Table Constraints for Integer Programming

Hendrik Bierlee[1]
KU Leuven, Belgium

Wout Piessens
KU Leuven, Belgium

Tias Guns
KU Leuven, Belgium

Peter J. Stuckey
Monash University, Melbourne, Australia
OPTIMA ITTC, Melbourne, Australia

Abstract

Global constraints are a central concept in *Constraint Programming* (CP), which allows modellers to compactly express complex relations, and allows solvers to efficiently handle them. Table constraints have especially been well-studied as they can express arbitrary finite relations, and are extensively used in CP benchmarks. In this paper we study how to best deal with table constraints when using *Integer Linear Programming* (ILP) solvers. We study two paradigms: linear encodings, and a lazy cut generation approach. For the encoding we propose a novel MDD-based flow encoding. For the cut generation, in which lazy constraints are generated on-demand during branch-and-cut search, we investigate different ways of generating such integer and fractional *cuts* as well as how to strengthen them through shrinking and cut lifting. We experimentally compare the different approaches on CP competition instances with a wide variety of table constraints, showing clear benefits over the standard integer encoding.

2012 ACM Subject Classification Software and its engineering → Constraint and logic languages; Mathematics of computing → Solvers; Applied computing → Operations research

Keywords and phrases Table constraints, integer programming, cut generation, modelling

Digital Object Identifier 10.4230/LIPIcs.CP.2026.37

Supplementary Material *Software (Source Code)*: `link`

Funding This research is partly funded by the European Research Council (ERC) under the EU Horizon 2020 research and innovation programme (Grant No 101002802, CHAT-Opt), by the Research Foundation-Flanders (FWO-Vlaanderen, G020525N), and by the Australian Government through the Australian Research Council Industrial Transformation Training Centre in Optimisation Technologies, Integrated Methodologies, and Applications (OPTIMA)

1 Introduction

Table constraints [4, 11, 19] are possibly the most well studied form of global constraint in the field of constraint programming. Indeed, Mackworth's original work on arc consistency [12] for (two variable) tables is one of the papers that originated the field. A (positive) table constraint allows us to specify an arbitrary finite relation among any number of variables, simply by listing all the possible solutions allowed.

[1] Corresponding author

© Hendrik Bierlee, Wout Piessens, Tias Guns, and Peter J. Stuckey;
licensed under Creative Commons License CC-BY 4.0

32nd International Conference on Principles and Practice of Constraint Programming (CP 2026).
Editor: Nicolas Beldiceanu; Article No. 37; pp. 37:1–37:20

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

▶ **Example 1.** Consider the following table constraint $\texttt{table}(\hat{x}, T)$ over a sequence of integer variables $\hat{x} \equiv [x_1, x_2, x_3]$ and integer matrix $T \equiv [[2, 1, 1], [3, 2, 2], [4, 3, 3], [1, 2, 3], [2, 1, 2]]$.

x_1	x_2	x_3
2	1	1
3	2	2
4	3	3
1	2	3
2	1	2

The only solutions to the table constraint are $x_1 = 2 \wedge x_2 = 1 \wedge x_3 = 1$, $x_1 = 3 \wedge x_2 = 2 \wedge x_3 = 2$, $x_1 = 4 \wedge x_2 = 3 \wedge x_3 = 3$, $x_1 = 1 \wedge x_2 = 2 \wedge x_3 = 3$, $x_1 = 2 \wedge x_2 = 1 \wedge x_3 = 2$; corresponding to the five rows in the table.

In this paper we only examine *positive tables*, where the table represents the set of solutions for the variables constrained. Table constraints arise naturally in many discrete optimisation problems, for example, in configurations where the capabilities and compatibilities of different components are naturally expressible using tables, and they often appear in *Constraint Programming* (CP) solver competition such as the XCSP3 competition [2].

While there have been many papers about how best to implement tables for constraint programming solvers and for propagating them, e.g. [4, 11], there has been little attention to how to implement tables in *Integer Linear Programming* (ILP) solvers. The widespread use of *solver-independent* modelling systems [14, 8] has meant that we now often write a high level CP model including `table` constraints, which can then subsequently be solved by an ILP solver; hence it is worth thoroughly investigating how to do this in the best way.

In this paper we investigate different approaches to implementing (positive) table constraints for ILP solvers. The contributions of the paper are:

- The first experimental comparison of different table encodings to ILP we are aware of.
- A novel approach to implementing tables in ILP using cut generation.
- An investigation of different choices available during cut generation.
- Experiments illustrating how different encodings perform, and when cut generation becomes beneficial.

Note that all technical proofs are included in the appendix (see Section A).

2 Preliminaries

This section will cover preliminaries and background. We will make use of *sequence notation* defined as follows: let $[]$ represent the empty sequence, and $[s_1, s_2, \ldots, s_n]$ represent a sequence of n objects $s_1, s_2, \ldots, s_n$ in order. We use $+\!\!+$ for sequence concatenation, e.g. $[1, 2] +\!\!+ [5, 2] = [1, 2, 5, 2]$. Given a sequence s, a prefix of s is any sequence p where there exists a sequence f such that $s = p +\!\!+ f$. We denote by $pre(s)$ the set of all prefixes of s, and extend this notation to sets of sequences S, $pre(S) = \bigcup_{s \in S} pre(s)$.

2.1 Constraint Programming (CP)

We cover the basics of CP following Rossi *et al.* [18]. We define a *Constraint Satisfaction Problem* (CSP) $P \equiv (\mathcal{X}, D, \mathcal{C})$ as a tuple of integer variables $\mathcal{X}$, with initial domains defined by D, and $\mathcal{C}$ a set of constraints. A constraint c with scope $scope(c) = [x_1, \ldots, x_n], x_i \in \mathcal{X}$ is defined (implicitly) as a set of solutions of the form $[\hat{v}_1, \ldots, \hat{v}_n]$ for its variable sequence. A *solution* of P is a valuation θ over variables $\mathcal{X}$ such that $\theta(x) \in D(x), x \in \mathcal{X}$ and $[\theta(x_1), \ldots, \theta(x_n)] \in c$ for all $c \in \mathcal{C}$ where $scope(c) = [x_1, \ldots, x_n]$.

2.2 Integer Linear Programming (ILP)

A finite-domain integer linear constraint has the form $\sum_{i=1}^{n} a_i x_i \ \# \ k$ where a_i, k are integer constants, x_i are finite-domain integer variables, and comparator $\# \in \{<, \leq, =, \geq, >\}$. An ILP model has exclusively linear constraints and a linear objective. We can *encode* a CSP P as an ILP Q by replacing the non-linear constraints in P with linear constraints in Q. Finding an effective encoding of CP constraints into ILP is challenging but important, as it can have a large impact on the solver's performance. An encoding often requires the introduction of additional variables, usually 01 integer variables to express complex constraints. See Belov *et al.* [3] for a discussion on encoding CP to ILP.

When encoding constraints to ILP, it often becomes useful to create a *domain encoding* (also known as direct, or unary encoding) of an integer variable x_i, where each domain value j that x_i can take is represented as a newly introduced *binary* (Boolean, 01) variable, $[\![x_i = j]\!]$ [17] which holds iff $x_i = j$:

$$\sum_{j \in D(x_i)} [\![x_i = j]\!] = 1 \tag{1a}$$

$$\sum_{j \in D(x_i)} j[\![x_i = j]\!] = x_i \tag{1b}$$

Equation (1a) enforces that exactly one value is selected for x. Depending on the encoding choices, the original integer variable x_i may be replaced entirely by its encoding variables. If instead x_i is in scope of other constraints in Q, it needs to be *channelled* to its encoding by Equation (1b).

An alternative to encoding a constraint c for ILP to linear constraints is to enforce the constraint by *(lazy) cut generation*. This is a form of (logic-based) Benders decomposition [10], where when a solution $\hat{v}$ of the relaxed problem is found, we test if it satisfies the constraint c. If so, nothing further is done for c. If not, then the solution $\hat{v}$ is called a *counterexample*. We add a linear constraint *cut*, which eliminates the counterexample, and is implied by c (so does not remove any solutions of c). Solving terminates when we find an optimal solution $\hat{v}^*$ which satisfies c. Cut generation is especially effective when a full encoding would require many constraints, such as the exponential number of sub-tour elimination constraints of a *Travelling Salesperson Problem* (TSP). ILP solvers support cut generation through callbacks, though this typically comes at the cost of more limited pre-solving being possible.

Linear constraints or generated cuts can be stronger or weaker. Given lower and upper bounds l_i and u_i on each integer variable $x_i \in \mathcal{X}$ (in an encoding of a CSP $l_i = \min D(x_i)$ and $u_i = \max D(x_i)$); we say a linear constraint $c_1 \equiv \sum_i a_i x_i \leq s$ is *stronger* than another linear constraint $c_2 \equiv \sum_i a'_i x_i \leq s'$ if every real solution of $c_1 \wedge \bigwedge_{x_i \in \mathcal{X}} (l_i \leq x_i \wedge x_i \leq u_i)$, is also a real solution of $c_2 \wedge \bigwedge_{x_i \in \mathcal{X}} (l_i \leq x_i \wedge x_i \leq u_i)$. A stronger constraint leaves fewer solutions in the real hypercube defined by the original bounds.

2.3 The `table` constraint

The table constraint $\texttt{table}(\hat{x}, T)$, where $\hat{x}$ is a vector of length k and T is a matrix of k columns and m rows (a sequence of m sequences of length k), enforces that the vector of variables $\hat{x}$ take values given by one of the rows in T (see Example 1). Table constraints are a highly flexible way of specifying arbitrary constraints, and their use and implementation in CP solvers has been heavily investigated [4, 11, 19].

More formally, a table T with m rows and k columns is a matrix of integer values $T_{ri}, r \in 1..m, i \in 1..k$. A 01 or *binary* table T is a table where $T_{ri} \in \{0,1\}$. Define T_i to be i^{th} column of T. If T is a binary table, we will abuse notation and treat this as a set of indices $T_i \equiv \{r \mid r \in 1..m, T_{ri} = 1\}$. Define $\hat{T}_r$ to be the rth row of T. If T is a binary table we will again abuse notation and treat this as a set of indices of columns, $\hat{T}_r = \{i \mid i \in 1..k, T_{ri} = 1\}$. Similarly a vector $\hat{x}$ of 01 values will sometimes be treated as a set of indices of columns.

We can simplify a table constraint $\texttt{table}(\hat{x}, T)$ to define an equivalent constraint system. First we can remove all duplicate rows of T. Second we can assume all variables in $\hat{x}$ are unique. Suppose $x_i = x_j, i < j$ then we can build table T' which removes column j and removes all rows r where $T_{r,i} \neq T_{r,j}$. The constraint $\texttt{table}([x_1, \ldots, x_{j-1}, x_{j+1}, \ldots, x_{n-1}, x_n], T')$ is equivalent to the original table.

Similarly we can remove any *constant* column j which contains only one value $T_{rj} = d, \forall r$ in a table T. The constraints $\texttt{table}\ ([x_1, \ldots, x_{j-1}, x_{j+1}, \ldots, x_{n-1}, x_n], T') \wedge x_j = d$, where T' is T with column j removed, is equivalent to the original table constraint.

3 Encoding table constraints to linear constraints

There are various ways to encode an integer table constraint $\texttt{table}(\hat{x}, T)$ to ILP.

3.1 Integer encoding

The integer variables can be used directly in a column-wise fashion as proposed by Belov *et al.* [3]. This encoding, to which we will refer as the Int encoding, adds binary row variables $r_i, 1 \leq i \leq m$, where $r_i = 1$ enforces that the ith row is the chosen solution, by encoding the table constraint $\texttt{table}(\hat{x}, T)$ as:

$$\sum_{l \in 1..m} r_l = 1 \tag{2}$$

$$\sum_{l \in 1..m} T_{li} r_l = x_i, \qquad i \in 1..k \tag{3}$$

The advantages of Int is that it is compact and its LP relaxation enforces that each x_i only takes values in the range $\min(T_i)..\max(T_i)$. It introduces m new 01 variables and $k+1$ linear equations with at most $m+1$ terms per equation. It is used to encode tables to ILP by default in MiniZinc [14].

▶ **Example 2.** The Int encoding of the table in Example 1 is

$$\begin{array}{rrrrrcl} r_1 & +r_2 & +r_3 & +r_4 & +r_5 & = & 1 \\ 2r_1 & +3r_2 & +4r_3 & +r_4 & +2r_5 & = & x_1 \\ r_1 & +2r_2 & +3r_3 & +2r_4 & +r_5 & = & x_2 \\ r_1 & +2r_2 & +3r_3 & +3r_4 & +2r_5 & = & x_3 \end{array}$$

3.2 Booleanized encoding

A Booleanized adaption of Int is proposed by Petit [16]. The original formulation also supports reification and negation. We limit ourselves to the positive table encoding, which we will call Bool. It uses the 01 table and the direct encoding of the integer variables. Since table constraints are typically used over *nominal* integers whose value represent discrete

objects, it is likely that other constraints will similarly reason over specific values of the integers (i.e. $x_i = v$), and hence require the direct encoding as well.

After introducing the Boolean encoding of the integers $\hat{x}$, we can rewrite the table constraint $\texttt{table}(\hat{x}, T)$ into the equivalent Boolean 01 table constraint $\texttt{table}(\hat{b}, T^{\mathbb{B}})$ with Boolean column variables $\hat{b}$ and 01 table $T^{\mathbb{B}}$ (with $n = \sum_{i=1}^{k} |D(x_i)|$ columns and still m rows):

$$\texttt{table}([[\![x_i = j]\!] \mid i \in 1..k, j \in D(x_i)], [[\langle T_{li} = j\rangle \mid i \in 1..k, j \in D(x_i)] \mid l \in 1..m]) \qquad (4)$$

where we use Iverson brackets to convert the conditions $T_{li} = j$ to 0 or 1 (e.g. $\langle 0 \leq -1\rangle = 0$). Every column is now a Boolean variable $[\![x_i = j]\!]$. We introduce the notation $p()$ to partition the columns by their *parent integer variable index*, i.e. $p([\![x_i = j]\!]) = i$. We can now encode the 01 table as in the previous section by introducing Boolean row variables $r_l, l \in 1..m$ and constraints

$$\sum_{l \in 1..m} r_l = 1 \qquad (5)$$

$$\sum_{l \in 1..m} \langle T_{li} = j\rangle r_l = [\![x_i = j]\!], \qquad i \in 1..k, j \in D(x_i) \qquad (6)$$

▶ **Example 3.** Consider the table constraint of Example 1 defined over three integer variables x_1, x_2, x_3 with domains $D(x_1) = 1..4$ and $D(x_2) = D(x_3) = 1..3$. The Boolean encoding of the variables introduces new Boolean variables $[\![x_i = j]\!], i \in 1..3, j \in D(x_i)$. We will use this notation to represent the Boolean variables themselves. The Boolean encoding of the variables introduces the `exactly-one` constraints $[\![x_1 = 1]\!] + [\![x_1 = 2]\!] + [\![x_1 = 3]\!] + [\![x_1 = 4]\!] = 1 \wedge \ldots$ and channelling constraints $x_1 = [\![x_1 = 1]\!] + 2[\![x_1 = 2]\!] + 3[\![x_1 = 3]\!] + 4[\![x_1 = 4]\!] \wedge \ldots$ for each of the integer variables. The Boolean version of the table $T^{\mathbb{B}}$ is:

	$[\![x_1{=}1]\!]$	$[\![x_1{=}2]\!]$	$[\![x_1{=}3]\!]$	$[\![x_1{=}4]\!]$	$[\![x_2{=}1]\!]$	$[\![x_2{=}2]\!]$	$[\![x_2{=}3]\!]$	$[\![x_3{=}1]\!]$	$[\![x_3{=}2]\!]$	$[\![x_3{=}3]\!]$
	0	1	0	0	1	0	0	1	0	0
	0	0	1	0	0	1	0	0	1	0
	0	0	0	1	0	0	1	0	0	1
	1	0	0	0	0	1	0	0	0	1
	0	1	0	0	1	0	0	0	1	0
p	1	1	1	1	2	2	2	3	3	3

Note the columns are partitioned by p (here, $\{[\![x_1 = 1]\!], [\![x_1 = 2]\!], [\![x_1 = 3]\!], [\![x_1 = 4]\!]\}$, $\{[\![x_2 = 1]\!], [\![x_2 = 2]\!], [\![x_2 = 3]\!]\}$, and $\{[\![x_3 = 1]\!], [\![x_3 = 2]\!], [\![x_3 = 3]\!]\}$. Notice how in each row exactly one value in each partition holds. The linear encoding of the Boolean table yields

$$\begin{array}{llllll} r_1 & +r_2 & +r_3 & +r_4 & +r_5 & = 1 \\ & & & r_4 & & = [\![x_1 = 1]\!] \\ r_1 & & & & +r_5 & = [\![x_1 = 2]\!] \\ & & & & & \vdots \\ & r_2 & & & +r_5 & = [\![x_3 = 2]\!] \\ & & r_3 & +r_4 & & = [\![x_3 = 3]\!] \end{array}$$

The Booleanized encoding introduces m 01 row variables and $n + 1$ linear equations with $m + 1$ terms for Equation (5), and for each partition $i \in 1 \ldots k$, $|D(x_i)|$ constraints of

type Equation (6) containing at most $m + |D(x_i)|$ terms in total. In that sense it is not much larger than the integer encoding, assuming we already have the domain encoding of the integers. Bool is *equivalent* to Int on the original variables $\hat{x}$. Any real solution of the Booleanized encoding generates a real solution of Int, and vice versa.

▶ **Theorem 4.** *Given a table constraint* $\texttt{table}(\hat{x}, T)$*. The real solutions of variables* $\hat{x}$ *for* Int *and* Bool *together with domain encoding constraints Equations* (1a) *and* (1b) *are identical.*

The advantage of the Booleanized encoding is that, as long as the binary variables are used elsewhere in the model, they can generate a stronger model.

▶ **Example 5.** Consider the table of Example 1 where branching has forced $[\![x_1 = 2]\!] = 0$ and $[\![x_3 = 3]\!] = 0$. Then the Booleanized encoding sets $r_2 = 1$, and $x_1 = 3$, $x_2 = 2$ and $x_3 = 2$. With Int we may know that $x_3 \leq 2$ but we cannot represent $x_1 \neq 2$ linearly. The real solutions of the resulting polytope do not collapse to a single solution.

3.3 MDD-based flow encoding

A table can be encoded more compactly as a *Multivalued Decision Diagram* (MDD) [6]. From this, an MDD can be encoded into linear constraints using a flow formulation, where each MDD node is associated with a constraint enforcing that the sum of incoming edge variables equals the sum of outgoing edge variables, for a total flow of 1. The advantage of encoding an MDD with a flow formulation is that it is a flow network using only linear constraints, and the resulting linear encoding is *totally unimodular* [1] meaning if there were no other constraints, then all extreme points of the polytope of the LP relaxation are integer points. The case of no other constraints is unlikely in practice, but it is still likely to make ILP solving easier. The greatest advantage of the MDD encoding is that it can can be *exponentially smaller* than the Booleanized encoding, since the MDD representing a table can be exponentially smaller than the original table.

The MDD for T has nodes for each (equivalence class of a) row prefix in T, where prefixes which share exactly the same set of completing suffixes in T share the same node. Let $pre(T)$ be all prefixes of rows occurring in T. Define the prefix equivalence classes of prefix s, $E(s)$ as $\{p \in pre(T) \mid \{f \mid p +\!\!+ f \in T\} = \{f \mid s +\!\!+ f \in T\}\}$, that is all prefixes that share the same set of completing suffixes are in the same equivalence class. Note that by this definition all complete rows $\hat{T}_r \in T$ are in the same equivalence class $E(\hat{T}_r)$, which we label *snk*. The graph encoding the MDD is defined by vertices $VM = \{E(s) \mid s \in pre(T)\}$, and edges EM defined as follows. For each prefix s of row $\hat{T}_r$ there is an edge labelled $[\![x_i = j]\!]$ from node $E(s)$ to $E(s +\!\!+ [j])$ in EM where $s +\!\!+ [j]$ is the next longest prefix of $\hat{T}_r$, and i is the length of $s +\!\!+ [j]$.

Figure 1 shows the MDD for Example 1. Note that the paths through the MDD are exactly $\{[2, 1, 1], [2, 1, 2], [1, 2, 3], [4, 3, 3], [3, 2, 2]\}$ making it isomorphic to the table, in terms of *integer solutions*. Note also that prefixes $[1, 2]$ and $[4, 3]$ share the same set of suffixes in the table ($[3]$) and hence share a node in the MDD, as well as all the complete sequences (which share suffix $[]$).

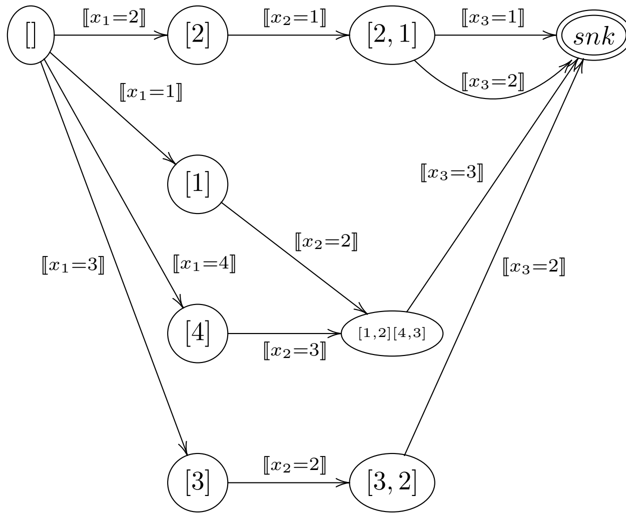


Figure 1 MDD encoding of Example 1.

The algorithm to convert a table to a reduced MDD, as in Figure 1, runs in $O(m \times k)$ [6].

Since a table constraint requires that exactly one row is active, we can encode the MDD as a flow network, where the flow from the root to the sink is exactly 1. This flow network directly leads to integer linear constraints. Since the MDD has at most one node per entry in the integer table (apart from the source node), the number of flow constraints is $O(m \times k)$. We create a set EV consisting of a 01 edge variable $e_{(s',j)}$ for each directed edge $(E(s), E(s +\!\!+ [j]))$ where we assume $s' = \min E(s)$ is the smallest representative of the equivalence class. The variable $e_{(s',j)}$ indicates whether there is a unit flow on the edge. Each edge variable $e_{(s,j)}$ is associated with a supporting Boolean variable $supp(e_{(s,j)}) = [\![x_i = j]\!]$, where $i = length(s) + 1$.

The encoding is

$$\begin{aligned}
&\sum_{(E(s),E(s+\!\!+[j]))\in EM} e_{(\min E(s),j)} \\
-&\sum_{(E(s+\!\!+[j]),E(s+\!\!+[j,k]))\in EM} e_{(\min E(s+\!\!+[j]),k)} &= 0, \quad E(s +\!\!+ [j]) \in VM \qquad (7)
\end{aligned}$$

$$\sum_{(E([]),E([j]))\in EM} e_{([],j)} = 1 \qquad (8)$$

$$\sum_{(E(s),E(s+\!\!+[j]))\in EM, E(s+\!\!+[j])=snk} e_{(\min E(s),j)} = 1 \qquad (9)$$

$$\sum_{e_{(s,j)}\in EV, supp(e_{(s,j)})=x_i} e_{(s,j)} = [\![x_i = j]\!], \quad i \in 1..k, j \in D(x_i) \qquad (10)$$

It consists of flow balance constraints for each internal node, requiring that the flow in equals the flow out (Equation (7)). We require one unit to flow out of node [] (Equation (8)) and one unit to flow into node *snk* (Equation (9)). We add constraints (Equation (10)) defining each $[\![x_i = j]\!]$: since every edge variable has a supporting encoding variable, we channel them by requiring that the encoding variable is true iff one of the edges it supports has a unit flow of one.

▶ **Example 6.** For the MDD shown in Figure 1 we generate the following linear constraints:

$$
\begin{array}{rcll}
[\![x_1 = 2]\!] & = & [\![x_2 = 1]\!] & ([2]) \\
[\![x_2 = 1]\!] & = & [\![x_3 = 1]\!] + e_{([2,1],2)} & ([2,1]) \\
[\![x_1 = 1]\!] & = & e_{([1],2)} & ([1]) \\
e_{([1],2)} + [\![x_2 = 3]\!] & = & [\![x_3 = 3]\!] & ([1,2],[4,3]) \\
[\![x_1 = 4]\!] & = & [\![x_2 = 3]\!] & ([4]) \\
[\![x_1 = 3]\!] & = & e_{([3],2)} & ([3]) \\
e_{([3],2)} & = & e_{([3,2],2)} & ([3,2]) \\
[\![x_1 = 1]\!] + [\![x_1 = 2]\!] + [\![x_1 = 3]\!] + [\![x_1 = 4]\!] & = & 1 & ([]) \\
e_{([2,1],2)} + [\![x_3 = 1]\!] + [\![x_3 = 3]\!] + e_{([3,2],2)} & = & 1 & (snk) \\
e_{([2,1],2)} + e_{([3,2],2)} & = & [\![x_3 = 2]\!] & \\
e_{([1],2)} + e_{[3],2)} & = & [\![x_2 = 2]\!] &
\end{array}
$$

Note that we obtain a larger number of constraints on the table of Example 1 than for INT and BOOL, but the constraints typically have fewer terms. We have applied a post-processing simplification in Example 6 where an edge variable supported by a single encoding variable is replaced by that encoding variable. Additionally, for MDD nodes $E(s +\!\!+ [j])$ with only one incoming and outgoing edge $e_{(\min E(s),j)}$ and $e_{(\min E(s++[j]),k)}$, we can remove the flow constraint and substitute $e_{(\min E(s),j)}$ for $e_{(\min E(s++[j]),k)}$ everywhere in the encoding: this is omitted from Example 6 for clarity.

We can show for the LP relaxation that the solutions of MDD and BOOL are equivalent.

▶ **Theorem 7.** *The solutions of the LP relaxation of the MDD encoding of* `table`$(\hat{x}, T)$ *on the variables* $[\![x_i = j]\!]$ *are the same as those for the* BOOL *encoding.*

Given a column permutation π, then `table`$(\hat{x}, T)$ is equivalent to `table`$(\pi(\hat{x}), \pi(T))$. For the previous encodings, the order of columns makes no difference, but for an MDD it can have a significant effect on its size. However, finding an optimal column ordering that minimizes MDD size is NP-complete [5]. Instead, we consider two column ordering heuristics. The first, MDD-INPUT, uses the original ordering. The second, MDD-DOM, orders the columns $i \in 1, \dots, k$ by increasing domain size of the associated variable $x_i, i \in 1, \dots, k$. The intuition behind MDD-DOM is that for variables with smaller domains, there are fewer choices to make, potentially causing more paths starting from the source to be merged.[2]

4 A cut generator for `table` constraints

In ILP, cut generation (see Section 2.2) has been successfully applied when an exponential number of constraints is required [10]. Unlike the exponential number of constraints which are required for e.g. TSP subtour elimination, all table encodings require at most a linear number of constraints in the size of the table. However, because tables can still be prohibitively large, it seems worthwhile to consider a *lazy* decomposition of `table` constraints.

We will now explore how to build an efficient cut generator for `table` constraints.

▶ **Example 8.** Consider the table of Example 1 again. Suppose the solver found an assignment $\hat{v} = [2, 2, 2]$ and calls the cut generator. The row does not appear in the table so $\hat{v}$ is a counterexample. The simplest way to generate a cut is to forbid this

[2] We experimented with other orderings, e.g. based on Fiedler [7], but this did not notably improve.

assignment: $\neg([\![x_1 = 2]\!] \wedge [\![x_2 = 2]\!] \wedge [\![x_3 = 2]\!])$ or equivalently the linear constraint $[\![x_1 = 2]\!] + [\![x_2 = 2]\!] + [\![x_3 = 2]\!] \leq 2$. A better cut is $[\![x_1 = 2]\!] + [\![x_2 = 2]\!] \leq 1$, since it also cuts off the counterexample without cutting off any row in the table. The latter is beneficial because it is strictly stronger, e.g. it also cuts off non-solutions $[2, 2, 1]$ and $[2, 2, 3]$ rather than only $[2, 2, 2]$.

Note that the use of the Boolean encoding variables $[\![x_i = j]\!]$ is vital for cut generation since without these variables there may be no valid cut. For example, there is no linear constraint $a_1x_1 + a_2x_2 + a_3x_3 \leq s$ over the integer variables x_i which cuts off solution $[2, 2, 2]$ without cutting off a row in the table of Example 1. The Boolean encoding variables always admit a cut, e.g. forbidding the complete assignment.

4.1 Initial cut generation

The cut generation algorithms will operate on a 01 table $T^{\mathbb{B}}$ as exemplified in Example 3, with a 01 counterexample $\hat{v}$ where $\hat{v}_j$ contains the value of the Boolean encoding variable b_j of 01 table column j. We will show how to generate a cut of the form $\sum_{j \in X} a_j b_j \leq s$ over a subset of the columns/Boolean variables $X \subseteq 1..n$. For now, the counterexample is assumed to be a *discrete* 01 assignment, $\hat{v} \in \mathbb{B}^n$, *fractional* counterexamples will be discussed in Section 4.2. For now, our cuts will have $a_j = 1, s = |X| - 1$, so they are defined by X alone. This type of cut forces at least one b_j to false, and thus encodes a logical disjunction of negative literals, $\bigvee_{j \in X} \neg b_j$. To generate the first cut from Example 8, we define $C(\hat{v}, l) = \{j \mid \hat{v}_j > 0 \wedge p(j) = l\}$ as all the non-zero variables/columns in partition l of $\hat{v}$. For discrete 01 counterexamples, C thus returns the single non-zero Boolean variable per partition. To generate the cut, we collect $X = \{j \mid l \in 1..k, j \in C(\hat{v}, l)\}$. This cut adds all variables assigned non-zero in $\hat{v}$; it cuts away the counterexample without cutting any assignment in the table (as none of its rows correspond to $\hat{v}$).

Algorithm 1 shows how we can generate stronger cuts of fewer terms by instead starting from an empty X, representing the *false* cut $0 \leq -1$ which cuts both $\hat{v}$ and all rows of $T^{\mathbb{B}}$. Then, we will add the non-zero Boolean variables from one column partition (i.e. one partition per integer variable) at a time, until no rows in $T^{\mathbb{B}}$ are cut. Invariantly, the cut represented by X will violate $\hat{v}$. In line 2, X and s are initialized accordingly. We use R to keep track of violated rows, initially all, and V to track which partitions are already used, initially none.

In the main loop of the algorithm in lines 4–9, we relax the cut by expanding X so that we can remove violated rows from R. While the cut still violates any row in the table, we choose a partition l in line 5 (i.e. an integer variable). Adding $C(\hat{v}, l)$ to X and incrementing s in lines 7–8 upholds the invariant (we defer the proof to Theorem 11 for a more general version of this algorithm). The new cut is weaker and may allow more rows in $T^{\mathbb{B}}$, so we remove those from the violated row set R in line 9, and repeat until R is empty.

▶ **Example 9.** Consider the (binary) table defined in Example 3. Suppose we wish to generate a cut to remove the discrete 01 counterexample $[\![x_1 = 2]\!] = [\![x_2 = 2]\!] = [\![x_3 = 2]\!] = 1$ and all others 0 (equivalently $[x_1, x_2, x_3] = [2, 2, 2]$). Initially $X = \varnothing, s = -1, R = 1..5, V = \varnothing$. Suppose we choose $l = 1$, corresponding to x_1, and add it to V. Then $C(\hat{v}, l) = \{2\}$, corresponding to $[\![x_1 = 2]\!]$, which is added to X, and s becomes 0. R is reduced to $\{1, 5\}$, the only two rows that contain value 1 in $[\![x_1 = 2]\!]$'s column, while the three rows removed from R would assign $[\![x_1 = 2]\!]$ to 0 which satisfies the cut. Next suppose we choose $l = 2$ and add it to V. Then $C(\hat{v}, l) = \{6\}$ which is added to X, s becomes 1, R is reduced to $\varnothing$ (intersected with $\{2, 4\}$). We return the valid cut $[\![x_1 = 2]\!] + [\![x_2 = 2]\!] \leq 1$.

Algorithm 1 The GENERATE algorithm for Boolean table constraints of m rows and n columns.

Require: Counterexample $\hat{v}$ violating $\texttt{table}(\hat{b}, T^{\mathbb{B}})$, with partitioning p.
1: **function** GENERATE($\hat{v}, \texttt{table}(\hat{b}, T^{\mathbb{B}})$)
2: **return** GENERATE($\hat{v}, \texttt{table}(\hat{b}, T^{\mathbb{B}}), \varnothing, -1, 1..m, \varnothing$)
3: **function** GENERATE($\hat{v}, \texttt{table}(\hat{b}, T^{\mathbb{B}}), X, s, R, V$)
4: **while** $R \neq \varnothing$ **do**
5: choose $l \in 1..k \setminus V$ ▷ Heuristically using Equation (11)
6: $V \leftarrow V \cup \{l\}$
7: $X \leftarrow X \cup C(\hat{v}, l)$
8: $s \leftarrow s + 1$
9: $R \leftarrow R \cap \bigcup_{j \in C(\hat{v},l)} T^{\mathbb{B}}_j$
10: **return** $\sum_{j \in X} b_j \leq s$

To choose a partition in line 5, we find one which leaves the fewest rows in R in line 9:

$$\arg\min_{l \in 1..k \setminus V} H(l) \text{ where } H(l) = |R \cap \bigcup_{j \in C(\hat{v},l)} T^{\mathbb{B}}_j| \tag{11}$$

This heuristic will often yield a small cut, but not necessarily a *minimal* one. To guarantee minimality, a SHRINK algorithm can be applied once we have obtained a valid cut in line 10. One approach, in the style of a deletion-based *Minimal Unsatisfiable Subset* algorithm, removes the term b_j if the cut without b_j still satisfies $T^{\mathbb{B}}$. Preliminary experiments found little benefit in doing so as few terms were removed at the cost of a relatively high overhead.

4.2 Fractional solutions

Algorithm 1 can be used generically for any solver producing discrete 01 solutions. However, when integrated as a callback procedure in an ILP solver, one can ask to be called also after the LP relaxation is solved, resulting in a fractional solution.

▶ **Example 10.** Suppose the fractional solution is $[0, 0.5, 0.5, 0, 0, 1, 0, 0, 1, 0]$ for variables $[[\![x_1 = 1]\!], \ldots, [\![x_3 = 3]\!]]$. Note how the sum of all variables belonging to the same integer/partition is still one due to Equation (1a). By analysing the four non-zero columns in Example 3, we can derive that while $[\![x_1 = 2]\!]$ holds in table rows 1 and 5, these rows do not satisfy the other non-zero columns, $[\![x_2 = 2]\!]$ and $[\![x_3 = 2]\!]$. Consequently, $\hat{v}$ can be cut by $[\![x_1 = 2]\!] + [\![x_2 = 2]\!] + [\![x_3 = 2]\!] \leq 2$ without cutting rows. Another fractional solution, $[0, 1, 0, 0, 1, 0, 0, 0.5, 0.5, 0]$, lies in the convex hull of the table, meaning it is a weighted average of table rows; hence each fractional column is supported by a row, so no separating cut exists.

The key idea is to identify such a fractional, non-zero column in $\hat{v}$, for which we can derive a cut. The algorithm is shown in Algorithm 2, which first collects all fractional column indices of $\hat{v}$ in F, and all indices of columns with value 1 in W. Line 6 then collects D as all *feasible* rows of the fractional solution, i.e. the rows in $T^{\mathbb{B}}$ that involve a subset of the non-zero columns of $\hat{v}$.

If there is now a fractional column that does not appear in any of the feasible rows, then we know that when selecting that column, there is no valid assignment to the non-zero columns of the other integer variables that would be supported in the table. In line 7, we collect all such columns in U, by checking they don't appear in any row of D. If however

$U = \varnothing$, we will not create a cut as $\hat{v}$ *may* appear in the convex hull of $T^{\mathbb{B}}$. While $\hat{v}$ could be outside the convex hull (see Lemma 16), this partial check is easier than a complete one. From any column in U we can generate a cut, and we use a heuristic similar to Equation (11) to select one. Let $X' = \bigcup_{l \in 1..k \setminus \{p(j)\}} C(\hat{v}, l)$ be the set of all binary columns of the integer variables/partitions except the partition $p(j)$ with non-zero column j. We know $\sum_{t \in X'} b_t = k - 1$ and that any extension of this involving b_j is not supported in the table, so $\sum_{t \in X'} b_t + b_j \leq k - 1$ does not eliminate a row from the table, but because b_j is non-zero in $\hat{v}$ it does cut the counterexample and hence it is a valid cut. We go one step further in the algorithm: we might not need the columns of all of the partitions in the cut. Hence, we start by adding column j to the cut, and call Algorithm 1 to expand it partition by partition until all rows are eliminated.

Algorithm 2 The FRACTIONAL algorithm for Boolean table constraints

Require: A (possibly fractional) counterexample $\hat{v}$, $\texttt{table}(\hat{b}, T^{\mathbb{B}})$ with partitioning p
1: **function** FRACTIONAL($\hat{v}, \texttt{table}(\hat{b}, T^{\mathbb{B}})$)
2: $F \leftarrow \{j \mid \hat{v}_j > 0 \land \hat{v}_j < 1\}$
3: **if** $F = \varnothing$ **then**
4: **return** GENERATE($\hat{v}, \texttt{table}(\hat{b}, T^{\mathbb{B}})$) ▷ Discrete 01 counterexample
5: $W \leftarrow \{j \mid \hat{v}_j = 1\}$
6: $D \leftarrow \{r \mid \hat{T}^{\mathbb{B}}_r \subseteq W \cup F\}$
7: $U = \{j \mid j \in F, \forall r \in D. T^{\mathbb{B}}_{rj} = 0\}$
8: **if** $U = \varnothing$ **then**
9: **return** *true*
10: $j \leftarrow \arg\min_{j \in U} |R \cap T^{\mathbb{B}}_j|$ ▷ Choose $j \in U$ heuristically similar to Equation (11)
11: **return** GENERATE($\hat{v}, \texttt{table}(\hat{b}, T^{\mathbb{B}}), \{j\}, 0, T^{\mathbb{B}}_j, \{p(j)\}$) ▷ Initializing X, s, R, and V

▶ **Theorem 11.** *FRACTIONAL($\hat{v}$, $\texttt{table}(\hat{b}, T^{\mathbb{B}})$) generates a correct cut eliminating $\hat{v}$ and no row of $T^{\mathbb{B}}$.*

▶ **Example 12.** Let us consider the fractional solution $\hat{v} = [0, 0.5, 0.5, 0, 0, 1, 0, 0.5, 0.5, 0]$. Then $F = \{2, 3, 8, 9\}$, $W = \{6\}$. There is only one feasible row, $D = \{2\}$, and it does not involve the fractional columns $U = \{2, 8\}$. Suppose we choose $j = 2$, then we construct $X = \{2\}, s = 0, R = \{1, 5\}$ and $V = \{p(2)\} = \{1\}$ and call Algorithm 1. Suppose it chooses $l = 2$, then X grows to $\{2, 6\}$ and $R = \varnothing$. The resulting cut is $[\![x_1 = 2]\!] + [\![x_2 = 2]\!] \leq 1$, the same as in Example 8, without the solver needing to find a discrete solution.

4.3 Cut lifting

A valid cut $\sum_{j \in X} a_j b_j \leq s$ can be made stronger by adding an unused j' to X, or increasing the coefficient $a_{j'}$ of an existing term, without raising the right-hand side s, called *lifting in a variable*. Suppose a *feasible* solution takes row r. Let $RS_r = \sum_{j \in X} a_j T^{\mathbb{B}}_{rj}$ be the value on the left-hand side of the cut. Since the original cut is valid, meaning it does not cut any row, we have $RS_r \leq s$. Lifting $a_{j'} b_{j'}$ will still satisfy row r, as long as either $T^{\mathbb{B}}_{rj'} = 0$, or if $a_{j'} < s - RS_r$. Since the lifted cut is at least as strong as the original, $\hat{v}$ remains cut.

Algorithm 3 efficiently finds candidate variables to lift. In line 3, we first calculate RS for all rows. In line 4, we collect *tight* rows R_{tight} for which the LHS sums to exactly s. In line 5, we find columns X' we cannot lift as they would increase RS_r for a tight row. In the main loop, we first choose a column j' we can lift. In line 9, we find a safe coefficient $a_{j'}$

such that no $RS_r > s$ after lifting. In line 10, we increment RS by $a_{j'}$ along the column j', avoiding recomputing RS by using a vector addition. In lines 11–13, we find the newly tightened rows N_{tight}, and update R_{tight} and X' accordingly. We prove cut lifting is correct if the original cut is valid, which is to say, the cut is a consequence (written $\models$) of the table because it does not eliminate any of its solutions/rows.

▶ **Theorem 13.** *Lifting is correct. Suppose* $T^{\mathbb{B}} \models \sum_{j\in X} a_j b_j \leq s$ *and* $T^{\mathbb{B}} \models b_{j'} \to \sum_{j\in X} a_j b_j \leq s - a_{j'}$ *then* $T^{\mathbb{B}} \models \sum_{j\in X\cup\{j'\}} a_j b_j \leq s$*.*

Algorithm 3 The CUTLIFT algorithm

Require: First argument is a valid cut for $T^{\mathbb{B}}$ with rows $1..m$

1: **function** CUTLIFT($\sum_{j\in X} a_j b_j \leq s, \mathtt{table}(\hat{b}, T^{\mathbb{B}})$)
2: $R \leftarrow 1..m$
3: $RS_r \leftarrow [\sum_{j\in X} a_j T^{\mathbb{B}}_{rj} \mid r \in R]$
4: $R_{\text{tight}} \leftarrow \{r \mid r \in R, RS_r = s\}$
5: $X' \leftarrow \bigcup_{r\in R_{\text{tight}}} \hat{T}^{\mathbb{B}}_r$
6: **while** $1..n \setminus X' \neq \varnothing$ **do**
7: choose $j' \in 1..n \setminus X'$ ▷ Heuristically using Equation (12)
8: $X \leftarrow X \cup \{j'\}$
9: $a_{j'} \leftarrow \min\{s - RS_r \mid r \in R \setminus R_{\text{tight}}, T^{\mathbb{B}}_{rj'} = 1\}$
10: $RS \leftarrow [RS_r + a_{j'} \mid r \in R, T^{\mathbb{B}}_{rj'} = 1]$
11: $N_{\text{tight}} \leftarrow \{r \mid r \in R \setminus R_{\text{tight}}, RS_r = s\}$
12: $R_{\text{tight}} \leftarrow R_{\text{tight}} \cup N_{\text{tight}}$
13: $X' \leftarrow X' \cup \bigcup_{r\in N_{\text{tight}}} \hat{T}^{\mathbb{B}}_r$
14: **return** $\sum_{j\in X} a_j b_j \leq s$

▶ **Example 14.** Consider the use of CUTLIFT on initial cut $[\![x_1 = 2]\!] + [\![x_2 = 2]\!] + [\![x_3 = 2]\!] \leq 2$. The $RS = [1, 2, 0, 1, 2]$ so $R_{\text{tight}} = \{2, 5\}$, so $X' = \{2, 3, 5, 6, 9\}$. Selecting $j' = 4$ we find $a_4 = 2$ and lift to get $[\![x_1 = 2]\!] + 2[\![x_1 = 4]\!] + [\![x_2 = 2]\!] + [\![x_3 = 2]\!] \leq 2$, then $RS = [1, 2, 2, 1, 2]$ and $R_{\text{tight}} = \{2, 3, 5\}$, so X' is extended by $\{4, 7, 10\}$. Repeating this twice with $j' = 8, a_8 = 1, N_{tight} = \{1\}$, then $j' = 1, a_1 = 1, N_{tight} = \{4\}$, terminates CUTLIFT with $X' = 1..n$ and the stronger cut $[\![x_1 = 1]\!] + [\![x_1 = 2]\!] + 2[\![x_1 = 4]\!] + [\![x_2 = 2]\!] + [\![x_3 = 1]\!] + [\![x_3 = 2]\!] \leq 2$.

To choose a column to lift, we base a heuristic on the *centre of mass* $\hat{c}$ of $T^{\mathbb{B}}$, the vector defined by each column's average value. The difference $\hat{c} - \hat{v}$ is the direction from $\hat{v}$ towards the centre. We lift the variable $b_{j'}$ which moves the cut most into this direction:

$$\underset{j'\in 1..n\setminus X'}{\arg\max} \frac{\sum_{r\in R} \hat{T}_{rj'}}{m} - \hat{v}_{j'} \tag{12}$$

4.4 A hybrid eager-lazy approach

Given a constraint specification, for each table constraint we can choose whether it should be eagerly encoded, or handled lazily with cut generation. Since lazy constraints are commonly used when the encoding is large, a sensible threshold is based on the number of rows. Secondly, if there are many rows, it is less likely that all rows are as relevant, e.g. many rows may not be feasible due to other constraints.

5 Experimental evaluation

We evaluate on a set of benchmarks with a wide variety of tables. We run each encoding described in Section 3, and perform a comparative study on the cut generation extensions described in Section 4.

Benchmarks We collect all instances from the CSP and COP tracks of XCSP3'22–25 competition [2]. We omit all instances which have no top-level table constraints, which leaves 180 CSP and 247 *Constrained Optimisation Problem* (COP) instances. This accounts for roughly 25% of all available CSP3 instances, showing the prevalence of this constraint in the competition. We check assignments of feasible results using the XCSP3 checker[3], and we check that there are no contradictions between solvers on feasibility results or on optimal objective values.

Solvers We use Gurobi v. 13.0.1 [9] as ILP solver, and CPMpy v0.10.0 [8] to load and linearize the XCSP3 instances, and to implement the novel encodings and cut generator. Gurobi is run with default parameters, with two exceptions. First, the `MipGap` and `MipGapAbs` parameters are tweaked for all approaches to prevent sub-optimal solutions from being reported as optimal. Secondly, whenever cut generation is used, the `LazyConstraints` flag is enabled. This disables certain Gurobi features incompatible with lazy constraints, which does give the eager approach a noticable advantage. The implementation is freely available.[4]

Resources We run each instance on a single core of an Intel(R) Xeon(R) CPU E5-2640 v3 @ 2.60GHz, and impose a 600 second time limit, and an 8 GB memory limit.

We aim to answer the following experimental questions:

Q1. What is the impact of each encoding described in Section 3?
Q2. What is the impact of each cut generation feature described in Section 4?
Q3. When does lazy cut generation become beneficial compared to using an eager encoding?

A full overview of the results is shown in Table 1. Supplemental performance profiles for a few selected approaches are provided in the appendix (see Figure 3 in Section B). On these XCSP3 instances, CP-SAT/OR-Tools (v. 9.15.6755) [15], which won 1st place in the 2025 COP track and 2nd place in the 2025 CSP track, can solve 115 CSP instances and 98 COP instances. This is considerably more than the ILP approaches in this paper, and we do not believe that improving the encoding of one global constraint would suddenly lead to ILP solvers outperforming native CP solvers on this benchmark. However, the XCSP3 instances form an appropriate dataset to investigate how improvements in encodings lead to improvements in ILP solver performance.

5.1 Impact of encodings (Q1)

To answer **Q1.**, we compare all table encodings described in Section 3.

For the CSP instances in Table 1a, we observe that Bool shows better performance than Int. This supports the idea that the table variables are *nominal* in nature, and benefit from a direct encoding that can be re-used throughout the model in the other constraints, as discussed in Section 3.2. As expected, Bool has more constraints than Int due to the larger

[3] https://xcsp.org/tools/
[4] https://github.com/ML-KULeuven/table-constraints-for-integer-programming

Table 1 Overview of results. The reported statuses are `POST` for linearizing and posting the model, `FEAS` for finding a feasible solution, or `SOLVED` for solved (i.e. reported unsatisfiable, or `FEAS` for CSPs, or optimal for COPs), within the time- and memory limit. We report the average solve time (including encode/post time) over all instances, with a PAR2 penalty for unsolved instances. We also show the average number of constraints required to encode the instances which were successfully posted by all approaches, and the average number of cuts to solve instances which were solved by all lazy approaches. The best result for the various columns is shown in **bold**.

(a) 180 CSP instances (mean rows: 20 918, stdev: 1 204 503), `FEAS` intentionally empty

	POST	FEAS	SOLVED	time	constraints	cuts
INT	171		55	851.7	8 249.3	–
BOOL	147		69	771.5	25 670.4	–
MDD-INPUT	154		67	780.8	38 800.8	–
MDD-DOM	154		76	728.5	30 884.2	–
GENERATE	174		36	988.6	8 639.7	1 857.5
FRACTIONAL	174		36	980.9	8 639.7	3 202.6
CUTLIFT	174		45	925.0	8 639.7	847.1
ALL	174		55	871.5	8 639.7	1 326.7
HYBRID (1000)	174		79	717.5	29 635.4	49.9
HYBRID (2000)	174		**80**	**715.6**	29 912.8	49.9
HYBRID (3000)	**175**		78	716.7	30 243.3	37.3

(b) 247 COP instances (mean rows: 9 548, stdev: 8 608 271)

	POST	FEAS	SOLVED	time	constraints	cuts
INT	**166**	139	73	864.3	9 525.9	–
BOOL	150	134	71	875.6	24 155.4	–
MDD-INPUT	151	133	**75**	859.0	36 284.4	–
MDD-DOM	157	137	**75**	**858.8**	35 563.0	–
GENERATE	**166**	112	58	936.4	15 504.4	1 436.3
FRACTIONAL	**166**	128	61	926.9	15 504.4	1 701.7
CUTLIFT	**166**	126	66	904.3	15 504.4	1 256.0
ALL	**166**	136	61	923.7	15 504.4	1 424.0
HYBRID (1000)	164	143	72	871.6	34 787.8	1.7
HYBRID (2000)	164	**145**	74	864.1	35 496.4	1.9
HYBRID (3000)	164	144	74	863.9	35 503.5	0.2

number of columns in the Boolean table $T^{\mathbb{B}}$ compared to the integer table T. We observe that MDD-INPUT outperforms INT by 12 instances. MDD-INPUT does not outperform BOOL. However, MDD-DOM does outperform BOOL by 7 instances, illustrating the benefit of this column ordering heuristic. MDD-INPUT has even more constraints than INT and BOOL, but usually with fewer terms (see Section 3.3), thereby still leading to a strong solving performance. The same goes for MDD-DOM, although we also observe that the number of constraints decreases compared to MDD-INPUT since more nodes in the MDD are merged.

For the COP instances shown in Table 1b, INT is somewhat better than BOOL, with the MDD-INPUT approach outperforming both in the number of instances proven optimal. While MDD-DOM does not improve this result further, it does obtain 4 additional feasible results. INT obtains even more feasible results. In general, the difference in performance is smaller compared to the CSP instances. For instance, consider that if an integer variable occurs both in the objective and in a table, having to link direct encoding variables to this integer variable with a channelling constraint could be less ideal compared to using INT.

5.2 Impact of cut generation (Q2)

To answer **Q2.**, we perform a comparative study on the cut generation algorithm GENERATE (Section 4.1), and the impact of adding the extensions FRACTIONAL (see Section 4.2) and CUTLIFT (see Section 4.3), individually, and altogether in ALL.

For the CSP instances shown in Table 1a, we can see that FRACTIONAL only slightly improves on GENERATE in time. Due to the fractional solutions, more cuts are required to solve instances. On the other hand, CUTLIFT improves on GENERATE with 9 additional feasible instances. We see this reflected in fewer cuts, indicating stronger ones were generated. When both extensions are combined in ALL, this outperforms CUTLIFT by 10 instances. For COP shown in Table 1b, again FRACTIONAL leads to a small and CUTLIFT to a larger improvement. Unlike for CSP, CUTLIFT outperforms ALL by proving optimality for 5 additional instances, however, ALL does find significantly more feasible solutions.

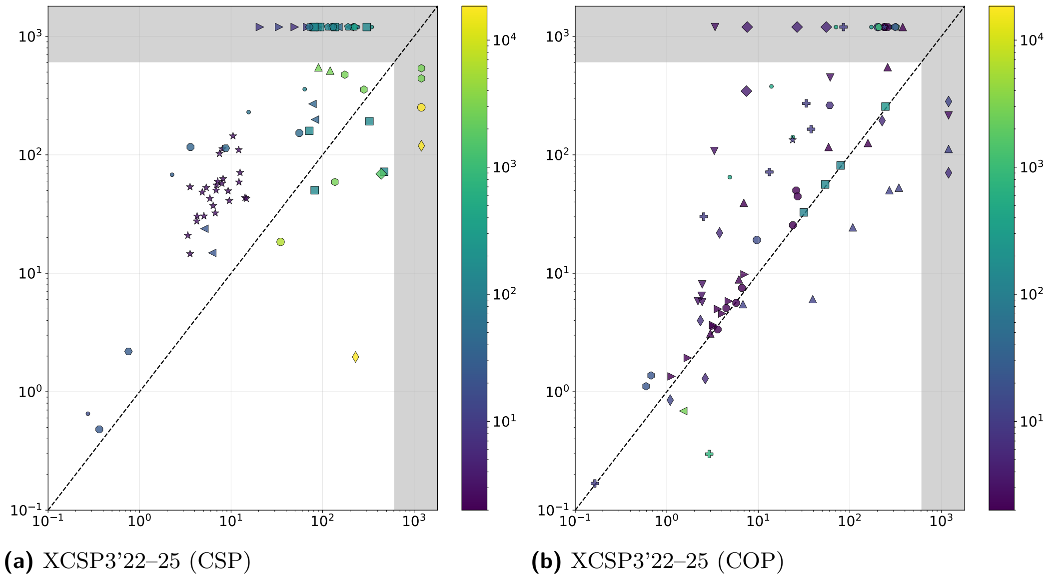
5.3 Comparison between encoding and cut generation (Q3)

To answer **Q3.**, we compare the eager and lazy approaches globally. Then, we analyse which instances benefit more from cut generation than from the eager encoding and v.v., and when hybridizing the approaches as discussed in Section 4.4 becomes worthwhile.

The results in Table 1 show that globally, the eager encodings outperform the lazy approach, especially for proving optimality. The lazy approach is able to post more instances in total. The number of constraints is low for lazy, yet the direct encoding of the integers and binary tables (see Section 2.3) still yields more constraints on average than INT.

In Figure 2, we compare the two best performing configurations of each approach for CSP, namely MDD-DOM and ALL. We show the median number of rows for each instance. For CSP instances in Figure 2a, we observe that MDD-DOM tends to perform better on smaller tables (with fewer rows), whereas ALL becomes advantageous for larger tables. For COP instances in Figure 2b, no correlation can be observed between table size and performance for the two paradigms. Nevertheless, some instances benefit more from eager encodings, while others benefit more from cut generation.

Figure 2 motivates the development of the HYBRID approach (see Section 4.4). We show results for three different table size thresholds: at least 1000, 2000, or 3000 rows for lazy. Back in Table 1, we can see the average number of constraints increasing again with the threshold, whereas the number of required cuts has been drastically reduced. In both tracks,



(a) XCSP3'22–25 (CSP) **(b)** XCSP3'22–25 (COP)

Figure 2 Scatter plots of solve time in seconds for all instances of the XCSP3'22–25 CSP and COP tracks. Compares ALL (x-axis) against MDD-DOM (y-axis), meaning bottom/right solves faster for ALL and top/left solves faster for MDD-DOM. Instances in the top/right grey area have reached timeout, and instances which timeout for both approaches are omitted. The colour of an instance indicates the median number of rows in its tables, and its shape indicates its problem class.

HYBRID (2000) strikes the best balance, as we see diminishing results from 1000 and 3000. A large improvement is made by HYBRID (2000) compared to ALL in solving CSPs, and to a lesser extent also for COPs. This is unsurprising given the strength of the MDD-based encoding. For CSPs, all HYBRID approaches outperform MDD-DOM. For COPs, HYBRID (2000) loses one solved instance from MDD-DOM, but finds the most feasible results of all approaches, outperforming INT by 6 additional feasible instances.

6 Conclusion and future work

We have studied how to tackle table constraints when using ILP solvers. Two paradigms were explored in detail: eager encodings into linear constraints, and a lazy cut generation approach. We experimented with different encodings and different extensions to the cut generation algorithm. The results show that the novel encodings convincingly outperform existing methods. The lazy approach is improved when both its extensions are enabled, and we find it better suited at finding feasible solutions rather than proving optimality. A more careful examination reveals the paradigms to be complementary, leading to a hybridization which improves the state-of-the-art for ILP solvers on this important constraint.

In future work, we hope to expand our understanding of when a table is suitable for which approach. For instance, a correlation may exist between the sparsity of a table and the performance of a method. Moreover, variations of the table constraint, such as *negative tables*, *smart tables* [13], and half- or fully reified tables, could benefit from a similar approach. For the MDD-based encoding, an algorithm for converting negative tables to MDDs already exists [6], analogous to the algorithm for positive tables, whereas it is an open question how to convert smart tables to MDDs so that their sizes scale proportionally. Our methods should easily extend to half-reification, but full reification will require additional work.

References

1. Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice hall, 1993.
2. Gilles Audemard, Christophe Lecoutre, and Emmanuel Lonca. Proceedings of the 2025 xcsp3 competition, 2025. URL: `https://arxiv.org/abs/2511.06918`, `arXiv:2511.06918`.
3. Gleb Belov, Peter J. Stuckey, Guido Tack, and Mark Wallace. Improved Linearization of Constraint Programming Models. In Michel Rueher, editor, *Principles and Practice of Constraint Programming*, pages 49–65, Cham, 2016. Springer International Publishing. `doi:10.1007/978-3-319-44953-1_4`.
4. Christian Bessière and Jean-Charles Régin. Arc consistency for general constraint networks: Preliminary results. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence, IJCAI 97, Nagoya, Japan, August 23-29, 1997, 2 Volumes*, pages 398–404. Morgan Kaufmann, 1997.
5. Beate Bollig and Ingo Wegener. Improving the variable ordering of obdds is np-complete. *IEEE Trans. Computers*, 45(9):993–1002, 1996. `doi:10.1109/12.537122`.
6. Kenil C. K. Cheng and Roland H. C. Yap. An mdd-based generalized arc consistency algorithm for positive and negative table constraints and some global constraints. *Constraints An Int. J.*, 15(2):265–304, 2010. URL: `https://doi.org/10.1007/s10601-009-9087-y`, `doi:10.1007/S10601-009-9087-Y`.
7. Miroslav Fiedler. A property of eigenvectors of nonnegative symmetric matrices and its application to graph theory. *Czechoslovak Mathematical Journal*, 25(4):619–633, 1975. URL: `http://eudml.org/doc/12900`.
8. Tias Guns. Increasing modeling language convenience with a universal n-dimensional array, cppy as python-embedded example. In *Proceedings of the 18th workshop on Constraint Modelling and Reformulation at CP (Modref 2019)*, volume 19, 2019.
9. Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2026. URL: `https://www.gurobi.com`.
10. John N. Hooker. Planning and scheduling by logic-based Benders decomposition. *Oper. Res.*, 55(3):588–602, 2007. URL: `https://doi.org/10.1287/opre.1060.0371`, `doi:10.1287/OPRE.1060.0371`.
11. Christophe Lecoutre. Str2: optimized simple tabular reduction for table constraints. *Constraints*, 16(4):341–371, 2011. `doi:10.1007/s10601-011-9107-6`.
12. Alan K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1977. URL: `https://www.sciencedirect.com/science/article/pii/0004370277900078`, `doi:10.1016/0004-3702(77)90007-8`.
13. Jean-Baptiste Mairy, Yves Deville, and Christophe Lecoutre. The smart table constraint. In Laurent Michel, editor, *Integration of AI and OR Techniques in Constraint Programming - 12th International Conference, CPAIOR 2015, Barcelona, Spain, May 18-22, 2015, Proceedings*, volume 9075 of *Lecture Notes in Computer Science*, pages 271–287. Springer, 2015. `doi:10.1007/978-3-319-18008-3_19`.
14. N. Nethercote, P.J. Stuckey, R. Becket, S. Brand, G.J. Duck, and G. Tack. Minizinc: Towards a standard CP modelling language. In C. Bessiere, editor, *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming*, volume 4741 of *LNCS*, pages 529–543. Springer-Verlag, 2007.
15. Laurent Perron and Frédéric Didier. Cp-sat. URL: `https://developers.google.com/optimization/cp/cp_solver/`.
16. Thierry Petit. On constraint linear decompositions using mathematical variables. In *29th IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2017, Boston, MA, USA, November 6-8, 2017*, pages 123–130. IEEE Computer Society, 2017. `doi:10.1109/ICTAI.2017.00030`.

17 Philippe Refalo. Linear Formulation of Constraint Programming Models and Hybrid Solvers. In Rina Dechter, editor, *Principles and Practice of Constraint Programming – CP 2000*, pages 369–383, Berlin, Heidelberg, 2000. Springer. doi:10.1007/3-540-45349-0_27.

18 Francesca Rossi, Peter van Beek, and Toby Walsh, editors. *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*. Elsevier, 2006. URL: https://www.sciencedirect.com/science/bookseries/15746526/2.

19 Roland H. C. Yap, Wei Xia, and Ruiwei Wang. Generalized arc consistency algorithms for table constraints: A summary of algorithmic ideas. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*, pages 13590–13597. AAAI Press, 2020. URL: https://doi.org/10.1609/aaai.v34i09.7086, doi:10.1609/AAAI.V34I09.7086.

A Proofs

▶ **Theorem 4.** *Given a table constraint* $\texttt{table}(\hat{x}, T)$*. Let* ES *be the encoding using Equations* (2) *and* (3)*. Let* BV *be the binary encoding of the integer variables* $\hat{x}$ *according to Equations* (1a) *and* (1b)*. Let* BS *be the encoding of the binarised version of table constraint* $\texttt{table}(\hat{x}, T)$ *defined by Equations* (5) *and* (6)*. Any solution of* ES *is extendible to a solution of* $BS \cup BV$ *and any solution of* $BS \cup BV$ *projected onto* $\hat{x}$ *and* $\hat{r}$ *is a solution of* ES*.*

Proof. Each equation of the form Equation (2) also appears in BS. We show that any equation $\sum_{l \in 1..m} T_{li} r_l = x_i \in ES$ of the form Equation (3) is a linear integer consequence of equations in $BV \cup BS$. Now $x_i = \sum_{j \in D(x_i)} j[\![x = j]\!] \in BV$. For each value $j \in D(x_i)$ we have a row $\sum_{l \in 1..m, T_{li}=j} r_l = [\![x_i = j]\!] \in BS$. Since each column r_l appears in at most one of these row constraints, if we replace $[\![x_i = j]\!]$ in $\sum_{j \in D(x)} j[\![x_i = j]\!]$ by the left hand side we exactly obtain $\sum_{l \in 1..m} T_{li} r_l$. As a consequence, we can always extend a solution of ES to a solution of $BS \cup BV$ by assigning $[\![x_i = j]\!]$ variables using the equations in BS. Similarly any solution of $BS \cup BV$ is a solution of ES. ◀

▶ **Theorem 7.** *Given a table constraint* $\texttt{table}(\hat{x}, T)$*. Let* BS *be the encoding of the binarised version of table constraint* $\texttt{table}(\hat{x}, T^{\mathbb{B}})$ *defined by Equations* (5) *and* (6)*. Let* MS *be the encoding of the MDD of table constraint* $\texttt{table}(\hat{x}, T^{\mathbb{B}})$ *defined by Equations* (7) *and* (10)*. Then the real solutions of* BS *and* ES *on the variables* $[\![x_i = j]\!], i \in 1..k, j \in D(x_i)$ *are equivalent.*

Proof. (Sketch) Consider a solution of the binarised encoding BS, gives a value for each r variable. We can set $e_{(s,j)}$ to equal to the sum of the row variables for rows which include the prefix $s' +\!\!+ [j]$, $s' \in E(s)$. This leads to the same solution on the $[\![x_i = j]\!]$ variables. Consider a flow solution θ on the variables $e_{(s,j)}$. This defines a solution on the Boolean encoding variables $[\![x_i = j]\!]$. We construct a solution for the row variables $\hat{r}$ that gives the same solution on $[\![x_i = j]\!]$ since flow variables $e_{(s,j)}$ that correspond to different rows with identical prefixes and identical sets of suffixes can always be broken into a sum of flows through the individual rows by the conservation of flow. ◀

▶ **Theorem 11.** *FRACTIONAL*$(\hat{v}, T^{\mathbb{B}})$ *generates a correct cut eliminating* $\hat{v}$ *and no row of* $T^{\mathbb{B}}$*.*

Proof. First note that the cut always cuts off $\hat{v}$ since at all stages $\sum_{j \in X} \hat{v}_j > s$. Initially this is true since either $X = \varnothing$ and $s = -1$ (if $F = \varnothing$) or $\hat{v}_j > 0, s = 0$, and in each iteration of the loop we add to X all columns for variable l, hence adding 1 to the sum, and we also add 1 to s.

We now show by induction that each row in R satisfies $\sum_{j \in X} b_j = s + 1$, and each row not in R satisfies $\sum_{j \in X} b_j \leq s$. Initially either $X = \varnothing$ and $s = -1$ and this holds trivially, or $X = \{j\}$ and each row $r \in R$ has $T^{\mathbb{B}}_{rj} = 1$ satisfying the equality condition. Each row $r' \notin R$ has $T^{\mathbb{B}}_{r'j} = 0$ thus satisfying the inequality condition.

In the loop iteration, we choose an unselected variable l and add $C(\hat{v}, l)$ to X and 1 to s. For rows r remaining in R we have that $T^{\mathbb{B}}_{rj} = 1$ for some $j \in C(\hat{v}, l)$ hence they continue to satisfy the equality condition. For rows r leaving R we have that $T^{\mathbb{B}}_{rj} = 0, \forall j \in C(\hat{v}, l)$ and hence they now satisfy the inequality condition. For rows r already not in R, we add at most one to the sum in the inequality condition (since all new variables added to X are from a single integer), and we add one to the right hand side, thus they continue to satisfy the inequality condition.

Finally, we show that the loop terminates, since we can always choose $X = \{j\} \cup \bigcup_{l \in 1..k \setminus \{p(j)\}} C(\hat{v}, l)$ to eliminate all rows R, because no rows in D include column j, and all other rows include a coefficient not in $X \subset W \cup F$. ◀

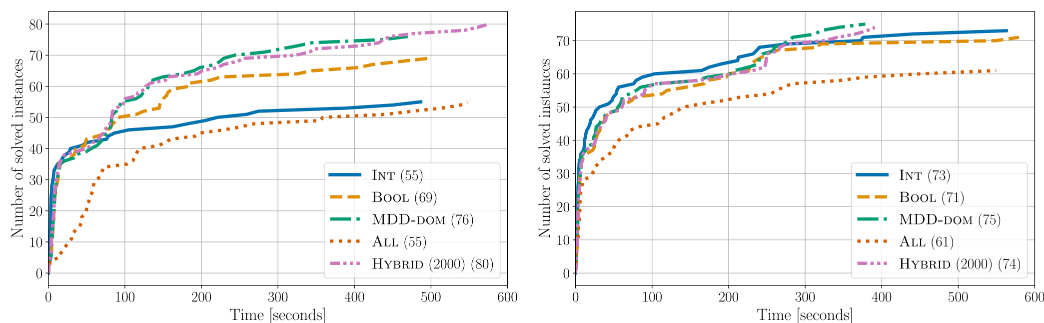
▶ **Theorem 13.** *Lifting is correct. Suppose $T^{\mathbb{B}} \models \sum_{j \in X} a_j b_j \leq s$ and $T^{\mathbb{B}} \models b_{j'} \rightarrow \sum_{j \in X} a_j b_j \leq s - a_{j'}$ then $T^{\mathbb{B}} \models \sum_{j \in X \cup \{j'\}} a_j b_j \leq s$.*

Proof. Suppose to the contrary the lifted cut $\sum_{j \in X \cup \{j'\}} a_j b_j \leq s$ is not a consequence of the table $T^{\mathbb{B}}$. Then there is a row that violates the constraint, say r. So $\sum_{j \in X \cup \{j'\}} a_j T^{\mathbb{B}}_{rj} > s$. Now in row r if $T^{\mathbb{B}}_{rj'}$ is true then by the implication $\sum_{j \in X} a_j T^{\mathbb{B}}_{rj} \leq s - a_{j'}$ and clearly the constraint cannot be violated. Otherwise if $T^{\mathbb{B}}_{rj'} = 0$ then since $\sum_{j \in X} a_j T^{\mathbb{B}}_{rj} \leq s$, by correctness of the original cut, we have a contradiction. ◀

▶ **Lemma 16.** *If $\hat{v}$ is in the convex hull of $T^{\mathbb{B}}$ then $U = \varnothing$.*

Proof. Suppose $\hat{v} = \bar{\lambda} T^{\mathbb{B}}$ then clearly $\lambda_r = 0$ for $r \notin D$ since otherwise we get a non-zero value in a column not in $\hat{v}$. Suppose to the contrary $\exists j \in U$ then $T^{\mathbb{B}}_{rj} = 0, \forall r \in D$, but $\hat{v}_j > 0$. Contradiction since $\lambda_r = 0$ for all rows where $T^{\mathbb{B}}_{rj} > 0$. ◀

B Performance profiles



(a) XCSP3'22–25 (CSP) **(b)** XCSP3'22–25 (COP)

Figure 3 Cactus plots for a few selected approaches, showing the number of instances solved on the y-axis by the solve time in seconds on the x-axis. Higher is better.