Generalized Arc Consistency Algorithms for Table Constraints: A Summary of Algorithmic Ideas

Roland H. C. Yap,[1] **Wei Xia, Ruiwei Wang**[1]
[1]School of Computing, National University of Singapore
{ryap, ruiwei}@comp.nus.edu.sg

Abstract

Constraint Programming is a powerful paradigm to model and solve combinatorial problems. While there are many kinds of constraints, the table constraint (also called a CSP) is perhaps the most significant—being the most well-studied and has the ability to encode any other constraints defined on finite variables. Thus, designing efficient filtering algorithms on table constraints has attracted significant research efforts. In turn, there have been great improvements in efficiency over time with the evolution and development of AC and GAC algorithms. In this paper, we survey the existing filtering algorithms for table constraint focusing on historically important ideas and recent successful techniques shown to be effective.

1 Introduction

"Constraints" is a surprisingly powerful notion. It is used successfully in Artificial Intelligence (AI) and in diverse areas in computer science, e.g., code generation, optimization, program synthesis, robotics, semantic web, simulation, software engineering, type checking, verification, etc. Historically constraints were developed in AI under the framework of Constraint Satisfaction Problems (CSP) (see (Freuder and Mackworth 2006) for a history). A CSP is given some relations over a finite set of variables—a canonical task is to determine satisfiability (see (Dechter 2003) for details). Another significant line of development comes from the integration of constraints into programming languages initiated by the work in Constraint Logic Programming (CLP) (see (Jaffar and Maher 1994)) which is now called Constraint Programming (CP). CP broadened the notion of constraints, e.g. real-valued constraints, complex global constraints (see (van Hoeve and Katriel 2006)), etc.

A key task is how to solve constraints, i.e. solving the CSP. We focus on finite domain (FD) constraints where the variables of the CSP take finite values which can be used to encode/model combinatorial problems. Since finite domain CSPs are NP-complete in general, the typical approach taken to solve them is to combine a local consistency algorithm with a search strategy to instantiate (or restrict) the variables. The consistency algorithm removes (filters) some incompatible values but to its local nature does not remove all such values, while the search instantiates further variables to simplify the problem till eventually we are certain about the values or the problem is determined to be unsatisfiable.

A canonical way of defining a FD constraint is simply to define the allowed (or disallowed) tuples of values, thus the constraint is defined as a table hence the term *table constraint*. The seminal work of Mackworth (Mackworth 1977) (it has $>$ 3600 citations in Google Scholar) defined a certain local consistency, namely, *arc consistency* (AC) together with algorithms, on (binary) table constraints. Since then there has been considerable research on AC and its more general form, *generalized arc consistency* (GAC). GAC algorithms have sped up by several orders of magnitude over the years. Figure 1 shows the algorithmic improvements over time, for selected GAC algorithms on a diverse set of instances.[1] Newer algorithms such as GAC-VA (Lecoutre and Szymanek 2006), STR2+ (Lecoutre 2008; 2011), STR3 (Lecoutre, Likitvivatanavong, and Yap 2012; 2015b), and Mddc (Cheng and Yap 2008; 2010) are faster than the older GAC-4 (Mohr and Masini 1988), GAC-V, and GAC-A (Bessière and Régin 1997) but slower than the even newer algorithms such as STRbit (Wang et al. 2016), CT (Demeulenaere et al. 2016) and HTAC (Wang and Yap 2019), e.g. on the Crossword-ogd-vg-11-13 instance—GAC-A timeouts, CT takes 49.78s and HTAC is the fastest at 21.97s (this problem instance is also in Figure 1).

There has been a large body of work on (generalized) arc consistency algorithms dating from the AC3 algorithm (Mackworth 1977) in 1977. Many different ideas in the development of (G)AC algorithms shown by the considerable progress in Figure 1. This paper surveys techniques and algorithms for (G)AC on table constraints ((G)AC refers to both GAC and AC). Table 1 gives an overview of the key techniques and algorithms that we will discuss in this paper. Our goal is to explain key ideas which have been historically important as well as review more recent algorithmic ideas over the past decades. While it is not possible to cover all algorithms and ideas due to space, we

[1]The benchmarks are intended to illustrate overall differences in the performance of the selected algorithms rather than being comprehensive.

Techniques	Algorithms
GAC-scheme based	GAC-scheme, GAC-V, GAC-A, GAC-nextIn, GAC-VA, GAC-nextDiff, Trie
Index	GAC-nextIn, GAC-VA, GAC-nextDiff, Trie, **AC5TC**
STR based	**STR, STR2, STR2+, STR3, STR-neg, STR2-C, STR3-C, shortSTR2, STR-slice, STR2w, smartSTR, STRbit(-C) CT, shortCT, CT-neg, smartCT**
Residue support	AC7, GAC-scheme based, (G)AC3.1/AC2001, AC3.2, AC3.3, AC3rm, **AC3^{bit+r}, STR3(-C), AC5TC, STR2w, STRbit(-C), CT, smartCT**
Bitwise Representation	**AC3^{bit},AC3^{bit+r}, STRbit, STRbit-C, CT, shortCT, CT-neg, smartCT, Compact-MDD, Compact-smartMDD, HTAC**
MDD based	**Mddc, incremental-MDD, MDD4, MDD4R, Compact-MDD, BDDF, Compact-smartMDD**
Incrementality	AC4, GAC4, AC5, AC6, AC7, GAC-scheme based, (G)AC3.1/AC2001, AC3.2, AC3.3, **STR based, MDD based, GAC4R, AC5TC**
Reset	**GAC4R, MDD4R, STRbit-C, CT, smartCT, Compact-MDD, Compact-smartMDD, HTAC**
Watched tuple	AC6, AC7, GAC-scheme, GAC-nextIn, GAC-nextDiff, Trie, **STR3, STR3-C, STR2w**
Compact representations	GAC-ctuple, **MDD-based, STR2-C, STR3-C, shortSTR2, STR-slice, smartSTR, STRbit-C, shortCT, smartCT**

Table 1: Overview (G)AC algorithms and techniques. The algorithms are sorted by year left to right. The algorithms proposed since STR (Ullmann 2007) are marked in **bold**.

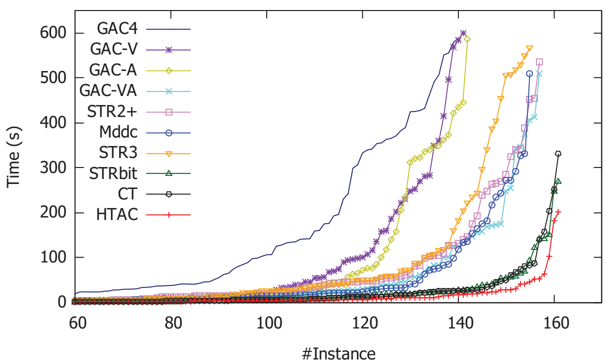


Figure 1: The runtime distribution of selected GAC algorithms giving the number of instances (x-axis) solved and total solving time (y-axis). There are 162 instances from 23 diverse benchmark series from the XCSP3 benchmarks (http://xcsp.org).

will attempt to explain how the development of improvements in (G)AC table algorithms has evolved and the key ideas. We review how algorithms have dealt with binary to general constraints and how they have exploited structure. We remark that some special global constraints such as `regular`, `grammar` and MDD constraints can also be viewed as a special form of table constraint. While several surveys and book chapters have been published on filtering algorithms of global constraints (van Hoeve 2001; Bessiere 2006; Régin 2011), they do not focus on table constraints. In this paper, we aim at providing the reader with a reasonably timely survey on the filtering algorithms for the table constraint (and CSPs) focusing on the important ideas in those algorithms.

2 Preliminaries

A *Constraint Satisfaction Problem* (CSP) P is a pair of (X, C) where X is a set of n *variables* $\{x_1, ..., x_n\}$ *and* C *a set of* e constraints $\{c_1, ..., c_e\}$. The variables in constraint c is denoted as $scope(c)$ and the allowed values for a variable $x \in X$ is its *domain*, $D(x)$. A constraint c is a relation, $rel(c)$, which can be defined as a set of tuples over the variables in $scope(c)$. The *arity* of constraint c is r where $r = |scope(c)|$. A constraint c is binary if the arity r is 2 and *non-binary* if $r > 2$. A tuple τ is *allowed* on c iff $\tau \in rel(c)$. A tuple is *valid* on c iff $\tau[x] \in D(x)$ for each $x \in scope(c)$ where $[x]$ denotes projection onto the variable x. A tuple τ is a *support* of (x, a) on c iff $\tau[x] = a$ and τ is valid and allowed by c. A *solution* to P is a valid tuple over X such that every constraint is satisfied. A CSP is unsatisfiable is it doesn't have a solution.

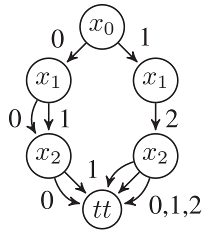
Definition 2.1. Generalized Arc Consistency (GAC). A value (x, a) is *generalized arc consistent* (GAC) (Dechter 2003) iff for any constraint c involving x, there exists at least one support τ for (x, a) in c. A constraint is GAC iff $\forall v \in D(x), \forall x \in scope(c)$ is GAC. A CSP $P(X, C)$ is GAC iff $\forall c \in C$ is GAC.

Constraint solvers usually use filtering algorithms to remove some form of inconsistent information from the CSP. Only some forms of information can be removed efficiently, which is usually called local consistency. GAC is perhaps the most basic non-trivial local consistency. Essentially, GAC attempts to remove certain values which cannot occur in a solution of the CSP. GAC is the most widely researched and successful form of local consistency. *Arc consistency* (AC) is GAC specialized for binary constraints. In this paper, local consistency/filtering which is stronger than (G)AC is called *stronger consistency*.

A *table constraint* c is defined on a positive (negative) table T and $scope(c)$, where T lists all the allowed (disallowed) tuples of c. Table constraints can be viewed as a general way of defining the constraint (relation). Traditionally CSPs were viewed as consisting of table constraints. One can think of the table as a logical definition but the actual representation need not be a table. One alternative representations of a table constraint is as a *multi-valued decision diagram* (MDD). Figure 2(a) gives an example of table constraint and its equivalent MDD representation in Figure 2(b). The tuples of table constraint adhere to paths from the root node x_0 to node tt (true terminal) in the MDD. Thus, a table constraint can be viewed as a table or it's equivalent MDD representation. Other alternate representations for a table are possible, such as the graph based representations: *deterministic finite automata* (DFA a.k.a `regular`) (Pesant 2004), *non-deterministic finite automata* (NFA) (Cheng, Xia, and Yap 2012), *context-free grammar* (CFG a.k.a `grammar`) (Sellmann 2006; Cheng, Xia, and Yap 2012). For example, a regular constraint is simply a DFA defining the set of tuples which can be generated by the automata over the variables in the constraint. Similarly,

	x_0	x_1	x_2
τ_0	0	0	0
τ_1	0	0	1
τ_2	0	1	0
τ_3	0	1	1
τ_4	1	2	0
τ_5	1	2	1
τ_6	1	2	2

(a) A table constraint.



(b) An MDD constraint.

	x_0	x_1	x_2
c-τ_0	0	{0,1}	{0,1}
c-τ_1	1	2	{0,1,2}

(c) c-tuple

	x_0	x_1	x_2
s-τ_0	0	0	0
s-τ_1	0	0	1
s-τ_2	0	1	0
s-τ_3	0	1	1
s-τ_4	1	2	*

(d) Short-Supports

Figure 2: Examples of a table constraint and alternate representations

there are various representations of table which simplify the description of the table by "compression": bitwise encoding, *c-tuple* (Katsirelos and Walsh 2007; Xia and Yap 2013) (the Cartesian product), *short-supports* (Jefferson and Nightingale 2013), *slice-tables* (Gharbi et al. 2014), and *smart-tables* (Mairy, Deville, and Lecoutre 2015). For example, Figure 2(c) and Figure 2(d) give the c-tuple and the short-support representations respectively for the table constraint in Figure 2(a). Specifically, the c-tuple c-τ_1 in Figure 2(c) and the short-support s-τ_4 in Figure 2(d) both represent the tuples τ_4, τ_5, τ_6 in Figure 2(a).

3 Classical Arc Consistency Algorithms

There has been considerable research in AC/GAC algorithms since the pioneering work of the AC3 algorithm (Mackworth 1977) in 1977. AC3 enforces AC at the granularity of a single constraint, i.e. *coarse-grained*, and propagates domain changes from variables to other constraints whose scope includes those variables. From the perspective of binary CSPs, a constraint is an edge in the constraint graph, thus a coarse-grained AC algorithm works on edges. A coarse-grained table approach for a G(AC) can be attractive as the algorithm can be simple with low overheads because of simple data structures. In contrast to AC3, AC4 (Mohr and Henderson 1986) works by enforcing AC at the granularity of a single variable's domain value, i.e. *fine-grained*, and was the first optimal AC algorithm in terms of time complexity. It tries to perform minimum work to maintain AC when a value is removed from a variable's domain. The tradeoff compared to the coarse grained AC3 is that AC4 maintains more fine grained information including a list of supports and counters for the number of supports of each variable-value pair. In practice, AC4 was found to be outperformed by AC3 eventhough AC3 is not an optimal algorithm (Wallace 1993) which highlights that it is not sufficient to consider worst case complexity for efficient and practical constraint solving. Meanwhile, AC3 and AC4 can be regarded as special cases of AC5 (Van Hentenryck, Deville, and Teng 1992). AC6 (Lecoutre and Cordier 1993; Bessière 1994) is also optimal but reduces the space of AC4 by a factor of d (variable's domain size). AC7 (Bessiere, Freuder, and Regin 1995) extends AC6 by exploiting bidirectionality, i.e. both values v_i and v_j of a valid support (v_i, v_j) can simultaneously support each other.

In 2001, more than two decades after AC3, an optimal coarse-grained algorithm AC3.1/AC2001[2] (Zhang and Yap 2001; Bessière and Régin 2001; Bessière et al. 2005) was found, which outperforms AC3. AC3.1/AC2001 gained efficiency with *residues*—residues are the supports found previously and saved for later use. Ordering supports and using residues amortizes support checking, reducing it by a factor of d making AC3.1/AC2001 optimal. AC3.2 and AC3.3 (Lecoutre, Boussemart, and Hemery 2003) extend AC3.1/AC2001 by partially and fully involving the bidirectionality of AC7 respectively. AC3$^{\mathrm{bit}}$ (Lecoutre and Vion 2008) is the first practical AC algorithm exploiting bitwise operations. AC3$^{\mathrm{bit+r}}$ (Lecoutre and Vion 2008) further extends AC3$^{\mathrm{bit}}$ to residues and multi-directionality, and is still a state-of-the-art AC algorithm.[3]

To summarize classical AC algorithms, we highlight that many can be simply extended to GAC. The older AC algorithms also contribute to the development of modern GAC algorithms, e.g., design choices such as propagation granularity, and optimizations like residue, bitwise representation, and multi-directionality.

4 Generalized Arc Consistency Algorithms

We now review the GAC algorithms for table constraints categorizing them by algorithm design choices, leveraging from optimization techniques to constraint representations.

GAC schema

GAC-scheme (Bessière and Régin 1997) is a framework enforcing GAC on any type of constraint by implementing a *seekingSupports* function, it searches supports for each domain value. If *seekingSupport* finds a support for a value, the value is GAC; otherwise the value is inconsistent. GAC-V and GAC-A are the two approaches following the GAC-scheme but with different ways of seeking supports. GAC-V iterates over valid tuples until one satisfying the constraint, while GAC-A iterates over the allowed tuples of the constraint until a valid one is found. In GAC-VA, it alternates traversal of the list of valid and allowed tuples. GAC-VA was shown to be more robust than GAC-A or GAC-V when constraint tightness is close to the both extremes, i.e. close to high end 1.0 or low end 0.0. GAC-nextln (Lhomme and Régin 2005), GAC-nextDiff(Gent et al. 2007) and Trie (Gent et al. 2007) are also based on the GAC-scheme, but use indexing.

[2]We use AC3.1/AC2001 to refer to the algorithm, as AC3.1 and AC2001 were proposed and named by different authors of the respective papers.

[3]AC3$^{\mathrm{bit+r}}$ is named as AC3$^{\mathrm{bit+rm}}$ in (Lecoutre and Vion 2008).

Indexing

Indexing is a classical way of optimizing data search by using an efficient data structure. It has also been used in GAC algorithms. GAC-nextln builds an index over a constraint c such that for each variable-value pair (x, a), the tuple of c containing (x, a) links to the following tuple also containing (x, a). GAC-nextDiff also builds indexes between tuples but links to the following tuple having different values for the same variable. Another indexing data structure is a trie (Gent et al. 2007) such that converting tables into trie can speed up the process of seeking supports. The AC5TC (Mairy, Hentenryck, and Deville 2014) algorithm and its variants also benefits from indexing on top of the generic AC5 algorithm. Experiments suggest that AC5TC variants can be more efficient than STR2+, STR3 and Mddc under low constraint arity but slower under high arity.

Residue Support

The idea of *residue support* is to record previously found supports, called residues, then seek supports from the residues to skip some checkings. As mentioned, (G)AC3.1/AC2001 uses residues to get optimality and efficiency. AC3.2, AC3.3, and AC3rm (Lecoutre, Boussemart, and Hemery 2003; Lecoutre and Hemery 2007) further revise residues to work in a multi-directional way, i.e. when a support is found, it can be used as residue for all values occurring in the support besides the value for which it was seeking support. Newer GAC algorithms also use the residue idea, e.g. STRbit (Wang et al. 2016) and CT (Verhaeghe, Lecoutre, and Schaus 2017). We will review these two algorithms in the following subsections.

Simple Tabular Reduction

Simple Tabular Reduction (STR) (Ullmann 2007) is one of the most successful techniques for filtering table constraints. At least 15 STR based algorithms were proposed for table constraints. The idea of STR is to remove invalid tuples from tables as search goes deeper, and restore them upon backtrack. STR reduces the number of tuples of a table as search goes deeper, saving unnecessary tuple checks. Experiments showed that STR can be very effective on shrinking tables—the average reduced table size can be smaller than the original table by 1-3 orders of magnitude (Lecoutre, Likitvivatanavong, and Yap 2012).

STR2 (Lecoutre 2008; 2011) optimises the basic STR algorithm in two ways. First, the variables whose domain has not changed since the last invocation are skipped when checking the validity of a tuple. Second, support checking is skipped when all domain values are consistent. STR2+ further adds a data structure to maintain the domain size of variable at the last time a particular constraint is processed during search. Experiments in (Lecoutre 2011) show that STR2+ is about two times faster than STR and even faster than GAV-VA for the most difficult instances in their benchmarks. STR2w (Lecoutre, Likitvivatanavong, and Yap 2015a) makes STR2 more efficient by using the *watched tuple* techniques.

STR3 (Lecoutre, Likitvivatanavong, and Yap 2012; 2015b) is a STR algorithm which uses a different representation, called dual table. In the dual table, each variable-value pair is mapped to a set of tuples containing the pair. Figure 3(a) shows an example of the dual table, which is equivalent to our earlier table constraint in Figure 2(a). For each table constraint and variable value, the supports of the variable value are recorded in the dual table, e.g., $\{\tau_0, \tau_1, \tau_2, \tau_3\}$ are the supports of $(x_0, 0)$. This illustrates a different change of representation in the data structures used to represent the table (AC4 also uses a fine grained data structure). Unlike STR2, this makes STR3 a fine-grained algorithm and it is designed to maintain GAC on the dual table during search, making it incremental. STR3 has the property of being *path-optimal*, i.e. each element of a table is traversed at most once along a search path. Their experiments show that the dynamic table size has the most significant factor affecting the performance of STR3, i.e. STR3 is faster than STR2 when tables remain large (table reduction is less effective) while STR3 is slower than STR2 if the table size becomes small. This also explains why STR2 is typically faster than STR3 in benchmarks since table reduction is often quite effective and the optimality properties of STR3 may be less significant in the actual benchmark.

Some variants of STR algorithms work on compressed table representations. STR2-C and STR3-C (Xia and Yap 2013) works on the Cartesian Product representation (c-tuple) of tuples to compress tables (the "-C" refers to a c-tuple version of the algorithm). The shortSTR algorithm (Jefferson and Nightingale 2013) works on *short support* which compresses table constraint by hiding the variables whose domain values are always supported. STR-slice (Gharbi et al. 2014) compresses tables by first grouping tuples of a table (slicing) and then decomposing each group (a subtable) into two tables with a smaller arity where the original subtable is obtained from the join of the two smaller tables. Such algorithms are competitive when there exists enough compression, e.g. Xia and Yap (Xia and Yap 2013) showed that STR2-C is faster than STR2 when the Cartesian product compresses tables by more than 75%. The state-of-the-art GAC algorithms STRbit (Wang et al. 2016) and CT (Verhaeghe, Lecoutre, and Schaus 2017) also use STR ideas among others. In addition, STR is also extended to handle negative tables, such as STR-neg (Li et al. 2013) and CT-neg (Verhaeghe, Lecoutre, and Schaus 2017).

Bitwise Representation

AC3^{bit} is the first practical AC algorithm using bitwise representation as an optimization. Essentially it uses bit vectors to represent the domain and supports. The speed advantage is due to the $(O(1))$ operations available on bit vectors giving speedups due to the word size.

Wang et al. (Wang et al. 2016) proposes a bitwise encoding of the dual table representations together with the algorithms STRbit and STRbit-C. To get the bitwise representation, the original table is first partitioned so that each subtable have w tuples where w corresponds to the natural word size of processor with $O(1)$ bit vector operations. Figure 3(b) gives an example of the bitwise representations of

x_0		x_1		x_2	
0	$\{\tau_0, \tau_1, \tau_2, \tau_3\}$	0	$\{\tau_0, \tau_1\}$	0	$\{\tau_0, \tau_2, \tau_4\}$
1	$\{\tau_4, \tau_5, \tau_6\}$	1	$\{\tau_2, \tau_3\}$	1	$\{\tau_1, \tau_3, \tau_5\}$
		2	$\{\tau_4, \tau_5, \tau_6\}$	2	$\{\tau_6\}$

(a) The table representation for STR3.

x_0		x_1		x_2	
0	$\{(\theta_0, 1111)\}$	0	$\{(\theta_0, 1100)\}$	0	$\{(\theta_0, 1010), (\theta_1, 1000)\}$
1	$\{(\theta_1, 1110)\}$	1	$\{(\theta_0, 0011)\}$	1	$\{(\theta_0, 0101), (\theta_1, 0100)\}$
		2	$\{(\theta_1, 1110)\}$	2	$\{(\theta_1, 0010)\}$

(b) The table representation for STRbit. θ_0 is the index of the bit vector for τ_0 to τ_3 and θ_1 is for τ_4 to τ_6.

Figure 3: Table Representations for STR3 and STRbit

Figure 3(a). In this example, we assume $w = 4$ and partition the table into two parts with θ_0 and θ_1 as the index. For each variable value and subtable, a word in a bit vector is used to record whether the tuples in the subtable are supports of the variable value, i.e., the i^{th} bit in the word is 1 if the i^{th} tuple in the subtable is a support, otherwise the i^{th} bit is 0. If there is no support from the subtable, the word equals 0 and can be skipped. Experiments showed that STRbit was one of the state-of-art algorithms, being up to 25X faster than STR3 and 70X faster than STR2.

Compact-table (CT) is another state-of-the-art algorithm, also based on bitwise representation. Both CT and STRbit(-C) use bit vectors (sparse sets (Briggs and Torczon 1993)) to record all valid tuples in a table (non-zero words in the bit vectors) during search. The difference between CT and STRbit(-C) is that for each table constraint, CT does not skip the zero words in the bit vectors recording supports of variable values, thus, those bit vectors have the same number of words and can share the sparse set which records non-zero words in the bit vector recording valid tuples. In addition, the STRbit(-C) and CT algorithms also record the last found support for each variable value, which is similar to algorithms such as (G)AC3.1/AC2001, AC3r (Lecoutre and Hemery 2007) and residue-based techniques (Lecoutre et al. 2008). From Figure 1, we can see the overall performance of CT and STRbit is quite close and clearly superior to the other algorithms. The principles of CT was also extended to the decision diagram based algorithm Compact-MDD in (Verhaeghe, Lecoutre, and Schaus 2018). Although Compact-MDD is still slower than CT, it reduces the gap between MDD based algorithms and CT. The HTAC (Wang and Yap 2019) algorithm, discussed in the last part, also uses bit vectors (sparse sets).

Multi-valued Decision Diagram

Besides the table-based representation, we can convert any table constraint into an equivalent MDD (Cheng and Yap 2010). In the best case, the MDD can get exponential compression. Like a tree/trie, a MDD can get compression due to sharing in the tree part of the graph but it also gets sharing of children. In practice, it was observed that the more compact the MDD, the faster the MDD based filtering algorithms become. In an analog to table support, a value is GAC if there exists a path of valid domain values from the root node to true terminal.

A number of MDD based algorithms have been proposed, including Mddc,incremental-MDD (Gange, Stuckey, and Szymanek 2011), MDD4 (Perez and Régin 2014), Compact-MDD (Compact-Diagram) (Verhaeghe, Lecoutre, and Schaus 2018; 2019), and BDDF (Vion and Piechowiak 2018). The Mddc is the first MDD based filtering algorithm. It recursively traverses the MDD in a top-bottom manner and enforces GAC during the process. At each MDD node, Mddc records the consistency of the node, i.e. whether the node can reach a true terminal through a path whose values are all present in variables' domains. This guarantees that Mddc explores each MDD node at most once. Another optimization technique is the *early-cutoff*, which remembers the level at and below of the MDD where all values in the domains of variables are consistent. As a result, Mddc can skip the unexplored sub-parts of any consistent node at or below the level. The MDD data structure also serves as a natural index. MDD4 (Perez and Régin 2014) adds incrementality by maintaining the validity of MDD edges kept as the supports of domain values. The *reset* technique is introduced (see later) which gives two algorithms: GAC4R and MDD4R.

Incrementality

For the most incrementality techniques, some additional data structures are maintained during search, in order to avoid repeatedly accessing invalid supports (tuples). A motivation is that if a support of a variable value is invalid, then it is still invalid after shrinking some variable domains.

Incrementality techniques appear in many (G)AC algorithms: (i) In the AC4 algorithm, the number of valid supports of variable values are recorded during search, correspondingly a variable value is not AC if the number of valid supports on a constraint is zero; (ii) In the AC6, AC7, GAC-scheme based, (G)AC2001/3.1/3.2/3.3, STR3, STRbit and STRbit-C algorithms, the last valid support of each variable value is maintained, thus, we only need to consider the supports before the last support when the last support becomes invalid; (iii) In the STR based algorithms, current tables, i.e., invalid tuples are removed, are maintained by using sparse (bit-)sets; (iv) In the GAC4 algorithm (Mohr and Masini 1988), the valid supports of each variable value are maintained by using linked lists or sparse sets; (v) In the Mddc, MDD4(R), incremental-MDD (Gange, Stuckey, and Szymanek 2011) and BDDF (Vion and Piechowiak 2018) algorithms, valid nodes are maintained by using various data structures, in addition, valid edges are also maintained in MDD4(R) and incremental-MDD. Note that a valid tuple corresponds to a path consisting of valid nodes and edges.

Reset

The *reset* technique is introduced in GAC4R and MDD4R, and used in many recent GAC algorithms, such as STRbit-C, CT, smartCT, Compact-MDD, Compact-smartMDD and HTAC. The idea is to choose whether to incrementally maintain deletions or to rebuild the data structures from scratch.

Watched Tuple

The *watched tuple* technique, inspired by SAT solvers, is to associate each watched tuple with a list of values that depend on this tuple as a proof of support. This simplifies checking whether a support is no longer correct. Should a tuple be removed, its watchers must be reattached, if possible, to another valid tuple. The algorithms applying such technique include AC6, AC7, GAC-scheme, GAC-nextIn, GAC-nextDiff, Trie, STR3, STR3-C and STR2w.

Other Representations

The intuition behind employing compact representations is that significant compression of tables should reduce running time for enforcing GAC. Thus besides MDD and bitwise based algorithms, more compact representations were proposed to revise existing GAC algorithms, such as the c-tuples, short-supports, slice-tables, and smart-tables (Mairy, Deville, and Lecoutre 2015). The corresponding GAC algorithms of different compact representations includes: GAC-ctuple (Katsirelos and Walsh 2007), STR2-C and STR3-C (Xia and Yap 2013), and STRbit-C (Wang et al. 2016) for c-tuples; shortSTR2 (Jefferson and Nightingale 2013) and shortCT (Verhaeghe, Lecoutre, and Schaus 2017) for short-supports; STR-slice (Gharbi et al. 2014) for slice-table; smartSTR (Mairy, Deville, and Lecoutre 2015) and smartCT (Verhaeghe et al. 2017) for smart-tables.

5 CSP Encodings

A CSP can also be solved by first transforming it into another CSP, which we will call an encoding.

Binary Encodings

There are two ways of solving a non-binary CSP. The first way, the typical one, is to solve the non-binary CSP instance with a solver employing filtering algorithms usually based on GAC. An alternative second way is to use a binary encoding transforming a non-binary CSP into a "solution equivalent" binary CSP so that binary-only techniques such as AC can be applied to the encoded binary CSP. Two well known binary encodings are *dual encoding* (Dechter and Pearl 1989) and *hidden variable encoding* (HVE) (Rossi, Petrie, and Dhar 1990). Most research (over the past decade and more) has been on the former as it was believed that binary encoding is not practical. The reason for this belief is shown with experiments in Wang and Yap (Wang and Yap 2019) showing that dual encoding takes too much space and HVE with the best AC algorithms ($AC3^{bit}$ and HAC (Mamoulis and Stergiou 2001; Samaras and Stergiou 2005)) are considerably outperformed by CT on the original non-binary CSP. However the HTAC algorithms in (Wang and Yap 2019) were shown to be competitive with state-of-art CT and STRbit.

HTAC is efficient being a specialized AC algorithm which exploits properties of binary encoding instances and employs techniques from modern GAC algorithms, such as CT and STRbit. At the same time, HTAC takes into account the effect of the filtering algorithm on the search heuristic. It allows the solver to do search on binary encoded models, and also the original model.

Stronger Consistency by Encodings

Another way of using encoding is to enforce a stronger consistency. The idea is that the encoded CSP is such that enforcing GAC on encoding leads to a stronger consistency on the original CSP. The benefit is that rather than designing a new higher-order consistency, existing GAC algorithms and solvers can be directly used.

The *factor encoding* (FE) (Likitvivatanavong, Xia, and Yap 2014) and *factor decomposition encoding* (FDE) (Likitvivatanavong, Xia, and Yap 2015) factor out the commonly shared variables between each pair of constraints, then create new variable for each set of the share variables. In FE, the new variable in the encoding returns back to the constraint where it comes from. One drawback of FE is that the scope of constraint is enlarged leading to (much) larger tables. To alleviate space issues, FDE was proposed to decompose the constraints by subtracting the created variable together with its original variables. GAC on FD and FDE is equivalent to the higher order consistency, *full pairwise consistency* (Lecoutre, Paparrizou, and Stergiou 2013) on the original CSP. The advantage of enforcing the stronger consistency with GAC on the encoded problem is that the size of the search space can be much smaller than with GAC on the original problem. In some cases, there is no search or a tiny search space.

6 Conclusion

Advances in (G)AC algorithms have led to orders of magnitude improvements in the efficiency of constraint solvers. We can see in Table 1 that newer algorithms such as the ones in bold use combinations of techniques and are more "sophisticated" than older ones. Older algorithms focused more on worst case time complexity while newer algorithms take advantage of features in CSP instances, e.g. constraint representations (e.g. compact representations), internal representations (e.g. dual table, bitwise). More sophisticated data structures are used, e.g. sparse sets, Index. Runtime optimizations which take advantage of making certain operations more efficient, e.g. STR, residues, efficient bit operations. The granularity of the processing has mostly shifted to be more fine-grained. Some forms of higher order consistency can also benefit from the improvements in G(AC) algorithms. In particular, the stronger consistency encodings tend to enlarge the arity of the encoded constraint, making them more suited for GAC algorithms which deal better with larger arity constraints.

Recent work with the HTAC algorithm (Wang and Yap 2019) suggests that although most research has focused on GAC algorithms rather than AC algorithms over the past decades, the binary case may need to be revisited. Indeed, HTAC uses techniques used in recent GAC algorithms, but specialised to the binary case and the hidden variable encoding. As more results and (stronger) consistencies have been developed for binary CSPs than for non-binary CSPs, it opens up possibilities for new consistency algorithms. Finally, global constraints is outside this survey but are important. Some of the constraints we discussed can be viewed as global constraints but an interesting and little explored direction is to have better ways of combining GAC algorithms for global constraint with the ones for various forms of table constraints.

Acknowledgements

This research is supported by the National Research Foundation Singapore under its AI Singapore Programme (Award Number: AISG-100E/RP/GC/RPKS-2018-005).

References

Bessière, C., and Régin, J.-C. 1997. Arc consistency for general constraint networks: preliminary results. In *Proceedings of the Fifteenth international Joint Conference on Artificial Intelligence*, 398–404.

Bessière, C., and Régin, J.-C. 2001. Refining the basic constraint propagation algorithm. In *Proceedings of the Seventeenth international Joint Conference on Artificial Intelligence*, 309–315.

Bessière, C.; Régin, J.-C.; Yap, R. H. C.; and Zhang, Y. 2005. An optimal coarse-grained arc consistency algorithm. *Artificial Intelligence* 165(2).

Bessiere, C.; Freuder, E. C.; and Regin, J.-C. 1995. Using inference to reduce arc consistency computation. In *Proceedings of the Fourteenth international Joint Conference on Artificial Intelligence*, 592–598. Morgan Kaufmann Publishers Inc.

Bessiere, C. 2006. Constraint propagation. In Rossi, F.; van Beek, P.; and Walsh, T., eds., *Handbook of Constraint Programming*. Elsevier Science. chapter 2.

Bessière, C. 1994. Arc-consistency and arc-consistency again. *Artificial Intelligence* 65(1):179 – 190.

Briggs, P., and Torczon, L. 1993. An efficient representation for sparse sets. *ACM Letters on Programming Languages and Systems* 2(1–4):59–69.

Cheng, K. C. K., and Yap, R. H. C. 2008. Maintaining generalized arc consistency on ad hoc r-ary constraints. In *Proceedings of the 14th international Conference on Principles and Practice of Constraint Programming*.

Cheng, K. C. K., and Yap, R. H. C. 2010. An MDD-based generalized arc consistency algorithm for positive and negative table constraints and some global constraints. *Constraints* 15(2):265–304.

Cheng, K. C.; Xia, W.; and Yap, R. H. 2012. Space-time tradeoffs for the regular constraint. In *International Conference on Principles and Practice of Constraint Programming*, 223–237. Springer.

Dechter, R., and Pearl, J. 1989. Tree clustering for constraint networks. *Artificial Intelligence* 38:353–366.

Dechter, R. 2003. *Constraint Processing*. Morgan Kaufmann.

Demeulenaere, J.; Hartert, R.; Christophe Lecoutre, G. P.; Perron, L.; Regin, J.-C.; and Schaus, P. 2016. Compact-table: Efficiently filtering table constraints with reversible sparse bit-sets. In *Proceedings of the 22nd international Conference on Principles and Practice of Constraint Programming*, 207–223.

Freuder, E. C., and Mackworth, A. K. 2006. Constraint satisfaction: An emerging paradigm. In Rossi, F.; van Beek, P.; and Walsh, T., eds., *Handbook of Constraint Programming*. Elsevier Science. chapter 3.

Gange, G.; Stuckey, P. J.; and Szymanek, R. 2011. Mdd propagators with explanation. *Constraints* 16(4):407.

Gent, I. P.; Jefferson, C.; Miguel, I.; and Nightingale, P. 2007. Data structures for generalised arc consistency for extensional constraints. In *Proceedings of the 22nd AAAI Conference on Artificial Intelligence*, 191–197.

Gharbi, N.; Hemery, F.; Lecoutre, C.; and Roussel, O. 2014. Sliced table constraints: Combining compression and tabular reduction. In *Proceedings of the 11th International Conference on the Integration of AI and OR Techniques in Constraint Programming*.

Jaffar, J., and Maher, M. J. 1994. Constraint logic programming: A survey. *Journal of Logic Programming* 19(20):503–581.

Jefferson, C., and Nightingale, P. 2013. Extending simple tabular reduction with short supports. In *Proceedings of the 23rd International Joint Conference on Artificial Intelligence*, 573–579.

Katsirelos, G., and Walsh, T. 2007. A compression algorithm for large arity extensional constraints. In *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming*.

Lecoutre, C., and Cordier, M.-O. 1993. Arc-consistency and arc-consistency again. In *Proceedings of the 11th National Conference on Artificial Intelligence*.

Lecoutre, C., and Hemery, F. 2007. A study of residual supports in arc consistency. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence*.

Lecoutre, C., and Szymanek, R. 2006. Generalized arc consistency for positive table constraints. In *Proceedings of the 12th International Conference on Principles and Practice of Constraint Programming*, 284–298.

Lecoutre, C., and Vion, J. 2008. Enforcing arc consistency using bitwise operations. *Constraint Programming Letters* 2:21–35.

Lecoutre, C.; Likitvivatanavong, C.; Shannon, S. G.; Yap, R. H. C.; and Zhang, Y. 2008. Maintaining arc consistency with multiple residues. *Constraint Programming Letters* 2:3–19.

Lecoutre, C.; Boussemart, F.; and Hemery, F. 2003. Exploiting multidirectionality in coarse-grained arc consistency algorithms. In *Proceedings of the 9th International Conference on Principles and Practice of Constraint Programming*, 480–494.

Lecoutre, C.; Likitvivatanavong, C.; and Yap, R. H. C. 2012. A path-optimal GAC algorithm for table constraints. In *Proceedings of the 20th European Conference on Artificial Intelligence*.

Lecoutre, C.; Likitvivatanavong, C.; and Yap, R. H. C. 2015a. Improving the lower bound of simple tabular reduction. *Constraints* 20:100–108.

Lecoutre, C.; Likitvivatanavong, C.; and Yap, R. H. C. 2015b. STR3: A path-optimal filtering algorithm for table constraints. *Artificial Intelligence* 220:1–27.

Lecoutre, C.; Paparrizou, A.; and Stergiou, K. 2013. Extending str to a higher-order consistency. In *Proceedings of the 27th AAAI Conference on Artificial Intelligence*.

Lecoutre, C. 2008. Optimization of simple tabular reduction for table constraints. In *Proceedings of the 14th international Conference on Principles and Practice of Constraint Programming*, 128–143.

Lecoutre, C. 2011. STR2: Optimized simple tabular reduction for table constraints. *Constraints* 16(4):341–371.

Lhomme, O., and Régin, J.-C. 2005. A fast arc consistency algorithm for n-ary constraints. In *Proceedings of the 20th National Conference on Artificial Intelligence*.

Li, H.; Liang, Y.; Guo, J.; and Li, Z. 2013. Making simple tabular reductionworks on negative table constraints. In *Proceedings of the 27th AAAI Conference on Artificial Intelligence*.

Likitvivatanavong, C.; Xia, W.; and Yap, R. H. C. 2014. Higher-order consistencies through GAC on factor variables. In *Proceedings of the 20th International Conference on Principles and Practice of Constraint Programming*, 497–513.

Likitvivatanavong, C.; Xia, W.; and Yap, R. H. C. 2015. Decomposition of the factor encoding for csps. In *Proceedings of the 24th International Joint Conference on Artificial Intelligence*, 353–359.

Mackworth, A. K. 1977. Consistency in networks of relations. *Artificial Intelligence* 8(1):99–118.

Mairy, J.-B.; Deville, Y.; and Lecoutre, C. 2015. The smart table constraint. In *Proceedings of the 12th International Conference on the Integration of AI and OR Techniques in Constraint Programming*.

Mairy, J.-B.; Hentenryck, P. V.; and Deville, Y. 2014. Optimal and efficient filtering algorithms for table constraints. *Constraints* 19(1):77–120.

Mamoulis, N., and Stergiou, K. 2001. Solving non-binary csps using the hidden variable encoding. In *Proceedings of the 7th International Conference on Principles and Practice of Constraint Programming*, 168–182. Springer.

Mohr, R., and Henderson, T. C. 1986. Arc and path consistency revisited. *Artificial Intelligence* 28(2):225–233.

Mohr, R., and Masini, G. 1988. Good old discrete relaxation. In *Proceedings of the 8th European Conference on Artificial Intelligence*, 651–656. Pitman Publishing, Inc.

Perez, G., and Régin, J.-C. 2014. Improving GAC-4 for table and mdd constraints. In *Proceedings of the 20th International Conference on Principles and Practice of Constraint Programming*, 606–621.

Pesant, G. 2004. A regular language membership constraint for finite sequences of variables. In *Proceedings of the 10th International Conference on Principles and Practice of Constraint Programming*, 482–495. Springer.

Régin, J.-C. 2011. *Global Constraints: A Survey*. New York, NY: Springer New York. 63–134.

Rossi, F.; Petrie, C.; and Dhar, V. 1990. On the equivalence of constraint satisfaction problems. In *Proceedings of the 9th European Conference on Artificial Intelligence*, 550–556.

Samaras, N., and Stergiou, K. 2005. Binary encodings of non-binary constraint satisfaction problems: Algorithms and experimental results. *Journal of Artificial Intelligence Research* 24:641–684.

Sellmann, M. 2006. The theory of grammar constraints. In *Proceedings of the 12th International Conference on Principles and Practice of Constraint Programming*, 530–544. Springer.

Ullmann, J. R. 2007. Partition search for non-binary constraint satisfaction. *Information Science* 177(18):3639–3678.

Van Hentenryck, P.; Deville, Y.; and Teng, C.-M. 1992. A generic arc-consistency algorithm and its specializations. *Artificial Intelligence* 57(2-3):291–321.

van Hoeve, W.-J., and Katriel, I. 2006. Global constraints. In Rossi, F.; van Beek, P.; and Walsh, T., eds., *Handbook of Constraint Programming*. Elsevier Science. chapter 2.

van Hoeve, W. J. 2001. The alldifferent constraint: A survey. In *Proceedings of Sixth Annual Workshop of the ERCIM Working Group on Constraints*.

Verhaeghe, H.; Lecoutre, C.; Deville, Y.; and Schaus, P. 2017. Extending compact-table to basic smart tables. In *Proceedings of the 23rd International Conference on Principles and Practice of Constraint Programming*.

Verhaeghe, H.; Lecoutre, C.; and Schaus, P. 2017. Extending compact-table to negative and short table. In *Proceedings of the 31st AAAI Conference on Artificial Intelligence*, 3951–3957.

Verhaeghe, H.; Lecoutre, C.; and Schaus, P. 2018. Compact-MDD: Efficiently filtering (s)MDD constraints with reversible sparse bit-sets. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence*.

Verhaeghe, H.; Lecoutre, C.; and Schaus, P. 2019. Extending compact-diagram to basic smart multi-valued variable diagrams. In *Proceedings of the 16th International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, 581–598.

Vion, J., and Piechowiak, S. 2018. From mdd to bdd and arc consistency. *Constraints* 23(4):451–480.

Wallace, R. J. 1993. Why AC-3 is almost always better than AC-4 for establishing arc consistency in CSPs. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence*, 239–247.

Wang, R., and Yap, R. H. 2019. Arc consistency revisited. In *Proceedings of the 16th International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, 599–615. Springer.

Wang, R.; Xia, W.; Yap, R. H. C.; and Li, Z. 2016. Optimizing simple tabular reduction with a bitwise representation. In *Proceedings of the 25th International Joint Conference on Artificial Intelligence*, 787–795.

Xia, W., and Yap, R. H. C. 2013. Optimizing STR algorithm with tuple compression. In *Proceedings of the 19th International Conference on Principles and Practice of Constraint Programming*, 724–732.

Zhang, Y., and Yap, R. H. C. 2001. Making AC-3 an optimal algorithm. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence*, 316–321.