Arc Consistency Revisited

Ruiwei Wang and Roland H.C. Yap

National University of Singapore, Singapore
{ruiwei,ryap}@comp.nus.edu.sg

Abstract. Binary constraints are a general representation for constraints and is used in Constraint Satisfaction Problems (CSPs). However, many problems are more easily modelled with non-binary constraints (constraints with arity > 2). Several well-known binary encoding methods can be used to transform non-binary CSPs to binary CSPs. Historically, work on constraint satisfaction began with binary CSPs with many algorithms proposed to maintain Arc Consistency (AC) on binary constraints. In more recent times, research has focused on non-binary constraints and efficient Generalized Arc Consistency (GAC) algorithms for non-binary constraints. Existing results and "folklore" suggest that AC algorithms on the binary encoding of a non-binary CSP do not compete with GAC algorithms on the original problem. We propose new algorithms to enforce AC on binary encoded instances. Preliminary experiments show that our AC algorithm on the binary encoded instances is competitive to state-of-the-art GAC algorithms on the original non-binary instances and faster in some instances. This result is surprising and is contrary to the "folklore" on AC versus GAC algorithms. We believe our results can lead to a revival of AC algorithms as binary constraints and resulting algorithms are simpler than the non-binary ones.

Keywords: Binary constraint · Binary encoding · Arc Consistency · Generalized Arc Consistency · CSP

1 Introduction

Binary constraint is a general representation for constraints and is used in Constraint Satisfaction Problems (CSPs) to model/solve any discrete combinatorial problem. Historically, work on constraint satisfaction began with binary CSPs, problems with at most two variables per constraint and many algorithms have been proposed to maintain Arc Consistency (AC) on binary constraints. The seminal work of Mackworth [16] proposed a basic local consistency, arc consistency, which has been the main reasoning technique used in constraint solvers for CSPs. However, many problems are more naturally modelled with non-binary constraints (constraints with arity $>$ 2). Several well-known binary encoding methods can be used to transform non-binary CSPs to binary CSPs. Non-binary CSPs can be also solved directly which would require non-binary constraint solvers, Generalized Arc Consistency (GAC) is the natural extension of AC. In more recent times, research has focused on non-binary constraints and efficient Generalized Arc Consistency (GAC) algorithms using clever algorithms, representations and data structures [22, 10, 12, 18, 3, 25, 27, 9, 8, 6, 26, 5, 24, 23]. Improvements in GAC algorithms have led to a "folklore belief" that AC algorithms on the binary encoding of a non-binary CSP do not compete with GAC.[1] It has also spurred major developments in GAC algorithms.

We first show with experimental comparisons of binary encoding with state-of-art GAC algorithms reasons why binary encoding with existing AC algorithms are outperformed by GAC on non-binary constraints. We propose new algorithms to enforce AC on binary encoded instances which address these factors: (i) a more efficient propagator for hidden variable binary constraints; and (ii) control the interaction between the search heuristic and the binary encoded model. Preliminary experiments show that our AC algorithm can be much faster than state-of-the-art AC algorithms for non-binary CSPs, CT[5] and STRbit [26], on their binary encoded instances. This result is surprising and is contrary to the "folklore" on AC vs GAC algorithms. We believe our results can lead to a revival of AC algorithms since binary constraints and resulting algorithms are simpler than the non-binary ones. For example, many stronger consistencies were proposed to handle binary constraints and these have been more extensively studied in the case of binary constraints. Many fundamental works studying properties of CSPs are often also studied in the binary case.

2 Background

A *CSP* (*Constraint Satisfaction Problem*) $\mathcal{P}$ is a pair $(\mathcal{X}, \mathcal{C})$ with n variables, $\mathcal{X} = \{x_1, x_2, \dots\ x_n\}$, and m constraints, $\mathcal{C}\ \{c_1, c_2, \dots\ c_m\}$. The variable domains are finite, $D(x_i)$ is the domain of $x_i \in \mathcal{X}$. We distinguish the current domain of x_i, $dom(x_i) \subseteq D(x_i)$, the domain may shrink during search when solving the CSP. The variables in each constraint c_i is called the *constraint scope*, $scp(c_i) = \{x_{i_1}, x_{i_2}, \dots\ x_{i_r}\}$ and r is the (constraint) arity. The constraint is a relation defined over the constraint scope, $rel(c_i) \subseteq \prod_{j=1}^{r} D(x_{i_j})$. In this paper, we only consider non-trivial constraints, hence, $r > 1$. A constraint c is a *binary constraint* iff $r = 2$, i.e $scp(c) = \{x, y\}$, otherwise, c is a *non-binary constraint* iff $r > 2$. A *binary CSP* only has binary constraints; otherwise the CSP is a *non-binary CSP*. An assignment $\mathcal{A} = \{(x_1, a_1), (x_2, a_2), \dots\ (x_n, a_n)\}$ satisfies a constraint c iff $\mathcal{A}[scp(c)] \in rel(c)$ where the notation $[v]$ denotes projection on the set of variables v. Then $\mathcal{A}$ is a solution satisfying $(\mathcal{X}, \mathcal{C})$ iff $\mathcal{A}$ satisfies all constraints in $\mathcal{C}$ and $\mathcal{A} \in \prod_{i=1}^{n} dom(x_i)$. Following [19], we say a CSP $\mathcal{P}_1$ is equivalent to $\mathcal{P}$ if they are mutually reducible. A CSP $\mathcal{P}$ is *reducible* to another CSP $\mathcal{P}_1$ if the solution of $\mathcal{P}$ can be obtained from the solution of $\mathcal{P}_1$, by mapping the variable values in one CSP to variable values in the other.

A tuple $\tau \in rel(c)$ is *valid* iff $\tau[x] \in dom(x)$ for all $x \in scp(c)$. We say (x, a) is a *support* of tuple $\tau \in rel(c)$ iff $\tau[x] = a$. A variable value (x, a) is *generalized*

[1] We focus on AC and GAC algorithms for the general finite domain CSPs with table constraints. Global constraints with special semantics and special GAC algorithms exploiting the semantics are outside our scope.

arc consistent (GAC) on c iff (x, a) has a valid support in $rel(c)$. A variable x is GAC on c iff for all value $a \in dom(x)$, (x, a) is GAC. A constraint c is GAC iff all variables in $scp(c)$ is GAC on c. A CSP $(\mathcal{X}, \mathcal{C})$ is GAC iff all constraints in $\mathcal{C}$ is GAC. A binary CSP $\mathcal{P}$ is *arc consistent* (AC) iff $\mathcal{P}$ is GAC, i.e. arc consistency is a special case of GAC. For a binary constraint c, arc consistency uses a simpler definition of support: a value $a \in dom(x)$ has a valid support in rel(c) iff (x, a) has a valid support b in $dom(y)$ such that $\{(x, a), (y, b)\} \in rel(c)$, where $scp(c) = \{x, y\}$. M(G)AC is used to denote maintaining (G)AC during search. In this paper, we focus on MGAC and MAC and simply say GAC or AC.

2.1 Binary encodings

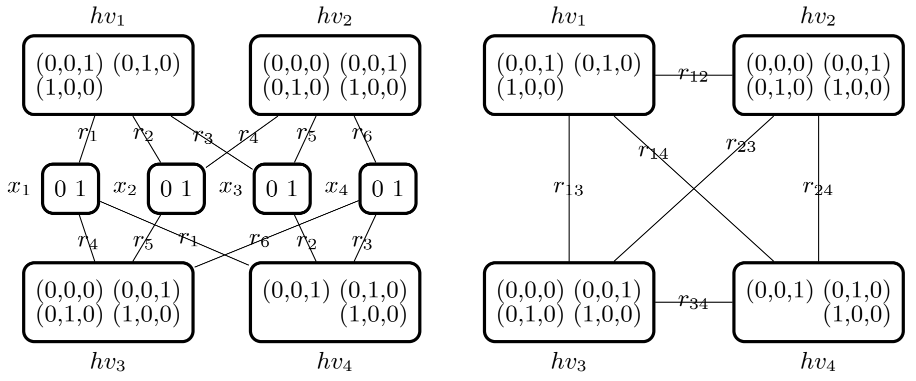
A non-binary CSP $\mathcal{P}_1 = (\mathcal{X}_1, \mathcal{C}_1)$ can be solved through transformation by encoding into an "equivalent" binary CSP $\mathcal{P}_2 = (\mathcal{X}_2, \mathcal{C}_2)$ such that $\mathcal{P}_2$ is reducible to $\mathcal{P}_1$. This means there are two options to solving a non-binary CSP P_1: (i) directly solving P_1; or (ii) indirectly by solving P_2. There are two well known binary encodings, namely, the *dual encoding* [4] and the *hidden variable encoding* (HVE) [19]:
- the dual encoding of P_1 is a binary CSP $(\mathcal{H}, \mathcal{DC})$
- the HVE of P_1 is a binary CSP $(\mathcal{X}_1 \cup \mathcal{H}, \mathcal{HC})$

with new variables $\mathcal{H} = \{hv_i | c_i \in \mathcal{C}_1\}$ where the domain of hv_i is the tuples of the c_i itself, $D(hv_i) = rel(c_i)$. Variables $\mathcal{H}$ are called *hidden variables* and also sometimes called dual variables [21, 20]. In the dual encoding, the new constraints are $\mathcal{DC} = \{c_{ij} | s = scp(c_i) \cap scp(c_j) \neq \emptyset\}$, i.e. $scp(c_{ij}) = \{hv_i, hv_j\}$, and $rel(c_{ij}) = \{(\tau_1 \in rel(c_i), \tau_2 \in rel(c_j)) \mid \tau_1[s] = \tau_2[s]\}$. The hidden variable encoding has constraints $\mathcal{HC} = \{c_i^x | x \in scp(c_i)\}$, one new constraint per variable in c_i, i.e. $scp(c_i^x) = \{x, hv_i\}$, and $rel(c_i^x) = \{(a \in D(x), \tau \in rel(c_i)) | \tau[x] = a\}$.

Example 1 *Consider a CSP P $(\mathcal{X}, \mathcal{C})$, where $\mathcal{X} = \{x_1, \ldots, x_4\}$, $D(x_i) = \{0, 1\}$ and $\mathcal{C} = \{c_1 : x_1 + x_2 + x_3 = 1,\ c_2 : x_2 + x_3 + x_4 < 2,\ c_3 : x_1 + x_2 + x_4 < 2,\ c_4 : x_1 + x_3 + x_4 = 1\}$. Figure 1(a) gives the HVE CSP instance of P, and Figure 1(b) is the dual encoding instance. Every node in the figure is a variable, each edge corresponds to a binary constraint, and the label of the edge denotes the relation of the binary constraint. E.g. $D(hv_1) = D(hv_4) = \{1', 2', 4'\}$ and $D(hv_2) = D(hv_3) = \{0', 1', 2', 4'\}$, where the values $0'$, $1'$, $2'$ and $4'$ represent $(0,0,0), (0,0,1), (0,1,0)$ and $(1,0,0)$ respectively (the figure uses the tuples notation). The constraint $r_1 = \{(0, 1'), (0, 2'), (1, 4')\}$ is the relation in the HVE constraints with scope $\{x_1, hv_1\}$ and $\{x_1, hv_3\}$ in the HVE while $r_{13} = \{(1', 2'), (2', 4'), (4', 0'), (4', 1')\}$ is the relation in the dual encoding with scope $\{hv_1, hv_2\}$.*

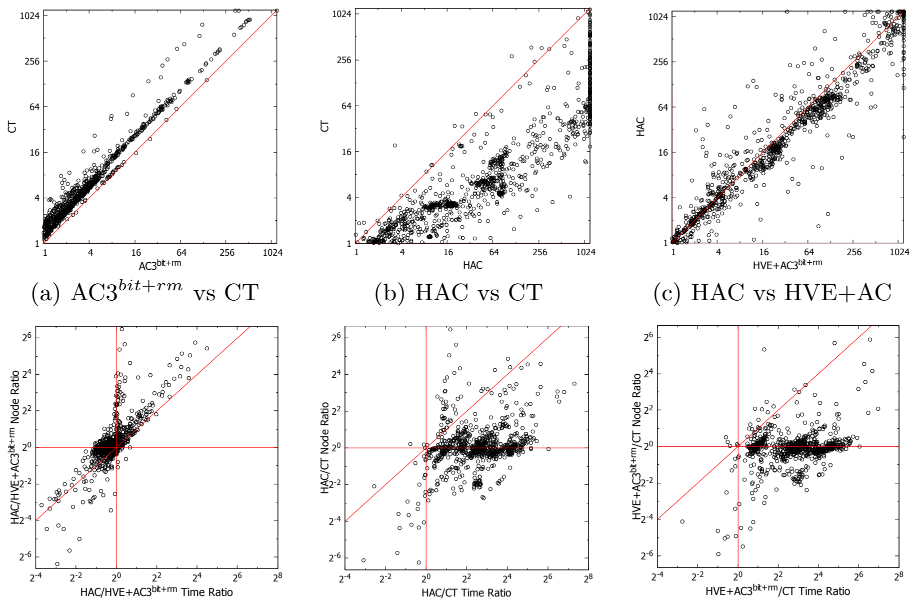
3 History and The Problem

In this paper, we revisit the question whether non-binary CSPs are better solved directly using a non-binary solver or the non-binary CSP is encoded to a new



(a) Hidden variable encoding (b) Dual encoding

Fig. 1. Binary encodings



(a) AC3^{bit+rm} vs CT (b) HAC vs CT (c) HAC vs HVE+AC

(d) HAC vs HVE+AC(nodes+time) (e) HAC vs CT(nodes+time) (f) HVE+AC vs CT(nodes+time)

Fig. 2. AC vs GAC algorithm: (a)-(c) use time on the axis while (d)-(f) use time and node ratios

Series	100%	80%	60%	40%	20%
Rand-5-12	50/50	50/50	50/50	50/50	50/50
Rand-15-23	25/25	25/25	25/25	25/25	25/25
Rand-10-60	50/50	50/50	50/50	50/50	50/50
Nonogram	88/170	78/170	71/170	53/170	40/170
Tsp	15/45	15/45	15/45	15/45	15/45
Jnh	50/50	40/50	25/50	2/50	0/50
Mdd-7-25	25/25	25/25	25/25	25/25	25/25
Rand-3-20	30/50	0/50	0/50	0/50	0/50

Table 1. Dual encoding memory size results: m/n means m instances were memory-out of n total, x% are fraction of original constraints

binary CSP and solved by a binary constraint solver. We focus on the comparison between binary constraints and table constraints which are the most general representation for constraints in CSPs. We start with a chronology of binary encodings and corresponding consistency algorithms (if any). In 1998, [1] showed on some instances, forward checking (FC) with backtracking search on binary encoded instances can be faster than solving the non-binary instances directly. In 1999, experimental results in [21] showed that enforcing AC on the dual encoding instances is very expensive. In 2001, [17] proposed HAC showing that MHAC is competitive with M(G)AC, and M(G)AC algorithms can be mapped to the corresponding AC algorithms on binary encoded instances. In 2005, [20] showed that binary encodings are competitive with the non-binary representation. It also showed that a higher order consistency PW-AC can work well on binary encoding instances. MHAC-2001 and PW-AC can be faster than MGAC-2001 in some cases. However, they only tested some special cases for the dual encoding. In 2011, [10] showed that the dual and double encodings run out of memory on many instances, and STR2+ can outperform HAC and HVE+AC3^{bit+rm}.[2]

During the past decades, many AC and table GAC algorithms were proposed. AC algorithms check whether a variable value has a valid support on another variable domain, and many methods are proposed to reduce the cost of AC consistency algorithm during search. Over the period from 2007-2018, there has been considerable research efforts expanded on GAC algorithms, but little work on AC algorithms given the shift to GAC algorithms. Many of the GAC algorithms use special ideas to make GAC more efficient. An incomplete list is as follows: reducing the size of tables during search, e.g. algorithms using simple table reduction during search[22, 10, 12], algorithms using decision diagrams [18, 3, 25], algorithms using compressed tables [27, 9, 8, 6], and bit set representations [26, 5, 24, 23]. Some of the state-of-the-art GAC algorithms are CT[5] and STR-bit[26] which combine bit set with simple table reduction can be much faster than STR2+ [10].

[2] (HVE+AC3^{bit+rm} means using the AC3^{bit+rm} on the HVE binary instances).

In order to understand and revisit the results from existing papers such as the ones above, we test various kinds of instances to compare different existing algorithms. Experimental details are given in Section 5 to avoid repetition. The main drawback of dual encoding is the large constraints which lead to the solver running out of memory, which we call *memory-out*. This was also shown in [10]. We also tested with various instances, Table 1, where the 100% column is the original instance and a random set of constraints removed creating a smaller CSP. Many instances are simply memory-out, e.g. with jnh instances at 60% (40% constraint removal) still 25 instances were memory-out. In this paper, we focus on the hidden variable encoding as the dual encoding starts to become infeasible as the constraints become larger.

We revisit GAC vs binary encodings for hidden variables with results in Figure 2. Each dot in the graphs is a problem instance. For AC, we employ $AC3^{bit}$[15] with residues [11] denoted as $AC3^{bit+rm}$. It has been shown to be efficient in practice for AC because of the bit representation [13]. Figure 2(a) compares time (in seconds) of AC^{bit+rm} with CT on binary CSP instances. It shows that $AC3^{bit+rm}$ can be faster than a recent state-of-the-art GAC algorithm, CT, on binary CSP instances. While it might not be surprising that a binary AC algorithm is faster than a non-binary algorithm, it highlights the special nature of binary constraints. The special case of binary constraints is simpler than the non-binary case which typically has more complex algorithms and data structures. For example, the $AC3^{bit+rm}$ algorithm in Figure 2(a) is much simpler than the CT algorithm and uses simpler data structures.

We now compare different ways of solving non-binary CSP instances. We first compare solving non-binary CSP instances with the HAC algorithm on the hidden variable encoding of the non-binary instances with the CT algorithm on the original non-binary instances. Figure 2(b) compares the time of HAC[3] with CT and shows that HAC is much slower than CT on non-binary CSP instances. This result is the opposite of the binary-only instances in Figure 2(a) and suggests what is known in the folklore that encoding a non-binary constraint to binary form to be solved using a binary solver is much slower than using a (modern) GAC solver on the non-binary constraints directly. The HVE encoding with AC (using $AC3^{bit+rm}$) is also slow, shown in the comparison of time between HAC and HVE+AC in Figure 2(c). Figure 2(d) compares time versus search nodes of HAC with HVE+AC with the y-axis giving the number of search nodes of HAC/(HVE+AC) and x-axis giving the time of HAC/(HVE+AC). It shows that the special propagator of HAC for HVE is more efficient than $AC3^{bit+rm}$ on HVE.

Figure 2(e) and 2(f) reveals important factors behind why the performance of HVE encoding + AC algorithm is worse than GAC (with CT) on the non-binary instances:

(i) The concentration of points around the x-axis to the right of the y-axis shows that for a similar number of search nodes, the search time is significantly slower than CT (yet Figure 2(a) shows $AC3^{bit+rm}$ is faster than CT

[3] In this paper, we use HAC to implicitly refer to HVE+HAC.

for binary constraints which do not come from the hidden variable encoding). This suggests that the CT propagator is more efficient than the AC propagators on HVE instances;

(ii) There are many differences in the search nodes for the binary encoding versus the original instance. Many instances have more search nodes in the binary encoding as shown by the density of points above the x-axis.

As many points are far to the right, we see that the propagator efficiency may be the factor for the superiority of CT though the difference in search nodes is also a factor.

Search heuristics and consistency propagators are the main components in a constraint solver. The results above show that with CT (and also other modern GAC propagators)[4] both the efficiency of the constraint propagator and effectiveness are reasons for the folklore superiority of GAC on the non-binary instance over AC on the encoded instance. Furthermore, we have seen that the binary constraints in HVE instances are very special. In this paper, we focus on improving two problems identified for binary encoding instances:

1. Designing a special AC propagator which is more efficient for the binary constraints in HVE instances.
2. Avoiding making the search heuristic on HVE instances worse than on the original instances.

4 The hidable model transformation propagator

We saw that in Section 3, results illustrating the folklore suggesting it is better to solve a non-binary CSP directly using GAC rather than with binary encoding. The goal in this paper is to dispel this folklore notion. We also saw that binary encoding can interact poorly with the search heuristic and that the binary constraints from the hidden variable encoding are special.

To deal with the search heuristic problem, we "virtualize" the binary encoding so that the interaction between the binary encoded constraints can be hidden from the search heuristic making it behave like the search heuristic for the original non-binary constraint. This allows us to investigate the search heuristic which behaves like in the non-binary instance but also have an alternative where the search heuristic works on the HVE instance. We incorporate ideas from modern GAC algorithms to get a more efficient AC propagator for the special kinds of binary constraints in HVE instances.

4.1 The hidable model transformation

The hidden variable encoding is a special transformation of a CSP P_1 to a P_2 where P_2 is reducible to P_1. We generalize this idea to allow different kinds of transformations and search strategies,

[4] Experimental results for STR2+ are also similar to CT and have not been shown.

Definition 1 *$(\mathcal{X}_1 \cup \mathcal{X}_2, \mathcal{C}_2)$ is a* GAC hidable transformation *of $(\mathcal{X}_1, \mathcal{C}_1)$ iff $\mathcal{C}_2$ has a partition $\{s_1, ..., s_m\}$ such that for each $c_i \in \mathcal{C}_1$, $scp(c_i) \subseteq scp(s_i)$ and c_i is GAC iff all constraints in s_i are GAC, where $scp(s_i) = \bigcup_{c \in s_i} scp(c)$.*

Corollary 1 *If $(\mathcal{X}_1 \cup \mathcal{X}_2, \mathcal{C}_2)$ is a GAC hidable transformation of $(\mathcal{X}_1, \mathcal{C}_1)$, then $(\mathcal{X}_1, \mathcal{C}_1)$ is GAC iff $(\mathcal{X}_1 \cup \mathcal{X}_2, \mathcal{C}_2)$ is GAC.*

If $P_2 = (\mathcal{X}_1 \cup \mathcal{X}_2, \mathcal{C}_2)$ is an GAC hidable transformation of P_1, then the solver does not need to search the variables in $\mathcal{X}_2$, since GAC on P_2 can check whether an assignment on $\mathcal{X}_1$ is a solution of P_1. As such, the search algorithm only needs to consider $\mathcal{X}_1$ where the GAC propagators on P_2 are a "black box". In this paper, we only consider binary encodings, i.e. P_2 is a binary CSP.

Corollary 2 *HVE is a GAC hidable model transformation.*

Proof. For a non-binary constraint c_i, we can set $s_i = \{c_i^x \in \mathcal{HC} | x \in scp(c_i)\}$, since $(scp(s_i), s_i)$ is the HVE of $(scp(c_i), \{c_i\})$. The HVE encoding by construction of c_i^x already meets the requirements of Definition 1. The HVE transformation is only on non-binary constraints. For a binary constraint c_i, we set $s_i = \{c_i\}$, also meeting the definition.

4.2 A propagator for the hidable model transformation

Algorithm 1 gives the *HTAC* algorithm to enforce AC on hidable binary encoding instances. HTAC adds a variable $x \in \mathcal{X}_1 \cup \mathcal{X}_2$ to the propagation queue $\mathcal{Q}$ if x may be used to update the domains of other variables. Then HTAC iteratively picks a variable x from $\mathcal{Q}$, and then use AC algorithms to enforces AC on all constraints in every subset $s_i \in \mathcal{S}$ such that $x \in scp(s_i)$. For different s_i, we can use different AC algorithms. If c_i is a binary constraint in $\mathcal{C}_1$ and $s_i = \{c_i\}$, then HTAC can use any efficient AC algorithms, e.g. AC3^{bit+rm}, to enforce AC on c_i. For a GAC hidable model transformation, we give special AC-H algorithms exploiting the nature of constraints used in the transformation. In section 4.3, we present a AC-H algorithm to handle the constraints used in HVE transformation. HTAC is different from HAC [20]: HTAC adds original variables to $\mathcal{Q}$ while HAC only adds hidden variables to $\mathcal{Q}$. When the domain of a variable x is changed, HAC directly updates the domains of all hidden variables constrained by x and does not add x to $\mathcal{Q}$. HTAC uses a reversible bit set to represent the domain of a hidden variable (see Section 4.3), but HAC does not; The revise functions used in AC-H are also different from HAC.

For a GAC hidable model transformation $(\mathcal{X}_1 \cup \mathcal{X}_2, \mathcal{C}_2)$, the solver only needs to search the variables in $\mathcal{X}_1$. Search heuristics which use information from the constraint structure can choose to use the structure of the GAC hidable model transformation or the original model. For example, the wdeg/dom [2] heuristic records a weight w for each constraint, and increasing w by one if the constraint causes inconsistency. Variables are selected based on the weights of constraints. For the HVE transformation, we propose two alternatives for wdeg/dom:

Algorithm 1: HTAC $(\mathcal{X}_1, \mathcal{C}_1)$

```
let $(\mathcal{X}_1 \cup \mathcal{X}_2, \mathcal{C}_2)$ be the hidable model transformation of $(\mathcal{X}_1, \mathcal{C}_1)$;
let $\mathcal{S}$ be a partition of $\mathcal{C}_2$ making $(\mathcal{X}_1 \cup \mathcal{X}_2, \mathcal{C}_2)$ hidable;
$\mathcal{Q} \leftarrow \mathcal{X}_1$;
while $\mathcal{Q} \neq \emptyset$ do
    pick and delete $x$ from $\mathcal{Q}$;
    for $s_i \in \mathcal{S}$ s.t. $x \in scp(s_i)$ do
        if $|s_i| = 1$ then
            // $s_i = \{c_i\}$
            if $\neg AC(s_i)$ then
                return false;
        else if $\neg AC\text{-}H(s_i)$ then
            return false;
return true;
```

A. using wdeg/dom with the original model, we record a weight w_i for each $s_i \in \mathcal{S}$. Thus, w_i is regarded as a weight for a virtual constraint representing the weight of $c_i \in \mathcal{C}_1$. Weight w_i is incremented if AC(-H)(s_i) is not consistent;

B. using wdeg/dom with the HVE transformation, we record a weight w_i^x for each $c_i^x \in \mathcal{C}_2$ and w_i^x is incremented if AC(-H)(s_i) is not consistent, where x is picked from $\mathcal{Q}$ and $x \in scp(s_i)$.

We call HTAC as *HTAC1* if the heuristics use the original non-binary model (A); and *HTAC2* if the heuristics use the transformation model (B).

4.3 The AC-H algorithm for HVE

We first introduce the data structures used in the algorithm which incorporates data structures used by (modern) GAC and AC algorithms [14, 10, 15, 5]:

1. For a original variable x:
   - $dom(x)$ uses an "ordered link" data structure proposed in [14] to represent the current domain of x.
   - $bitDom(x)$ uses a bit set to represent the domain of a variable x [15].
   - $bitSup(c, x, a)$ is used to represent all supports of variable value (x, a) in $bitDom(y)$, where $scp(c) = \{x, y\}$ [15].
   - $lastSize(c, x)$ is used to record the size of $dom(x)$ after the last update on the domain of x based on c [10].
2. For a hidden variable x:
   - $bitDom(x)$ uses a bit set to represent the domain of x, if $bitDom(x)$ is changed, the old states of $bitDom$ are recorded in a stack so that it can be undone on backtracking.
   - $wordDom(x)$ is a sparse set used to record the non-ZERO words in $bitDom(x)$. It uses the reversible sparse bit-set idea in [5].

– $prevDom(x)$ is a copy of $bitDom(x)$, we use $prevDom(x)$ to check whether $bitDom(x)$ is changed.
– $buf0$ is a bit set, where all words in $buf0$ are initialized as ZERO (ZERO is the bit set with all zeroes).

Algorithm 2 is to enforce AC on a set (s_i) of constraints for HVE instances, where $s_i = \{c_i^x \in \mathcal{HC} | hv_i \in scp(c_i^x)\}$. The HVE transformation has a star structure constraint graph (a special tree). This allows the AC-H algorithm to update the domains of variables in two passes: (i) from leaves $(x \in c_i)$ to the root (hv_i), and (ii) from the root to the leaves. The first phase of revise is with function $revise2$ to (partially) update the domain of hv_i based on the current domains of variables in $scp(c_i)$. Then function $update$ updates $wordDom$ representation of hv_i. If $wordDom(hv_i) = \emptyset$, the instance is not AC. The second revise phase uses function $revise1$ to update the domains of all variables in $scp(c_i)$. If the domain of a variable x is changed, x is added to Q. We do not add hv_i to Q, since the domains of variables constrained with hv_i are updated in the AC-H algorithm.

Algorithm 2: AC-H (s_i)

```
let $hv_i$ be the hidden variable constrained with binary constraints in $s_i$;
for each $c_i^x \in s_i$ do
    revise2($c_i^x, hv_i$);
if $\neg$update($hv_i$) then
    return false;
for each $c_i^x \in s_i$ do
    if revise1($c_i^x, x$) then
        $\mathcal{Q} \leftarrow \mathcal{Q} \cup \{x\}$;
return true;
```

Due to lack of space, we briefly sketch correctness of AC-H and associated functions. The overall structure is similar to any AC algorithm using revise except that we exploit the star constraint graph as explained above. The current domain of hv_i is only updated by the function $revise2$. Meanwhile, the function $revise2$ deletes the values which do not have valid supports in the current domains of the variables in $scp(c_i)$ from the current domain of hv_i. If the function $update$ return false, all words in $bitDom(hv_i)$ become ZERO, which means that it is not AC. Finally. the function $revise1$ is similar to AC3^{bit+rm}.

4.4 Revise functions

In AC algorithms, the $revise(c, x)$ functions are used to update $dom(x)$ based on $dom(y)$, where $\{x, y\} = scp(c)$, i.e removing the values in $dom(x)$ which don't have valid supports in $dom(y)$. We give two revise functions used in AC-H to handle the reversible bit-set domains. The function $revise1$ updates original variable domains using function $seekSupport$ which is similar to that in AC3^{bit+rm} algorithm. The difference is that our $seekSupport$ only check the words in $wordDom$. For hidden variables, we do not use the $seekSupport$ function, since each value in hidden variable domains (by construction) only has one support in $rel(c)$, i.e. an hidden variable hv functionally determines the values in the domains of original variables constrained with hv (see [1]), which make $bitSup$ useless. Using the ideas from GAC algorithms CT[5] and STRbit[26], function $revise2$ applies the $delete$ and $reset$ functions to update $bitDom$. If the number of values deleted from $dom(x)$ is larger than $|dom(x)|$, then function $delete$ is used to remove all supports of the deleted values from $bitDom(hv)$. Otherwise, function $reset$ is used to build a new $bitDom(hv)$ based on current $dom(x)$. After updating $bitDom(hv)$ of a hidden variable hv, the function $update$ is used to check $bitDom(hv)$ for domain wipeout, for each word w in $bitDom(hv)$, if w is changed, the old value of $preDom(w)$ in saved on a stack for backtracking, and if $w = ZERO$ it is removed from $wordDom$.

Function $revise1(c, x)$

```
$size \leftarrow |dom(x)|$;
for each $a \in dom(x)$ do
    if $\neg seekSupport(c, x, a)$ then
        remove $a$ from $dom(x)$;
return $|dom(x)| \neq size$;
```

Function $seekSupport(c, x, a)$

```
Let $y$ be the variable such that $scp(c) = \{x, y\}$;
$w \leftarrow rm[c, x, a]$;
if $(bitSup[c, x, a][w] \& bitDom[y, w]) \neq ZERO$ then
    return true;
for each $w \in wordDom(y)$ do
    if $(bitSup[c, x, a][w] \& bitDom[y, w]) \neq ZERO$ then
        $rm[c, x, a] \leftarrow w$;
        return true;
return false;
```

Function $revise2(c, hv)$

```
Let $x$ be the variable such that $scp(c) = \{x, hv\}$;
$dn \leftarrow lastSize(c, x) - |dom(x)|$;
if $dn > |dom(x)|$ then
    $reset(c, hv)$;
else if $dn > 0$ then
    $delete(c, hv, dn)$;
```

Function *delete*(c, hv, dn)

```
Let $x$ be the variable such that $scp(c) = \{x, hv\}$;
for $i = 0$ to $dn$ do
    Let $a$ be the last $i$ value deleted from $dom(x)$;
    for each $w \in wordDom(hv)$ do
        $bitDom[hv, w] \leftarrow bitDom[hv, w] \& \neg bitSup[c, x, a]$;
```

Function *reset*(c, hv)

```
Let $x$ be the variable such that $scp(c) = \{x, hv\}$;
for each $a \in dom(x)$ do
    for each $w \in wordDom(hv)$ do
        $buf0[w] \leftarrow buf0[w] | bitSup[c, x, a]$;
for each $w \in wordDom(hv)$ do
    $bitDom[x, w] \leftarrow buf0[w] \& bitDom[hv, w]$;
    $buf0[w] \leftarrow ZERO$;
```

Function *update*(x)

```
for each $w \in wordDom(x)$ do
    if $bitDom[x, w] \neq prevDom[x, w]$ then
        save $prevDom[x, w]$ in a stack;
        $prevDom[x, w] \leftarrow bitDom[x, w]$;
        if $bitDom[x, w] = ZERO$ then
            remove $w$ from $wordDom(x)$;
return $wordDom(x) \neq \emptyset$;
```

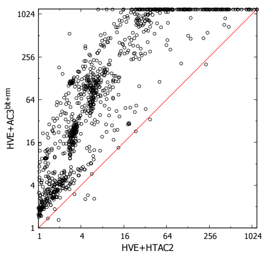
5 Experiments

We present experiments to evaluate HTAC on the hidden variable encoding (HVE+HTAC1 and HVE+HTAC2) compared with HVE+AC3^{bit+rm}, HAC, STR2+, CT and STRbit algorithms. HTAC, HVE+AC3^{bit+rm} and HAC maintain AC (MAC) on the hidden variable encoding instances. STR2+, CT and STRbit maintain GAC (MGAC) on the original non-binary instances. CT and STRbit are the state-of-the-art GAC algorithms, and HAC is the best existing algorithm for binary encoding from Section 3. We used binary and non-binary instances from the XCSP3 website (http://xcsp.org). Instances from XCSP3 which timeout for all compared algorithms are ignored. In total, we evaluated 1328 binary and 2431 non-binary problem instances. The binary CSP series are:

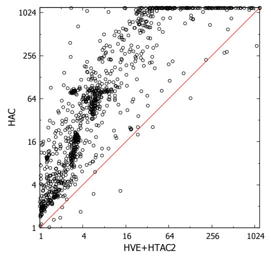
> QCP, QWH, Geometric, Rlfap, Driver, Lard, Queens, RoomMate, Prop-Stress, QueensKnights, KnightTour, Random

The binary instances are discussed in Section 3 in Figure 2(a). The non-binary CSP series are:

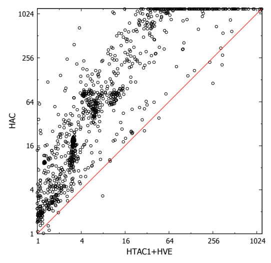
Kakuro, Dubois, PigeonsPlus, MaxCSP,Renault, Aim, Jnh, Cril, Tsp, Various, Nonogram, Bdd-{15,18}-21, mdd-7-25-{5,5-p7,5-p7}, reg2ext,Rand-3-24-24, Rand-3-24-24f, Rand-5-12-12, Rand-5-{2,4,8}X, Rand-10-20-10, Rand-10-20-60, Rand-15-23-3, Rand-5-10-10, Rand-5-12-12t, Rand-7-40-8t, Rand-8-20-5, Rand-3-20-20, Rand-3-20-20f



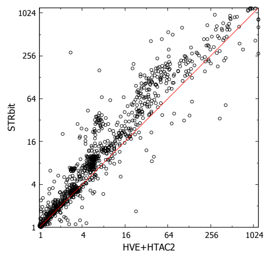
(a) HTAC2 vs AC3^{bit+rm}



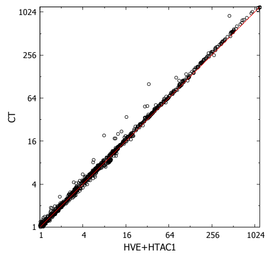
(b) HTAC2+HVE vs HAC



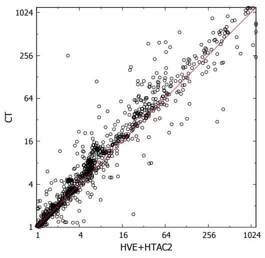
(c) HTAC1+HVE vs HAC



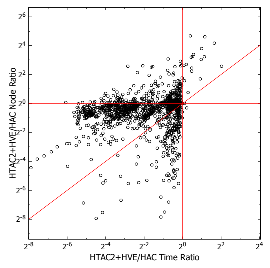
(d) HTAC2+HVE vs STRbit



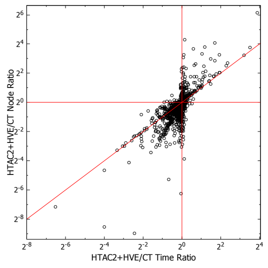
(e) HTAC1+HVE vs CT



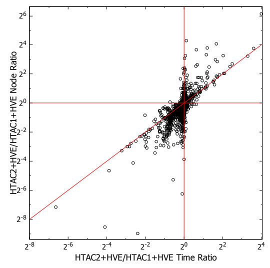
(f) HTAC2+HVE vs CT



(g) HTAC2+HVE vs HAC (nodes+time)



(h) HTAC2+HVE vs CT (nodes+time)



(i) HTAC2 vs HTAC1 (nodes+time)

Fig. 3. HTAC vs other algorithms: (a)-(f) use time on the axis while (g)-(i) use node and time ratios in the axis

This section focuses on the non-binary instances. Results on the non-binary series are also given in Section 3 comparing HAC, HVE+AC3^{bit+rm} and CT. The experiments were run on a 3.20GHz Intel i7-8700 machine. We implemented

HTAC in the Abscon solver[5] which has the other algorithms implemented. In addition, we optimized the Abscon CT and HAC implementation to be a little faster.[6] The variable search heuristic used is *wdeg*/*dom* and the value heuristic used is lexical value order. The *wdeg*/*dom* with restart is considered one of state-of-the-art heuristics in classic constraint solvers [7]. The restart policy was geometric restart (the initial $\mathit{cutoff} = 10$ and $\rho = 1.1$)[7]. CPU time is limited to 1200 seconds per instance and memory to 8GB.

Figures 3 shows 9 scatter plots to compare HVE+HTAC with other algorithms. Each dot in the plots is an instance. Figures 3(a) to 3(f) compare the time[8] of different algorithms. Meanwhile, Figures 3(g), 3(h) and 3(i) compare the time ratio and node ratio, where the time (node) ratio of A/B means the ratio "the time (number of search nodes) of algorithm A" to "the time (number of search nodes) of algorithm B". Figures 3(a),3(b) and 3(c) show that HVE+HTAC2 and HVE+HTAC1 can outperform the other binary algorithms. From Figure 3(g), giving the time ratio and node ratio of (HTAC2+HVE)/HAC (see the discussion of ratio graphs in Section 3), we see that HTAC is generally much faster than HAC, since most points around the x-axis are at the left of the y-axis. For most instances, the search nodes of HTAC2 is less than HAC.

Figures 3(d), 3(e) and 3(f) show HTAC is competitive with the state-of-the-art GAC algorithms CT and STRbit. HTAC1+HVE using wdeg/dom on the original model is competitive with CT, being faster than CT on some instances. HTAC2+HVE using wdeg/dom on the HVE transformed model is faster than STRbit and CT on many instances. Figure 3(h) combines node ratio and time ratio to show the runtime and search nodes tradeoffs of HTAC2 with CT with more instances having less nodes and time. Figure 3(i) compares HTAC2 with HTAC1, the performance of HTAC1 is similar to CT.

Figure 4 shows the runtime distribution of the problem instances solved by the different algorithms. The y-axis is the solving time (in seconds) and the x-axis is the number of instances solved within the time limit. It shows firstly, the folklore that binary encodings are slower with HVE+AC3^{bit+rm} and HAC being behind STR2+ (also show in [10]). There is a separation between STR2+ and newer GAC algorithms (STRbit and CT). The performance of our HTAC algorithms is competitive with STRbit and CT, in particular, HTAC2 is faster on some instances.

[5] https://www.cril.univ-artois.fr/~lecoutre/#/softwares

[6] While implementing HTAC, we found some optimizations for the existing CT and HAC code.

[7] $cutoff$ is the allowed number of failed assignments for each restart. After restart, $cutoff$ increases by ($cutoff \times \rho$)

[8] For binary encoding instances, the time includes solving time and model transformation time.

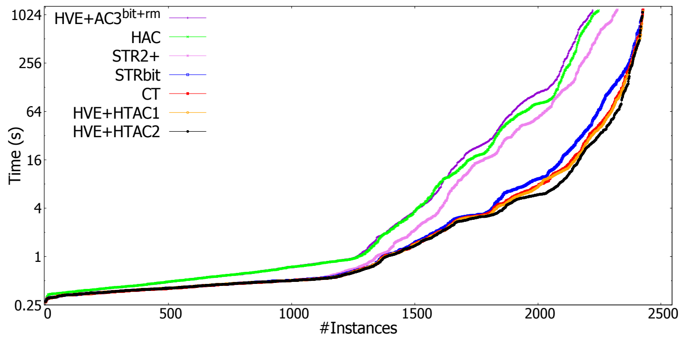


Fig. 4. Runtime Distribution: HTAC, HAC, HVE+AC3^{bit+rm}, CT, STRbit, STR2+

6 Conclusion

We first show experimental results which can explain the folklore that it is better to solve a non-binary CSP instance directly with GAC than by a binary encoding of the instance and using AC. We show that this folklore is misleading, solving with the binary encoding can be improved by having a more efficient propagator for binary constraints from the HVE instances and preventing poor interaction of the HVE model with the search heuristic.

We propose a new propagator HTAC. By using the GAC hidable binary encoding with HTAC, we can address the differences in search nodes so that the search space on the binary instance behaves as in the non-binary instance but it also allows search on the binary encoded model. The HTAC propagator gains efficiency by using properties of binary constraints in the HVE. It is also efficient as we apply ideas from modern GAC algorithms. Experiments show that HTAC on the binary encoded instance is competitive with state-of-the-art GAC algorithms on the original instances, in some cases, HTAC is faster. Not only have we shown that solving with the binary encoding is viable and competitive, we believe that it opens new directions for modelling and solver algorithms while still retaining the original non-binary instance. Binary instances and constraints are special being simpler so the algorithms can also be simpler. Many transformations and higher consistencies can be applied directly to binary instances.

Acknowledgements

We acknowledge the support of grant MOE2015-T2-1-117.

References

1. Bacchus, F., Van Beek, P.: On the conversion between non-binary and binary constraint satisfaction problems. In: AAAI/IAAI. pp. 310–318 (1998)

2. Boussemart, F., Hemery, F., Lecoutre, C., Sais, L.: Boosting systematic search by weighting constraints. In: European Conf. on Artificial Intelligence. pp. 146–150. IOS Press (2004)
3. Cheng, K., Yap, R.H.C.: An MDD-based generalized arc consistency algorithm for positive and negative table constraints and some global constraints. Constraints **15**(2), 265–304 (2010)
4. Dechter, R., Pearl, J.: Tree clustering for constraint networks. Artificial Intelligence **38**(3), 353–366 (1989)
5. Demeulenaere, J., Hartert, R., Lecoutre, C., Perez, G., Perron, L., Régin, J.C., Schaus, P.: Compact-table: Efficiently filtering table constraints with reversible sparse bit-sets. In: International Conference on Principles and Practice of Constraint Programming. pp. 207–223. Springer (2016)
6. Gharbi, N., Hemery, F., Lecoutre, C., Roussel, O.: Sliced table constraints: Combining compression and tabular reduction. In: Proceedings of the 11th International Conference on Integration of Artificial Intelligence and Operations Research techniques in Constraint Programming. pp. 120–135 (2014)
7. Hebrard, E., Siala, M.: Explanation-based weighted degree. In: International Conference on AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems. pp. 167–175. Springer (2017)
8. Jefferson, C., Nightingale, P.: Extending simple tabular reduction with short supports. In: Proceedings of the 23rd International Joint Conferences on Artificial Intelligence. pp. 573–579 (2013)
9. Katsirelos, G., Walsh, T.: A compression algorithm for large arity extensional constraints. In: Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming. pp. 379–393 (2007)
10. Lecoutre, C.: STR2: optimized simple tabular reduction for table constraints. Constraints **16**(4), 341–371 (2011)
11. Lecoutre, C., Hemery, F., et al.: A study of residual supports in arc consistency. In: IJCAI. vol. 7, pp. 125–130 (2007)
12. Lecoutre, C., Likitvivatanavong, C., Yap, R.H.C.: A path-optimal GAC algorithm for table constraints. In: Proceedings of the 20th European Conference on Artificial Intelligence. pp. 510–515 (2012)
13. Lecoutre, C., Likitvivatanavong, C., Yap, R.H.C.: STR3: A path-optimal filtering algorithm for table constraints. Artificial Intelligence **220**, 1–27 (2015)
14. Lecoutre, C., Szymanek, R.: Generalized arc consistency for positive table constraints. In: International conference on principles and practice of constraint programming. pp. 284–298. Springer (2006)
15. Lecoutre, C., Vion, J.: Enforcing arc consistency using bitwise operations. Constraint Programming Letters **2**, 21–35 (2008)
16. Mackworth, A.K.: Consistency in networks of relations. Artificial intelligence **8**(1), 99–118 (1977)
17. Mamoulis, N., Stergiou, K.: Solving non-binary csps using the hidden variable encoding. In: International Conference on Principles and Practice of Constraint Programming. pp. 168–182. Springer (2001)
18. Perez, G., Régin, J.C.: Improving GAC-4 for table and MDD constraints. In: Proceedings of the 20th International Conference on Principles and Practice of Constraint Programming. pp. 606–621 (2014)
19. Rossi, F., Petrie, C.J., Dhar, V.: On the equivalence of constraint satisfaction problems. In: ECAI. vol. 90, pp. 550–556 (1990)

20. Samaras, N., Stergiou, K.: Binary encodings of non-binary constraint satisfaction problems: Algorithms and experimental results. Journal of Artificial Intelligence Research **24**, 641–684 (2005)
21. Stergiou, K., Walsh, T.: Encodings of non-binary constraint satisfaction problems. In: AAAI/IAAI. pp. 163–168 (1999)
22. Ullmann, J.R.: Partition search for non-binary constraint satisfaction. Information Sciences **177**, 3639–3678 (2007)
23. Verhaeghe, H., Lecoutre, C., Deville, Y., Schaus, P.: Extending compact-table to basic smart tables. In: International Conference on Principles and Practice of Constraint Programming. pp. 297–307. Springer (2017)
24. Verhaeghe, H., Lecoutre, C., Schaus, P.: Extending compact-table to negative and short tables. In: AAAI. pp. 3951–3957 (2017)
25. Verhaeghe, H., Lecoutre, C., Schaus, P.: Compact-mdd: Efficiently filtering (s) mdd constraints with reversible sparse bit-sets. In: IJCAI. pp. 1383–1389 (2018)
26. Wang, R., Xia, W., Yap, R.H., Li, Z.: Optimizing simple tabular reduction with a bitwise representation. In: IJCAI. pp. 787–795 (2016)
27. Xia, W., Yap, R.H.C.: Optimizing STR algorithms with tuple compression. In: Proceedings of the 19th International Conference on Principles and Practice of Constraint Programming. pp. 724–732 (2013)