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

Roland H. C. Yap,1 Wei Xia, Ruiwei Wang1
1School 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 Regin 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

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

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 \( \tau \) is a support of \( (x, a) \) on \( c \) iff \( \tau[x] = a \) and \( \tau \) 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 if 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 \( \tau \) for \( (x, a) \) in \( c \). A constraint \( c \) is GAC iff \( \forall \tau \in D(x), \forall x \in scope(c) \) 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 searched 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 alternative representations for a table are possible, such as the graph based representations: deterministic finite automata (DFA a.k.a regular) (Pessay 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,
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 even though 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 (Bessière, 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 Régis 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$^{bit}$ (Lecoutre and Vion 2008) is the first practical AC algorithm exploiting bitwise operations, AC3$^{bit+rm}$ (Lecoutre and Vion 2008) further extends AC3$^{bit}$ 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 residues, 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 Régis 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-nextLin (Lhomme and Régis 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$^{bit+rm}$ is named as AC3$^{bit+rm}$ 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 Mdde 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 AC3m (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

AC3bit 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
the table into two parts with Figure 3(a). In this example, we assume in the subtable is a support, otherwise the bit vectors (sparse sets).

MDD in (Verhaeghe, Lecoutre, and Schaus 2018). Although tended to the decision diagram based algorithm Compact-(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.

<table>
<thead>
<tr>
<th>$x_0$</th>
<th>$x_1$</th>
<th>$x_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>${t_0, t_1}$</td>
<td>${t_0, t_1}$</td>
</tr>
<tr>
<td>1</td>
<td>${t_4, t_5, t_6}$</td>
<td>${t_4, t_5, t_6}$</td>
</tr>
</tbody>
</table>

(a) The table representation for STR3.

<table>
<thead>
<tr>
<th>$x_0$</th>
<th>$x_1$</th>
<th>$x_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>${t_0, 1111}$</td>
<td>${t_0, 1100}$</td>
</tr>
<tr>
<td>1</td>
<td>${t_1, 1111}$</td>
<td>${t_0, 0001}$</td>
</tr>
</tbody>
</table>

(b) The table representation for STRbit. $\theta_0$ is the index of the bit vector for $t_0$ to $t_1$ and $\theta_1$ is for $t_4$ to $t_6$.

Figure 3: Table Representations for STR3 and STRbit.

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.
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\textsuperscript{fast} 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 a 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.
nally, 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: A1Sg-100E/RP/GC/RPKS-2018-005).

References


13596


