldots, D_i$ are handled by a standard CSP propagator (e.g., a GAC propagator). Both UP and GAC communicate with each other and with an underlying solver. The communication required is simple. Whenever GAC (the solver) prunes a value d from $dom[V]$ (assigns $V = d$) it informs the UP engine that the associated literal $\neg A_{V=d}$ ($A_{V=d}$) has become true. UP can then propagate this literal which might result in other variables $A_{X=a}$ being forced true or false. If $A_{X=a}$ becomes true, then the assignment $X = a$ is forced and all of X's other values have been pruned. The GAC engine can then propagate the fact that $X \neq d$ for all $d \neq a$. If $A_{X=a}$ becomes false, then the GAC engine can propagate the pruned value $X \neq a$. Either engine might be able derive a contradiction, at which point the search engine can backtrack. On backtrack the search engine will inform the GAC and UP engines to backtrack their state. For UP this is the simple process of unassigning all literals set to be true since the backtrack point.


Backtracking Search Engine
UP Engine
GAC Propagator
Clauses encoding constraints $C_1,\ldots,C_k$ and variable domains for variables in these constraints
Propagators for constraints $D_1,\ldots,D_i$

Note that UP can also return a clausal reason for every assignment variable it forces. These clausal reasons can then be utilized by the search engine to perform non-chronological backtracking and clause learning. Since these clauses can contain both positive and negative instances of the assignment variables they are more general than standard nogoods (which contain only positive instances of assignment variables). As shown in [12] this added generality can yield a super-polynomial speedup in search. With the GAC engine, however, we have to modify each specialized propagator to allow it to produce an appropriate clausal reason for its pruned values. [12] gives some examples of how this can be accomplished.

5 Constraint specific UP encodings

GAC propagators are one of the fundamental enablers of GAC in CSP solvers. Propagators are constraint specific algorithms that exploit the special structure of a constraint so as to compute GAC in time polynomial in the constraint's arity.

Our main contribution is to demonstrate that constraint specific structure can also be exploited by a UP engine for a range of constraints. In particular, we present specialized clausal encodings for three different constraints, **regular**, **among**, and **gen-sequence**, such that UP on these encodings achieve GAC as efficiently as currently known propagators.

There has been some previous work on exploiting structure via specialized CNF encodings of constraints, e.g., [2, 15, 10, 1]. These works however are mostly aimed at obtaining better performing CNF encodings for use in a SAT solver. In particular, either UP on these encodings does not achieve GAC or when it does it is not as efficient as standard GAC propagators.

5.1 Regular

A **regular** constraint over a sequence of k variables asserts that the sequence of values assigned to those variables falls into a specific regular language. By varying the language the regular constraint can be used to capture a number of other constraints (e.g., the **stretch** constraint) [13]. It is not always possible to achieve GAC on a regular constraint in time polynomial in its arity. In particular, the complexity of achieving GAC also depends on the complexity of the regular language being recognized. Nevertheless, UP on our clausal encoding can achieve the same complexity guarantees as the propagator given in [13].

More formally, each regular language $\mathcal{L}$ has an associated Deterministic Finite Automaton (DFA) M that accepts a string iff that string is a member of $\mathcal{L}$. M is defined by the tuple $\langle Q, \Sigma, \delta, q_0, F\rangle$, where Q is a finite set of automaton states, Σ is an input alphabet of symbols, δ is a transition function mapping state-input symbol pairs to new states $Q \times \Sigma \mapsto Q$, q_0 is the initial state, and F is a set of accepting states. A DFA takes as input a string of characters. Each character $c \in \Sigma$ causes it to transition from its current state q to the new state $\delta(q, c)$. The DFA starts off in the state q_0 and a string S over the alphabet Σ is said to be accepted by the DFA M if S causes M to transition from q_0 to some accepting state in F.

Let $\mathcal{L}$ be a regular language over the alphabet of symbols $\bigcup_{i=1}^{k} dom[V_i]$, and M be a DFA that accepts $\mathcal{L}$. The regular constraint over the sequence of variables $\langle V_1, \ldots, V_k\rangle$ and the language $\mathcal{L}$, $\textbf{regular}_{\mathcal{L}}(\langle V_1, \ldots, V_k\rangle)$ is satisfied by a sequence of assignments $\langle V_1 = d_1, \ldots V_k = d_k\rangle$ iff the sequence of values $\langle d_1, \ldots, d_k\rangle$ is accepted by the DFA associated with $\mathcal{L}$.

Our clausal encoding of regular utilizes the insight of [13] that GAC can be achieved by looking for paths in a layered directed graph. The graph has k layers each of which represents the possible states the DFA can be in after s inputs ($0 \leq s \leq k$). Any input string generates a path in this graph, and an accepted string must generate a path starting at q_0 and ending in some state of F. Our encoding captures the structure of this graph in such a way that UP can determine whether or not any particular value $d \in dom[V_i]$ lies on an accepting path (is part of a satisfying tuple). The encoding contains the assignment variables $A_{V_i=d}$ ($d \in dom[V_i]$) along with the **VDom** clauses encoding the variable domains. In addition it also contains the following variables and clauses:

1. For each step s of M's processing, $0 \leq s \leq k$, and each state $q_i \in Q$, the state variable q_i^s. This variable is true if M is in state q_i after having processed s input symbols.
2. For each transition $(q_i, d) \mapsto q_j \in \delta$ and each step s, $1 \leq s \leq k$ such that $d \in dom[V_s]$, the transition variable $t^s_{q_i \langle d \rangle q_j}$. This variable is true if M's s-th input symbol is d and on processing this symbol it transitions from state q_i to q_j.

3. For each transition variable the clauses $(\neg t^s_{q_i\langle d\rangle q_j}, q_i^{s-1})$, $(\neg t^s_{q_i\langle d\rangle q_j}, q_j^s)$, and $(\neg t^s_{q_i\langle d\rangle q_j}, A_{V_s=d})$. These clauses capture the condition that if the transition $t^s_{q_i\langle d\rangle q_j}$ is true, M must be in state q_i at step $s-1$, q_j at step s, and V_s must be assigned the value d.
4. For each state variable q_i^s let $\{t^{s+1}_{q_i\langle *\rangle *}\}$ be the set of transition variables representing transitions out of q_i at step $s+1$ and $\{t^s_{*\langle *\rangle q_i}\}$ be the set of transition variables representing transitions into q_i at step s. For each q_i^s the encoding contains the clauses $(\{t^{s+1}_{q_i\langle *\rangle *}\}, \neg q_i^s)$ and $(\{t^s_{*\langle *\rangle q_i}\}, \neg q_i^s)$, capturing the condition that at least one incoming and one outgoing transition must be true for state q_i to hold at step s. For step 0 we omit the clauses for incoming transitions, and for step k we omit the clauses for outgoing transitions.
5. For each assignment variable $A_{V_s=d}$ let $\{t^s_{*\langle d\rangle *}\}$ be the set of transition variables representing transitions enabled by the assignment $V_s = d$ at step s. Then for each assignment variable the encoding contains the clause $(\{t^s_{*\langle d\rangle *}\}, \neg A_{V_s=d})$ capturing the condition that at least one supporting transition must be true for this assignment to be possible.
6. For each state variable q_i^0 such that $i \neq 0$, the encoding contains the clause $(\neg q_i^0)$, capturing the condition that M must start in step 0 in the initial state q_0. For each state variable q_i^k such that q_i is not an accepting state of M, the encoding contains the clause $(\neg q_i^k)$, capturing the condition that M must finish in some accepting state.

Theorem 2. *If the constraint* ***regular***$_{\mathcal{L}}(\langle V_1, \ldots, V_k\rangle)$ *is satisfied by some tuple of assignments, UP on the above encoding achieves GAC on the constraint in time $O(km|Q|)$, where k is the arity of the constraint, m is the maximum sized domain, and $|Q|$ is the number of states of the DFA accepting $\mathcal{L}$. That is, the assignment $V_i = d$ is GAC-inconsistent for* ***regular***$_{\mathcal{L}}(\langle V_1, \ldots, V_k\rangle)$ *iff UP on this encoding forces $\neg A_{V_i=d}$. If* ***regular***$_{\mathcal{L}}(\langle V_1, \ldots, V_k\rangle)$ *has no satisfying assignment UP will generate the empty clause (GAC will generate an empty domain).*

Proof: First we address the size of the encoding. There are $|Q|k$ state variables (where k is the arity of the constraint and $|Q|$ is the number of states of M), and if m is the maximal sized domain of any of the variables V_i, at most $km|Q|$ transition variables. In particular, M is deterministic, hence for each transition variable $t^s_{q_i\langle d\rangle q_j}$ if any two of the three parameters q_i, d, or q_j are fixed the last parameter can only have a single value (s ranges from 1 to k).

Associated with each transition variable are 3 clauses from item 3 for a total size of $O(km|Q|)$. Associated with each of the $k|Q|$ state variables is a clause of length $|\{t^{s+1}_{q_i\langle *\rangle *}\}| + 1$ and a clause of length $|\{t^s_{*\langle *\rangle q_i}\}| + 1$, from item 4. These clauses are at most $m + 1$ in length (given the input symbol the other (input or output) state is determined): again a total size of $O(km|Q|)$. Finally, associated with each of the $O(km)$ assignment variables is a single clause of length $|\{t^s_{*\langle d\rangle *}\}| + 1$. These clauses can be most length $|Q| + 1$ (fixing the input or output state determines the other state): once again a total size of $O(km|Q|)$. Hence the total size of the clausal encoding is $O(km|Q|)$ which gives the space as well as the time complexity of UP (UP runs in time linear in the total size of the encoding).[2] This is the same time and space complexity of the propagator given in [13].

Now we prove that UP achieves GAC assuming that the constraint is satisfied by some tuple of assignments. First, if $V_i = d_i$ is not pruned by GAC then UP cannot force $\neg A_{V_i=d_i}$. If $V_i = d_i$ is not pruned by GAC it must have a supporting tuple, $\langle V_1 = d_1, \ldots, V_i = d_i, \ldots, V_k = d_k\rangle$. This sequence of inputs to M will cause M to transition through a sequence of states $\langle q_{\pi(0)}, q_{\pi(1)}, \ldots, q_{\pi(k)}\rangle$ starting at the initial state $q_{\pi(0)} = q_0$ and ending at an accepting state $q_{\pi(k)} \in F$. Setting all of the corresponding assignment variables $A_{V_i=d_i}$, state variables $q^s_{\pi(s)}$, and transition variables $t^s_{q_{\pi(s-1)}\langle d_s\rangle q_{\pi(s)}}$ to be true, and and all other variables to be false, we observe that all clauses become satisfied. Hence $A_{V_i=d_i}$ is part of a satisfying truth assignment and since UP is sound it cannot force $\neg A_{V_i=d_i}$. We conclude by contraposition that if UP forces $\neg A_{V_i=d_i}$ then GAC must prune $V_i = d_i$

Second, if UP does not force $\neg A_{V_i=d_i}$ then GAC will not prune $V_i = d_i$, and by contraposition we have that if GAC prunes $V_i = d_i$ then UP must force $\neg A_{V_i=d_i}$. Given that UP has been run to completion and $\neg A_{V_i=d_i}$ has not been forced, then by the clauses of item 5 there must exist some transition variable $t^i_{q_h\langle d_i\rangle q_j}$ that also has not be falsified by UP. By the clauses of item 2, it must also be the case that the state variables q^{i-1}_h and q^i_j have not been falsified. By the clauses of item 4 there must be corresponding transition variables $t^{i-1}_{q_g\langle d_{i-1}\rangle q_h}$ incoming to q^{i-1}_h and $t^{i+1}_{q_j\langle d_{i+1}\rangle q_k}$ outcoming from q^i_j, neither of which have been falsified. By the clauses of item 2 the assignment variables $A_{V_{i-1}=d_{i-1}}$ and $A_{V_{i+1}=d_{i+1}}$ cannot be falsified. Continuing this way we arrive at a input sequence that includes $V_i = d_i$ and that causes M to transition from q_0 to an accepting state. That is, $V_i = d_i$ has a supporting tuple, and GAC will not prune it.

Finally, if **regular**$_{\mathcal{L}}(\langle V_1, \ldots, V_k\rangle)$ has no satisfying assignment then no value $d \in dom[V_i]$ is supported (for any variable V_i). By the above UP will force $\neg A_{V_i=d}$ for every $d \in dom[V_i]$ thus making the **VDom** clause $(A_{V_i=d_1}, \ldots, A_{V_i=d_m})$ empty. ■

Corollary 2 *If after initial GAC we additionally prune the assignment $V_i = d_i$, then adding the unit clause $\neg A_{V_i=d_i}$ and again performing UP re-establishes GAC.*

That is, we can incrementally maintain GAC on **regular**$_{\mathcal{L}}(\langle V_1, \ldots, V_k\rangle)$ by simply falsifying the assignment variables for every pruned assignment and then redoing UP. As noted in the introduction UP is inherently incremental, so no changes need to be made to achieve an incremental propagator. This corollary can be proved by observing that the proof of the theorem continues to apply even if some of the assignment variables have initially been set to be false.

In [14] a CNF encoding for the **grammar** constraint has been developed. That encoding is based on the grammar rules view of languages, and uses CNF to encode a dynamic programming parsing algorithm. UP on the encoding is able to achieve GAC for context free grammar constraints. Since regular languages are a subset of context free languages, this encoding supplies an alternate way of using UP to achieve GAC on **regular**. However, as demonstrated below our encoding for **regular** can be extended to provide an encoding for **gen-sequence** whereas the encoding of [14] does not provide

[2] As discussed Section 2.1 we are exploiting the fact that the km^2 **VDom** clauses can be represented and propagated in $O(km)$ space and time.

an immediate solution for **gen-sequence**. Furthermore, the argument that UP might be a useful general way of achieving GAC is not put forward in that paper.

5.2 Among

An **among** constraint over a sequence of k variables, $\langle V_1, \ldots, V_k \rangle$ asserts that at least $\min$ and at most $\max$ of these variables can take on a value from a specific set S. More formally, **among**$(\langle V_1, \ldots, V_k \rangle, S, \min, \max)$ is satisfied by a tuple of assignments $\langle d_1, \ldots, d_k \rangle$ iff $\min \leq |\langle d_1, \ldots, d_k \rangle \cap S| \leq \max$.

To simplify our notation we can consider **among**$(\langle V_1, \ldots, V_k \rangle, \{1\}, \min, \max)$ in which all of the variables in $\langle V_1, \ldots, V_k \rangle$ have domain $\{0, 1\}$ and S is $\{1\}$. Any **among** constraint **among**$(\langle X_1, \ldots, X_k \rangle, S', \min, \max)$ can be reduced to this case by introducing the $\langle V_1, \ldots, V_k \rangle$ as new variables and imposing a constraint between X_i and V_i ($1 \leq i \leq k$) such that $V_i = 1$ iff X_i has a value in S'. In fact, although we omit the details here, the constraint between the original X_i variables and the V_i variables can be captured with a CNF encoding and enforced with the same UP engine used to enforce **among**.

We can specify a CNF encoding for **among** by constructing a DFA that has $\max$ states, $q_0, \ldots, q_{\max}$. The states $q_{\min}, \ldots q_{\max}$ are all accepting states. The input alphabet is the two conditions V_i and $\neg V_i$ (the V_i are binary values thus they can be treated as propositions with $V_i \equiv V_i = 1$ and $\neg V_i \equiv V_i = 0$). The transition function is simply defined by the condition that an V_i input causes a transition from q_j to q_{j+1}, while a $\neg V_i$ input causes a transition from q_j back to q_j. In other words, the states of the DFA simply keep track of the number of variables taking values in S and accepts iff the total number over all k variables lies in the range $[\min, \max]$. Thus the **among** constraint can be encoded as a CNF on which UP achieves GAC by using the previously specified encoding for this DFA.

Encoding **among** as a CNF is not particularly practical, as **among** has a very simple propagator that is more efficient than UP on this CNF encoding. The real use of the CNF encoding of **among** comes from applying it to conjunctions of **among** constraints. It can also be noted that [1, 15] both provide CNF encodings for the Boolean Cardinality Constraint which is equivalent to the above **among** constraint. Their encodings allow UP to achieve GAC, but as with the above encoding, their encodings are not as efficient as the standard propagator.

5.3 Generalized Sequence Constraint

A **gen-sequence** constraint over a sequence of variables $\langle V_1, \ldots, V_k \rangle$ is a conjunction of **among** constraints. However, it is not an arbitrary collection of **among** constraints. In particular, all of the **among** constraints are over sub-sequences of the same global sequence $\langle V_1, \ldots, V_k \rangle$. Furthermore, the **among** constraint all count membership in a fixed subset of domain values S.[3] More formally,

$$\textbf{gen-sequence}(\langle V_1, \ldots, V_k \rangle, S, \langle \sigma_1, \ldots, \sigma_m \rangle, \langle \min_1, \ldots, \min_m \rangle, \langle \max_1, \ldots, \max_m \rangle) \\ \equiv \bigwedge_{j=1}^{m} \textbf{among}_j(\sigma_j, S, \min_j, \max_j),$$

[3] The algorithm provided in [16] also requires that the set S be fixed over all **among** constraints.

where each σ_j is a sub-sequence of $\langle V_1, \ldots, V_k \rangle$.

We present an encoding on which the failed literal test achieves GAC. That is, with this encoding, each value of each variable can be tested to determined if it is supported by doing a single run of UP. It is particularly important that modern implementations of UP are decremental almost without cost. Thus all that needs to be done between testing successive variable values is to undo the truth assignments forced by the previous run of UP. Using the failed literal test to achieve GAC on this constraint achieves the same time complexity guarantee as the propagator given in [16].

Since the set S is fixed across all of the **among** constraints, we can apply the same transformation used in the previous section to convert them to **among** constraints over variables with domain $\{0, 1\}$ and $S = \{1\}$. With this simplifying transformation our CNF encoding for **gen-sequence** consists of the encoding generated by a simple DFA along with some additional clauses. In particular, to obtain a CNF encoding for **gen-sequence**$(\langle V_1, \ldots, V_k \rangle, \{1\}, \langle \sigma_1, \ldots, \sigma_m \rangle, \langle \min_1, \ldots, \min_m \rangle, \langle \max_1, \ldots, \max_m \rangle)$ (abbreviated as **gen-sequence**$(\langle V_1, \ldots, V_k \rangle)$) where $S = \{1\}$ and the variables are all have domain $\{0, 1\}$, we first generate the CNF encoding for **among**$(\langle V_1, \ldots, V_k \rangle, S, 0, k)$.

This encoding captures a DFA that simply keeps track of the number of variables in $\langle V_1, \ldots, V_k \rangle$ that lie in S: all states are accepting. In the CNF encoding each step s $(0 < s \leq k)$ contains s state variables q_i^s $(0 \leq i \leq s)$ indicating that i of the first s variables $V_1, \ldots, V_s$ took a value in S. The transition variables $t^s_{q_i \langle V_s \rangle q_{i+1}}$ and $t^s_{q_i \langle \neg V_s \rangle q_i}$ capture the transitions $q_i \rightarrow q_{i+1}$ made when V_s takes a value in S and $q_i \rightarrow q_i$ made when V_s takes a value not in S.

In addition to the clauses encoding this "base" DFA, for every constraint **among**$_j(\sigma_j$, S, $\min_j$, $\max_j)$ in the **gen-sequence** we have the following clauses.

1. Let σ_j be the subsequence of variables $\langle V_g, \ldots, V_h \rangle$, for every state variable at step $g{-}1$, q_i^{g-1} $(0 \leq i \leq k)$ we have the clause $(q^h_{i+\min_j}, \ldots, q^h_{i+\max_j}, \neg q_i^{g-1})$ encoding the condition that if the count by time we have reached step h does not lie in the range $[i + \min_j, i + \max_j]$ then we cannot have started with the count at i at step $g{-}1$.
2. The clause $(q^{g-1}_{i-\max_j}, \ldots, q^{g-1}_{i-\min_j}, \neg q_i^h)$ encoding the condition that if the count by time we reach step h is i then we must have started off in the range $[i - \max_j, i - \min_j]$ at step $g{-}1$.

Theorem 3. *If the constraint* ***gen-sequence***$(\langle V_1, \ldots, V_k \rangle)$ *is satisfied by some tuple of assignments, the failed literal test on the above encoding achieves GAC on the constraint in time* $O(mk^3)$*, where* k *is the arity of the constraint, and* m *is the number of* ***among*** *constraints in the* ***gen-sequence*** *constraint. That is, the assignment* V_i *(i.e.,* $V_i = 1$*) is GAC-inconsistent iff the failed literal test on* V_i *yields a contradiction. Similarly for the assignment* $\neg V_i$ *(i.e.,* $V_i = 0$*). If* ***gen-sequence***$(\langle V_1, \ldots, V_k \rangle)$ *has no satisfying assignment then UP (without use of the failed literal test) will generate the empty clause (GAC would generate an empty domain).*[4]

[4] Note that in [16] the claim is made that their propagator runs in time $O(k^3)$. However, a close look at their algorithm demonstrates that it in fact requires $O(mk^3)$. In particular, each time a PUSHUP operation (at most $O(k^2)$) in their CHECKCONSISTENCY algorithm is performed

Proof: We show that V_i is GAC inconsistent iff the failed literal test on $\neg V_i$ yields a contradiction. The argument for $\neg V_i$ is similar.

Consider the relevant parts of the CNF encoding of **gen-sequence**. These are the clauses encoding the base DFA that counts the number of variables taking values in S, along with the clauses of item 1 and 2 above capturing the relationship between the initial and final counts for each **among** constraint.

If GAC does not prune V_i then there must be some supporting tuple of values containing it $\langle V_1, \ldots, V_k \rangle$. We can then generate a path through the base DFA from this input sequence of V_j values. As with the proof of Theorem 2 this sequence can be used to assign **true** to the state, and transition lying along this accepting path, with all other variables assigned **false**. It can then be observed that all clauses of the DFA will be satisfied, and further since the sequence also satisfies all m **among** constraints so will all clauses for the m **among** constraints (items 1 and 2 above). Thus UP cannot not force $\neg V_i$ since V_i is part of a satisfying assignment. Furthermore, the failed literal test on V_i cannot yield a contradiction. By contraposition we conclude that if the failed literal test yields a contradiction then GAC will prune the value.

To prove the other direction consider what happens after UP (but not failed literal) has been run. There will be some **set** of unfalsified state variables Q_s at each step s. If any of these sets, say Q_i is empty, UP must have produced an empty clause. In particular, the clauses of item 3 of the DFA encoding (specifying that a transition cannot be true if its final state is false) will allow UP to falsify all incoming transitions variables into states of step s, and this in turn will allow UP to falsify both "alphabet symbols" V_i and $\neg V_i$ using the clauses of item 5 of the DFA. This will give rise to a contradiction and an empty clause. By the previous paragraph this means that the constraint has no satisfying tuple, as each satisfying tuple yields a model of the clauses.

Say that none of these sets Q_s is empty, let $\langle q_0, q_{\pi(1)}, \ldots, q_{\pi(k)} \rangle$ be the sequence of states such that $\pi(i)$ is the minimal indexed state variable of Q_i. This sequence of states corresponds to the minimal unfalsified counts in the DFA after UP. We claim that this sequence of states corresponds to a satisfying tuple for **gen-sequence**. First, we have that either $\pi(i{+}1) = \pi(i)$ or $\pi(i{+}1) = \pi(i) + 1$, that is these counts can increase by zero or one at each step. If $\pi(i{+}1) < \pi(i)$, then the only transitions into $q^{i+1}_{\pi(i+1)}$ are from q^i_j with $j < \pi(i)$. However, all such state variables have been falsified as $q^i_{\pi(i)}$ is the minimal indexed state variable at step i, hence the clauses of item 3 of the DFA would have falsified all transitions variables coming into $q^{i+1}_{\pi(i-1)}$, and the clauses of item 4 of the DFA would falsify $q^{i+1}_{\pi(i+1)}$: a contradiction. Similarly, if $\pi(i{+}1) > \pi(i){+}1$ then all outgoing transitions from $q^i_{\pi(i)}$ would be falsified and so would $q^i_{\pi(i)}$. Second, if $\pi(i{+}1) = \pi(i)$ the variable $\neg V_i$ cannot be false and if $\pi(i{+}1) = \pi(i){+}1$ the variable V_i cannot be false: if these variables were false then one or both of $q^i_{\pi(i)}$ and $q^{i+1}_{\pi(i+1)}$ would fail to have any incoming or outgoing transitions and would thus be falsified by UP. Finally, this sequence must satisfy each **among** constraint because of the clauses from each of these constraints. Consider the clauses $(q^h_{\pi(g+\min_j)}, \ldots, q^h_{\pi(g+\max_j)}, \neg q^{g-1}_{\pi(g-1)})$

it could require first checking all m **among** constraints to see if any are violated. This yields an $O(mk^2)$ complexity for CHECKCONSISTENCY and an overall $O(mk^3)$ complexity for achieving GAC.

and $(q^{g-1}_{\pi(h-\max_j)}, \ldots, q^{g-1}_{\pi(h-\min_j)}, \neg q^{h}_{\pi(h)})$ from **among** constraint j. Since $q^{g-1}_{\pi(g-1)}$ is the minimal indexed state variable at step g and $q^{h}_{\pi(h)}$ is the minimal indexed state variable at step h, we can see that if $\pi(g) - \pi(h)$ lies outside of the range $[\min_j, \max_j]$ then one of $q^{g-1}_{\pi(g-1)}$ or $q^{h}_{\pi(h)}$ will be falsified by UP from these clauses. For example, if $\pi(g) - \pi(h) > \max_j$ then all of the step h state literals in $(q^{h}_{\pi(g+\min_j)}, \ldots, q^{h}_{\pi(g+\max_j)}, \neg q^{g-1}_{\pi(g-1)})$ would be falsified and thus $q^{g-1}_{\pi(g-1)}$ would also be falsified by UP. Hence, these minimal states and the corresponding transitions and V_j variables between them define a satisfying tuple for **gen-sequence**. We also see that if the constraint has no satisfying tuple, at least one of the sets Q_s must be empty and by the previous paragraph UP yields the empty clause. This proves the last claim of the theorem.

If we apply the failed literal test to V_i, this will force V_i and thus if there is no contradiction the minimal path after UP must contain V_i in the satisfying tuple it defines. Thus V_i is supported, and so GAC will not prune this assignment. By contraposition, if GAC prunes V_i then the failed literal test must generate a contradiction.

Turning now to the complexity, note that the encoding for the base DFA is of size (total length of clauses) $O(k^2)$; in particular $|Q| = k$ and $m = 2$ since the base DFA only has to symbols V_j and $\neg V_j$ in its alphabet. The clauses encoding the **among** constraints total m (the number of **among** constrains) times $O(k^2)$; each **among** constraint generates k clauses from item 1 and k clauses of item 2, each of these clauses is of length $O(k)$. So in total we have a CNF theory of total size $O(mk^2)$. At most we have to do $O(k)$ failed literal tests each of which runs in time $O(k^2)$. This gives us a total complexity for achieving GAC of $O(mk^3)$ in the worst case. ■

Some additional comments can be made about the practicality of this approach to GAC on **gen-sequence**. First, the failed literal test can be quite efficient. In particular, no work needs to be done to reset the "data structures" (i.e., the clauses) between successive failed literal tests. Second, UP on this encoding achieves incremental GAC, it is not difficult to see that if we prune values from the domains of the variables V_i so as to force the value of V_i, UP can incrementally re-establish GAC. Third, the initial UP and each successive failed literal test yields a supporting tuple. If V_i or $\neg V_i$ appears in any of these tuples, we need not do a failed literal test on it: it is already supported. Furthermore, by analogous reasoning it is not hard to see that the sequence of **maximum** indexed state variables in the sets Q_s also forms a satisfying tuple. Thus the initial UP and each subsequent failed literal test can yield up to two satisfying tuples, each of which can be used to avoid testing some as yet untested V_i literals.

6 Conclusions

In this paper we have shown that GAC can be efficiently achieved by using the approach of converting a constraint to a CNF and then employing a UP engine to do propagation in that CNF. As discussed in the introduction this approach has a number of advantages, including the fact that it is immediately incremental and decremental. We have also shown that special structure in a constraint can in some cases be exploited so that UP can achieve GAC with the same complexity as a constraint specific propagation algorithm.

A number of questions remain open. First, there is the empirical question of how this approach performs in practice. On that front we are quite optimistic given the highly tuned nature of publically available UP engines. Second, there is the question of just how general is this approach; can it be applied to other well known constraints? One thing to note with respect to this question is that **regular** is already quite a general constraint. Nevertheless, further research along this line is definitely required. It should be noted that the CNF encodings presented here exploits structure that had already been uncovered and exploited in previous propagators. So it is feasible that other propagation algorithms could similarly exploited.

References

1. Olivier Bailleux and Yacine Boufkhad. Efficient cnf encoding of boolean cardinality constraints. In *Proc. of CP2003*, pages 108–122, 2003.
2. Olivier Bailleux and Yacine Boufkhad. Full cnf encoding: The counting constraints case. In *Proc. of SAT-2004*, 2004.
3. H. Bennaceur. A comparison between sat and csp techniques. *Constraints*, 9:123–138, 2004.
4. C. Bessière and J.C. Régin. Arc consistency for general constraint networks: Preliminary results. In *Proc. IJCAI*, pages 398–404, 1997.
5. Mark Chavira and Adnan Darwiche. Compiling bayesian networks with local structure. In *Proc. IJCAI-2005*, pages 1306–1312, 2005.
6. Y. Dimopoulos and K Stergiou. Propagation in csp and sat. In *Proc. of CP2006*, volume 4204 of *LNCS*, pages 137–151. Springer-Verlag, 2006.
7. Niklas En and Niklas Srensson. Minisat solver. `http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat/MiniSat.html`.
8. J. W. Freeman. *Improvements to Propositional Satisfiability Search Algorithms*. PhD thesis, University of Pennsylvania, 1995.
9. Ian Gent. Arc consistency in sat. In *ECAI*, pages 121–125, 2002.
10. Ian Gent and Peter Nightingale. A new encoding of all-different into SAT. In *Proc. 3rd International Workshop on Modelling and Reformulating Constraint Satisfaction Problems*, pages 95–110. 2004.
11. E. Hebrard, C. Bessière, and T. Walsh. Local consistencies in sat. In *Proc. of SAT-2003*, pages 400–407, 2003.
12. G. Katsirelos and F. Bacchus. Generalized nogoods in csps. In *Proc. of AAAI*, pages 390–396, 2005.
13. G. Pesant. A regular language membership constraint for finite sequences of variables. In *Proc. of CP-2004*, 2004.
14. Claude-Guy Quimper and Toby Walsh. Decomposing global grammar constraints. In *Proc. of CP2007*, 2007.
15. Carsten Sinz. Towards an optimal cnf encoding of boolean cardinality constraints. In *Proc. of CP2005*, pages 827–831, 2005.
16. W.J. van Hoeve, G. Pesant, L.M. Rousseau, and A. Sabharwal. Revisiting the Sequence Constraint. In *Proc. of CP-2006*, 2006.
17. Toby Walsh. Sat v csp. In *Proc. of CP2000*, number 1894 in Lecture Notes in Computer Science, pages 441–456. Springer-Verlag, New York, 2000.