The Extensional Constraint

Hélène Verhaeghe[1]

UCLouvain, ICTEAM, Place Sainte Barbe 2, 1348 Louvain-la-Neuve, Belgium,
{helene.verhaeghe}@uclouvain.be

Abstract. Table constraints and Multi-Valued Decision Diagrams (MDDs) are very useful for modeling combinatorial constrained problems, and thus play an important role in Constraint Programming (CP). During the last decade, many algorithms have been proposed for enforcing the property known as Generalized Arc Consistency (GAC) on such constraints. A state-of-the-art GAC algorithm called Compact-Table (CT), which has been recently proposed, significantly outperforms all previously proposed algorithms for the extensional constraint.
In my thesis, we first extend the CT algorithm in order to deal with basic smart table (i.e., tables with unary expressions such as $= v$, $*$, $\neq v$, $\leq v$, $\geq v$ and $\in S$) and negative tables (i.e., tables containing conflicts instead of solutions). The ideas behind Compact-Table are also used to construct Compact-Diagram (CD), an algorithm handling MDDs and, in general, any ordered layered diagram, with or without decision nodes. Compact-Diagram is also extended to handle basic smart diagrams. Our experimental results show the practical interest of our approaches.

Keywords: Compact-table, Bitwise operations, Table constraint, MDD constraint

1 Introduction

Efficiently representing constraints under extensional forms such as tables and decision diagrams has been a hot research topic for the last decade; concerning MDDs (Multi-Valued Decision Diagrams), see e.g., [1–4, 9–11, 29]. Two main lines of improvements have been followed when handling extensional forms of constraints. Firstly, high effective filtering techniques have been proposed over the years, such as those based on tabular reduction [14, 17] and bitwise operations [8, 12, 34]. Secondly, compact representation techniques have been intensively studied, mainly by allowing simple constraints to be put in tables as in [13, 20] or by directly using decision diagrams such as MDDs [7, 23, 24].

My thesis consisted of studying the possibility to extend the Compact-Table algorithm [8] to handle some of the compact representation. The main contributions of my Ph.D. thesis, summarised in this paper, are:

- an extension of CT to handle basic smart tables (CT^{bs}) [32]
- an extension of CT to handle negative tables (CT_{neg}) [33]
- an adaptation of CT to handle diagrams (CD) [30]

– an extension of CD to handle basic smart diagrams (CD^{bs}) [31]

The remainder of this paper is structured as follows. The next section recall some background, then Section 3 briefly recalls the Compact-Table. Sections 4, 5, 6 and 7 respectively introduce the CT^{bs}, the CT$_{neg}$, the CD and CD^{bs} algorithms. Finally, the global results of the experiments are given in Section 8.

2 Technical Background

A *constraint network* is composed of a set of variables and a set of constraints. Each *variable* x has an associated (ordered) domain $dom(x)$ containing the values that can be assigned to it; the *current* domain is included in the *initial* domain $dom^0(x)$. We respectively denote by $min(x)$ and $max(x)$ the smallest and greatest values in $dom(x)$. Each *constraint* c involves an ordered set of variables, called the *scope* of c and denoted by $scp(c)$, and is semantically defined by a *relation* $rel(c)$ containing the tuples allowed for the variables involved in c. The *arity* of a constraint c is $|scp(c)|$. When the domain of a variable x is (becomes) singleton, we say that x is *bound*.

Given a sequence $\langle x_1, \ldots, x_r \rangle$ of r variables, an r-tuple τ on this sequence of variables is a sequence of r values $\langle a_1, \ldots, a_r \rangle$, where the individual value a_i is also denoted by $\tau[x_i]$. An r-tuple τ is *valid* on an r-ary constraint c iff $\forall x \in scp(c), \tau[x] \in dom(x)$, and τ is *allowed* by c iff $\tau \in rel(c)$. A *support* of c is a tuple which is both valid on c and allowed by c. A *literal* is a pair (x, a) where x is a variable and a a value. A literal (x, a) is *Generalized Arc-Consistent* (GAC) on c iff there is a support τ on c such that $\tau[x] = a$. A constraint c is GAC iff any literal (x, a) such that $x \in scp(c)$ and $a \in dom(x)$ is GAC on c.

A directed graph is composed of a set of nodes and a set of arcs. Each arc has an orientation from one node, the *tail* of the arc, to another node, the *head* of the arc. For a given node ν, the set of arcs with ν as tail (resp., head) is called the set of *outgoing* (resp., *incoming*) arcs of ν. A labeled directed graph is a directed graph such that a label $l(\omega)$ is associated with each arc ω. A node is *in-d* (in-deterministic) iff it does not have two incoming arcs with the same label, *in-nd* otherwise. A node is *out-d* (out-deterministic) iff no two outgoing arcs have the same label, *out-nd* otherwise. A directed acyclic graph (DAG) is a directed graph with no directed cycles. An MVD (Multi-valued Variable Diagram) [1] *for* a constraint c (called an MVD constraint) is a layered DAG, with one special root node at level 0, denoted by `ROOT`, r layers of arcs, one layer $L(x_i)$ for each variable x_i of the scope $\langle x_1, \ldots, x_r \rangle$ of c, and one special sink node at level r, denoted by `SINK`. The arcs in $L(x_i)$ going from level $i-1$ to level i are *on* the variable x_i: any such arc is labelled by a value in $dom^0(x_i)$. A *valid path* in such an MVD is a path p from the root to the sink such that for each variable x_i in $scp(c)$ the label of the arc going in p from level $i-1$ to i is a value in $dom(x_i)$. The set of supports of an MVD constraint c corresponds to the valid paths in the MVD for c. One classical type of MVD is the Multi-valued Decision Diagram (MDD) [6], which guarantees that each node is out-d (each node at level i has

at most $|dom^0(x_i)|$ outgoing arcs, labelled with different values), but possibly in-nd.

3 Compact-Table

This section briefly introduces Compact-Table (CT), a state-of-the-art filtering algorithm [8] initially introduced for enforcing GAC on positive (ordinary) table constraints. It first appeared in Or-Tools [21], the solver developed at Google, and is now implemented in OscaR [22], AbsCon and Choco [25]. CT benefits from well-established techniques: bitwise operations [5, 18], residual supports [15, 16, 19], tabular reduction [14, 17, 28], reversible sparse sets [27] and resetting operations [24].

The core structure of CT, when applied to a constraint c, is a reversible sparse bitset, called `currTable`, responsible for keeping track of the current supports of c: the ith bit of `currTable` is set to 1 iff the ith tuple τ_i of the table of c is currently valid. It is updated by means of precomputed bitsets: for each value (x, a) of c, `supports`$[x, a]$ is the bitset that identifies the set of tuples that are initially supports of (x, a) on c.

Algorithm 1 presents a simplified version of CT, which consists of two main steps. First, updateTable(), iterates for each variable x involved in c over either the set Δ_x of values that have been removed from the domain of x since the last invocation of the algorithm (incremental update) or the current domain of x (reset update), and use the appropriate bitsets `supports` to update `currTable` using bitwise operations. The incremental update is done by intersecting the `supports` corresponding to the removed values. This intersection is then removed from `currTable`. The reset update is done by intersecting the `supports` corresponding to the remaining values. The negation of this intersection is then removed from `currTable`. Secondly, filterDomains(), iterates over every value (x, a) of c and use the corresponding bitset `supports`$[x, a]$ to verify whether the value is still supported or not. This is done by doing the intersection between `currTable` and `supports`$[x, a]$. An empty intersection corresponding to an unsupported value.

In the experiments, described in [8], Compact-Table outperformed all the previous state-of-the-art algorithms for table or MDD constraint such as STR2, STR3, MDD4R,... However, when dealing with compressed tables or MDDs as input, a decompression phase was required to use Compact-Table. The next sections detailed the extensions of CT aiming to handle the basic smart tables (one kind of compressed table) or MDDs as input while using their structure to improve the propagation.

4 CT^{bs}: Compact-Table for Basic Smart Tables

A basic smart table is composed of smart tuples containing expressions of the following forms: $*$, <op> v, $\in S$ and $\notin S$, with <op> $\in \{<, \leq, =, \neq, \geq, >\}$. We need to extend CT (which handles only $= v$, often write only v in that case

Algorithm 1: Class ConstraintCT

```
1  Method enforceGAC()
2      updateTable()
3      filterDomains()
4  Method updateTable()
5      updateMasks()
6  Method updateMasks()
7      foreach variable x ∈ {x ∈ scp : |Δ_x| > 0} do
8          if |Δ_x| < |dom(x)| then
9              IncrementalUpdate()
10         else
11             ResetUpdate()
```

Algorithm 2: updateMasks() from CT^{bs} and CD^{bs}

```
1  Method updateMasks()
2      foreach variable x ∈ {x ∈ scp : |Δ_x| > 0} do
3          if |Δ_x| < |dom(x)| and there is no ∈ S used for the variable then
4              IncrementalUpdate()
5              LowerBoundUpdate()
6              UpperBoundUpdate()
7          else
8              ResetUpdate()
```

for simplicity) to handle the smart tuples in a basic smart table. The resulting algorithm is called CT^{bs}. All the modification lies in the updateMasks() method given in Algo. 2, and presented i a high level description.

Handling $*$ and $\neq v$

To handle $*$ and $\neq v$, only the incremental update requires modifications. Instead of using `supports`, another bitset, called `supports`*, is used. It is defined as `supports` for tuple with $= v$ and filled with 0 for other types of labels.

Correctness for expressions $*$ and $\neq v$ is proved by showing that the table is always properly updated. Let us cover all possible cases. When $|dom(x)| = 0$, the solver detect the failure through the variables. For the case of the reset-based update, as support precisely depicts the acceptance of values by tuples, this is necessarily correct. In the incremental update, we will necessary have $|dom(x)| \geq 2$. This comes from the structure of the algorithm when the full version, as described in [8], is used. In this case, the tuple is always support for the given variable and as the corresponding bits in `supports`* are set to 0 by construction, the tuple is not removed from `currTable`.

Handling <op> v, with <op> $\in \{<, \leq, \geq, >\}$

First, note that it is sufficient to focus on expressions of the form $\geq v$ and $\leq v$ since $> v$ and $< v$ are equivalent to $\geq v+1$ and $\leq v-1$. We first introduce two additional arrays of bitsets: `supportsMin` for $\leq v$ and `supportsMax` for $\geq v$. For each value (x, a) of c, the ith bit of `supportsMin`$[\mathtt{x}, \mathtt{a}]$ (resp., `supportsMax`$[\mathtt{x}, \mathtt{a}]$) is 1 iff $\tau_i[x]$ allows at least one value $\geq a$ (resp., $\leq a$).

To handle <op> v, with <op> $\in \{\leq, \geq\}$, a lower bound update and an upper bound update are added in addition to the incremental update.

Correctness is shown for $\leq v$, considering all cases at column x for tuple τ. The case $|dom(x)| = 0$ is as trivial as in the last section. For the case of the reset-based update, as `supports` precisely depicts the acceptance of values by tuples, this is necessarily correct. Finally, in the incremental update (resp. upper bound update), due to the constructions of the bitsets, i.e., the bit for τ in `supports`* (resp. `supportsMax`) is always set to 0 (resp. 1), updating depends only on `supportsMin` (lower bound update). By definition, if $dom(x)$.`min` is $\leq v$, meaning still supported, the bit for τ in `supportsMin` is 1, keeping τ in `currTable`. If $dom(x)$.`min` $> v$, bit for τ is 0, removing τ. Dealing with $\geq S$ can be conducted similarly.

Handling $\in S$ (and $\notin S$)

There is no easy way to handle expressions of the form $\in S$ using an update based on the Δ. We then propose to systematically execute reset-based updates as they do in [34] for passing from STRbit to STRbit-C. More precisely, as soon as a variable is involved in an expression of the form $\in S$ in one of the tuples of the basic smart table, a reset-based update is forced. Dealing with $\notin S$ can be conducted similarly.

5 CT$_{neg}$: Compact-Table for Negative Tables

This section discusses the modifications brought to CT for dealing with negative tables, i.e., tables containing conflict tuples. We keep working with the bitset `currTable` that indicates which tuples from the initial table of c are still valid, and we introduce bitsets `conflicts` that are computed the same way as bitsets `supports` were. Simply, as the context is different, the meaning is different: instead of permanently updating the table of supports in `currTable` through bitsets `supports`, we permanently update the table of conflicts in `currTable` through bitsets `conflicts`.

For filtering, the basic idea is to count for each value (x, a) of c how many valid tuples containing (x, a) are in the current table of c (hence, representing the number of conflicts for (x, a) on c) and to compare this number with the number of valid tuples containing (x, a). When these two numbers are equal, it simply means that all valid tuples containing (x, a) correspond to conflicts, and consequently that no support for (x, a) on c exists. Computing, in the context

of a constraint c, the number of valid tuples for any value in the domain of a variable x is simple. This is:

$$\Pi_{y \in scp(c): y \neq x} |dom(y)| \tag{1}$$

When Method enforceGAC(), is called, the first step is to update the current table, exactly as it is done for positive table constraints, except that the bitsets `conflicts` are used instead of `supports`. For filtering domains, we verify whether values have still support or not. We compute the number of bits set to 1 on the bitwise intersection of `currTable` and $\texttt{conflicts}[\texttt{x}, \texttt{a}]$ to compute the number of conflicts for (x, a) on c. The rest of the algorithm is similar to CT.

6 Compact-Diagram: Compact-Table for the diagram constraint

Compact-Diagram, or CD, is a filtering algorithm (propagator) that uses bitwise operations for MVD constraints. It is based on some ideas behind both Compact-Table [8] and MDD4R [24], a propagator for MDD constraints.

The idea is to keep track of the arcs that remain valid during the filtering process; namely by introducing (reversible sparse) bitsets, one per layer of the MVD (and so, per variable of the constraint). At layer i, one bit, in the bitset $\texttt{currArcs}[x_i]$, is associated with each arc: when the bit is set to 1, it means that the arc is considered as valid. This way, the current MVD, which can be seen as a subgraph of the initial MVD, can be identified and used to remove the values without any supports left.

To ease computations, at each level there are three types of precomputed bitsets: these bitsets are never modified. First, $\texttt{supports}[x, a]$ indicates for each arc on the variable x regardless of whether the value a is initially supported by this arc (bit is set to 1 iff a is supported). Second, $\texttt{arcsT}[\nu, x]$ and $\texttt{arcsH}[x, \nu']$ indicate for each arc on x whether ν and ν' are respectively the tail and the head of this arc. Finally, a temporary bitset $\texttt{mask}[\texttt{x}_\texttt{i}]$ is associated with each variable x_i to store the results of intermediate computations.

The pseudo-code for enforcing GAC on an MVD constraint is given by Algo. 3, which is, for simplicity, a very simplified version of the one given in [30]. In method updateGraph(), after initializing all masks, all arcs that can be trivially removed are handled by calling updateMasks(). This method assumes access to the set of values Δ_x removed from $dom(x)$ since the last call to enforceGAC()[1]. There are two ways of updating the masks (before updating `currArcs` from these masks, later): either incrementally or from scratch after resetting. In case of an incremental update, we perform the union of the arcs to be removed, whereas, in case of a reset-based update, we perform the union of the arcs to be kept, followed by a reverse operation. Next, we need to determine which arcs

[1] In [26], a sparse-set domain implementation for obtaining Δ_x without overhead is described.

Algorithm 3: Class ConstraintCD

```
1 Method enforceGAC()
2     updateGraph()
3     filterDomains()
4 Method updateGraph()
5     updateMasks()
6     propagateDown()
7     propagateUp()
```

can be subsequently removed: this is achieved by calling the methods propagateDown() and propagateUp(), which, similarly to MDD4R, perform two passes on the diagram. During the downward (resp., upward) pass, each level is examined from the root (resp., sink) to the sink (resp., root). When there are no more valid arcs entering (resp. exiting) a node, it becomes unreachable and all arcs exiting (resp. entering) the node becomes invalid. Identifying unreachable nodes is done by testing if the intersection between `currArcs` and `arcsT` (for outgoing arcs) or `arcsH` (for incoming arcs) is empty.

The process of filtering domains is very similar to the one described in CT [8]. This is given by method filterDomains() in Algo. 3. For each unbounded variable x and each value a in $dom(x)$, the intersection between the valid arcs on x, `currArcs`$[x]$, and the arcs allowing value a, `supports`$[x, a]$, determines if a is still supported. An empty intersection means that a can be deleted from $dom(x)$.

7 CD^{bs}: Compact-Diagram for *bs*-MVDs

CD and CT are quite similar in terms of design. Both of them use the bitsets called `supports` to respectively find the arcs and tuples that must be discarded. The CT^{bs} [32] algorithm, which can deal with *bs*-tables, was proposed as an extension of CT, by only modifying the update procedure. In the same spirit, we show how similar ideas can be reused to adapt the method updateMask() of CD to define CD^{bs}.

As in CT^{bs}, in addition to bitsets `supports`, we introduce auxiliary bitsets:

- `supports`$^*[x, a]$, the exclusive supports: for each arc for which the label of arc ω is exactly a ('$= a$'), the bit is set to 1,
- `supportsMin`$[x, a]$, the lower bound supports: for each arc which would be still valid if the minimum of the domain was a, the bit is set to 1,
- `supportsMax`$[x, a]$, the upper bound supports: for each arc which would be still valid if the maximum of the domain was a, the bit is set to 1.

Algorithm 2 displays the method updateMasks() for the simple version of CD^{bs}. This is for Compact-Diagram a simple adaptation of the modifications made to pass from CT to CT^{bs}. Resetting (and recomputing) is performed when

the number of removed values (i.e., values in Δ_x) is too large by collecting the supports of every value in the current domain or if there was some $\in S$ expression used. Otherwise, an incremental update is performed. Notice that contrarily to the reset-based update, one needs to also collect invalid arcs for operators in $\{<, \leq, >, \geq\}$ using `supportsMin` (lower bound update) and `supportsMax` (upper bound update). The time complexity of one call to updateMasks(), for a given variable x, is $\Theta(dt)$ where t is the number of valid words and d is $\min(|\Delta_x|, |dom(x)|)$ if there was no $\in S$ used and $|dom(x)|$ if yes.

This is the simple version of CD^{bs}. The optimized version, fully described in the paper, strongly relies on a partition of the arcs at each level i.

8 Experimental Results

This section gives an overview of the results that we obtained from the experiments carried out on the different algorithms. Only the big picture is given here. For a more detailed analysis, see the papers [30–33].

The experiments about CT^{bs} led to the following results: to overcome the overhead of the additional operations, the basic smart table should contain more than a few basic smart elements.

The negative table experiments showed an improvement in the resolution time of CT$_{neg}$ in comparison to STRne (i.e., the negative table version of STR2).

On the diagram point of view, CD improved the timing of MDD4R. The CD^{bs} algorithm showed how the compression of the graph leads to a speedup. However, when comparing the CD on the MDD to CT on the corresponding table, CT is still outperforming the diagram-based propagation.

To conclude the experiments, introducing compression in the table allowed us to reduce the search time on the problems. The introduction of the bitwise technique to the diagram-based propagator allowed us to improve the timing. The gap between the table-based and diagram-based method is reduced but still existent.

9 Conclusion

The starting point of this thesis was the Compact-table algorithm. It was extended following three axes. First from table to basic smart table, secondly from positive to negative tables and third from table to diagram. All the propagators detailed were proven efficient on the tested benchmarks. For more details, we refer the reader to papers [30–33].

Acknowledgement

This thesis is supervised by Pierre Schaus and Christophe Lecoutre.

References

1. Amilhastre, J., Fargier, H., Niveau, A., Pralet, C.: Compiling CSPs: A complexity map of (non-deterministic) multivalued decision diagrams. International Journal on Artificial Intelligence Tools **23**(04) (2014)
2. Andersen, H., Hadzic, T., Hooker, J., Tiedemann, P.: A constraint store based on multivalued decision diagrams. In: Proceedings of CP'07. pp. 118–132 (2007)
3. Bergman, D., Ciré, A., van Hoeve, W.: MDD propagation for sequence constraints. Journal of Artificial Intelligence Research **50**, 697–722 (2014)
4. Bergman, D., Ciré, A., van Hoeve, W., Hooker, J.: Decision diagrams for optimization. Springer (2016)
5. Bliek, C.: Wordwise algorithms and improved heuristics for solving hard constraint satisfaction problems. Tech. Rep. 12-96-R045, ERCIM (1996)
6. Bryant, R.: Graph-based algorithms for Boolean function manipulation. IEEE Transactions on Computers **35**(8), 677–691 (1986)
7. Cheng, K., Yap, R.: An MDD-based generalized arc consistency algorithm for positive and negative table constraints and some global constraints. Constraints **15**(2), 265–304 (2010)
8. 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: Proceedings of CP'16. pp. 207–223 (2016)
9. Gange, G., Stuckey, P., Szymanek, R.: MDD propagators with explanation. Constraints **16**(4), 407–429 (2011)
10. Hadzic, T., Hooker, J., O'Sullivan, B., Tiedemann, P.: Approximate compilation of constraints into multivalued decision diagrams. In: Proceedings of CP'08. pp. 448–462 (2008)
11. Hoda, S., van Hoeve, W., Hooker, J.: A systematic approach to MDD-Based constraint programming. In: Proceedings of CP'10. pp. 266–280 (2010)
12. Ingmar, L., Schulte, C.: Making compact-table compact. In: International Conference on Principles and Practice of Constraint Programming. pp. 210–218. Springer (2018)
13. Le Charlier, B., Khong, M.T., Lecoutre, C., Deville, Y.: Automatic synthesis of smart table constraints by abstraction of table constraints
14. Lecoutre, C.: STR2: Optimized simple tabular reduction for table constraints. Constraints **16**(4), 341–371 (2011)
15. Lecoutre, C., Boussemart, F., Hemery, F.: Exploiting multidirectionality in coarse-grained arc consistency algorithms. In: Proceedings of CP'03. pp. 480–494 (2003)
16. Lecoutre, C., Hemery, F.: A study of residual supports in arc consistency. In: Proceedings of IJCAI'07. pp. 125–130 (2007)
17. Lecoutre, C., Likitvivatanavong, C., Yap, R.: STR3: A path-optimal filtering algorithm for table constraints. Artificial Intelligence **220**, 1–27 (2015)
18. Lecoutre, C., Vion, J.: Enforcing arc consistency using bitwise operations. Constraint Programming Letters **2**, 21–35 (2008)
19. Likitvivatanavong, C., Zhang, Y., Bowen, J., Freuder, E.: Arc consistency in MAC: a new perspective. In: Proceedings of CPAI'04 workshop held with CP'04. pp. 93–107 (2004)
20. Mairy, J.B., Deville, Y., Lecoutre, C.: The smart table constraint. In: Proceedings of CPAIOR'15. pp. 271–287 (2015)
21. van Omme, N., Perron, L., Furnon, V.: or-tools user's manual. Tech. rep., Technical report, Google (2014), available from `https://github.com/google/or-tools`

22. OscaR Team: OscaR: Scala in OR (2012), available from `https://bitbucket.org/oscarlib/oscar`
23. Perez, G.: Decision diagrams: constraints and algorithms. Ph.D. thesis, Université de Nice (2017)
24. Perez, G., Régin, J.C.: Improving GAC-4 for Table and MDD constraints. In: Proceedings of CP'14. pp. 606–621 (2014)
25. Prud'homme, C., Fages, J.G., Lorca, X.: Choco3 documentation. TASC, INRIA Rennes, LINA CNRS UMR **6241** (2014)
26. le Clément de Saint-Marcq, V., Schaus, P., Solnon, C., Lecoutre, C.: Sparse-sets for domain implementation. In: Proceeding of TRICS'13. pp. 1–10 (2013)
27. de Saint-Marcq, V.L.C., Schaus, P., Solnon, C., Lecoutre, C.: Sparse-sets for domain implementation. In: (TRICS) Workshop on Techniques for Implementing Constraint programming Systems (2013)
28. Ullmann, J.: Partition search for non-binary constraint satisfaction. Information Science **177**, 3639–3678 (2007)
29. de Uña, D., Gange, G., Schachte, P., Stuckey, P.J.: Compiling cp subproblems to mdds and d-dnnfs. Constraints **24**(1), 56–93 (2019)
30. Verhaeghe, H., Lecoutre, C., Schaus, P.: Compact-mdd: Efficiently filtering (s) mdd constraints with reversible sparse bit-sets. In: IJCAI. pp. 1383–1389 (2018)
31. Verhaeghe, H., Lecoutre, C., Schaus, P.: Extending compact-diagram to basic smart multi-valued variable diagrams. In: International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research. pp. 581–598. Springer (2019)
32. Verhaeghe, H., Lecoutre, C., Deville, Y., Schaus, P.: Extending compact-table to basic smart tables. In: Proceedings of CP'17. pp. 297–307 (2017)
33. Verhaeghe, H., Lecoutre, C., Schaus, P.: Extending compact-table to negative and short tables. In: Proceedings of AAAI'17 (2017)
34. Wang, R., Xia, W., Yap, R., Li, Z.: Optimizing Simple Tabular Reduction with a bitwise representation. In: Proceedings of IJCAI'16. pp. 787–795 (2016)