Table Constraints in Clause Learning CSP Solvers

Ozan Erdem (student), George Katsirelos, and Fahiem Bacchus (supervisor)

Department of Computer Science, University of Toronto,
Toronto, Ontario, Canada, M5S 3H5
{ozan,fbacchus}@cs.toronto.edu

Abstract. We investigate alternative methods for implementing table constraints in clause learning CSP solvers (CL solvers). CL solvers have been an important development in CP solving as they can provide important performance improvements on some problems. Furthermore, table constraints remain an important and useful modeling tool in CP. Hence, it is important to be able to utilize table constraints in CL solvers effectively. Here we compare different ways of achieving GAC propagation over table constraints in a CL solver. These methods require different representations of the constraint. First we utilize a CNF encoding of the table constraint which has the property that unit propagation achieves GAC. We compare this with the use of a traditional GAC propagation algorithm for tables, Simple Tabular Reduction (STR). To utilize STR in a CL solver we also develop a method for extracting clausal explanations for pruned values from it. We also develop and test a negative version of STR which more compactly represents tables that have fewer falsifying than satisfying tuples, which also generates clausal explanations. We implement these different methods in the CL solver minicsp, and analyze their performance empirically.

1 Introduction

A number of robust and powerful CSP solvers have been developed and are publicly available. A recent development in CSP solver technology has been the importation of ideas from SAT, specifically clause learning [5, 6, 8, 9]. Clause learning improves the theoretical power of a CSP solver, and recently well engineered clause learning CSP solvers (CL solvers) have demonstrated very good empirical performance on a range of problems (see, e.g., the results of recent MiniZinc Challenges, `http://www.minizinc.org/`).

One of the main technologies exploited by CL solvers is the ability to generate clausal explanations from global constraints. Once the solver can obtain a clause for each pruned value, it can perform clause learning much like a regular SAT solver. In particular, when a contradiction is detected the CL solver must be able to obtain the clausal reasons for the various domain prunings that lead up to the contradiction, and resolve these reasons against the base contradiction.

A considerable amount of work has been done showing how to generate clausal reasons from propagators for various global constraints, e.g., [5, 10, 3,

2]. Methods for generating explanations from table constraints were examined in [5], but from a practical point of view these methods have not been previously examined in the light of the performance tradeoffs of modern CL solvers.

Table constraints are of course very important in constraint programming. They are easy for inexperienced users to use, problem domains often contain special ad-hoc constraints that are most easily encoded extensionally as table constraints, and information stored in databases is often accessed most conveniently as a table constraint.

In this paper we examine some different options for implementing table constraints in a CL solver so as to efficiently achieve GAC. We look at a clausal decomposition, i.e., representing the table constraint as a set of clauses (CNF). The main inference method available on clauses in a CL solver is unit propagation, hence we examine a clausal encoding on which unit propagation is sufficient to achieve GAC. As mentioned above, all that the CL solver needs is the ability to generate clausal reasons for the values pruned by the constraint. We examine one of the most efficient and simplest "propagators" for table constraints, Simple Tabular Reduction [7]. Finally, we examine a new STR-like algorithm for propagating table constraints represented by their falsifying rather than satisfying tuples. We give an algorithm for achieving GAC on this negative STR table. We compare these different ways of handling table constraints in CL solvers and draw some conclusions about their relative effectiveness in practice.

2 Background

A constraint problem $\mathcal{P}$ consists of a finite set $\mathcal{V} = \{V_1, ..., V_n\}$ of variables, each with a finite domain of values $Dom[V_i]$ and a finite set of constraints $\mathcal{C} = \{c_1, \ldots, c_m\}$. Each constraint c_i is over some subset of variables, $scope(c_i) \subseteq \mathcal{V}$. An assignment $\mathcal{A}$ is a set of variable value assignments $\{V^1 = d^1, \ldots, V^k = d^k\}$ in which any variable is assigned at most a single value. Let $vars(\mathcal{A}) \subseteq \mathcal{V}$ denote the set of variables assigned in $\mathcal{A}$. $\mathcal{A}$ is said to **cover** a constraint c if $scope(c) \subseteq \mathcal{A}$.

A constraint c can be viewed as a Boolean function from assignments $\mathcal{A}$ that cover it to *true* (in which case the $\mathcal{A}$ is said to be satisfying) or *false* ($\mathcal{A}$ is said to be falsifying). Often we consider $scope(c)$ to be ordered, and then for assignments $\mathcal{A}$ such that $vars(\mathcal{A}) = scope(c)$ we can order its variable assignments in the same way and then without loss of information remove the variables leaving only a sequence of values. This ordered sequence of values is called a **tuple** for c. If $\mathcal{A}$ is satisfying then its corresponding tuple is said to be a positive tuple (p-tuple) for c, otherwise it is a negative tuple for c (n-tuple). If t is a tuple we let $vars(t)$ denote the variables t specifies values for. If $V \in vars(t)$ we let $t[V]$ denote the value assigned to V in t, and we say that t is **valid** iff for all $V \in vars(t)$ we have that $t[V] \in Dom[V]$. A valid p-tuple t for c is said to be a **support** for the value $d \in Dom[V]$ (where $V \in scope(c)$) if $t[V] = d$.

A positive table constraint (p-table) is a constraint that is specified by a set of p-tuples, a negative table constraint is specified by a set of n-tuples. In

particular these sets of tuples are complete: t satisfies a p-table constraint T if and only if $t \in T$ (similarly for n-tables).

Finally, a constraint c is said to be generalized arc-consistent (GAC) if $\forall V \in scope(c), d \in Dom[V]$ there exists a valid p-tuple t for c such that $t[V] = d$.

A *propositional variable* is a variable with domain $\{0, 1\}$. By convention, if x is a propositional variable, we write x for $x = 1$ and $\neg x$ for $x = 0$. x and $\neg x$ are *literals* of the variable x and they are called complementary to each other. If l is a literal, we denote its complement by $\bar{l}$. A *clause* is a constraint which is a set of literals, interpreted as their logical disjunction, e.g., $(x, y, \overline{z})$, which is interpreted as $x \vee y \vee \neg z$.

In a CL solver, for each variable we maintain (implicitly or explicitly) a *clausal encoding* of its domain. We use the order encoding, as suggested in [8]. For every multi-valued variable V with $Dom[V] = \{d_1, \ldots, d_k\}$ we have k Boolean assignment variables $A_{V=d_j}$ and $k + 1$ *order* variables $A_{V \leq d_j}$, $0 \leq j \leq k$ The variable $A_{V=d_j}$ is true when $V = d_j$, and is false when d_j has been pruned from V's domain. The variable $A_{V \leq d_j}$ is true when all values greater than d_j have been pruned from the domain of V and false when all values less than or equal to d_j have been pruned. We also have $O(k)$ clauses that encode $A_{V=d} \iff A_{V \leq d} \wedge \neg A_{V \leq d-1}$ and $A_{V \leq d} \rightarrow A_{V \leq d+1}$ and the unit clauses $(A_{V \leq d_k})$ and $(\neg A_{V \leq d_0})$.

3 Clausal Decompositions of Table Constraints

We aim to examine methods for implementing table constraints in CL solvers. Such solvers have efficient mechanisms for handling clauses, as they can learn many clauses during solving, and it is very easy to use the same mechanisms to deal with an initial set of input clauses. Hence, one way of implementing table constraints is to convert them into a set of input clauses. The only restriction is that the solver only has access to unit propagation (UP) to reason about these clauses. Thus to achieve GAC we must use an encoding on which UP achieves GAC.

3.1 Support Tuple Encoding

In [1] a CNF encoding for a table constraint c was given on which UP achieves GAC. This encoding was an adaptation of encodings presented in [4].

To encode the constraint c we utilize additional propositional variables t_1, ..., t_m, each one representing one of the m different p-tuples of c. Let τ_i be the p-tuple represented by the propositional variable t_i. Using the t_i variables we can write the clauses capturing C as follows. 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 τ_i such that $\tau_i[V] = d$ (these are the supports of $V = d$). For each variable and value $V = d$ we have the clause $(s_1, \ldots, s_i, \neg A_{V=d})$ ($V = d$ must be false if it has no support). Finally, we have for each p-tuple of C, τ_i, and assignment $V = d \in \tau_i$ 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 and also the condition that if τ_i holds then so do all of its variable assignments.

It has been shown [1] that this encoding enforces GAC on a table constraint in linear size of the constraint's p-table Hence UP on this encoding will operate in time linear in the size of the constraint's p-table representation.

4 Clausal Reasons from Table Propagators

In this section we examine an algorithmic representation for achieving GAC on a table constraint. First we discuss the standard STR algorithm that works on p-table constraints. Then we turn our attention to a negative version of STR that works on n-table constraints, which is new.

4.1 Positive STR

Simple Tabular Reduction (STR) is an efficient GAC algorithm which dynamically maintains tables in order to keep track of supports. It was first introduced by [11], and used in the context of a backtracking search algorithm by [7] along with a number of optimizations under the name STR2+. In this paper we will refer to STR2+ as *positive STR*, and as *STR* when the context is clear. In positive STR, the table t of a constraint is divided into its upper and lower parts called $top(t)$ and $bottom(t)$ with all the tuples in *bottom* being invalid.

As described in [7] achieving GAC with STR involves processing the tuples in *top* to determine if they are valid. If a tuple t is found to be invalid it is moved to *bottom*, while if t is found to be valid all of the values it assigns are marked as having a support (i.e., these values are GAC). After all tuples in *top* are processed, the unsupported variable values are pruned. The key contribution of STR is that the invalid tuples need never be examined, the valid tuples need be examined only once, and backtracking can be achieved by simply restoring the variable domains and moving the marker that divides *top* from *bottom*.

Our addition to STR is to compute clausal reasons for the values pruned,which can be computed lazily. When processing the tuples in *top* for any invalid tuple t we detect we remember the variable value pair that made t invalid. This is some $X = d$ such that $t[X] = d$ and $d \notin Dom[X]$. Hence, each tuple in *bottom* is marked with a proposition $A_{V=d_j}$. Then if we prune a value, e.g., $X = d$ to compute a reason we scan all tuples in *bottom*, locate those tuples t with $t[X] = d$ and accumulate their propositional reasons into a set. The conjunction of these propositions then implies the loss of all supporting tuples for $X = d$, and thus implies $\neg A_{X=d}$. This implication is a clause and supplies the reason we want.

Computing reasons can be done lazily by simply restoring the marker between *top* and *bottom* to its state when the value was pruned and then examing all tuples in *bottom*. Alternatively, by using more memory these reasons could be accumulated as the tuples are detected to be invalid.

4.2 Negative STR

The new STR-based algorithm that we describe for negative table constraints c reuses the table data structures of the positive STR and whenever a tuple becomes invalidated we swap it to the bottom part of the table. To compute the prunings, we keep track of the following: For each variable value pair (X, d), we keep a count of the forbidden tuple counts $ftc(X, d)$ and for each variable X an array $dprod$ of size $|scope(c)|$ where $dprod(i)$ is the number of valid tuples that contain each value of variable i. Whenever $ftc(X, d) == dprod(i)$, we prune the assignment $X = d$. Full details of the algorithm is given in a paper we have submitted.

To compute clausal reasons from n-STR is more complex to implement. We chose to implement a simple but non-minimal way of computing reasons. If n-STR c prunes $X = d$ then it is easy to see that the conjunction of all of the other pruned values for the other variables in c's scope is a sufficient reason.

5 Empirical Results

To evaluate these three methods, we have implemented them in the CL solver `minicsp` (`http://www7.inra.fr/mia/T/katsirelos/minicsp.html`). The evaluation was performed using XCSP benchmarks (`http://www.cril.univ-artois.fr/~lecoutre/benchmarks.html`).

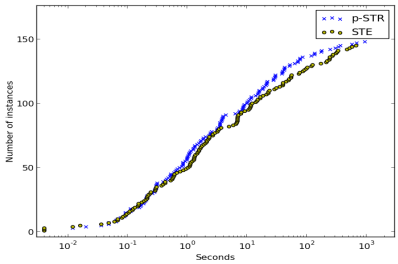

Fig. 1. Real instances

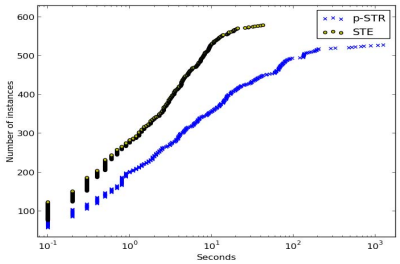

Fig. 2. Patterned instances

Figure 1 reports the number of solved real instances and Figure 2 reports the number of solved patterned instances using the CNF support tuple encoding (STE) and the positive STR method (p-STR). We also generated random instances of table constraints of 50% tightness, Figure 3 shows the comparison between our p-STR implementation and the support tuple encoding (STE). Finally, we tested the n-STR algorithm against a naive encoding where each conflicting tuple is directly represented as a clause which is shown in Figure 4.

In general, it would seem that once the simplicity of using the clausal encoding is considered for CL solvers a CNF encoding will probably be the best

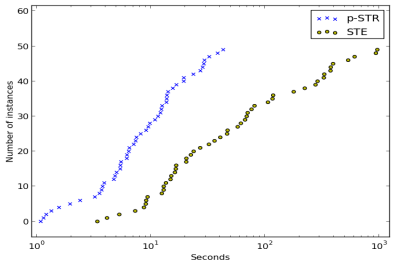

Fig. 3. Random instances

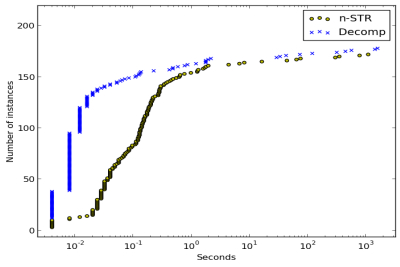

Fig. 4. Instances with negative tables

choice for representing table constraints. The results could change, however, depending on future developments: (1) potentially a more memory expensive way of computing reasons from p-STR tables might benefit p-STRs; (2) better reasons could potentially be computed from n-STRs. These are useful questions for future research.

References

1. Bacchus, F.: Gac via unit propagation. In: Principles and Practice of Constraint Programming (CP). pp. 133–147. Springer-Verlag, New York (2007)
2. Downing, N., Feydy, T., Stuckey, P.J.: Explaining flow-based propagation. In: Integration of AI and OR Techniques in Contraint Programming for Combinatorial Optimzation Problems (CPAIOR). pp. 146–162 (2012)
3. Gange, G., Stuckey, P.J.: Explaining propagators for s-dnnf circuits. In: Integration of AI and OR Techniques in Contraint Programming for Combinatorial Optimzation Problems (CPAIOR). pp. 195–210 (2012)
4. Hebrard, E., Bessière, C., Walsh, T.: Local consistencies in sat. In: Proceedings of Theory and Applications of Satisfiability Testing (SAT). pp. 400–407 (2003)
5. Katsirelos, G.: Nogood Processing in CSPs. Ph.D. thesis, University of Toronto (2008)
6. Katsirelos, G., Bacchus, F.: Generalized nogoods in csps. In: Proceedings of the AAAI National Conference (AAAI). pp. 390–396 (2005)
7. Lecoutre, C.: Str2: optimized simple tabular reduction for table constraints. Constraints 16(4), 341–371 (2011)
8. Ohrimenko, O., Stuckey, P.J., Codish, M.: Propagation = lazy clause generation. In: Principles and Practice of Constraint Programming (CP). pp. 544–558 (2007)
9. Ohrimenko, O., Stuckey, P.J., Codish, M.: Propagation via lazy clause generation. Constraints 14(3), 357–391 (2009)
10. Schutt, A., Feydy, T., Stuckey, P.J., Wallace, M.G.: Explaining the cumulative propagator. Constraints 16(3), 250–282 (2011)
11. Ullmann, J.R.: Partition search for non-binary constraint satisfaction. Inf. Sci. 177(18), 3639–3678 (Sep 2007), http://dx.doi.org/10.1016/j.ins.2007.03.030