Journal of Artificial Intelligence Research 82 (2025) 1999-2056 Submitted 08/2024; published 03/2025

TabID: Automatic Identification and Tabulation of Subproblems in Constraint Models

Özgür Akgün ozgur.akgun@st-andrews.ac.uk
Ian P. Gent Ian.Gent@st-andrews.ac.uk
School of Computer Science, University of St Andrews
St Andrews, Fife KY16 9SX, UK

Christopher Jefferson cjefferson001@dundee.ac.uk
School of Science and Engineering, University of Dundee
Dundee, DD1 4HN, UK

Zeynep Kiziltan zeynep.kiziltan@unibo.it
Department of Computer Science and Engineering
University of Bologna
Mura Anteo Zamboni 7, 40126, Bologna, Italy

Ian Miguel ijm@st-andrews.ac.uk
School of Computer Science, University of St Andrews
St Andrews, Fife KY16 9SX, UK

Peter Nightingale peter.nightingale@york.ac.uk
Department of Computer Science, University of York
Heslington, York YO10 5GH, UK

András Z. Salamon Andras.Salamon@st-andrews.ac.uk
School of Computer Science, University of St Andrews
St Andrews, Fife KY16 9SX, UK

Felix Ulrich-Oltean felix.ulrich-oltean@york.ac.uk
Department of Computer Science, University of York
Heslington, York YO10 5GH, UK

Abstract

The performance of a constraint model can often be improved by converting a subproblem into a single table constraint (referred to as tabulation). Finding subproblems to tabulate is traditionally a manual and time-intensive process, even for expert modellers. This paper presents TabID, an entirely automated method to identify promising subproblems for tabulation in constraint programming. We introduce a diverse set of heuristics designed to identify promising candidates for tabulation, aiming to improve solver performance. These heuristics are intended to encapsulate various factors that contribute to useful tabulation. We also present additional checks to limit the potential drawbacks of suboptimal tabulation.

We comprehensively evaluate our approach using benchmark problems from existing literature that previously relied on manual identification by constraint programming experts of constraints to tabulate. We demonstrate that our automated identification and tabulation process achieves comparable, and in some cases improved results. We empirically evaluate the efficacy of our approach on a variety of solvers, including standard CP (Minion and Gecode), clause-learning CP (Chuffed and OR-Tools) and SAT solvers (Kissat).

Our findings highlight the substantial potential of fully automated tabulation, suggesting its integration into automated model reformulation tools.

1. Introduction

Constraint programming provides an efficient means of solving complex combinatorial problems across a wide variety of disciplines, such as scheduling, planning, routing, and configuration (Rossi et al., 2006). In order to solve a problem using the constraint programming paradigm, it must first be *modelled* in a format suitable for input to a constraint solver. This involves determining the set of decision variables that represent the choices that must be made to solve the problem, and formulating a set of constraints over the variables so as to allow only valid combinations of decisions. For example, in a scheduling scenario we might employ a decision variable per task to represent the start time of that task, with constraints to disallow a number of simultaneous tasks that would exceed the resources available.

Once a model has been chosen, a constraint solver automatically searches for a solution: a complete assignment of values to the decision variables that satisfies all of the constraints. Search is interleaved with inference known as *constraint propagation*, where deductions are made based on the constraints and the current set of assignments made via search. These deductions serve to narrow down the choices for the variables as yet unassigned by search, and therefore reduce the search required. Generally, there are many ways in which a given problem may be modelled, and the model chosen has a significant effect on the performance of the constraint solver in searching for solutions. Therefore, automated methods for improving constraint models are valuable (Leo & Tack, 2015; Nightingale et al., 2017).

In order to improve the performance of a constraint model, a common step is to reformulate the expression of a subset of the problem constraints, either to strengthen the inferences made during search by the constraint solver by increasing constraint propagation, or to maintain the level of propagation while reducing the cost of propagating the constraints. One such method is *tabulation*, the aggregation of a set of constraint expressions into a single table constraint (Mohr & Masini, 1988). Such a table constraint explicitly lists the allowed tuples of values for the decision variables involved. This allows us to exploit efficient table constraint propagators that enforce generalised arc consistency (GAC) (Bessiere, 2006), typically a stronger level of inference than is achieved for a logically equivalent collection of separate constraints. Successful examples of this approach where the reformulation has been performed by hand include Black Hole patience (Gent, Jefferson, Kelsey, Lynce, Miguel, Nightingale, Smith, & Tarim, 2007) and Steel Mill Slab Design (Gargani & Refalo, 2007).

Dekker et al. (2017) presented a method for the automation of tabulation (there called 'auto-tabling'). In their approach a predicate (a Boolean function) expressed in the MiniZinc language (Nethercote et al., 2007) may be annotated. Such an annotation requests that the predicate be converted into a table constraint. While called 'auto-tabling', note that the choice of predicate to be tabulated is done manually by the modeller. IBM ILOG CPLEX Optimization Studio (IBM, 2024) and ECLiPSe (Le Provost & Wallace, 1992) have similar facilities to generate table constraints, while other approaches target alternatives such as Multi-valued Decision Diagrams (MDDs) and regular constraints (de Uña et al., 2018; Löffler

et al., 2020). In all of these approaches, the crucial first step of identifying promising parts of a given model for tabulation is left to the human modeller.

In this work we present an entirely automatic tabulation method, TabID, situated in the constraint modelling tool Savile Row (Nightingale et al., 2017). The core function of Savile Row is to translate the constraint modelling language Essence Prime (Nightingale, 2024) into the input languages of solvers, including constraint programming (CP) and Boolean satisfiability (SAT) solvers. A set of heuristics is employed to identify in an Essence Prime model some candidate sets of constraints for tabulation, which are then tabulated automatically. Resource limits are applied to this process, ensuring that only the most useful candidate sets of constraints are tabulated. In order to demonstrate the effectiveness of our approach, we first examine the same four case studies that were used by Dekker et al. (2017) to showcase the utility of tabulation from explicit model annotations. We show that our automated approach can identify the same opportunities to improve the models by tabulation. We also study nine additional problem classes that show that TabID is effective on a wider range of problems. This paper builds on (and entirely includes) an earlier conference publication (Akgün et al., 2018).

1.1 Contributions

The primary contribution of TabID is the automation of a hitherto difficult manual task: the recognition of opportunities to tabulate parts of a constraint model in order to increase constraint propagation and therefore reduce search. In support of this primary contribution, we contribute the following:

- A set of heuristics to identify common tabulation opportunities, such as expressions that will propagate weakly.
- A caching system to avoid tabulating equivalent subproblems multiple times.
- A system of progress checks and work limits for the situation where a heuristic identifies a constraint that is too large to be tabulated.
- An empirical study across several solvers of the frequency with which our heuristics identify effective tabulation opportunities.

1.2 Motivating Examples

We consider two motivating examples from the literature: Black Hole and Knight's Tour.

Black Hole is a single-player card game (variously termed 'patience' or 'solitaire' depending on the variety of English spoken) where cards are played one by one into the 'black hole' from seventeen face-up fans of three cards. All cards can be seen at all times. A card may be played into the black hole if it is adjacent in rank to the previous card. Figure 1 shows part of a Black Hole game, illustrating the rank adjacency condition. Black Hole was modelled for a variety of solvers by Gent, Jefferson, Kelsey, Lynce, Miguel, Nightingale, Smith, and Tarim (2007) and a table constraint was used in the CP model.

A constraint model of Black Hole is presented in Figure 2. We use the simplest and most declarative model of Dekker et al. (2017) where two variables `a` and `b` (with cards coded as integers $\{0 \dots 51\}$) represent adjacent cards iff `|a-b| % 13 in {1,12}` (the adjacency

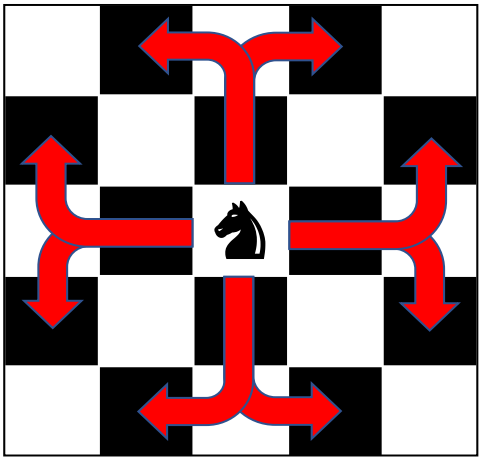

Figure 1: Left: illustration of part of a Black Hole patience setup, showing only 3 (of 17) fans and the 'black hole' (Ace of Spades) at the beginning of the game. The King of Hearts and the 2 of Clubs may be played onto the black hole at this point, because they are rank-adjacent to the card on top of the black hole. The 2 of Hearts cannot be played because it is covered. Right: illustration of the moves available for a knight in chess.

constraint). The model in the figure uses the available arithmetic and logical operators in the language to capture the rank adjacent condition in the game. The cards in each fan must be played in order. This constraint is captured by ordering variables in the `cardSeq` matrix which represents the time step that each card is played.

The adjacency constraint is a relatively complex expression containing sum, modulo, and absolute value operators. Typically this constraint will propagate weakly because some solvers implement modulo and absolute value poorly, and also (depending on the solver used) the sum operator may implement bound consistency.

Dekker et al. (2017) show that manually tabulating this constraint significantly improves the solving performance. This problem is therefore a good candidate for evaluating our work and indeed we will see that the tabulation opportunities present in this model can be identified and exploited automatically by TabID.

Our second motivating example is Knight's Tour, a classic puzzle studied by Euler (1759). More recently, Schwenk (1991) determined the set of board sizes (including rectangular boards) that have a knight's tour. Given a natural number n and a starting square on an $n \times n$ chess board, the task is to find a sequence of moves for the knight visiting each of the remaining squares of the board exactly once. The moves of a knight are illustrated in Figure 1. We use the Hamiltonian path version of Knight's Tour: the last square visited is not required to be a knight's move from the starting square.

A constraint model of the Knight's Tour is presented in Figure 3. We will refer to this as the *sequence* model to distinguish it from an alternative presented in Section 5.2.4. The

```
given initialStacks : matrix [int(0..50)] of int(1..51)
find blackHole : matrix [int(0..51)] of int(0..51)
find cardSeq : matrix [int(0..51)] of int(0..51)
such that
blackHole[0] = 0,  cardSeq[0] = 0,
allDiff(cardSeq),
allDiff(blackHole),
forAll step : int(1..51).
    (|blackHole[step-1] - blackHole[step]| % 13 in {1,12}),
forAll card : int(0..50). (card % 3 != 2) ->
    (cardSeq[initialStacks[card]] < cardSeq[initialStacks[card+1]]),
forAll step : int(0..51). forAll card : int(0..51).
    (blackHole[step] = card) <-> (cardSeq[card] = step)
```

Figure 2: A model of Black Hole in the constraint modelling language Essence Prime (Nightingale, 2024). The parameters to the model (the initial layout of the cards, named `initialStacks`) are introduced with the keyword `given`. The decision variables (the single-dimensional matrices `blackHole` and `cardSeq`) are introduced with the keyword `find`. Playing cards are numbered $0 \ldots 51$, with spades in the range $0 \ldots 12$, followed by hearts, clubs, and diamonds.

model in the figure is a formulation of the problem from the 2008 CSP solver competition[1], investigated experimentally by Rendl et al. (2009) using the available arithmetic and logical operators in the language to capture the legal moves of the knight. However, its performance in terms of constraint propagation is weak, as we will discuss below.

The location of the knight is encoded as a single integer $(nx + y)$ where (x, y) are the coordinates of the knight on the board (from 0). The *sequence* model simply has a one-dimensional matrix of variables (`tour`), with `tour[i]` representing the location of the knight at timestep i. The constraints enforce that initially the knight is at the given location, the moves are all different, and each adjacent pair `tour[i]` and `tour[i+1]` corresponds to a knight's move. As presented in the figure, the knight's move constraint contains two location variables and is naturally expressed using integer division and modulo to obtain the x and y coordinates. The constraint states that the absolute difference in the y coordinates is 1, and for x coordinates is 2, or vice versa.

The knight's move constraint is a relatively complex expression: a disjunction of conjunctions of reified arithmetic expressions, in which the absolute value of the difference between scaled adjacent `tour` variables is compared with a constant. Typically this constraint will propagate poorly for two different reasons. Firstly, some solvers implement modulo and absolute value poorly. Secondly, the constraint as a whole will propagate poorly, as most solvers will wait until the value of one side of the disjunct is known before propagating the other.

1. Archived as of 28 July 2009 at https://web.archive.org/web/20090728111748/http://www.cril.univ-artois.fr/~lecoutre/research/benchmarks/benchmarks.html

```
given n: int
given startCol,startRow : int(0..n-1)
find tour : matrix indexed by [int(0..n*n-1)] of int(0..n*n-1)
such that
allDiff(tour),
tour[0] = startCol + (startRow)*n,
forAll i : int(0..n*n-2) .
    ((|tour[i]%n-tour[i+1]%n| = 1) /\ (|tour[i]/n-tour[i+1]/n| = 2)) \/
    ((|tour[i]%n-tour[i+1]%n| = 2) /\ (|tour[i]/n-tour[i+1]/n| = 1))
```

Figure 3: The *sequence* model of the Knights Tour in Essence Prime. The parameters to the model (board size n and the starting square for the knight) are introduced with the keyword `given`. The decision variables (the single-dimensional matrix `tour`) are introduced with the keyword `find`.

There are tradeoffs involved in choosing subproblems to tabulate. Tabulating just the expression `(|tour[i]%n - tour[i+1]%n| = 1)` may well reduce search, but tabulating the entirety of the expression between two adjacent `tour` variables in a single step will produce a table of a similar size (as only two variables are involved), while reducing search much more. We could consider tabulating multiple adjacent moves as a single table — while this may reduce search further, the resulting tables would grow rapidly as a function of the number of variables involved. As we will see, the tabulation opportunities present in this naïve model can be identified and exploited automatically by the methods presented in this paper.

In both of these motivating examples, solver performance is improved by tens or hundreds of times (depending on the solver type) completely automatically by TabID, without the need to apply constraint modelling expertise.

1.3 Organisation

The rest of the paper is organised as follows. In Section 2 we review necessary background for the paper. In particular, we provide details of our modelling language, tools, and the abstract syntax tree data structure which our algorithms manipulate. We then discuss our approach to finding promising parts of the problem where tabulation could be applied, in Section 3. Here we cover each of our four heuristics and how they are adapted to different situations that occur in the abstract syntax tree. In Section 4 we describe the tabulation algorithm, its progress checks and work limits, and the caching system.

We then evaluate our system: in Section 5 in terms of how well the heuristics framework identifies promising subproblems; in Section 6 in terms of how overall performance is improved compared to the models without tabulation; in Section 7 with respect to how well our system performs on a large set of models where tabulation might not a priori be expected to improve performance; and finally in Section 8 we consider how tabulation scales as we allow the arity of generated tables to increase. We discuss related work in Section 9 and conclude in Section 10.

2. Background

This section gives the necessary background on constraint propagation, the constraint modelling language we employ in this paper, and the Savile Row tool in which we situate TabID.

2.1 Constraint Problems, Consistency, and Constraint Propagation

A constraint satisfaction problem (CSP) consists of a set of variables, each with a finite domain of values, and a set of constraints that specify the allowed values for given subsets of variables. A solution to a CSP is an assignment of values to the variables that satisfies all of the constraints. To find such solutions, constraint solvers typically combine a search through the space of partial assignments with constraint propagation, a form of deduction that helps to reduce the search required. A constrained optimisation problem (COP) is a CSP with an objective function that must be maximised or minimised.

Constraint propagation usually operates by establishing a *consistency* property on the constraints and variables. Generalised arc consistency is a common, powerful consistency property used in this paper, which we define in what follows. The *scope* of a constraint c, named scope(c), is the set of variables that c constrains. A *literal* is a decision variable-value pair (written $x \mapsto v$). A literal $x \mapsto v$ is *valid* iff v is in the domain of decision variable x. A *support* of constraint c is a set of valid literals containing exactly one literal for each variable in scope(c), such that c is satisfied by the assignment represented by these literals. A constraint c is Generalised Arc Consistent (GAC) (Bessiere, 2006) if and only if there exists a support for every valid literal of every variable in scope(c). GAC is established by identifying all literals $x \mapsto v$ for which no support exists and removing v from the domain of x.

There exist efficient constraint propagation algorithms to establish GAC on a table constraint, such as Compact Table (Demeulenaere et al., 2016; Ingmar & Schulte, 2018) where tuples are bit-packed, Tries (Gent, Jefferson, Miguel, & Nightingale, 2007), and Simple Tabular Reduction (Lecoutre, 2011). Table constraints are widely available in constraint solvers.

2.2 Essence Prime

The importance of modelling is widely recognised in constraint programming (CP) as well as the related fields of propositional satisfiability (SAT) and integer linear programming (ILP). In CP several constraint modelling languages have been developed, including OPL (Van Hentenryck, 1999), MiniZinc (Nethercote et al., 2007), and Essence Prime (Nightingale, 2024) in order to aid in the statement of constraint models and abstract away from the details of particular constraint solvers. Herein, we focus on Essence Prime, which is comparable with OPL and MiniZinc.

Essence Prime provides the facility to model parameterised *classes* of problems, where an individual problem instance is specified by giving values for the class parameters, for example the integers `n`, `startCol` and `startRow` in Figure 3. The language supports Boolean and integer finite-domain decision variables, both singly and collected into multidimensional matrices, such as the one-dimensional matrix of integers `tour` in Figure 3. Constraints over

these variables are expressed via arithmetic and logical expressions, as can also be seen in the figure. Quantification and comprehension enable the concise statement of such expressions. Essence Prime supports a number of *global* constraints (Rossi et al., 2006) that capture common patterns in constraint modelling, including the all-different constraint present in the figure (Régin, 1994), Global Cardinality Constraint (GCC) (Régin, 1996), and the table constraint that is the focus of this paper.

2.3 Savile Row, Tailoring, and the Abstract Syntax Tree

We investigate TabID as a component of the constraint model reformulation tool Savile Row (Nightingale et al., 2017). Savile Row is essentially a multi-pass term rewriting system. It represents a model internally using several abstract syntax trees (ASTs) representing the constraints, the objective function, the domain of each decision variable and parameter (or matrix thereof), and other statements in the model. The parser reads a model in the Essence Prime language, along with a parameter file giving a value to each of the problem class parameters. There are several backends targeting different solvers, including mature backends for CP and SAT solvers and a prototype ILP backend. The system has a number of different passes, some of which are always performed, others are required for specific backends, and others are optional reformulations intended to improve the performance of the solver. TabID is one optional reformulation. We refer to the entire process of transforming a model into input for a solver as *tailoring* the model. The early steps of tailoring prior to TabID are as follows:

- Problem class parameters and other constants (defined by `letting` statements) are substituted into the model;
- All quantifiers and matrix comprehensions are unrolled;
- Matrices of decision variables are replaced with individual decision variables;
- Multi-dimensional matrix indexing is replaced with single-dimensional indexing of a one-dimensional version of the matrix if required;
- Global constraints are identified by simple aggregation steps (Nightingale et al., 2017), e.g. collecting a clique of not-equal constraints into a single all-different constraint.

In addition, *simplifiers* are applied after each pass to perform partial evaluation and to maintain a normal form. In particular, negation is pushed towards the leaves of the AST (similar to *negation normal form*), double negation is removed, and some operators are rewritten when negated (for example, negated $=$ is rewritten to $\neq$). Variable domains are filtered using an external constraint solver prior to tabulation, and any assigned variables are deleted. Details of simplifiers, the normal form, and domain filtering are given by Nightingale et al. (2017).

As an example, consider the Knight's Tour problem and its *sequence* model (shown in Figure 3), with an 8×8 board. Part of the AST for this model is shown in Figure 4. The `tour` matrix has been replaced with individual variables `tour0`, `tour1`, etc. The `forAll` quantifier of Figure 3 has been unrolled to create $n^2 - 1$ knight's move constraints, each

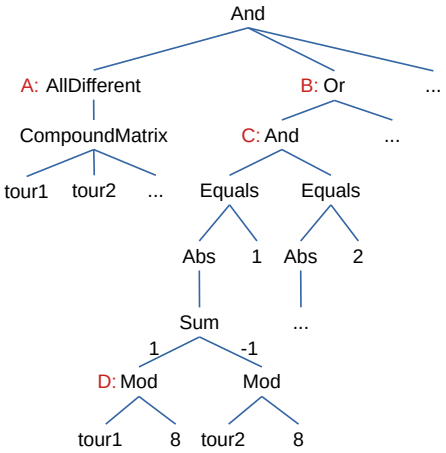

Figure 4: Part of the AST representing the constraints of the *sequence* model of Knight's Tour (Figure 3) where $n = 8$, showing the `allDiff` constraint and a part of one of the knight's move constraints linking two adjacent `tour` variables. Four AST nodes are labelled A to D.

containing two adjacent `tour` variables. A fragment of one of the knight's move constraints is shown in the example AST. Also, the variable `tour0` has been deleted because it was assigned, and its assigned value has been removed from the other variable domains.

For TabID we consider Boolean and integer expressions, for example those represented by nodes A, B, C, and D in Figure 4 (where A, B, and C are Boolean and D is integer). We distinguish between *top-level constraints* and other Boolean expressions. Top-level constraints are Boolean expressions directly beneath the top And node (e.g. nodes A and B in Figure 4). We refer to other Boolean expressions as *nested* (e.g. node C). Integer expressions are always nested, they cannot be directly contained in the top-level conjunction. In terms of the AST a table constraint is a Boolean expression with two arguments: a one-dimensional matrix of decision variables, and a two-dimensional matrix representing a list of satisfying tuples. Following the optional TabID pass, some further steps are required to tailor the model for the target solver. We use the default settings of Savile Row which can be summarised as follows, with further details given by Nightingale et al. (2017):

- Decomposition of constraint types that the target solver does not support;
- Common subexpression elimination (Active CSE) which factors out identical (or semantically equivalent) expressions replacing them with a new decision variable (for example, in the Knight's Tour *sequence* model in Figure 3 there are identical absolute value, division, and modulo expressions that would all be factored out by Active CSE);
- General flattening to extract nested expressions where the nesting is not allowed by the target solver (for example, in Figure 3 the absolute value operator contains a sum which (if not already extracted by CSE) would be extracted and replaced with a new decision variable).

In addition the SAT backend has a final encoding step where each integer variable is encoded using *order*, *direct*, or both as required for the constraints containing the variable. The remaining constraints (such as linear, element, and table constraints) are encoded to CNF. Others such as all-different and GCC are decomposed before reaching the SAT backend. Further details of the SAT encoding are given in the Savile Row manual (Nightingale, 2024).

3. TabID: Identifying Promising Subproblems for Tabulation

We have designed four heuristics for TabID to identify cases where expert modellers might experiment with tabulation to improve the performance of a model. The heuristics operate on the AST, identifying AST nodes that are candidates for tabulation. The heuristics are applied somewhat differently for top-level constraints, nested Boolean expressions, and integer expressions, and we describe these cases in Section 3.2 below. First we describe the types of changes tabulation can make to the AST.

Note that the heuristics to identify promising subproblems do not take account of the size of the resulting table, nor the work required to generate it. The heuristics can and frequently do identify candidates that would be impractical to tabulate. Progress checks (described in Section 4) allow tabulation to be abandoned early and play an important role in avoiding overhead.

Figure 5 provides an overview of the system, showing how the set of heuristics are combined with other components described in Section 4.

3.1 AST Modifications

The AST that TabID acts upon is described in Section 2.3 with an example in Figure 4. Tabulation modifies the AST in one of two ways depending on the type (integer or Boolean) of the node to be replaced. When the AST node is of type Boolean, the subtree rooted at the node is directly replaced with one table constraint. For example, if node B of Figure 4 were to be replaced, the resulting tree would be Figure 6 (upper) in which B has been replaced with node E. Node B is a top-level constraint with variables `tour1` and `tour2` in scope, and its replacement is a top-level constraint with the same scope. If the node to be replaced is a nested Boolean expression, such as node C in Figure 4, then the replacement table constraint will also be nested. Some solver types do not allow nested table constraints and in this case it would be extracted and replaced with a new Boolean variable, however this occurs in another pass after TabID is complete.

When the node to be replaced is of type integer, the node is replaced with a new variable and a top-level constraint is created to link the new variable to the variables in scope of the tabulated expression. For example, node D of Figure 4 is replaced with new auxiliary variable `aux1` in Figure 6 (lower), and a new table constraint G is added to the top-level conjunction with `aux1` and `tour1` in scope. The new table constraint is generated from an equality between the new variable and the expression to be tabulated, which in this example is `aux1 = tour1%8`. The domain of the auxiliary variable is generated from the expression using the extended domain filtering method (Nightingale et al., 2017) (the default method for all variables introduced by Savile Row).

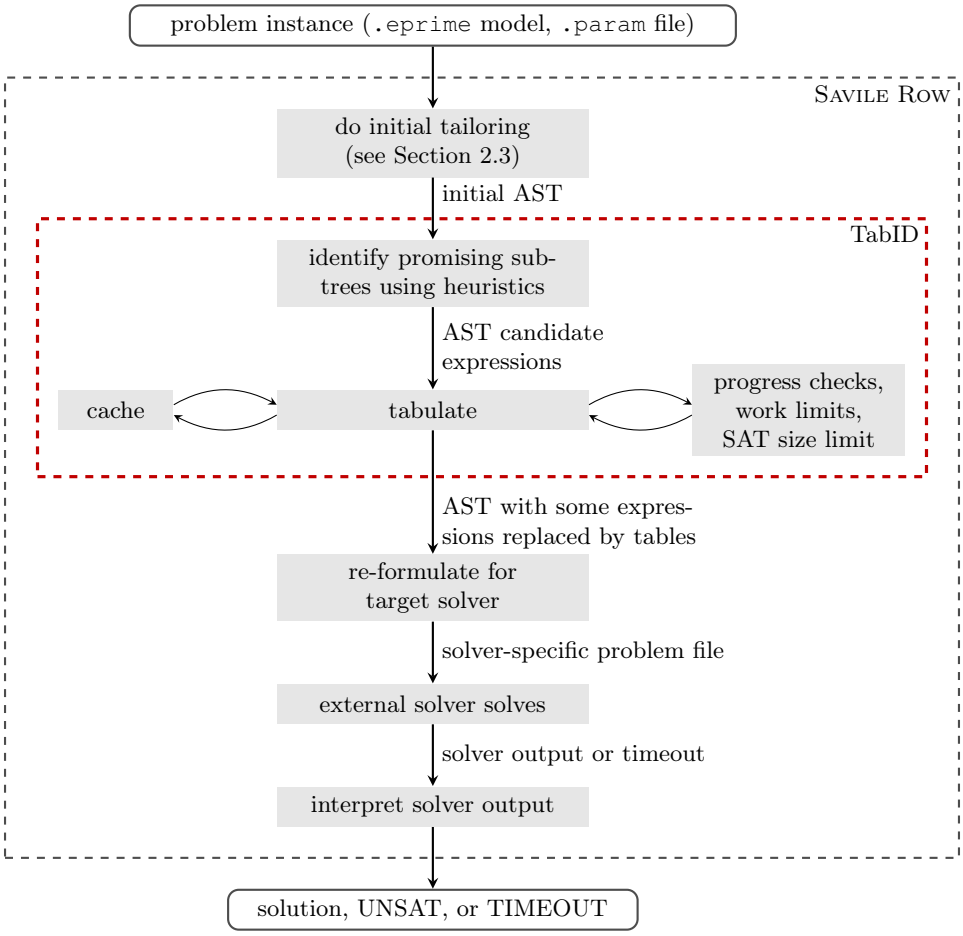

Figure 5: An overview of the TabID process as part of the constraint solving pipeline in Savile Row. The white boxes with solid borders represent data; the grey boxes represent processes.

Finally, both examples in Figure 6 have an identifier (`aux2`) in place of the table of satisfying tuples; this is because matrices of constants are cached to avoid duplication.

3.2 Identifying Promising Constraints

First we define the four heuristics as they apply to top-level constraints (i.e. constraints in the top-level conjunction). The heuristics are as follows, in the order that they are applied:

Identical Scopes identifies sets of two or more constraints whose scopes contain the same set of decision variables.

Duplicate Variables identifies constraints with at least one variable occurring more than once in the constraint.

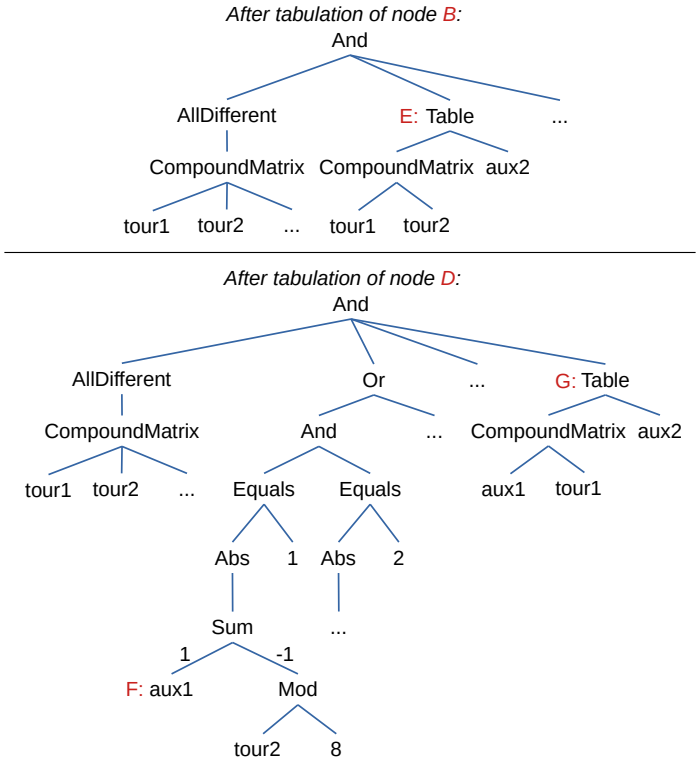

Figure 6: Two examples of tabulation applied to the AST shown in Figure 4. Upper: Node B of Figure 4 is replaced with a table constraint labelled E. Both top-level constraints and nested Boolean expressions are directly replaced when they are tabulated. Lower: Node D of Figure 4 (an integer expression) is replaced with a new auxiliary variable (F), and a table constraint G is attached to the top-level And node. The scope of the table constraint is the scope of the tabulated expression plus the new variable (`aux1` in this case).

Large AST identifies a constraint where the number of nodes in the AST is greater than 5 times the number of distinct decision variables in scope.

Weak Propagation identifies a constraint c_1 that is likely to propagate weakly (i.e. less than GAC), such that there is another constraint c_2 that propagates strongly, with at least one variable in the scope of both c_1 and c_2. The method of estimating whether a constraint will propagate strongly is described in Section 3.3. Note that only c_1 is identified for tabulation.

Subproblems containing more than 20 distinct variables are not candidates for tabulation because they would almost certainly be impractical to tabulate. Each of the four heuristics is based on a simple rationale regarding either propagation strength or propagation speed

of the constraint(s). The constants used in these heuristics were chosen by hand based on preliminary experiments.

First we consider the Identical Scopes heuristic. It is well known that multiple constraints on the same scope may not propagate strongly together, even if each constraint individually does propagate strongly. The Identical Scopes heuristic is intended to collect such sets of constraints into a single table constraint that may propagate more strongly and also may be faster to propagate. An extreme example would be two contradictory constraints on the same scope (e.g. $x < y$ and $y < x$) which would be replaced by a trivially false table constraint. The idea behind Identical Scopes is not new: constraint networks are defined as *normalized* iff no pair of constraints has the same scope (Bessiere, 2006).

The Duplicate Variables heuristic identifies constraints that are likely to propagate weakly even when the target solver has a strong propagator for the constraint type. In most cases a GAC propagator will enforce GAC only when there are no duplicate variables. In some cases it is intractable to enforce GAC with duplicate variables. GAC on the Global Cardinality Constraint (GCC) is known to be NP-hard with duplicate variables (Bessière et al., 2007), therefore Régin's polynomial-time GAC propagator (Régin, 1996) achieves GAC only when there are no duplicate variables. The knight's move constraint of the Knight's Tour *sequence* model (Section 1.2) triggers the Duplicate Variables heuristic: each decision variable in scope is mentioned four times. In the evaluation we show that tabulating this constraint improves solver performance very substantially.

The Large AST heuristic identifies constraints that are not compactly represented in the AST. The knight's move constraint of the Knight's Tour *sequence* model also triggers the Large AST heuristic. It has two decision variables in scope, and the AST representation (part of which is illustrated in Figure 4) has 43 nodes. In this case, tabulating the constraint avoids the need to create auxiliary variables during CSE and flattening (Section 2.3), and also strengthens propagation of the constraint. In general the rationale behind the heuristic is that a table propagator may be more efficient while achieving the same or stronger propagation.

Finally we consider the Weak Propagation heuristic. It is intended to catch cases where the weak propagation of one constraint is hindering strong propagation of another. The knight's move constraint of the Knight's Tour *sequence* model also triggers the Weak Propagation heuristic: the representation of the knight's move constraint for a CP solver is not expected to enforce GAC (because it contains arithmetic such as sum and modulo), and it overlaps with an `allDiff` constraint that is expected to enforce GAC. To implement the Weak Propagation heuristic we need to estimate which constraint expressions are expected to propagate strongly. Section 3.3 describes how we do this.

Each of the four heuristics proves to be valuable: each one is triggered on at least one of the problems that we study in the evaluation. The heuristics are applied in the order that they are listed in this section.

In some cases it can be useful to tabulate an expression within a constraint without tabulating the entire top-level constraint. One example is where the top-level constraint is out of reach of tabulation (i.e. tabulation would produce an impractically large table or take too long). We have adapted the heuristics in two ways (to identify nested Boolean and integer expressions respectively) as described in Appendix A. The goals remain the same: to strengthen propagation, or to replace an unwieldy expression to improve the efficiency

of propagation. The adapted heuristics will be distinguished from top-level heuristics by adding (Nested) or (Integer) to the name (e.g. Large AST (Nested), Large AST (Integer)).

3.3 GAC Estimate

Given an expression e, the *GAC estimate* is a heuristic to estimate whether the representation of e for a conventional CP solver will propagate strongly. In cases where CP solvers vary, we use Minion (Gent et al., 2006) as a reference. When e is a top-level constraint, the GAC estimate is simply an estimate of whether the solver will enforce GAC on e. Nested Boolean expressions are treated identically to top-level constraints. When e is an integer expression, the GAC estimate is applied to the constraint $a = e$ (where a is a new auxiliary variable), with the rationale that e will in many cases be extracted by flattening and replaced with a (see Section 2.3), creating the constraint $a = e$.

The definition of the GAC estimate is recursive on the AST representing the expression e. To avoid undue complexity the same GAC estimate is used regardless of target solver. At the leaves of the AST, constants and references to variables are defined to be strong. Each type of internal AST node (such as sum or product) has its own rules to define when it is weak or strong. For example, the `allDiff` constraint often has a GAC propagator so it is defined to be strong iff all its children are strong. Therefore the constraint $\texttt{allDiff}(x_1, x_2, x_3)$ is strong. A sum is defined to be strong when all its children are strong, and for each child c the interval of possible values $[a, b]$ of c satisfies $b - a \leq 1$. The reason is that sums are usually implemented with bound consistency propagators that are in general weaker than GAC but equivalent to GAC in this specific case. The constraint $\texttt{allDiff}(x_1 - x_2, x_3 - x_4, x_5 - x_6)$ on integer variables $x_1, \ldots, x_6 \in \{1 \ldots 3\}$ is therefore defined to be weak. Its representation for a CP solver is unlikely to enforce GAC on the variables $x_1, \ldots, x_6$ even when `allDiff` has a GAC propagator.

4. TabID: Tabulation, Caching, Progress Checks, and Limits

Having identified promising candidates, the next step is to perform tabulation efficiently and with appropriate work limits to avoid impractically large tables and long tabulation times. The method to generate tables is a straightforward depth-first search and is described in Appendix B. For efficiency we have implemented a cache to avoid repeated generation of identical tables for expressions that are semantically equivalent (up to renaming of decision variables). The cache relies on a normal form for expressions, and both the cache and the normal form are described in Section 4.1. Finally, in Sections 4.2 and 4.3 we describe work limits and progress checks applied to tabulation. Figure 5 provides an overview showing how the various components of TabID fit together.

4.1 Caching

We use caches to avoid generating identical tables for constraints that are semantically equivalent (up to renaming of decision variables). To store or retrieve a table for an expression e, we first place e into a normal form as follows. First the expression is simplified and normalised as described in Section 2.3. Then all associative and commutative k-ary expressions (such as sums) and commutative binary operators (e.g. $=$) within e are sorted.

Alphabetical order is used because it will group together references to the same matrix (all else being equal) and place references to different matrices in a consistent order regardless of the indices. The normal form is not only used for accessing the caches. It is also applied before generating a table for an expression e to ensure that tables are generated with columns in the correct order for storing in the cache.

After applying the normal form, the expression is traversed in depth-first, left-first order to collect a sequence of decision variables (without duplication), and the variables in the sequence are then renamed to a canonical sequence of names to create e'. Thus the actual variable names in e do not affect e', only their relative positions. e' and the variable domains together are used as a key to store and retrieve tables in the caches.

As an example, the e' expression for the Knight's Tour move constraint from Figure 3 with $n = 8$ is shown below.

```
((1=|((x0%8) - (x1%8))|) /\ (2=|((x0/8) - (x1/8))|)) \/
((1=|((x0/8) - (x1/8))|) /\ (2=|((x0%8) - (x1%8))|))
```

There are two in-memory caches: the first contains tables, and the second stores cases where tabulation failed because it failed a progress check or reached the node limit. The in-memory caches do not persist after one tailoring process on one problem instance. We have also implemented a persistent filesystem cache but this is disabled for the evaluation because it would cause timings to change depending on the order of processes.

4.2 Progress Checks and Work Limits

In some cases a heuristic will identify a constraint that is too large to be tabulated. Simple work limits (such as those applied in our earlier work (Akgün et al., 2018) where we limited the depth-first search to generate at most 10,000 tuples, and to fail and backtrack at most 100,000 times) are not ideal because time can be wasted attempting to tabulate constraints that are far beyond reach. As an alternative we propose progress checks where the progress of the algorithm through the assignment space is compared to the total size of the space, and if the algorithm seems to be making insufficient progress then the search is stopped early. The depth-first search algorithm progresses through the assignment space in lexicographic order, making it straightforward to calculate the number of total assignments explored so far from the current (partial) assignment.

Suppose we have a constraint on variables $x_1, \ldots, x_r$ with domains $D_1, \ldots, D_r$, and we reached a partial assignment setting variables $\langle x_1, \ldots x_k \rangle$ to values $\langle v_1, \ldots, v_k \rangle$ where $k \leq r$. The partial assignment is completed by filling in the minimum value of the domain for each unassigned variable:

$$\tau = \langle v_1, \ldots, v_k, \min(D_{k+1}), \ldots, \min(D_r) \rangle$$

The formulas below assume that each domain is a single contiguous range of integers $D_i = \{0 \ldots \max(D_i)\}$. The implementation has an additional step to map domain values into a single contiguous range. The last assignment number A and current position C are:

$$A = \sum_{i=1}^{r} \left[\max(D_i) \times \prod_{j=i+1}^{r} |D_j| \right]$$

$$C = \sum_{i=1}^{r} \left[\tau_i \times \prod_{j=i+1}^{r} |D_j| \right]$$

The progress check uses the current node count (*nodeCount*) as well as the parameter *nodeLimit*. It compares the progress made so far to the proportion of the search node limit that has been used so far, effectively using a linear extrapolation to estimate whether the search will complete within the node limit. The search is abandoned if:

$$\frac{C}{A} < \frac{nodeCount}{nodeLimit}$$

The term $\frac{C}{A}$ represents the progress made so far through the search space, and $\frac{nodeCount}{nodeLimit}$ is the proportion of the search node budget used so far. The progress checks are pessimistic: if a search is expected to slightly exceed the node limit it is abandoned, even though the search algorithm is unlikely to progress through the assignment space at a constant rate. A more optimistic strategy could be obtained by adjusting the formula above (e.g. by multiplying the left-hand side by an additional parameter that is > 1), or instead a stochastic estimation procedure could be used (Knuth, 1975). Progress checks are carried out after 1000 and 10,000 nodes, and then after every 10,000 nodes. In addition to the progress checks, the search is terminated if it reaches *nodeLimit* nodes.

One further limit is applied when tabulating nested Boolean expressions. For solvers or encoding backends that do not support reified or nested table constraints, we limit A to be no more than *nodeLimit*. The reason is that a nested Boolean expression would be replaced with a nested table constraint by tabulation (as in Section 3.1) which would then be replaced with a new Boolean variable and a top-level reified table constraint. If the solver does not support reified table constraints, it is converted to a conventional table constraint with exactly A satisfying tuples. This final step is problematic if A is large, therefore we limit A.

4.3 SAT Size Limit

Encoding to SAT and applying a state-of-the-art SAT solver can be very effective for some problem classes. Savile Row has a SAT backend (briefly described in Section 2.3) that includes encodings or decompositions for all constraints in Essence Prime, and with a choice of two encodings for table constraints. Applying TabID prior to SAT encoding can improve solver performance on amenable problem classes, as we will show in Section 6. However, in some cases applying TabID can dramatically increase the size of the SAT formula and as a consequence reduce solver performance.

The *SAT size limit* is an optional limit in TabID that prevents large increases in the SAT encoding size (measured by the number of clauses). Given a candidate Boolean expression e, the SAT size limit first estimates the encoding size of e, then limits the encoding size of the generated table constraint to be no more than 2 times larger than that of e. A small increase is allowed because a table constraint encoding may have better properties than other encodings (for example, unit propagation simulating GAC).

To estimate the encoding size of e, a CSP instance C is made containing only e and decision variables in the scope of e. C is tailored for encoding to SAT (as described in

Section 2.3), creating C'. C' is encoded to SAT[2] and the encoding size is recorded (excluding the encoding of variables in C). Finally, the encoding size of the generated table constraint is limited by bounding the number of tuples generated (if the encoding size is a function of the number of tuples), or simply by encoding the generated table constraint and rejecting it if the encoding is too large.

5. Feasibility Evaluation of TabID

We now come to the first part of our evaluation, to show that TabID can identify and tabulate promising subproblems of a wide range of models. Here we are evaluating TabID's set of heuristics together with the progress checks and work limits, to show that they are of use in successfully tabulating automatically. For this feasibility evaluation, we do not consider the effectiveness of tabulations: in Section 6, we will show that in many cases applying TabID strongly improves the total time to tailor and solve an instance (including time taken to identify candidates and perform tabulation).

This feasibility evaluation is divided into two. We first examine, in Section 5.1, what we call 'Baseline Problems' where we have identified examples of tabulation in the literature. Tabulation (whether performed manually or with tool support) is a well-established technique, and for some problem classes and models there are examples in the literature of subproblems that can profitably be tabulated. For these models we compare with the literature, in addition to comparing the original model to the version after applying TabID. We then examine, in Section 5.2, seven new case studies where tabulation has not been previously identified in the literature but where TabID was able to tabulate automatically. In Section 5.3 we summarise the results of our study and how each heuristic performs.

Following some preliminary experiments, we have set the parameter *nodeLimit* to 100,000 for our feasibility evaluations. Also, we chose not to apply the SAT size limit. The effect of the node limit can be seen (for example) with the Killer Sudoku problem where some constraints of arity 5 (on variables with domain size 16) fail a progress check but others are successfully tabulated. The base models (with parameter files) are publicly available online (Akgün et al., 2024) alongside a version of Savile Row that implements TabID.

5.1 Baseline Problems

The set of baseline problems consists of four problems presented by Dekker et al. (2017), and two others: Sports Scheduling Completion and Maximum Density Still Life. In the first four cases we show that TabID can automatically identify and tabulate the same subproblems that Dekker et al. identified by hand and found to be useful, as well as finding some other opportunities for tabulation. Similarly, for Sports Scheduling Completion, TabID identifies and tabulates the same constraint as in the literature. For Maximum Density Still Life, there is no exact equivalent in the literature but TabID tabulates the "easy formulation" constraint from the model of Bosch and Trick (2004, Section 1.1).

2. With some efficiency measures, e.g. generating the entire encoding of C' is not always necessary.

5.1.1 Black Hole

Black Hole was introduced as a variant of patience in Section 1.2, along with a constraint model of the problem. To recap, the model has two matrices of variables: `blackHole`, the sequence of cards played into the black hole; and its inverse `cardSequence` (the index of each card in `blackHole`). We post the adjacency constraint on each pair of adjacent variables in `blackHole`. Less-than constraints on `cardSequence` ensure that the cards in a fan are not played out of order. Both matrices have an `allDiff` constraint, and they are linked by channelling constraints. An instance of Black Hole is a permutation of the 52 cards. We experimented with the 102 randomly-generated instances from CSPLib (Nightingale, 2018).

All adjacency constraints trigger the Weak Propagation heuristic because they overlap with the `allDiff` on the `blackHole` matrix. In addition the Identical Scopes (Nested) heuristic is triggered by a small number of equalities (no more than 9) within the channelling constraints. No other constraint triggers any heuristic, so our set of candidates is very similar to those identified by hand (Gent, Jefferson, Kelsey, Lynce, Miguel, Nightingale, Smith, & Tarim, 2007). All candidates are successfully tabulated.

5.1.2 Block Party Metacube Problem

The Block Party Metacube Problem is a puzzle in which eight small cubes are arranged into a larger *metacube*, such that the visible faces on each of the six sides of the metacube form a 'party'. Each small cube has a symbol at each corner of each of its faces (24 symbols per cube in total), and each symbol has three attributes, with each attribute in turn taking one of four values. To form a valid party (the *party constraint*), the four small cubes forming a visible face of the large cube must be arranged so that the four symbols in the middle of the visible face are either all different, or all the same, for each of the three attributes.

The model we use is closely based on Dekker et al. (2017) and we use the same set of instances. The model has two matrices of decision variables, `cubeAt` and `symAt`. Matrix `cubeAt` encodes a permutation of the 8 cubes (numbered 1 to 8), representing the relative locations of the cubes in the metacube. Further, `symAt` represents for each of the 4 symbol positions located in the middle of each of the 6 faces of the metacube (24 symbol positions) which symbol is visible in that position. A hand-computed matrix `pp` encodes how the 24 positions in which symbols are placed on a cube occur together at corners of a cube. A set of channelling constraints link each `cubeAt` variable to three `symAt` variables.

There are some notable differences between our model and Dekker et al. They introduce a variable for each attribute at each of the 24 symbol positions, whereas in our model the expression for the attribute (one of the following: `symAt[i]/16`, `symAt[i]%4`, or `(symAt[i]%16)/4` with constant `i`) is used wherever it is needed. Where necessary Savile Row will introduce a variable for an attribute expression during tailoring. Also, Dekker et al. introduced a local *rotation* variable for each small cube in their non-tabulated model. The rotation variables are not present in their tabulated model. Essence Prime does not have local decision variables so we used an existential quantifier in place of each rotation variable.

Dekker et al. tabulated the 8 channelling constraints linking cubes and symbols. The Duplicate Variables, Large AST, and Weak Propagation heuristics all identify the same set of channelling constraints (with arity 4) and they are all successfully tabulated. The Weak Propagation (Nested), Large AST (Nested), or Weak Propagation (Integer) heuristics

identify the attribute expressions (e.g. `symAt[i]/16`) or an equality of two attribute expressions, contained in the party constraints. All such expressions are successfully tabulated (and all have arity 2). Larger nested sub-expressions of the party constraints are identified by Identical Scopes (Nested), but almost all fail the progress check at 1,000 nodes. At most 4 are tabulated in any given instance.

5.1.3 Handball Tournament Scheduling

The Handball Tournament Scheduling problem is to schedule matches of a tournament, while respecting the rules governing the tournament, and minimising a cost function related to the availability of venues. We use the model and the same 20 instances of Dekker et al. (2017), which is simplified from the full model (Larson et al., 2014) by omitting some constraints. The problem has 14 teams (in two divisions of 7), and briefly it requires constructing one round-robin tournament for each division (in periods 1-7), followed by a round-robin tournament for all teams (periods 8-20) then its mirror image (periods 21-33). Constraints are either structural (such as balancing home and away games) or seasonal (such as respecting venue unavailability, given as a parameter). The main sets of variables represent: the *home-away pattern* (`HAP`) indexed by row and period, with values *home*, *away*, or *bye* (i.e. does not play); the `break` period for each row of `HAP`, when the alternating home-away pattern is broken by the team playing at home twice or away twice in sequence; the `contestant` (opponent) for each entry in `HAP`; the team assigned to each row of `HAP`, named `teamof`; and the cost of each row (`rowcost`), which contributes to the objective.

Dekker et al. reformulated the original model in three steps. First, they generated a `table` constraint to channel between a row of `HAP` and its corresponding `break` variable (and reported no significant speed-up from this). Second, they reformulated the row cost constraint for each row of `HAP` to be in terms of `break`, `teamof`, and `rowcost` (replacing `HAP` variables with `break`). Thirdly, they applied tabulation to the row cost constraints (generating arity 3 `table` constraints).

TabID is not capable of reformulating a constraint to change its scope, so we have performed the first two steps manually and we evaluate whether TabID can do the final step automatically. The model presented to TabID closely follows Dekker et al., except that local decision variables declared within the row cost constraint are replaced with an existential quantifier (because Essence Prime does not have local decision variables). The quantifier unrolls to a disjunction where each disjunct corresponds to one value of the `break` variable.

When TabID is disabled, using a disjunctive version of the row cost constraint would be artificially bad so in this case we use a different formulation that is stated on a row of the `HAP` matrix, `teamof`, and `rowcost`. The formulation of the objective is similar to the baseline (non-tabulated) model of Dekker et al. The rest of the model is identical to the version presented to TabID.

The Duplicate Variables, Large AST, and Weak Propagation heuristics identify the row cost constraints and all are tabulated. Also, unary constraints on `contestant` are identified by Weak Propagation, while Identical Scopes (Nested) identified equalities containing a single `contestant` variable. Weak Propagation (Integer) triggered for shift expressions used in matrix indexing and linear expressions involving `contestant` and `HAP`. All candidates mentioned here were tabulated.

5.1.4 JP Encoding Problem

The JP Encoding problem was introduced in the MiniZinc Challenge 2014. In brief, the problem is to find the most likely encoding of each byte of a stream of Japanese text where multiple encodings may be mixed. The encodings considered are ASCII, EUC-JP, SJIS, UTF-8, or unknown (this choice incurs a large penalty). Once again our model closely follows that of Dekker et al. (2017). We use all 10 instances in the MiniZinc benchmark repository. The instances are from 100 to 1900 bytes in length. Each byte has four variables: the `encoding`, a `byte_status` variable that combines the encoding with the byte's position within a multibyte character, a `char_start` variable indicating whether the byte begins a new multibyte character, and the `score` which contributes to the objective.

Dekker et al. tabulate three subproblems. The first connects two adjacent `status` variables, and the Identical Scopes heuristic triggers on this. The second links `status`, `encoding`, and `char_start`, and we found that the Identical Scopes heuristic separately links `status` to `encoding`, and `status` to `char_start`. The `encoding` and `char_start` variables are both functionally defined by `status` so no propagation is lost with two binary table constraints compared to one ternary table. Thirdly Dekker et al. tabulate the constraint linking the score to the encoding. The Duplicate Variables and Large AST heuristics trigger on this. In summary, the set of subproblems identified by TabID is not identical to Dekker et al., but is exactly equivalent in propagation strength (assuming GAC is enforced on table constraints). All candidates are successfully tabulated (creating binary tables).

5.1.5 Maximum Density Still Life

The Maximum Density Still Life problem, CSPLib problem 32 (Smith, 2015), is to maximise the number of live cells in an $n \times n$ grid in such a way that applying the rules of John Conway's Game of Life would leave the grid unchanged. Cells outside the grid are assumed to be dead. The rules state that a live cell survives if it has two or three live neighbours, but dies otherwise. A dead cell will only become alive if it has exactly three live neighbours. Maximum Density Still Life has been solved for extremely large grids (Chu & Stuckey, 2012) using a sophisticated CP model and other techniques. For sufficiently large grids, a provably optimal solution is constructed from a finite set of tiles, thus providing a solution for any n.

Our focus is not on solving Still Life for large n but in evaluating tabulation, and so we have simply modelled the problem using an $n \times n$ matrix `g` of Boolean variables (where *true* means live), surrounded by a border of width 2 of cells that are *false*. The rules are implemented with two implication constraints, one setting the cell to *true* if exactly three of the eight neighbours are alive; the other setting the cell to *false* if fewer than 2 or more than 3 neighbours are alive; leaving cells with exactly two live neighbours unconstrained, as they would be unchanged in the next step of the game. The constraints are applied to each cell in the $n \times n$ matrix and to the inner layer of border cells. We used the instances where $n \in \{6..15\}$. Letting `sum(neighbours)` abbreviate the sum of the 8 neighbours of `g[i,j]`, the constraints for one cell `g[i,j]` are written as follows:

```
sum(neighbours) = 3 -> g[i,j],
(sum(neighbours) > 3 \/ sum(neighbours) < 2) -> !g[i,j]
```

For each cell within the $n \times n$ matrix, the Identical Scopes heuristic identifies the two implication constraints (together) as a candidate for tabulation, and they are successfully

tabulated (with arity 9 for cells not touching the edges). For the cells in the inner border (and away from the corners), each cell has 3 unassigned neighbours and Savile Row simplifies the two constraints to a single sum $\neq 3$ constraint which is not a candidate for tabulation. When tabulation is disabled, the three occurrences of `sum(neighbours)` are removed by common subexpression elimination (Nightingale et al., 2017) and replaced with a new variable.

Our model is logically equivalent to the "easy formulation" constraint model of Bosch and Trick (2004, Section 1.1). Tabulation seemed promising as an improvement to this model without needing to reformulate the constraints to take into account additional mathematical insights (Chu & Stuckey, 2012; Smith, 2002). The constraints may seem to be natural candidates for tabulation, but in fact (as we will see in Section 6.3) tabulation provides no benefit, slowing down solving in most cases. We therefore include Maximum Density Still Life as an example of tabulation where the result negatively affects performance.

5.1.6 Sports Scheduling Completion

The Sports Scheduling Completion problem is to construct a schedule of $n(n-1)/2$ games among n teams (n must be even), where each team plays every other team once. The schedule is divided into $n-1$ weeks, in each week there are $n/2$ periods, and one game is played in each period of each week. No distinction is made between home and away games. Each team plays at most twice in a period, and each team plays exactly once each week (Van Hentenryck et al., 1999).

The schedule is represented explicitly with a matrix `schedule[w,p,i]`, indexed by the week `w`, period `p`, and `i` which is 1 or 2 for the two teams in the game. Some symmetry is broken by ordering the two teams in each game in the `schedule` matrix. `allDiff` constraints are used to ensure each team plays once each week, and a set of `gcc` constraints (one per period) ensure each team plays at most twice in each period (with an auxiliary variable `card` indicating the number of games played by each team in each period). A second matrix `game[w,p]` (also indexed by week and period) represents each game with a single integer. The two representations are channelled with the following constraints:

```
forAll w : WEEKS . forAll p : PERIODS .
    game[w,p] = n*(schedule[w,p,1]-1) + schedule[w,p,2]
```

Finally an `allDiff` constraint is posted on the `game` matrix to ensure every team plays every other team exactly once.

In Sports Scheduling Completion, we start with a partial schedule where some of the slots in `schedule` are assigned. 10 instances were generated with $n = 12$ and 10 slots assigned a team at random with uniform distribution. Trivially unsatisfiable instances were filtered out.

The Weak Propagation heuristic identifies each of the channelling constraints (on 3 variables), and all are tabulated. Van Hentenryck et al. (1999) manually tabulated the same constraint in their OPL model of Sports Scheduling. Weak Propagation also identifies linear equality constraints over the `card` variables for each period (introduced automatically as implied constraints from `gcc`), and these are also tabulated.

5.2 New Case Studies

In this section we present seven case studies that were not featured in the literature to our knowledge. In each case we briefly describe the model and discuss the expressions that trigger the TabID heuristics.

5.2.1 Accordion Patience

'Accordion' (BVS Development Corporation, 2017) is a single-player (patience or solitaire) card game. The game starts with the chosen cards in a sequence, each element of which we consider as a 'pile' of one card. Each move we make consists of moving a pile on top of either the pile immediately to the left, or three to the left (i.e. with two piles between the source and destination) such that the top cards in the source and destination piles match by either rank (value of the card, e.g. both 7) or suit (clubs, hearts, diamonds, or spades). The result of each move is to reduce the number of piles by 1 and change the top card of the destination pile. The empty space left at the position of the source pile is deleted. The goal is to keep making moves until just one pile remains. We consider the 'open' variant where the positions of all cards are known before play starts, in a variant studied by Ross and Knuth (1989) where we play with a randomly chosen subset of $n \leq 52$ cards. A problem instance consists of the subset of cards in play, and their initial positions. We have generated 5 random instances for each value of $n \in \{11, 12, 13, 14, 15, 16\}$.

We model accordion with a matrix called `piles` of $(n-1) \times n$ variables with domain $\{0 \ldots 51\}$, the element `piles[i,j]` representing the top card of the stack in position j in the sequence after move i. Only the top card in each stack is represented; others are not relevant. We have $2(n-1)$ decision variables for the $n-1$ moves, expressing which pile is moved to which other pile (named `from` and `to`). We also have two variables per move representing the top card of the stack that is moved (`fromcard`), and the top card of the stack it is moved onto (`tocard`). Frame axioms ensure that unmoved cards are copied from one timestep to the next, and that the unused slots at each timestep fill up with zeroes. A set of constraints link `to` with `tocard`, and `from` with `fromcard` by indexing the `piles` matrix. Finally the move is implemented by indexing into `piles` with `from` and `to`.

There are two key constraints on the moves. The first (Move1) is that the move is of either one or three places, written as follows:

```
forAll t : int(1..n-1) . to[t] = from[t] - 1 \/ to[t] = from[t] - 3
```

The other key constraint (Move2) ensures that the top cards of the two piles are of the same rank or suit. This is expressed by stating that the relevant two cards either have the same value modulo 13 or integer-divided by 13, as follows:

```
forAll t : int(1..n-1) .
  fromcard[t]%13 = tocard[t]%13 \/ fromcard[t]/13 = tocard[t]/13
```

All Move1 and Move2 constraints are identified by the Duplicate Variables, Large AST, and Weak Propagation heuristics and are tabulated, creating binary tables. The Weak Propagation (Integer) heuristic identifies expressions of the form `x-c` where `c` is a constant. These expressions come from indexing into the `piles` matrix with `from` and `to`, and they are tabulated. Identical Scopes identifies a small number of frame axiom constraints which

are tabulated. Preliminary experiments showed that almost all benefit came from tabulating Move1 and Move2 constraints.

5.2.2 Coprime Sets

Erdős and Sárközy (1993) studied a range of problems involving coprime sets. A pair of numbers a and b are coprime if there is no integer $n > 1$ which is a factor of both a and b. The Coprime Sets problem of size k is to find the smallest m such that there is a subset of k distinct numbers from $\{m/2 \dots m\}$ that are pairwise coprime. In our model, the set is represented as a sequence `V` of integer variables. Each pair of variables `V[i]` and `V[j]` has a set of coprime constraints, one for each potential factor in $\{2 \dots m\}$:

```
forAll d : int(2..m) . ((V[i]%d != 0) \/ (V[j]%d != 0))
```

Adjacent variables in the sequence are ordered with less-than constraints to break symmetry. Also, the lower bound of $m/2$ is enforced with constraints `V[i]>=(V[k]/2)` for each `i` from 1 to `k-1`. Finally the variable `V[k]` is minimised.

In the experiments, we used the instances where $k \in \{8 \dots 25\}$. For each pair of variables, the Identical Scopes heuristic is triggered by the coprime constraints, a symmetry breaking constraint if one exists, and a lower-bound constraint if one exists. For instances of size $8 \dots 19$, tabulation is successful for each candidate set of constraints, and all original constraints are replaced with binary tables. For the larger instances, Large AST (Nested) and Weak Propagation (Nested) identify sub-expressions of the coprime constraint: `(V[i]% d != 0)`, and these are tabulated.

5.2.3 Killer Sudoku

Killer Sudoku, CSPLib problem 57 (Nightingale, 2015), is a popular puzzle similar to the classical Sudoku, where an empty 9×9 grid is filled in with numbers $1 \dots 9$, such that each row, column and the nine non-overlapping 3×3 sub-squares take different values. In Killer Sudoku there are also clues (which differ between instances). Clues are sets of cells that sum to a given value (and also take different values). We use a straightforward model where each cell of the grid has one decision variable with domain $\{1 \dots 9\}$. The variables of each row, column and non-overlapping 3×3 sub-square are constrained by `allDiff`. Traditional 9×9 Killer Sudoku instances are trivial for a constraint solver, so we use an existing model for the 16×16 case, and an existing set of 100 16×16 instances (Nightingale et al., 2017). The instances are all satisfiable but unlike conventional Killer Sudoku puzzles, they may have more than one solution.

Each clue is a set of cells (from 1 to 5 cells) that are contiguous. For each clue, the model contains both an `allDiff` (except for size 1 clues) and a sum equality constraint on the same scope. In a conventional constraint solver (without clause learning), the model propagates poorly and solving times can be poor. It was shown by Nightingale et al. (2017) that associative-commutative common subexpression elimination (AC-CSE) (when combined with implied sum constraints generated from the `allDiff` constraints) can improve solving times substantially by connecting the two constraints on each clue, and also connecting the clues to the rules. The constraints below represent a size 3 clue from one of the instances.

```
(field[3,4] + field[4,4] + field[4,5]) = 26,
allDiff([field[3,4], field[4,4], field[4,5]])
```

For each clue of size 3 to 5, the two clue constraints are (together) identified as a candidate by the Identical Scopes heuristic. For clues of size 2, the clue `allDiff` is removed prior to tabulation because it is subsumed by a larger `allDiff`, leaving just the sum equality constraint. The sum equality triggers the Weak Propagation heuristic. Clues of size 2 to 4 are tabulated, but in some cases clues of size 5 cannot be tabulated. For example, the first instance (named `sol1` in the repository) has 9 clues of size 5, of which 6 were successfully tabulated. The other 3 failed the progress check after 10,000 nodes (as described in Section 4.2). In each case the arity of the generated table is equal to the size of the clue.

5.2.4 Knight's Tour Problem

The Knight's Tour Problem was described in Section 1.2. Recall that we use the Hamiltonian path version of Knight's Tour, i.e. the last square visited is not required to be a knight's move from the first square visited. An instance defines n and the starting location of the knight. We experiment with instances where $n \in \{6 \dots 12, 15, 20, 25, 30, 35\}$ and with two starting locations, $(0, 0)$ and $(0, 1)$, for 24 instances in total.

We use two models, and in both the location of the knight is encoded as a single integer $(nx + y)$ where (x, y) are the coordinates of the knight on the board (from 0). The first model is the *sequence* model introduced in Section 1.2 (see Figure 3), in which we have a one-dimensional matrix of variables (`tour`), with `tour[i]` representing the location of the knight at time-step `i`. The constraints enforce that initially the knight is at the given location, it never revisits a location (via `allDiff`), and each adjacent pair `tour[i]` and `tour[i+1]` corresponds to a knight's move. The knight's move constraint uses integer division and modulo to obtain the x and y coordinates. For convenience we re-cap this constraint here:

```
((|tour[i]%n - tour[i+1]%n| = 1) /\ (|tour[i]/n - tour[i+1]/n| = 2)) \/
((|tour[i]%n - tour[i+1]%n| = 2) /\ (|tour[i]/n - tour[i+1]/n| = 1))
```

The second model (named *successor*) has a matrix of variables `next` which indicate the successor of each location. The `next` variables are constrained by `allDiff`. In addition, it has all variables and constraints of the *sequence* model and a set of channelling constraints connecting `next` to `tour`, ensuring there are no cycles in `next`.[3] The channelling constraints are as follows:

```
forAll i : int(0..tourLength-2). next[tour[i]] = tour[i+1]
```

The knight's move constraint triggers the Duplicate Variables, Large AST, and Weak Propagation heuristics. For both models, when $n \leq 15$, all knight's move constraints are tabulated (creating a binary table). For instances where $n = 20$, the first 15 or 16 are tabulated (where variable domains have been reduced by preprocessing), and the number

3. The standard CP model of a Hamiltonian path problem uses a successor viewpoint and a global constraint (e.g. `path` in Gecode (Gecode Team, 2024)). Savile Row and Minion do not have the path constraint. Instead the *successor* model loosely follows an example in Gecode (Gecode Team, 2024) (credited to Gert Smolka) which has successor, predecessor, and *jump* variables that give the index of each location in the sequence. In preliminary experiments our model performed slightly better than the Gert Smolka model when using Minion. The size of both of our models and Gert Smolka's model (as the sum of domain sizes) is $\Theta(n^4)$.

reduces further to 9 when $n = 35$. For the remaining knight's move constraints, each division and modulo operator is identified by Weak Propagation (Integer), extracted and tabulated (creating a binary table constraint). Each unique division and modulo expression is tabulated once and the auxiliary variable is reused as described in Appendix A.2. When TabID is disabled, identical common subexpression elimination (CSE), which is part of the default configuration (Nightingale et al., 2017), improves the knight's tour constraint by adding auxiliary variables for the division, modulo, and absolute value expressions.

Finally, the channelling constraints in the *successor* model are translated to an `element` constraint that indexes from 1. If the lower bound of `tour[i]` is not 1 (after domain filtering) then a shifted index expression `tour[i]+c` is created.[4] The `tour[i]+c` expressions are identified by Weak Propagation (Integer) and tabulated for the 8 instances where $n \leq 15$ (also creating binary tables). When $n > 15$ a small proportion of them are tabulated.

5.2.5 N-Linked Sequence and Optimal N-Linked Sequence

This puzzle, proposed by Itay Bavly (Roeder, 2017b), requires arranging as many as possible out of the first 100 positive integers into a sequence, so that in every pair of adjacent numbers one is a multiple of the other. The longest possible sequence was found to consist of 77 numbers (Roeder, 2017a). The question was also asked for 1000 numbers, and in this case a sequence with 418 numbers was constructed. We use the name *n-linked sequence* as proposed by William Gasarch (2019), who also introduced the parameter n for the largest integer.

We consider two versions of the puzzle. The first version is a decision problem in which we ask whether a sequence exists of some given length, where the length `len` is a fraction of n chosen to be challenging (close to the unsatisfiability threshold, but still satisfiable). The second version is an optimization problem where we simply seek a longest sequence as in the original problem description. Both versions are modelled with a one-dimensional matrix `seq` of variables with domain $\{1 \dots n\}$ representing the sequence. In the decision version, `seq` is of length `len`, which is a parameter. In the optimization version, `len` is a variable that is maximised. In both versions the entire matrix `seq` is `allDiff`. The divisibility constraint for the optimization problem is shown below. In the decision problem, the condition `(i<=len)` is omitted.

```
(i<=len) -> ((seq[i]%seq[i-1] = 0) \/ (seq[i-1]%seq[i] = 0))
```

For the decision version of the problem, we used the 15 instances where $n \in \{60, 70, 80\}$ and $len \in \{n-30, n-25, n-20, n-15, n-10\}$. For the optimization version, we used instances $n = \{12, 14, 16, \dots, 42\}$. For the optimization problem, the Duplicate Variables and Weak Propagation heuristics identify each arity 3 divisibility constraint and they are all tabulated. For the decision problem, Duplicate Variables, Large AST, and Weak Propagation heuristics identify each arity 2 divisibility constraint and all are tabulated.

5.2.6 Peaceable Armies of Queens

This puzzle is also named 'Peaceably Coexisting Armies of Queens' (Smith et al., 2004). The problem asks how to place two equal-sized armies of queens on a chessboard so that the white queens do not attack the black queens, and vice versa. On a standard 8 by 8 board,

4. For instances where the starting location is $(0, 1)$ most `tour[i]` variables have a lower bound of 0.

there are 71 non-isomorphic solutions and their number grows quickly with board size. Some early results and discussion are due to Stephen Ainley (1977, Problem C5). A sophisticated model that makes use of the *regular* constraint was used in the MiniZinc Challenge 2022.

We use a very simple model, based on the *Basic Model* of Smith et al. (2004), that takes a single parameter n for the board size (defining an $n \times n$ board). For each square on the board we have a variable with domain $\{0, 1, 2\}$ with the values indicating no queen, white, and black respectively. As in the third model of Smith et al., we also use one additional variable *armySize* to indicate the number of queens in each army, and this is maximised. There are two `sum` constraints stating that there are `armySize` occurrences of values 1 and 2 respectively. The vast majority of the constraints are to prevent an attack between a pair of queens from opposing armies. Supposing the board is named `b`, and squares (i, j) and (k, l) share a row, column, or diagonal, then the *attack* constraint `b[i,j]+b[k,l] != 3` is posted.

We experimented with instances where $n \in \{4 \dots 11\}$. The two army size constraints are identified (together) by the Identical Scopes heuristic. For $n \geq 5$ the candidate has more than 20 variables and is discarded, otherwise tabulation fails the progress check after 1,000 nodes. Each attack constraint is identified by the Weak Propagation heuristic and all are tabulated (creating binary table constraints). The attack constraints are identical apart from the variable names, so after the first is tabulated the rest are retrieved from the cache. Without tabulation a new integer variable is introduced for each sum `b[i,j]+b[k,l]` in an attack constraint. Note that the MiniZinc Challenge 2022 model has the same viewpoint and the non-attack requirements are implemented with *regular* constraints which would usually propagate strongly, suggesting that tabulation of the attack constraints may be useful.

5.2.7 Strong External Difference Families

A Strong External Difference Family (SEDF) is an object defined on a group, with applications in communications and cryptography (Paterson & Stinson, 2016). For the purposes of this paper, a group is a set G with an associative and invertible binary operation $\times$. Also, G must contain an identity element e, which means that $e \times g = g$ for every $g \in G$. The model of SEDF described below (with additional symmetry breaking constraints) was used to find a number of previously undiscovered SEDFs, including the first in non-Abelian groups (Huczynska et al., 2021).

Given a finite group G on a set of size n, an (n, m, k, λ) SEDF is a list $A_1, \dots, A_m$ of disjoint subsets of size k of G such that, for all $1 \leq i \leq m$, the multi-set $M_i = \{xy^{-1} \mid x \in A_i, y \in A_j, i \neq j\}$ contains λ occurrences of each non-identity element of G.

The parameters of the SEDF problem are (n, m, k, λ), the group G given as a multiplication table `tab` (which is an $n \times n$ matrix of integers), and `inv`, a one-dimensional table which maps each group element to its inverse. The SEDF is represented as an $m \times k$ matrix `sedf`. The entire `sedf` matrix is contained in a single `allDiff` constraint. It has row symmetry (as the sets are not ordered) (Flener et al., 2002) and also each row has symmetry (as each row represents a set): any two rows may be exchanged in a solution while preserving solutionhood; also any two elements within a row may be exchanged while preserving solutionhood. The variables within each row and the first column are ordered with $<$ constraints to partially break the two kinds of symmetry.

To ensure each multi-set M_i has λ occurrences of each non-identity element, we use the `gcc` on the comprehension below. Value 1 has cardinality 0, and all other values have cardinality λ. Each `gcc` contains all variables in `sedf`. Note that both `sedf[i,p]` and `sedf[j,q]` are individual variables, and that the inner expression is equivalent to `tab[X, inv[Y]]` for integer variables `X` and `Y`.

```
[ tab[sedf[i,p], inv[sedf[j,q]]] |
        p:int(1..k), q:int(1..k), j:int(1..m), j!=i ]
```

We use a set of 12 instances where $19 \leq n \leq 29$, $3 \leq m \leq 8$, and $2 \leq k \leq 5$. The instances originate from research into SEDFs (Huczynska et al., 2021) and are provided in the experimental repository.

The Identical Scopes heuristic identifies the `allDiff` and all `gcc` constraints together. Duplicate Variables, Large AST, and Weak Propagation heuristics identify each `gcc` individually. In each case tabulation fails the progress check at 1,000 nodes. Identical Scopes (Integer) identifies cases where `tab[X,inv[Y]]` within a `gcc` overlaps with `X<Y` or `Y<X` ordering the first column (where `X` and `Y` are variables in the `sedf` matrix); Large AST (Integer) and Weak Propagation (Integer) identify the rest of the `tab[X,inv[Y]]` expressions and all are successfully tabulated for all instances (creating `table` constraints of arity 3).

5.3 Summary of Results

We have shown the utility of all four TabID heuristics when applied to top-level constraints, and in most cases their nested and integer versions as well. Table 1 summarises the set of heuristics applied to each problem class. The table indicates when a heuristic fires for at least one instance of a problem class, *and* at least one of the subproblems identified by the heuristic is tabulated. All heuristics except Duplicate Variables (Nested) and Duplicate Variables (Integer) meet these criteria for at least one problem class. The top-level heuristics are most important and fire on all but one of the problems, while the other heuristics are also contributing and are essential for SEDF. Nested and Integer heuristics tend to become relevant when top-level constraints are beyond the reach of tabulation, for example in SEDF and the largest instances of Knight's Tour.

For 9 of the 15 models, the generated tables are predominantly binary. Examples include games (e.g. Accordion Patience), maths problems such as Coprime Sets, and puzzles (e.g. Peaceable Armies of Queens). There are four examples of tabulation (predominantly) producing ternary table constraints: Handball Tournament Scheduling; Optimal N-Linked Sequences; SEDF; and Sports Scheduling Completion. BPMP has two sets of expressions that are tabulated: the channelling constraints of arity 4, and division and modulo expressions where an arity 2 table is produced. Killer Sudoku has 2, 3, 4, and 5 arity tables matching the sizes of the clues in the puzzle. Finally, with Still Life tabulation produces tables of arity 9, albeit on Boolean variables. The *nodeLimit* parameter is set to a conservative value (100,000), minimising the time cost of tabulation but also limiting the arity of generated tables. However the preponderance of arity 2 and 3 tables comes from the structure of the models rather than the effect of *nodeLimit*.

Problem	Top-level				Nested				Integer			
	IS	DV	LAST	WP	IS	DV	LAST	WP	IS	DV	LAST	WP
BlackHole				✓	✓							
BPMP		✓	✓	✓	✓		✓	✓				✓
Handball		✓	✓	✓	✓							✓
JPEncod.	✓	✓	✓									
MDSL	✓											
SportsSchC				✓								
Accordion	✓	✓	✓	✓								✓
Coprime	✓						✓	✓				
KillerSu.	✓			✓								
KnTourSeq		✓	✓	✓								✓
KnTourSucc		✓	✓	✓								✓
NLinkedSeq		✓	✓	✓								
NLinkedOpt		✓		✓								
PAQueens				✓								
SEDF									✓		✓	✓

Table 1: Heuristics that trigger for each problem class: Identical Scopes (IS); Duplicate Variables (DV); Large AST (LAST); and Weak Propagation (WP). A tick indicates that the heuristic identifies at least one subproblem for at least one instance of the problem class *and* the subproblem is tabulated.

6. Experimental Evaluation of TabID

Having shown that TabID identifies and tabulates promising subproblems of each of the benchmark problems, we now evaluate whether TabID actually speeds up solving of these problems. We test this hypothesis with two conventional CP solvers (Minion and Gecode, taking advantage of extensive research into table propagators) and two CP solvers with conflict learning (OR-Tools and Chuffed). We also test the hypothesis in a different setting, where problem instances are encoded into SAT and solved with a recent CDCL SAT solver (Kissat). In this case we experiment with two different encodings of the table constraints. We first give experimental details and describe the statistical method, and then look at each of the problems of Section 5 in turn. TabID is configured in the same way as in Section 5: *nodeLimit* is set to 100,000; and the SAT size limit is *not* applied except where stated otherwise.

6.1 Experimental Details

We evaluate tabulation with five solvers in six configurations, as follows:

Minion Version 1.9.2 of Minion (Gent et al., 2006), with ascending value and static variable orderings. The Trie propagator is used for all table constraints (Gent, Jefferson, Miguel, & Nightingale, 2007).

Gecode Commit `d1bd874` on the `release/6.3.0` branch of Gecode (Gecode Team, 2024) with the Compact Table propagator (Ingmar & Schulte, 2018). The search order is the same as for Minion.

Chuffed Version 0.13.0 of Chuffed (Chu et al., 2018) with free search enabled (i.e. the solver is free to use dynamic search ordering heuristics). Chuffed encodes all table constraints into SAT internally: it uses the support encoding for binary table constraints (Gent, 2002) and the Bacchus encoding otherwise (Bacchus, 2007).

OR-Tools Version 9.7.2996 of OR-Tools (Perron & Didier, 2023) CP-SAT solver with free search enabled. The support encoding is used for binary table constraints (Gent, 2002). Each non-binary table constraint is compressed to a set of short tuples or c-tuples (Katsirelos & Walsh, 2007) which is then encoded using basic Boolean constraints similarly to Akgün et al. (2016).

Kissat Version 3.1.1 of Kissat (Biere & Fleury, 2022) with Savile Row's default SAT encodings for all constraint types unless otherwise stated. The support encoding is used for binary table constraints (Gent, 2002) and the Bacchus encoding for other tables (Bacchus, 2007).

Kissat (MDD) As above, but non-binary tables are compressed into Multi-valued Decision Diagrams (MDDs) which are then encoded with the GenMiniSAT encoding (Abío et al., 2016).

The solvers and configurations were chosen to include: the recent Compact Table propagator, as well as an earlier table propagator; conventional CP (Minion and Gecode), clause learning CP, and SAT solvers; and both static and dynamic search orders. We used Savile Row 1.10.0 for the experiments, extended with the new TabID method. Each reported time is the median of five runs on a 134-node cluster, each node having two 48-core AMD EPYC3 processors and 512 GB RAM; jobs were submitted requiring 1 CPU core and 6 GB RAM. A time limit of 1 hour was applied. Reported times are total times (unless otherwise stated) and include the time taken by Savile Row to tailor the instance, the time taken to apply TabID (if activated), and the backend solver time. Software, models and parameter files for all experiments are publicly available online (Akgün et al., 2024).

6.2 Statistical Analysis

To compare two configurations A and B of Savile Row (for example, where A does not include TabID and B does), we first take the median of the total time for each instance and each configuration. The median was chosen because it is less affected by outliers than the mean. Instances where *both* configurations timed out are discarded. For the remaining timeouts (i.e. median total time is >3600 s) we apply a PAR2 penalty, so their median total time is considered to be 7200 s. For each instance, we take the quotient of the two medians ($\frac{A}{B}$). We take the geometric mean of the set of quotients to obtain s, a single statistic to compare A and B. If $s > 1$ then B is considered to be better than A. The geometric mean is more appropriate than the arithmetic mean in this case (Nightingale et al., 2017).

Where s is close to 1, it may not be clear whether the difference between A and B is statistically significant. As in Nightingale et al. (2017), we use the bootstrap method to

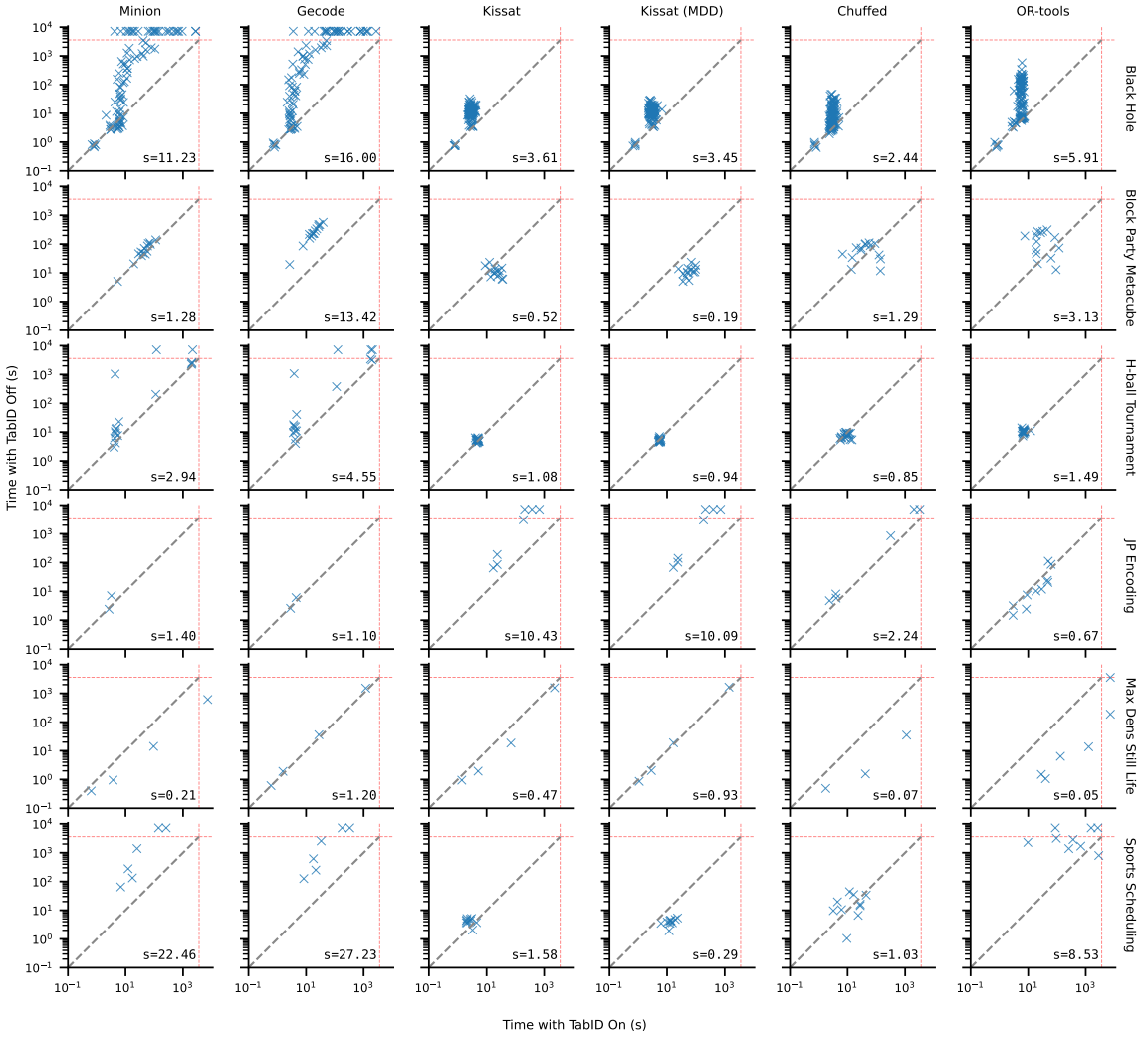

Figure 7: **TabID without SAT Size Limit vs No TabID.** Total runtime (including both Savile Row and the solver) with each of the six solvers for the problem classes: Black Hole, Block Party Metacube Problem, Handball Tournament Scheduling, JP Encoding, Maximum Density Still Life, and Sports Scheduling Completion. Time with TabID is shown on the x-axis and without TabID on the y-axis. The red dotted lines indicate the time limit of 1 hour; points beyond the line timed out and had the PAR2 penalty applied. Points above the $x = y$ diagonal were solved faster with TabID than without. The s value on each plot is the geometric mean of speedup quotients.

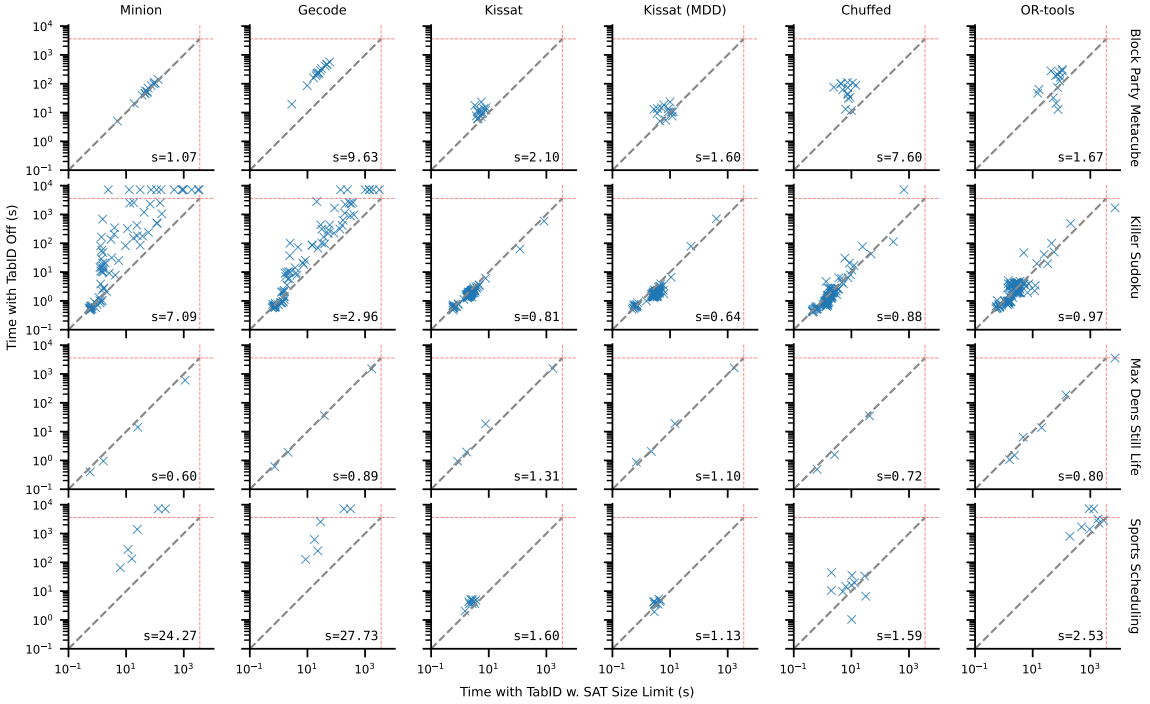

Figure 8: **TabID with SAT Size Limit vs No TabID.** Total runtime (including both Savile Row and the solver) with each of the six solvers for the problem classes: Block Party Metacube Problem, Killer Sudoku, Maximum Density Still Life and Sports Scheduling. Time with TabID using the SAT size limit is shown on the x-axis and without TabID on the y-axis. Other details are the same as in Figure 7.

compute a 95% confidence interval of s with 100,000 bootstrap samples. We consider the difference between A and B to be statistically significant when the 95% confidence interval does not include 1.

6.3 Evaluation of Baseline Problems

First we present results for six problem classes where table constraints have been applied in the literature to improve a constraint model. Four of these classes were used in Dekker et al. (2017) to evaluate their auto-tabling method, while the other two are from earlier constraint modelling literature. Results are shown in Figure 7. For each problem class we use the set of instances described in Section 5.1. In this section we give a summary of results. Further analysis can be found in Appendix C.1.

For Black Hole, TabID significantly speeds up all six solvers with the most substantial improvements seen in solvers without clause learning. The results are mixed for BPMP, where Minion, Gecode, and OR-Tools show a significant speedup, Chuffed does not reach significance and the SAT solvers are slowed down by TabID. For Handball Tournament

Scheduling there is a clear benefit for the CP solvers without learning, but for all other solvers the instances are too easy to show a clear difference. On the JP Encoding problem, TabID substantially improved the SAT solver's performance but made much less difference with the CP solvers. Maximum Density Still Life might seem an obvious candidate, however we found that TabID does not strengthen propagation (node counts for Minion and Gecode are not reduced) and results are mostly negative. Finally, for Sports Scheduling TabID substantially improved the performance of the CP solvers without learning (as expected) but results with the learning solvers are mixed.

For problems where there are substantial negative results ($s < 0.5$ for any solver), adding the SAT size limit can help. Results with the SAT size limit enabled are shown in Figure 8. For BPMP, we found that adding the SAT size limit improves performance for Kissat, Kissat (MDD), and Chuffed. For Maximum Density Still Life and Sports Scheduling, switching on the SAT size limit broadly improves the performance of TabID.

6.4 Evaluation of New Case Studies

In this section we give experimental results for seven new case studies described in Section 5.2. Results are shown in Figures 9 and 10, and for all problem classes the set of instances is described in Section 5.2. We give an overview of results here: further discussion of results can be found in Appendix C.2.

For Accordion Patience, TabID dramatically reduces search with the non-learning CP solvers, however with the learning solvers the difference is small or even negative. On the Coprime Sets problem, TabID allows a few additional instances to be solved by the non-learning CP solvers but makes little difference to the other solvers despite TabID changing the model extensively. On the Killer Sudoku problem, TabID strongly improves performance of the non-learning CP solvers but also increases the SAT encoding size quite substantially and slows down all learning solvers on average. Adding the SAT size limit moves all the average speedups closer to 1 for Killer Sudoku (Figure 8). On the *sequence* model of Knight's Tour, TabID causes a very clear speedup for all solvers. When $n \leq 15$ TabID produces an improved model with much stronger propagation and no auxiliary variables. With the *successor* model the results are mainly positive but complex, and the reasons for that are discussed in Appendix C.2.

TabID speeds up all solvers quite substantially on the N-Linked Sequence problem, however Kissat and OR-Tools mainly benefit for the easier instances suggesting these solvers are able to improve the formulation over time by conflict learning. Results for Optimal N-Linked Sequence are similar but with reduced speedups due to the generation of larger tables. For Peaceable Armies of Queens, the tabulated model propagates much better leading to big speedups for the non-learning CP solvers, however the learning solvers are almost unchanged. Finally, SEDF demonstrates that large speedups are possible (for all solvers) even when no top-level constraints are tabulated.

6.5 Discussion

With the conventional CP solvers Gecode and Minion, we expected TabID to improve solving time, and it does in the vast majority of cases. Three of the four groups of heuristics (Identical Scopes, Duplicate Variables, Weak Propagation) are based on strength of propaga-

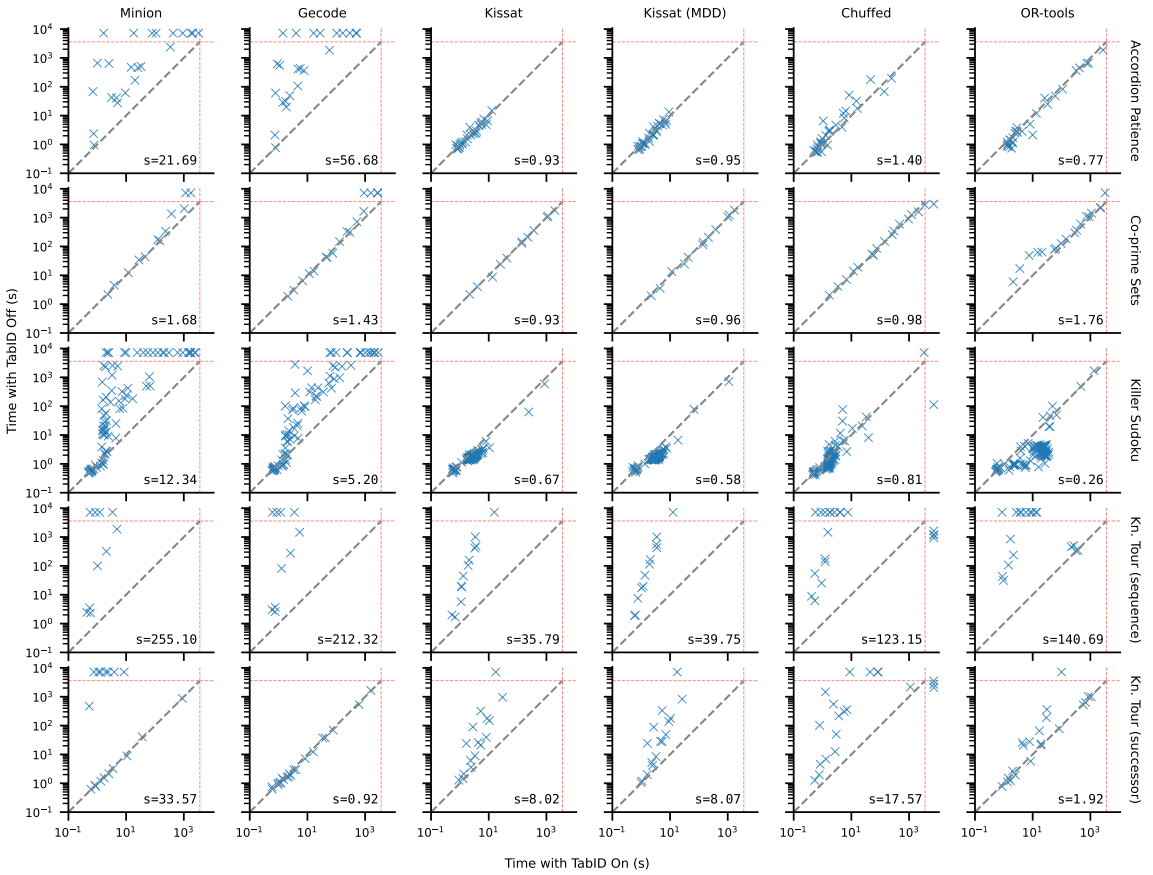

Figure 9: **TabID without SAT Size Limit vs No TabID.** Total runtime (including both Savile Row and the solver) with each of the six solvers for the problem classes: Accordion Patience, Coprime Sets, Killer Sudoku, and Knight's Tour (with two models). Time with TabID is shown on the x-axis and without TabID on the y-axis. Other details are the same as in Figure 7.

tion, i.e. identifying sub-problems where propagation is expected to be weak. Assuming the heuristics are accurate, the question then is whether the benefit of obtaining GAC outweighs the cost of generating and propagating table constraints. Gains range from minor efficiency improvements (e.g. Block Party Metacube Problem with Minion) to dramatic reductions in search (e.g. Black Hole, Killer Sudoku, N-Linked Sequence). The other group of heuristics (Large AST and its integer and nested versions) aim to improve efficiency by replacing a large, cumbersome expression with a table. Large AST heuristics are triggered on several problem classes but typically fire together with other heuristics (see Table 1). One example is SEDF, and in this case TabID is worthwhile (particularly for the most difficult instances of SEDF which are sped up by over 40 times). We also have a few cases where TabID slows down search: the Knight's Tour *successor* model with Gecode (because Gecode has an

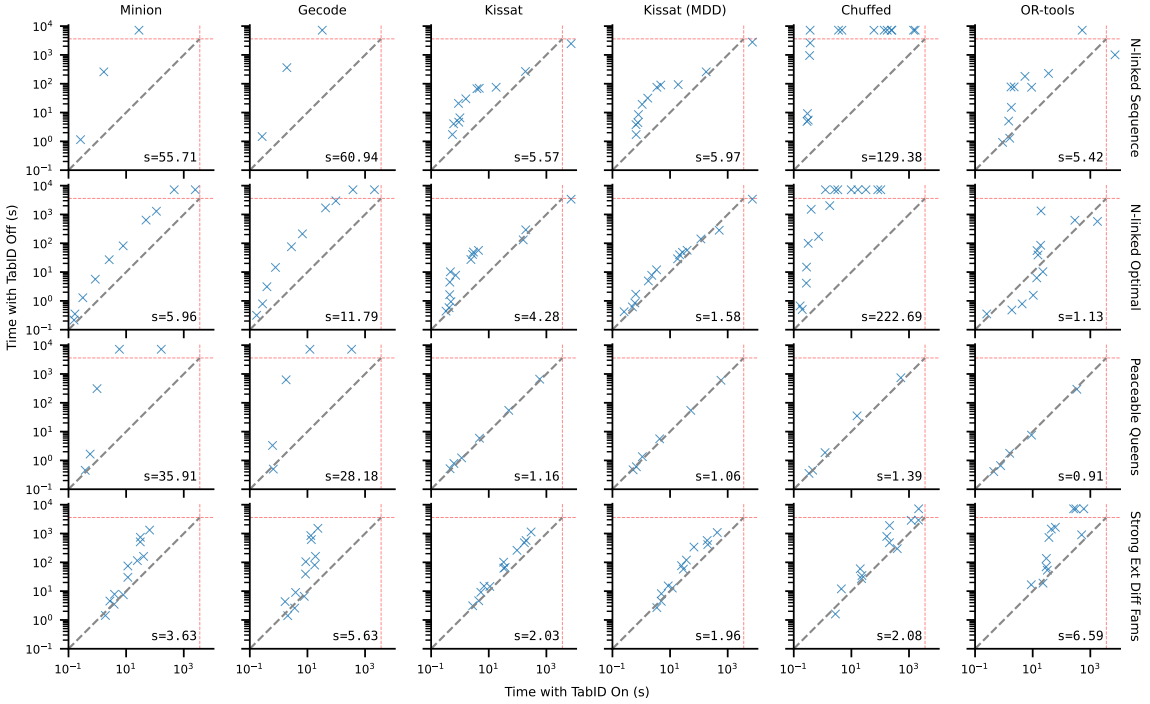

Figure 10: **TabID without SAT Size Limit vs No TabID.** Total runtime (including both Savile Row and the solver) with each solver for the problem classes: N-Linked Sequence, Optimal N-Linked Sequence, Peaceable Armies of Queens, and Strong External Difference Families. Time with TabID is shown on the x-axis and without TabID on the y-axis. Other details are the same as in Figure 7.

efficient GAC propagator for the shift constraint), and Still Life with Minion (where TabID does not strengthen propagation and the table constraints are simply slower to propagate).

For the solvers with conflict learning (Kissat, Kissat (MDD), Chuffed, OR-Tools) the picture is not as simple. None of them use a native table propagator. Table constraints are encoded into SAT and the size of the encoding affects solver performance. Also, conflict learning may be able to mitigate weak propagation. For some problem classes, TabID is clearly worthwhile: for Black Hole, the solving time becomes approximately constant and the most difficult instances (without TabID) are sped up by approximately 10 times for the SAT solvers and more for Chuffed and OR-Tools; both models of Knight's Tour are sped up by orders of magnitude (except the successor model with OR-Tools); and N-Linked Sequence exhibits large speed-ups for some instances. In each case the generated table constraints are binary and are encoded compactly without any additional SAT variables. SEDF (where the generated table constraints have arity 3) also benefits from TabID to a smaller degree.

All problem classes where TabID is notably worse with the conflict learning solvers have non-binary constraints (BPMP for some solvers/instances, Sports Scheduling Completion for some solvers/instances, Still Life, and Killer Sudoku). The results suggest that arity,

SAT encoding size, or both are important. Adding the SAT size limit to TabID improves average performance on all four of these problem classes, for most of the conflict learning solvers. The exceptions are BPMP and Sports Scheduling with OR-Tools. Figure 8 shows the performance using TabID with the SAT size limit applied for these four problem classes. Full results with the SAT size limit are given in Appendix D.

Finally, the overhead of TabID proved to be acceptable in these experiments. All runtimes reported in this section are total times, but further analysis presented in Appendix E shows that the time spent applying TabID (as a proportion of total time) is usually small.

7. Experimental Evaluation of TabID on Other Problem Classes

In this section we evaluate TabID on a large set of models and problem instances from an existing collection. The purpose is to evaluate the effect of TabID on models where we do not necessarily expect it to improve the model. The key questions are whether TabID represents an unacceptable overhead, and whether the heuristics and limits in TabID accurately identify only those subproblems where tabulation will be useful. The *nodeLimit* parameter is set to 100,000 and the SAT size limit (Section 4.3) is applied for all 6 solvers and all problem classes.

We used all 50 models and 596 instances from Nightingale et al. (2017), including models and instances that were used in other experiments in this paper. The models are of a variety of problems including combinatorial designs (e.g. BIBD, OPD), puzzles (such as English Peg Solitaire), and industrial design problems (such as SONET). All models and instances are available in the experiments repository.

Results are plotted in Figure 11. In this case we have one plot for each solver, and all 50 models are plotted together. TabID is a modest win on average for Minion and Gecode, with Killer Sudoku and Peaceable Armies of Queens showing the largest gains. For the other four solvers, TabID (with the SAT size limit) adds a small overhead on average. Each plot in Figure 11 is annotated with the proportion of instances where speedup s of the instance is less than 0.9, where it is close to 1 ('close'), and where it is greater than 1.1. Approximately half the instances lie close to the $x = y$ line in each case (i.e. $0.9 \leq s \leq 1.1$).

The geometric mean speedup for each solver is close to 1, showing that the overhead of TabID does not deteriorate performance excessively. For Gecode and Minion, a modest overall speedup is observed, with 95% confidence interval values of $[1.05, 1.18]$ and $[1.17, 1.39]$ respectively. The learning solvers incur a small overhead, with intervals of $[0.88, 0.96]$ for Chuffed, $[0.94, 0.99]$ for Kissat, $[0.88, 0.94]$ for Kissat (MDD), and $[0.87, 0.96]$ for OR-Tools. However, we should not simply turn off TabID whenever the chosen solver has conflict learning. Results presented in Section 6 show that large improvements in performance are possible with conflict learning solvers.

8. Scalability of TabID

In this section we investigate how our approach scales as the arity of a generated table increases. The SAT size limit is switched off and *nodeLimit* is set to 100,000 for this experiment. Even though large-arity tables may not be of practical interest in general, there might be cases where they are useful and we want our method to successfully generate

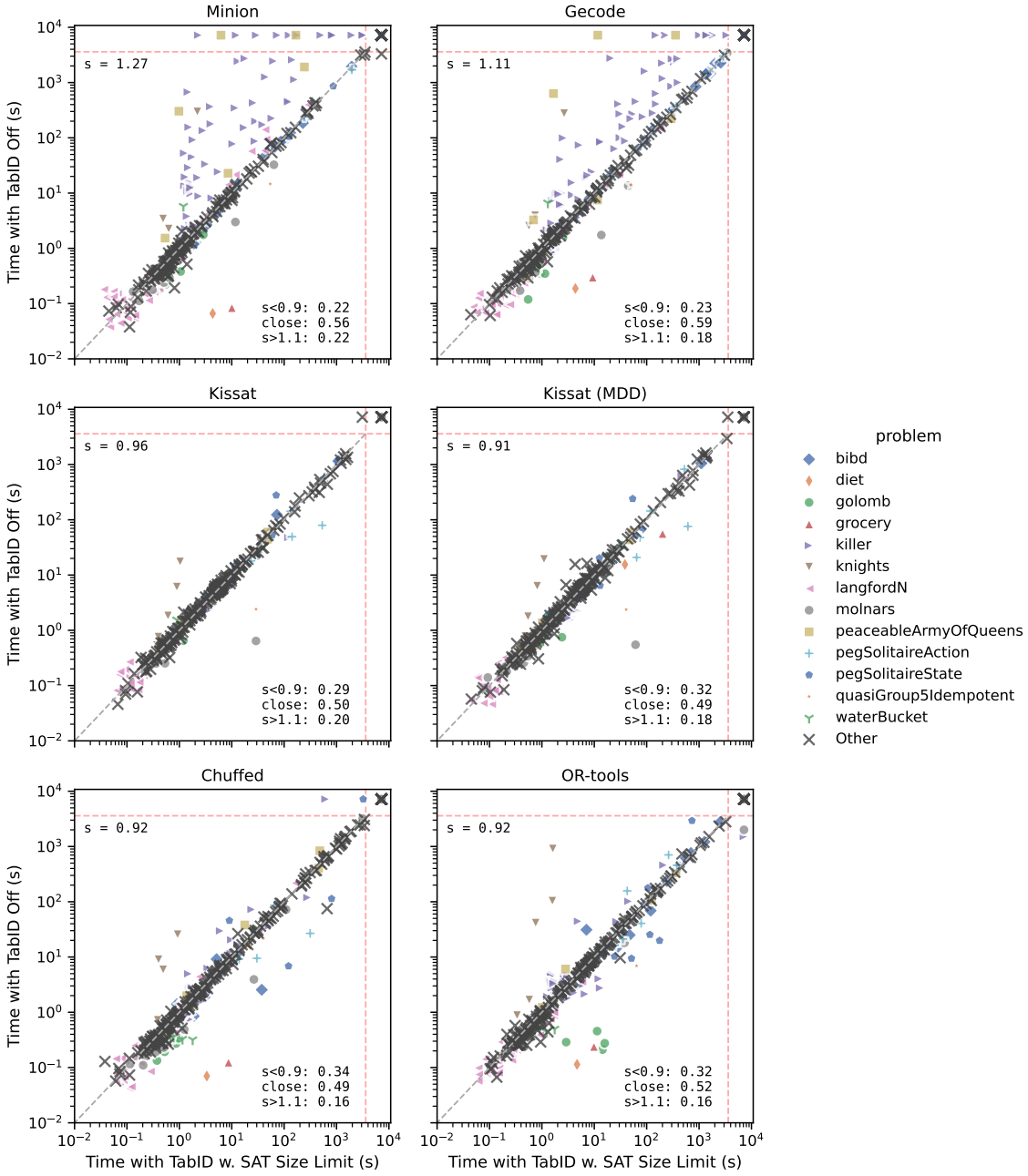

Figure 11: For each of the six solvers, the figure plots the total time (including both Savile Row and the solver) with TabID using the SAT size limit on the x-axis and without TabID on the y-axis for a set of 50 models. Other details are the same as in Figure 7. The problem classes with a distinct symbol and colour have at least one point away from the $x = y$ diagonal on at least one of the plots.

Sequence	Arity	Tabulation		Time (s)			Solver	Auto
variables	r	nodes	tuples	Tabulation	SR Total	Solver	nodes	
—	—	—	—	—	0.22	0.36	174,171	—
2	3	724	285	0.01	0.24	0.03	2435	Yes
3	4	4,999	3,697	0.04	0.32	0.04	588	Yes
4	5	60,454	51,689	0.23	0.89	0.37	330	No
5	6	835,789	762,213	0.85	1.52	6.34	203	No
6	7	12,268,984	11,396,505	13.88	21.57	129.98	143	No

Table 2: Results of tabulating subsequences of lengths 2, 3, 4, 5, and 6 (generating constraints of lengths 3, 4, 5, 6, and 7) of the optimal N-Linked sequence problem with $n = 15$ and solving with Minion. The first line shows no tabulation. The total node count, number of tuples generated, and time are reported for the tabulation process in columns 3-5. Savile Row total time, solver time and solver nodes are reported in columns 6-8. Finally the Auto column indicates whether tabulation would have completed with the work limit and progress checks switched on.

such tables. For this purpose, we use the optimal n-linked sequence problem introduced in Section 5.2.5 and investigate the time and search nodes required to generate tables of various arities and also the effect of them on a CP solver.

Here we experiment with the instance where $n = 12$, and we scale tabulation up by increasing the number of sequence variables in the table constraint. In particular, we use 2 to 6 adjacent variables in the table, thus table arities r range from 3 to 7, and we maximize the sequence length from 6 upwards. To achieve this we have written a separate model for each value of r, and in these models the divisibility constraint is extended to cover $r - 1$ sequence variables. For example, when $r = 4$ the divisibility constraint (for each `i` in $\{3 \dots n\}$) is as follows:

```
!active[i] \/ ( ((seq[i]%seq[i-1] = 0)   \/ (seq[i-1]%seq[i] = 0))
             /\ ((seq[i-1]%seq[i-2] = 0) \/ (seq[i-2]%seq[i-1] = 0)))
```

We also post an `allDiff` constraint over the same set of `seq` variables:

```
active[i] -> allDiff([seq[i-2],seq[i-1],seq[i]])
```

The Identical Scopes heuristic identifies these two constraints together as a candidate. For each value of r, 6 constraints of arity r are tabulated, and 5 binary constraints are tabulated between adjacent pairs of variables within the first 6 variables in the sequence (where `active[i]` is true and the conjunction decomposes into $r - 2$ binary constraints before tabulation).

The results of the experiments are shown in Table 2. The first row corresponds to the case when no tabulation is invoked. The 'tabulation nodes' in the third column is the number of nodes generated to tabulate all candidates. The arity r candidate is tabulated once and the other 5 are retrieved from the cache. *Auto* on the last column says whether automatic tabulation could be done with the progress checks and work limit on. For the *No* rows, TabID would stop at the first progress check at 1,000 nodes.

The results show that the number of nodes explored decreases significantly as we generate larger tables. However, tabulation explores more nodes, and tabulation, Savile Row and solver times increase. While the stronger inference comes with a cost, the incurred runtimes seem reasonable until we reach 762,213 tuples (arity 6). With 11 million tuples (arity 7), we observe a notable cost in tabulation, but still we are able to generate and use the tables.

In summary, the overhead of generating the tables scales as expected and is not prohibitive until we have millions of tuples ($r = 7$). The best balance of propagation strength against overhead is found with 285 tuples ($r = 3$), while $r = 4$ is the largest size allowed with the progress checks and work limit switched on.

9. Related Work

In accordance with reformulating a subset of the problem constraints for stronger and/or cheaper constraint propagation, the Globalizer tool (Leo et al., 2013) of MiniZinc helps detect opportunities to use global constraints in constraint models. For instance, it can detect that a set of disequality constraints can be converted to an all-different constraint. However, the approach is not completely automatic. First, the user provides some instances, then the tool solves them and analyzes the solutions to find properties that match global constraints. The detected global constraints are provided as a suggestion to the user because they are not guaranteed to be correct for all problem instances.

The tool proposed by Dekker et al. (2017) can convert any predicate (Boolean function) in MiniZinc into a table constraint, but the user must annotate the predicates to be tabulated. In the same vein, the IBM ILOG CPLEX Optimization Studio software supports `strong` annotations to indicate that the solver should find a precomputed table constraint corresponding to a specified set of variables; the resulting table constraint is then added to the model as an implied constraint (IBM, 2024). The Propia library performed a similar step for an annotated goal in ECLiPSe (Le Provost & Wallace, 1992). In all of these approaches, the user is required to identify the promising parts of a given model for tabulation.

Ek (2018) proposed to automatically encapsulate sub-problems of MiniZinc models and to identify the strongest candidates for tabulation using a set of heuristics based on numerical scores. One example is minimising *argument modesty*, a measure of the number and domain size of arguments (decision variables and problem class parameters) of the sub-problem. The implementation builds on MiniZinc auto-tabling (Dekker et al., 2017). The goal of the system is very similar to TabID, but the approach is quite different: encapsulation and scoring are applied to the problem class model, and the heuristics are entirely different to ours. The system was evaluated with models of Black Hole patience and the Block Party Metacube Problem (BPMP). The preliminary results reported were not positive, however there are many ways in which this work could be developed further.

With the same goals as tabulation, researchers have studied replacing a set of constraints with other constraints. De Uña et al. (2018) considered alternative data structures to store the contents of tables. Specifically, they proposed the use of Multivalued Decision Diagrams (MDDs) and Deterministic Decomposable Negation Normal Forms (d-DNNFs), motivated by the fact that creating a table can be costly when the reformulated subproblem has a large number of solutions. The experimental results show that while building compact structures like MDDs or d-DNNFs can be substantially faster than building tables in certain problem

classes, it is not clear which structure yields the best total solving time overall. The paper suggests that a more robust technique could choose between the three structures (table, MDD, d-DNNF) depending on an estimate of the size of the solution set of the subproblem. In any case, the presented approach does not automatically identify candidate subproblems. The heuristics we present in this paper could be used to detect automatically opportunities for generating MDD and d-DNNF constraints.

As an alternative, Löffler et al. (2020) considered transforming subproblem constraints to a regular constraint. The experimental results demonstrate that the best total solving times are achieved either by a table constraint or by the combination of regular and table constraints. Again, there is currently no specific algorithm to detect candidate subproblems automatically. The approach either relies on the heuristics presented in our earlier work (Akgün et al., 2018) or is applied manually.

Our approach could also generate tables in other compact representations where ordinary tuples are replaced by compressed tuples (Katsirelos & Walsh, 2007; Régin, 2011; Xia & Yap, 2013), short tuples (Nightingale et al., 2011, 2013; Jefferson & Nightingale, 2013), or smart tuples (Mairy et al., 2015). The latter generalizes classical, compressed and short tuples, can lead to exponentially smaller tables, and can encode compactly many constraints, including a dozen well-known global constraints. Under a very reasonable assumption about the acyclicity of smart tuples, a polynomial time GAC algorithm was introduced and shown to be effective in practice (Mairy et al., 2015). In addition, Le Charlier et al. (2017) proposed automatically synthesizing smart table constraints from (ordinary) table constraints. While the theoretical worst-case time complexity of the algorithm is quadratic in the size of the input table, it was shown to have quasi linear execution time on the considered benchmarks. Segmented tuples (Audemard et al., 2020) also generalise compressed tuples by allowing tuples to contain sub-tables over a restricted scope. They demonstrate good performance of segmented tables compared to MDD and regular formulations of a crossword design problem.

Propagation of multiple overlapping table constraints can be stronger than GAC on each constraint individually. For example, Schneider and Choueiry (2018) proposed a filtering algorithm based on Compact Table for full PWC, a level of consistency that is significantly stronger than GAC. More recently Wang and Yap (2020) presented a decomposition to binary constraints which achieves full PWC in some cases.

Finally, advances in propagating or encoding table constraints could improve performance of TabID. Wang and Yap (2023) survey formalisms for ad-hoc constraints (compressed tables, MDDs, among others) focusing on expressive power. SAT encodings continue to improve, e.g. Wang and Yap (2022) present some faster alternatives to MDD encodings.

10. Conclusions and Future Work

In this paper we have demonstrated that a small set of heuristics can successfully identify promising sub-problems in a constraint model for tabulation, and that these opportunities can be effectively exploited. The entire process is automated in TabID, situated in the constraint modelling system Savile Row. Our heuristics identify the same tabulation opportunities as recent work by Dekker et al. (2017) where manual annotations of a constraint model are used. In addition we have presented nine other case studies demonstrating the efficacy of our heuristics and automated tabulation. We evaluated the method with SAT,

learning CP, and conventional CP solvers, on a wide variety of models. The results vary considerably between problem classes, and between solvers on the same problem class. In some cases, the method produces orders of magnitude improvements in solving time and this is achieved completely automatically. In general we have observed more gains with CP solvers that have native, efficient table propagators. When using a SAT solver, the arity of the generated tables and the size of the encoding are important to solver performance.

We have also evaluated the method on a set of 50 models (from a pre-existing collection) where for most models we expected there to be no opportunities for useful tabulation. In the large majority of cases we found that TabID makes very little difference to total time, the best possible outcome for TabID when there are no opportunities for useful tabulation. In this experiment we applied a SAT encoding size limit in addition to the work limit that is always part of TabID. In the final experiment we investigated how tabulation scales as the arity of a constraint is increased, showing (for the Optimal N-Linked Sequence problem) that table generation can scale far beyond the table size (i.e. number of tuples) that has the best trade-off between propagation strength and overhead. Currently, a potential user of TabID would be well advised to experiment with a few instances of their problem class to determine whether TabID is useful.

There are two main avenues of future work. Firstly, there is opportunity to refine and extend the collection of heuristics in TabID. Machine learning could be employed to help decide which expressions are promising candidates, as well as to tune the parameters of the tabulation procedure. It has already been applied to predict whether a problem class is generally amenable to tabulation (Cena et al., 2023), as well as to predict whether a `regular` or a `table` constraint works faster in a CSP with a single constraint (Löffler et al., 2021). Secondly, it would be interesting to investigate generating compressed table representations such as MDDs (with a work limit and progress checks adapted to the representation). For some compressed representations (including MDDs) there already exist algorithms to generate them (surveyed in Section 9). Integrating compressed representations promises to improve the scalability of the method in some cases, allowing more of the identified candidates to be tabulated.

Acknowledgments

We thank EPSRC for grants numbered EP/P015638/1, EP/P026842/1, EP/R513386/1, EP/W001977/1, and EP/V027182/1 which have supported the authors at various times while this research was undertaken. Dr Jefferson held a Royal Society University Research Fellowship. This project was undertaken on the Viking Cluster, which is a high performance compute facility provided by the University of York. We are grateful for computational support from the University of York High Performance Computing service, Viking and the Research Computing team.

Appendix A. Identifying Promising Nested Expressions

A.1 Identifying Promising Nested Boolean Expressions

In this section we adapt the four heuristics to identify candidates for tabulation from the set of nested Boolean expressions. The heuristics are adapted as follows (in the order they are applied).

Identical Scopes (Nested) identifies a Boolean expression c_1 that has the same scope as a top-level constraint c_2, where c_2 does not contain c_1.

Duplicate Variables (Nested), and Large AST (Nested) are unchanged apart from applying to nested Boolean expressions rather than top-level constraints.

Weak Propagation (Nested) identifies a Boolean expression c_1 that is likely to propagate weakly (Section 3.3), and there exists a top-level constraint c_2 that propagates strongly, with at least one variable in the scope of both c_1 and c_2.

Once again, subproblems containing more than 20 distinct variables are not considered as candidates for tabulation. All four heuristics except Identical Scopes (Nested) identify a single Boolean expression to be tabulated. Identical Scopes (Nested) is somewhat different: it identifies a single Boolean expression (named c_1) to be replaced, but the table constraint is generated from the conjunction of c_1 with all top-level constraints that have the same scope as c_1. For example, suppose that $x \neq y$ is a nested Boolean expression c_1, and $x \leq y$ is a top-level constraint c_2 (the only top-level constraint with scope $\{x, y\}$). $x \neq y$ would trigger the Identical Scopes (Nested) heuristic. A table would be generated for $(x \neq y) \wedge (x \leq y)$ and c_1 would be replaced with the new table constraint, while c_2 would remain unchanged.

The nested Boolean expression heuristics are applied after the top-level constraint heuristics. They are applied to the AST in a top-down order: all four heuristics are applied to an AST node n before any of the descendants of n. The rationale is to tabulate the largest possible Boolean expression to get the most benefit (within the limits described in Section 4.2). If an expression is identified by one of the heuristics but tabulation fails (e.g. by exceeding a work limit), then parts of the expression (i.e. descendant nodes in the AST) may still be tabulated. This occurs in the Knight's Tour problem for example, described in Section 5.2.4.

A.2 Identifying Promising Integer Expressions

Finally we adapt the heuristics to apply to integer expressions. The integer expression heuristics are applied last. The AST is traversed in the same order as for Boolean nested expressions (i.e. parents before children). Tabulating an integer expression has the additional step of introducing a new auxiliary variable. To avoid the overhead of introducing unnecessary variables, we only consider expressions that would be extracted by general flattening (Section 2.3). The model may contain identical integer expressions, and to avoid tabulating multiple identical expressions (and introducing multiple auxiliary variables for them) we use a cache mapping expressions to auxiliary variables. Before applying the heuristics to an expression e_1, we first check the cache, and if another expression identical to e_1 has already been tabulated with auxiliary variable a_1 then e_1 is replaced with a_1.

Prior to applying the heuristics to expression e_1, a *temporary* auxiliary variable a_{temp} is created, and the constraint c_{temp}: $a_{temp} = e_1$ is made (but not attached to the AST). In terms of e_1 and c_{temp}, the heuristics are as follows:

Identical Scopes (Integer) identifies an integer expression e_1 that contains more than one variable and has the same scope as a top-level constraint c_2 where c_2 does not contain e_1.

Weak Propagation (Integer) identifies an integer expression e_1 where either: (a) e_1 is likely to propagate weakly (as in Section 3.3), and the top-level constraint c_2 that contains e_1 is likely to propagate strongly when e_1 is temporarily replaced with a_{temp}; or (b) the constraint c_{temp} would trigger the top-level Weak Propagation heuristic.

Duplicate Variables (Integer), and Large AST (Integer) are identical to the heuristics Duplicate Variables and Large AST applied to c_{temp}.

If e_1 triggers any heuristic except Identical Scopes (Integer), tabulation is attempted on the constraint c_{temp}. If tabulation is successful, then a_{temp} becomes permanent, e_1 is replaced with a_{temp}, and the new table constraint is attached to the AST (as described in Section 3.1). Identical Scopes (Integer) is similar: the only difference is that tabulation is attempted on the conjunction of c_{temp} with all top-level constraints that have the same scope as e_1. As with top-level and nested heuristics, subproblems containing more than 20 distinct variables (including a_{temp}) are not considered as candidates for tabulation.

Appendix B. Generating Tables

The algorithm for generating a table is implemented entirely within Savile Row and operates directly on an arbitrarily nested Boolean expression, avoiding the need to tailor a candidate expression for an external solver. In preliminary experiments we also implemented tabulation subject to resource limits in Minion, to take advantage of the propagation mechanism in the solver. However, we found that this was slower than our approach of doing tabulation entirely within Savile Row. In our discussion we therefore focus only on the internal generation approach.

Given a Boolean expression e to tabulate, we first traverse the AST of e (in depth-first, left-first order) to collect a list of its variables without duplication. A table is generated by depth-first search with a static variable ordering (the order of the variable list) and branching on each value in turn. At each node of search the expression is simplified (Section 2.3); if it evaluates to `false` then the search backtracks (regardless of whether all variables have been assigned). At each leaf node of search that evaluates to `true`, we store the corresponding assignment as a tuple in the table. This method adapts to the local structure of the expression, and it only considers the remaining subset of variables occurring in a subtree. This is in contrast to a generate-and-test approach to tabulation which would evaluate the expression for all possible assignments to every variable in the expression.

For example, consider the knight's move constraint of the Knight's Tour *sequence* model with $n = 4$. Table 3 shows part of the depth-first search to tabulate the constraint between `tour[7]` and `tour[8]`, the first pair of variables with domain $\{1 \dots 15\}$ (i.e. all values except

x	y	Simplified expression
—	—	`(|x%4-y%4|=1 /\ |x/4-y/4|=2) \/ (|x%4-y%4|=2 /\ |x/4-y/4|=1)`
1	—	`(|1-y%4|=1 /\ y/4=2) \/ (|1-y%4|=2 /\ y/4=1)`
1	1	`false`
	⋮	
1	6	`false`
1	7	`true` (tuple $\langle 1, 7\rangle$ added to table)
1	8	`true` (tuple $\langle 1, 8\rangle$ added to table)
1	9	`false`
	⋮	

Table 3: A fragment of the depth-first search to tabulate the knight's move constraint between variables `tour[7]` (denoted by `x`) and `tour[8]` (denoted by `y`) of the Knight's Tour *sequence* model, with $n = 4$ and starting location $(0, 0)$.

the starting position of 0). Once the first variable is assigned to 1, the expression becomes much shorter and simpler. When both variables are assigned, it evaluates to either `true` or `false`. The first two tuples to be added to the table are $\langle 1, 7\rangle$ and $\langle 1, 8\rangle$. The next would be $\langle 1, 10\rangle$. Tables are generated with tuples in lexicographic order, and with columns in the order of the variable list.

Appendix C. Detailed Experimental Results

In this appendix we provide more detailed analysis including confidence intervals and statistical significance results as described in Section 6.2. Some problem classes have 10 or fewer instances, in which case we report the speedup quotient for each instance instead of the 95% confidence interval.

C.1 Baseline Problems

C.1.1 Black Hole

TabID can speed up all six solvers. The solvers without clause learning show very substantial speedups with TabID and the effect is clearly significant. For example, Gecode has a geometric mean speedup of 16.00 with 95% confidence interval $[10.59, 24.01]$. Minion exhibits a small overhead, and some easy instances are slower with TabID than without, but on average there is a large speedup with 95% confidence interval of $[7.17, 17.73]$. The solvers with learning performed much better on Black Hole (with or without TabID), and the average speedup is significant, though smaller: the 95% confidence intervals for Kissat, Chuffed and OR-Tools are $[3.24, 4.01]$, $[2.09, 2.87]$, and $[4.55, 7.71]$, respectively. The two SAT solver configurations are in fact identical for this problem class because all table constraints are binary and the support encoding is used in both cases (see Section 6.1).

C.1.2 Block Party Metacube Problem

The solvers exhibit differing speed-ups, with the largest being Gecode where $s = 13.42$ with 95% confidence interval of $[11.90, 14.76]$. For Minion the speed-up is smaller but significant, with a 95% confidence interval of $[1.14, 1.42]$. We found no significant difference with Chuffed, where the 95% confidence interval is $[0.68, 2.26]$. In the results presented by Dekker et al. (2017), Chuffed performs badly without tabulation, whereas here it is solving all instances. Their non-tabulated model has additional *rotation* variables and others (see Section 5.1.2) which may explain the difference in performance. OR-Tools shows a larger (and statistically significant) improvement with confidence interval $[1.44, 6.46]$.

TabID substantially reduces performance of the SAT solver with both encodings. For this problem the sizes of the SAT encodings are increased by TabID. Adding the SAT size limit (Section 4.3) to TabID improves performance for Kissat, Kissat (MDD), and Chuffed, resulting in average speed-ups of 2.10, 1.60, and 7.60 respectively as shown in Figure 8.

C.1.3 Handball Tournament Scheduling

For this problem class there is a clear benefit for the conventional CP solvers: the 95% confidence intervals for Gecode and Minion are $[2.49, 9.66]$ and $[1.56, 6.62]$ respectively. For all other solvers, the instances are too easy to show a clear difference. Total times are around 10 seconds for every instance, both with and without TabID. The 95% confidence intervals for Kissat, Kissat (MDD), Chuffed, and OR-Tools are $[1.02, 1.16]$, $[0.89, 0.99]$, $[0.74, 0.96]$, and $[1.37, 1.62]$ respectively.

C.1.4 JP Encoding Problem

For JP Encoding the results are slightly positive for Gecode and Minion, however they can solve only two of the ten instances. Chuffed solves six instances with TabID and four without, and is somewhat faster with TabID. OR-Tools solves all instances with or without TabID, however TabID slows it down on average.

Savile Row's default SAT encoding for sums does not scale well when encoding the objective function. Instead we used the GGPW encoding (Bofill, Coll, Nightingale, Suy, Ulrich-Oltean, & Villaret, 2022) applied to the direct encoding of the variables (Nightingale, 2024). Kissat and Kissat (MDD) show the largest speed-ups of around 10 times on average. The speedup quotients for each solver on each instance (except double timeouts) are shown in the following table.

Solver	Sorted speedup quotients for tabulation
Minion	0.90, 2.18
Gecode	0.90, 1.35
Kissat	3.69, 3.82, 8.22, 10.63, 16.49, 19.18, 34.58
Kissat-MDD	4.03, 4.41, 5.84, 10.04, 16.60, 18.07, 34.04
Chuffed	1.45, 2.02, 2.06, 2.22, 2.71, 3.51
OR-Tools	0.28, 0.40, 0.42, 0.49, 0.52, 0.55, 0.81, 1.06, 1.38, 2.23

C.1.5 Maximum Density Still Life

Still Life might seem to be an obvious candidate for tabulation, however for Gecode and Minion we found that the node counts are identical for all non-timeout instances: TabID does not improve propagation. Minion is substantially slower with TabID, as are the learning solvers except Kissat (MDD). Gecode and Kissat (MDD) have approximately the same performance, as shown in Figure 7 and the following table.

Solver	Sorted speedup quotients for tabulation
Minion	0.08, 0.15, 0.26, 0.62
Gecode	1.01, 1.23, 1.27, 1.29
Kissat	0.27, 0.39, 0.68, 0.69
Kissat-MDD	0.72, 0.80, 1.14, 1.17
Chuffed	0.03, 0.04, 0.28
OR-Tools	0.01, 0.03, 0.03, 0.05, 0.05, 0.49

Adding the SAT size limit to TabID improves its performance for all solvers except Gecode, resulting in average speed-ups between 0.60 (Minion) and 1.31 (Kissat) as shown in Figure 8. Recall from Section 5.1.5 that the Identical Scopes heuristic identifies subproblems with arity 9 (for grid squares away from the inner border), arity 6 (for squares adjacent to the inner border), and arity 4 (for corner squares). The SAT size limit always prevents the arity 9 and 6 candidates being tabulated. It also prevents arity 4 candidates being tabulated when the MDD encoding of table is used. Instead, parts of the constraints are identified by Duplicate Variables (Nested) and they are tabulated with mixed results as shown in Figure 8.

C.1.6 Sports Scheduling Completion

TabID proves to be highly beneficial for the conventional CP solvers Gecode and Minion, as expected. However, with Chuffed the picture is mixed and the average speedup is very close to 1. Results with Kissat depend on the choice of encoding for the generated table constraints, with the support encoding performing much better than the MDD encoding (however, instances are too easy for firm conclusions). Finally, OR-Tools is less effective than the other learning solvers in general but TabID helps quite substantially. Plots and geometric mean speedups are shown in Figure 7, and the sorted speedup quotients for each solver on each instance (except double timeouts) are shown in the following table. Adding the SAT size limit to TabID improves Kissat (MDD) performance in particular but makes OR-Tools worse (see Figure 8).

Solver	Sorted speedup quotients for tabulation
Minion	7.64, 9.43, 22.48, 27.90, 51.09, 55.63
Gecode	11.53, 15.19, 21.79, 34.62, 40.22, 76.70
Kissat	0.63, 0.83, 1.56, 1.72, 1.73, 1.79, 1.87, 2.18, 2.23, 2.47
Kissat-MDD	0.16, 0.23, 0.24, 0.25, 0.28, 0.32, 0.33, 0.34, 0.38, 0.56
Chuffed	0.11, 0.28, 0.52, 0.59, 0.75, 1.70, 2.14, 3.00, 3.77, 4.38
OR-Tools	0.28, 2.51, 2.72, 4.64, 5.37, 7.85, 33.20, 81.58, 234.85

C.2 New Case Studies

C.2.1 Accordion Patience

For the two non-learning CP solvers, TabID speeds up solving considerably. Gecode has the largest geometric mean speedup of 56.68, with 95% confidence interval $[25.27, 126.75]$. In some cases TabID reduces search nodes by more than 1,000 times and this translates into very large speedups. For Chuffed the effect is small, the geometric mean speedup is 1.40 with 95% confidence interval $[1.14, 1.74]$. TabID seems to cause a slight slow-down for Kissat and Kissat (MDD) but neither reach significance (with 95% confidence intervals $[0.87, 1.01]$ and $[0.88, 1.03]$ respectively). OR-Tools is also somewhat slower, with confidence interval $[0.67, 0.88]$.

A preliminary experiment with a particular instance (named `cards_11_01.param`) and Minion (with a static variable and value ordering) showed that almost all the benefit comes from tabulating Move1 and Move2 constraints (described in Section 5.2.1). Without tabulation, Minion takes 644,774 nodes. With just Move1 and Move2 tabulated, Minion takes 35,760 nodes, and with TabID it takes 35,712 nodes (and total time was almost identical).

C.2.2 Coprime Sets

For the non-learning CP solvers, TabID is neutral for the smallest (and easiest) instances but starts to pay off for larger instances. The 95% confidence intervals are $[1.07, 1.98]$ for Gecode and $[1.22, 2.42]$ for Minion. In both cases some additional instances are solved with TabID. Kissat and Chuffed show very little difference overall, which we found surprising given the extensive changes that TabID makes to the model. Finally, OR-Tools benefits for easier instances but there is little difference after approximately 100 s. One additional instance is solved with TabID, and the confidence interval is $[1.36, 2.32]$.

C.2.3 Killer Sudoku

For the non-learning CP solvers, TabID makes a substantial difference. For Minion the average speedup is 12.34 with 95% confidence interval $[7.54, 20.47]$. The effect on Gecode is smaller, with geometric mean speedup of 5.20 and 95% confidence interval $[3.74, 7.28]$. However, for all the learning solvers, the geometric mean speedup is below 1 (e.g. with Kissat the average is 0.67 with 95% confidence interval $[0.63, 0.70]$). In this case TabID increases SAT encoding size quite substantially. Adding the SAT size limit to TabID prevents some of the identified subproblems being tabulated, and for all solvers it moves the geometric mean speedup closer to 1. OR-Tools benefits in particular, with geometric mean speedup of 0.97. The results with the SAT size limit are shown in Figure 8.

C.2.4 Knight's Tour Problem

For the *sequence* model, TabID causes a very clear speedup for all solvers. In this case, tabulation of instances where $n \leq 15$ produces a quite different model with no auxiliary variables and much stronger propagation. Kissat shows the least improvement, with a geometric mean speedup of 35.79 and 95% confidence interval $[14.08, 89.36]$.

With the *successor* model the results are more complex. For Minion, TabID makes little difference except for the 8 instances described in Section 5.2.4 where shift constraints are tabulated. For these 8 instances there is a speedup of more than 100 times. Gecode is slightly slower with TabID; it uses a GAC propagator by default for shift constraints so does not benefit from TabID in this case. The SAT solvers and Chuffed benefit from TabID, in terms of both time and number of instances solved. For example, Kissat has a geometric mean speedup of 8.02 with confidence interval $[3.78, 18.29]$. Finally, with OR-Tools we found that 7 instances were solved faster with TabID and it made almost no difference to the others. The 7 instances are of various sizes from 8 to 15 and have no obvious common features; only three have the shift constraints that made a large difference with Minion.

C.2.5 N-Linked Sequence and Optimal N-Linked Sequence

For the decision problem, TabID speeds up all solvers quite substantially. For example Chuffed has a geometric mean speedup of 129.38 with 95% confidence interval $[35.59, 509.32]$. Gecode and Minion struggle to solve many instances with or without TabID. All table constraints are binary so the two SAT configurations are the same (see Section 6.1). Kissat and OR-Tools are affected less than the other solvers, but both still reach statistical significance. For example, OR-Tools has confidence interval $[1.85, 13.97]$. For Kissat and OR-Tools, TabID seems to help more on the easier instances, suggesting that these solvers are able to improve the formulation of an instance over time by conflict learning.

TabID also appears to speed up all solvers for the optimization problem, but for Kissat (MDD) and OR-Tools the gain is small (and for OR-Tools it does not reach significance). Kissat and Kissat (MDD) have 95% confidence intervals of $[2.18, 8.17]$ and $[1.16, 2.10]$ respectively. Chuffed benefits the most, with 95% confidence interval $[70.02, 656.66]$. OR-Tools has confidence interval $[0.46, 3.12]$.

The table constraints generated for the optimization problem are of arity 3 and have more tuples for the same value of n. For the decision problem, with $n \in \{60, 70, 80\}$, the tables have between 462 and 656 tuples. For the optimisation problem (with $12 \leq n \leq 42$) the tables have between 202 and 2058 tuples, shifting the trade-off between TabID and the default encodings or propagators.

C.2.6 Peaceable Armies of Queens

The tabulated model propagates much better with the conventional CP solvers and they exhibit very large speedups as a result. The conflict learning solver performance is almost unchanged by TabID. Kissat and Kissat (MDD) are identical since all table constraints are binary (see Section 6.1). The sorted speedup quotients for each solver on each instance (except double timeouts) are shown in the following table.

Solver	Sorted speedup quotients for tabulation
Minion	1.19, 2.95, 43.43, 317.85, 1231.42
Gecode	0.76, 5.38, 21.12, 345.59, 591.67
Kissat	1.07, 1.11, 1.13, 1.13, 1.24, 1.27
Kissat-MDD	0.89, 0.91, 1.03, 1.04, 1.23, 1.29
Chuffed	1.01, 1.05, 1.42, 1.53, 2.23
OR-Tools	0.83, 0.86, 0.86, 0.92, 1.11

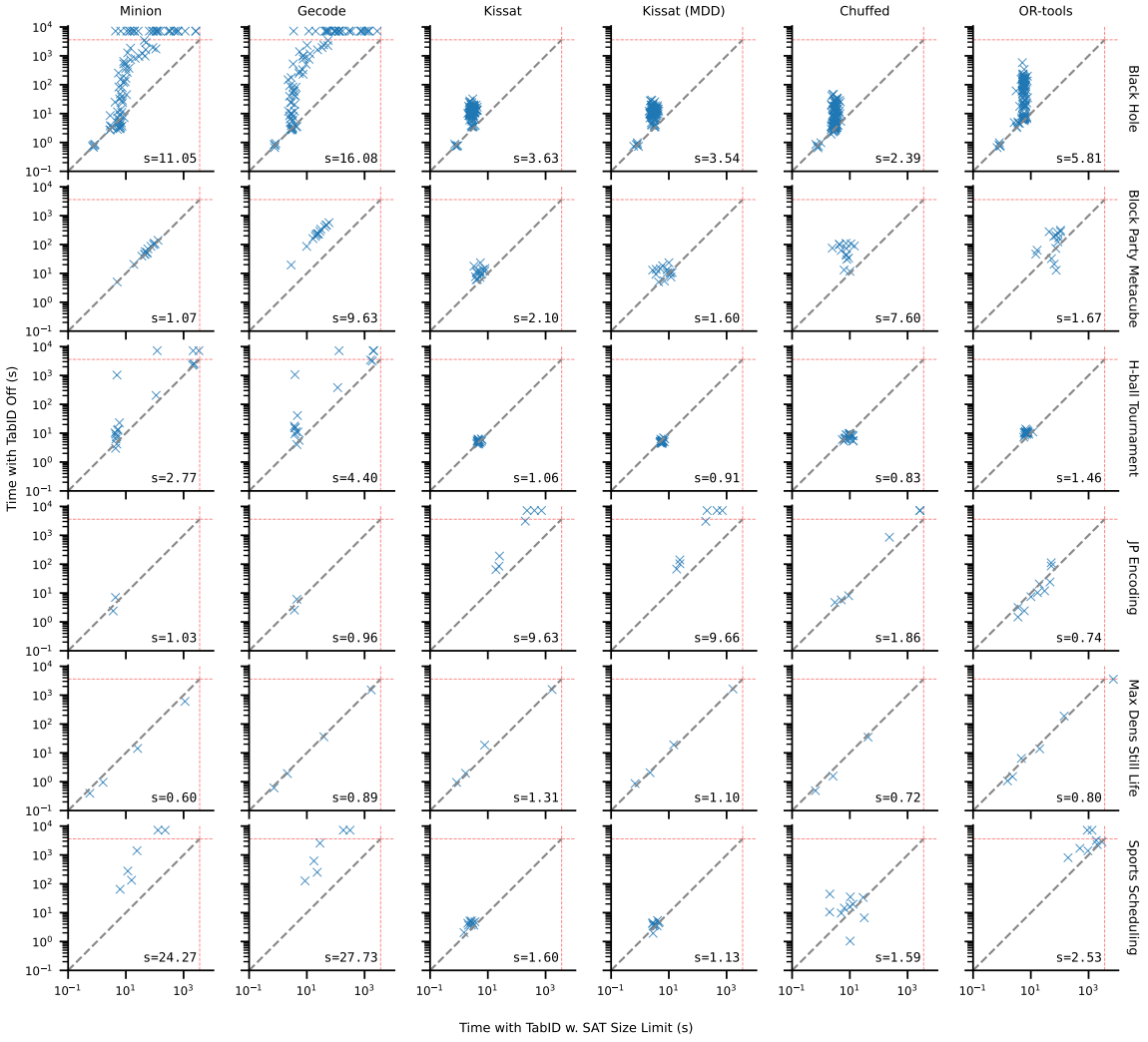

Figure 12: For each of the six problem classes: Black Hole, Block Party Metacube Problem, Handball Tournament Scheduling, JP Encoding, Maximum Density Still Life, and Sports Scheduling, and the six solvers, the figure plots the total time (including Savile Row and the solver) with TabID applying the SAT size limit on the x-axis and without TabID on the y-axis. Other details are as in Figure 7.

C.2.7 Strong External Difference Families

The results are positive for all solvers, but particularly for Gecode, Minion, and OR-Tools on larger instances. For example, with Gecode the geometric mean speedup is 5.63 with 95% confidence interval $[2.29, 14.16]$, and a peak speedup of 65 times. Gecode, Minion, Kissat, and Kissat (MDD) all have no time-outs, with and without TabID. Chuffed solves

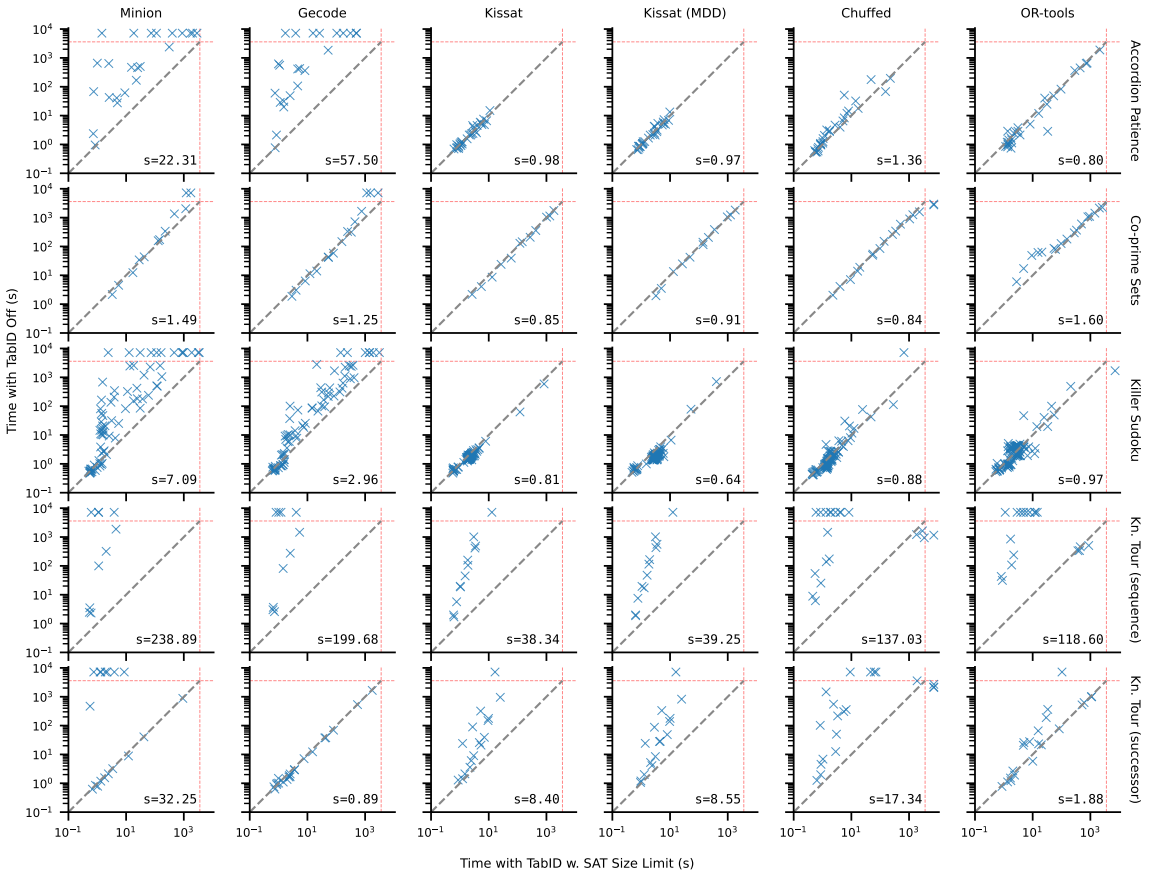

Figure 13: For each of the four problem classes: Accordion Patience, Coprime Sets, Killer Sudoku, and Knight's Tour (with two models), and the six solvers, the figure plots the total time (including both Savile Row and the solver) with TabID applying the SAT size limit on the x-axis and without TabID on the y-axis. Other details are the same as in Figure 7.

all instances with TabID, but times out on 1 instance without TabID. OR-Tools is similar but times out on 3 instances without TabID. The 95% confidence intervals for solvers Chuffed, Kissat, and Kissat (MDD) are $[1.38, 3.16]$, $[1.62, 2.54]$, and $[1.44, 2.64]$ respectively so all reach statistical significance. In SEDF all the tabulated expressions are of one kind: they are integer expressions within a `gcc` constraint. SEDF demonstrates that large speedups are possible even when no top-level constraints are tabulated.

Appendix D. Results with SAT Size Limit

In Section 4.3 we describe the option to apply a *SAT size limit* which avoids generating a table for a candidate expression if the resulting SAT encoding would be too large. In this

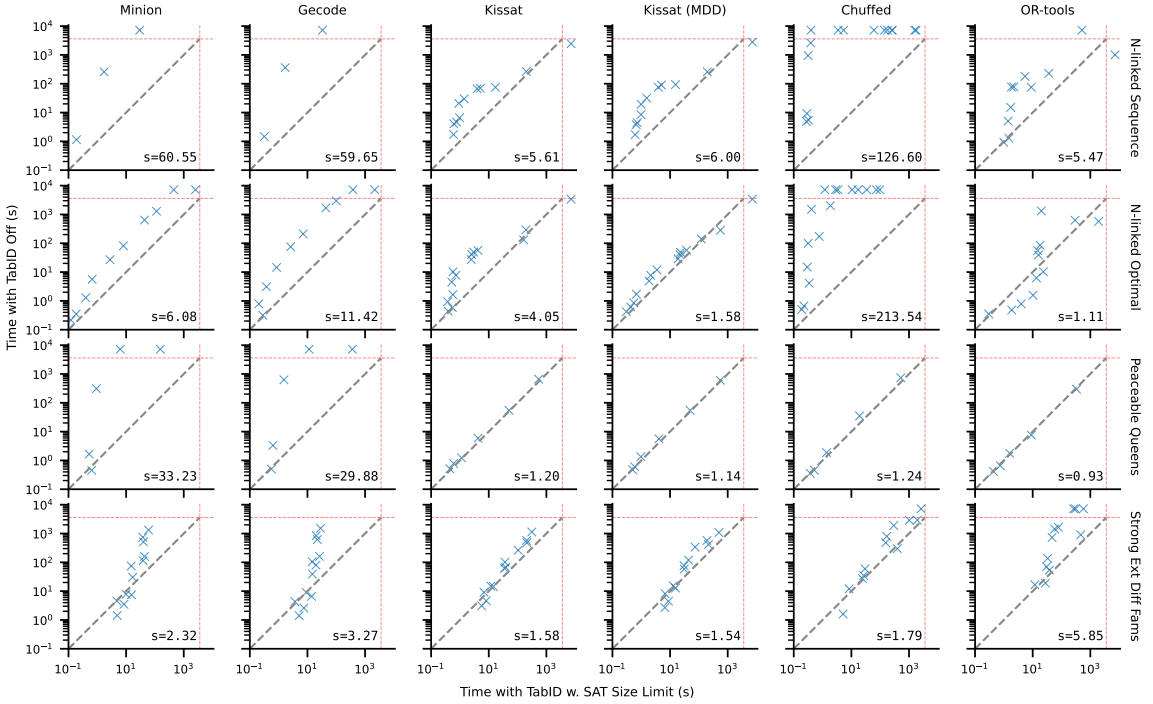

Figure 14: For each of the four problem classes: (Optimal) N-Linked Sequence, Peaceable Armies of Queens, and Strong External Difference Families (SEDF), and the six solvers, the figure plots the total time (including both Savile Row and the solver) with TabID applying the SAT size limit on the x-axis and without TabID on the y-axis. Other details are the same as in Figure 7.

appendix we present the results of solving problems without TabID versus TabID with the SAT size limit applied; Figures 12 to 14 show the comparison for the baseline problems as well as the new case studies (cross-reference with the results presented and analysed in Sections 6.3 and 6.4).

In Figure 12 we observe that the speedups with the SAT size limit applied is broadly similar for Minion, Gecode and OR-tools. However, when we consider the results with Kissat, Kissat(MDD) and Chuffed, we see a marked improvement for the Block Party Metacube problem with almost all instances being solved faster than when TabID is turned off, and with the geometric mean of speedups rising from (0.52, 0.19, 1.29) to (2.10, 1.60, 7.60). There are also improvements for the Maximum Density Still Life problem with the same three solvers.

Figure 13 shows the results for the first batch of new case studies. Here, the application of the SAT size limit does not change the results considerably in most cases. The exception is the Killer Sudoku problem, where we see that with the SAT size limit there is a deterioration in performance with Minion and Gecode, but a big improvement with OR-tools.

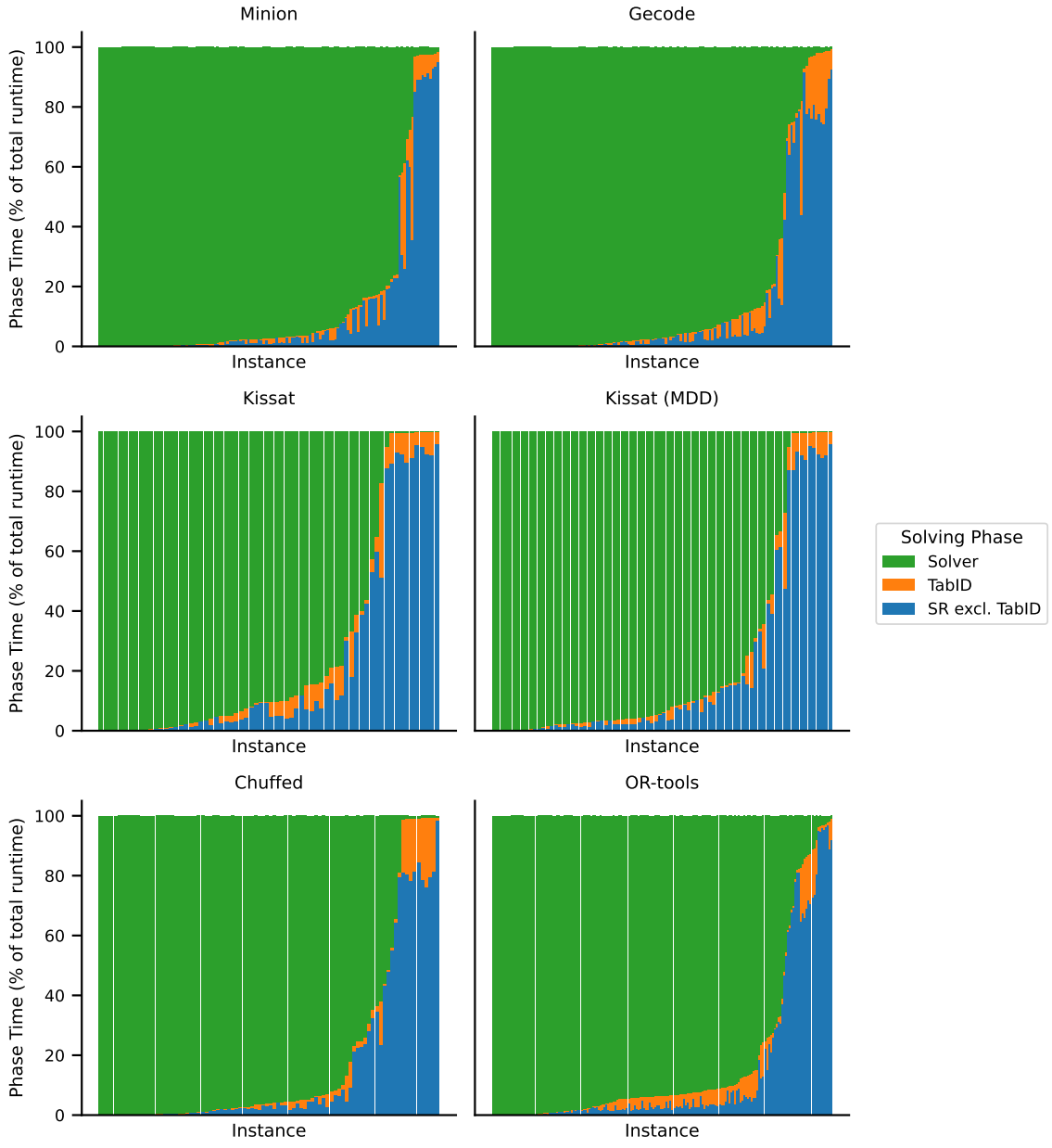

Figure 15: The time spent on the different phases of the solving process. Instances are ordered by total Savile Row time along the horizontal axis; for each instance the runtime is split into three phases displayed vertically: firstly Savile Row's pre-processing excluding TabID, secondly the time spent on TabID, and finally the time taken by the target solver. We show all instances from the featured problem classes in Section 6 whose total runtime is between 10 and 3600 seconds.

Finally in Figure 14 we have the results for the second batch of new case studies. Once again the results here are very similar to the ones presented in Section 6.4. The SAT size limit does have a slight negative impact on the Strong External Difference Families problem — in this case some speedups were achieved by TabID even though the SAT encoding size of some of the tables are over the threshold imposed by the SAT size limit.

Appendix E. Overhead of TabID

We present a brief analysis of the time taken by TabID as compared to the other phases in the solving pipeline. The SAT size limit is switched off and *nodeLimit* is set to 100,000 for this experiment. Figure 15 shows how the total runtime is divided between Savile Row and the target solver. The time spent in Savile Row is further split into two phases: TabID and everything else. The middle TabID portion (shown in orange) is relatively small in the vast majority of cases – sometimes it can take over half of the Savile Row time, but in those cases much longer is spent in the backend solver phase.

In most cases the majority of time is spent in the final solving phase carried out by the backend solver. However, there are several instances where considerable time is spent in Savile Row to reformulate the problem ready for the target solver which then solves it almost instantly.

References

Abío, I., Gange, G., Mayer-Eichberger, V., & Stuckey, P. J. (2016). On CNF encodings of decision diagrams. In *Proceedings of the 13th International Conference on AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR 2016)*, Vol. 9676 of *LNCS*, pp. 1–17. Springer. https://doi.org/10.1007/978-3-319-33954-2_1.

Ainley, S. (1977). *Mathematical Puzzles*. G. Bell & Sons.

Akgün, O., Gent, I. P., Jefferson, C., Kiziltan, Z., Miguel, I., Nightingale, P., Salamon, A. Z., & Ulrich-Oltean, F. (2024). Experimental data for TabID. University of York Research Database. https://doi.org/10.15124/c24d61e6-ee25-40ed-828d-b941158f862f.

Akgün, O., Gent, I. P., Jefferson, C., Miguel, I., & Nightingale, P. (2016). Exploiting short supports for improved encoding of arbitrary constraints into SAT. In *Proceedings of the 22nd International Conference on Principles and Practice of Constraint Programming (CP 2016)*, pp. 3–12. https://doi.org/10.1007/978-3-319-44953-1_1.

Akgün, O., Gent, I. P., Jefferson, C., Miguel, I., Nightingale, P., & Salamon, A. Z. (2018). Automatic discovery and exploitation of promising subproblems for tabulation. In *Proceedings of the 24th International Conference on Principles and Practice of Constraint Programming (CP 2018)*, pp. 3–12. https://doi.org/10.1007/978-3-319-98334-9_1.

Audemard, G., Lecoutre, C., & Maamar, M. (2020). Segmented tables: An efficient modeling tool for constraint reasoning. In *Proceedings of the 24th European Conference on*

Artificial Intelligence (ECAI 2020), pp. 315–322. https://doi.org/10.3233/FAIA200108.

Bacchus, F. (2007). GAC via unit propagation. In *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming (CP 2007)*, Vol. 4741 of *LNCS*, pp. 133–147. Springer. https://doi.org/10.1007/978-3-540-74970-7_12.

Bessiere, C. (2006). Constraint propagation. In *Handbook of Constraint Programming*, pp. 29–83. Elsevier. https://doi.org/10.1016/S1574-6526(06)80007-6.

Bessière, C., Hebrard, E., Hnich, B., & Walsh, T. (2007). The complexity of reasoning with global constraints. *Constraints*, *12*(2), pp. 239–259. https://doi.org/10.1007/s10601-006-9007-3.

Biere, A., & Fleury, M. (2022). Gimsatul, IsaSAT and Kissat entering the SAT Competition 2022. In *Proc. of SAT Competition 2022 – Solver and Benchmark Descriptions*, Vol. B-2022-1 of *Department of Computer Science Series of Publications B*, pp. 10–11. University of Helsinki. https://hdl.handle.net/10138/359079.

Bofill, M., Coll, J., Nightingale, P., Suy, J., Ulrich-Oltean, F., & Villaret, M. (2022). SAT encodings for Pseudo-Boolean constraints together with at-most-one constraints. *Artificial Intelligence*, *302*(103604). https://doi.org/10.1016/j.artint.2021.103604.

Bosch, R., & Trick, M. (2004). Constraint programming and hybrid formulations for three life designs. *Annals of Operations Research*, *130*, p. 41–56. https://doi.org/10.1023/B:ANOR.0000032569.86938.2f.

BVS Development Corporation (2017). Accordion solitaire. (Archive as of 31 Jan 2017). https://web.archive.org/web/20170131204634/https://www.bvssolitaire.com/rules/accordion.htm.

Cena, C., Akgün, O., Kiziltan, Z., Miguel, I., Nightingale, P., & Ulrich-Oltean, F. (2023). Learning when to use automatic tabulation in constraint model reformulation. In *Proceedings of the Thirty-Second International Joint Conference on Artificial Intelligence (IJCAI 2023)*, pp. 1902–1910. Main Track. https://doi.org/10.24963/ijcai.2023/211.

Chu, G., & Stuckey, P. J. (2012). A complete solution to the maximum density still life problem. *Artificial Intelligence*, *184–185*, pp. 1–16. https://doi.org/10.1016/j.artint.2012.02.001.

Chu, G., Stuckey, P. J., Schutt, A., Ehlers, T., Gange, G., & Francis, K. (2018). Chuffed. Github repository. https://github.com/chuffed/chuffed/.

de Uña, D., Gange, G., Schachte, P., & Stuckey, P. J. (2018). Compiling CP subproblems to MDDs and d-DNNFs. *Constraints*, *24*(1), pp. 56–93. https://doi.org/10.1007/s10601-018-9297-2.

Dekker, J. J., Björdal, G., Carlsson, M., Flener, P., & Monette, J.-N. (2017). Auto-tabling for subproblem presolving in MiniZinc. *Constraints*, *22*(4), pp. 512–529. https://doi.org/10.1007/s10601-017-9270-5.

Demeulenaere, J., Hartert, R., Lecoutre, C., Perez, G., Perron, L., Régin, J.-C., & Schaus, P. (2016). Compact-table: efficiently filtering table constraints with reversible sparse bit-sets. In *Proceedings of the 22nd International Conference on Principles and Practice of Constraint Programming (CP 2016)*, Vol. 9892 of *LNCS*, pp. 207–223. Springer. https://doi.org/10.1007/978-3-319-44953-1_14.

Ek, A. (2018). Automatic predicate encapsulation of potentially profitably presolvable sub-models in MiniZinc. Master's thesis, Uppsala Universitet. https://urn.kb.se/resolve?urn=urn%3Anbn%3Ase%3Auu%3Adiva-351965.

Erdös, P., & Sárközy, A. (1993). On sets of coprime integers in intervals. *Hardy-Ramanujan Journal*, *16*, pp. 1–20. https://hal.archives-ouvertes.fr/hal-01108688.

Flener, P., Frisch, A. M., Hnich, B., Kiziltan, Z., Miguel, I., Pearson, J., & Walsh, T. (2002). Breaking row and column symmetries in matrix models. In *Proceedings of the 8th International Conference on Principles and Practice of Constraint Programming (CP 2002)*, Vol. 2470 of *LNCS*, pp. 462–476. Springer. https://doi.org/10.1007/3-540-46135-3_31.

Gargani, A., & Refalo, P. (2007). An efficient model and strategy for the steel mill slab design problem. In *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming (CP 2007)*, Vol. 4741 of *LNCS*, pp. 77–89. Springer. https://doi.org/10.1007/978-3-540-74970-7_8.

Gasarch, W. (2019). Open problems column. *SIGACT News*, *50*(4), pp. 26–30. https://doi.org/10.1145/3374857.3374863.

Gecode Team (2024). Gecode: Generic constraint development environment. Accessed 25 Jul 2024. https://www.gecode.org/.

Gent, I. P. (2002). Arc consistency in SAT. In *Proceedings of the 15th European Conference on Artificial Intelligence (ECAI 2002)*, pp. 121–125. IOS Press. https://frontiersinai.com/ecai/ecai2002/pdf/p0121.pdf.

Gent, I. P., Jefferson, C., Kelsey, T., Lynce, I., Miguel, I., Nightingale, P., Smith, B. M., & Tarim, S. A. (2007). Search in the patience game 'Black Hole'. *AI Communications*, *20*(3), pp. 211–226. https://content.iospress.com/articles/ai-communications/aic405.

Gent, I. P., Jefferson, C., & Miguel, I. (2006). Minion: A fast scalable constraint solver. In *Proceedings of the 17th European Conference on Artificial Intelligence (ECAI 2006)*, pp. 98–102. IOS Press. http://ebooks.iospress.nl/volumearticle/2658.

Gent, I. P., Jefferson, C., Miguel, I., & Nightingale, P. (2007). Data structures for generalised arc consistency for extensional constraints. In *Proceedings of AAAI 2007*, pp. 191–197. AAAI Press. https://www.aaai.org/Papers/AAAI/2007/AAAI07-029.pdf.

Huczynska, S., Jefferson, C., & Nepšinská, S. (2021). Strong external difference families in abelian and non-abelian groups. *Cryptography and Communications*, *13*(2), pp. 331–341. https://doi.org/10.1007/s12095-021-00473-3.

IBM (2024). CP Optimizer file format reference manual: strong. ILOG CPLEX Optimization Studio 22.1.1 Documentation. https://www.ibm.com/docs/en/icos/22.1.1?topic=f-strong.

Ingmar, L., & Schulte, C. (2018). Making compact-table compact. In *Proceedings of the 24th International Conference on Principles and Practice of Constraint Programming (CP 2018)*, Vol. 11008 of *LNCS*, pp. 210–218. Springer. https://doi.org/10.1007/978-3-319-98334-9_14.

Jefferson, C., & Nightingale, P. (2013). Extending simple tabular reduction with short supports. In *Proceedings of 23nd International Joint Conference on Artificial Intelligence (IJCAI 2013)*, pp. 573–579. https://www.ijcai.org/Abstract/13/092.

Katsirelos, G., & Walsh, T. (2007). A compression algorithm for large arity extensional constraints. In *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming (CP 2007)*, Vol. 4741 of *LNCS*, pp. 379–393. Springer. https://doi.org/10.1007/978-3-540-74970-7_28.

Knuth, D. E. (1975). Estimating the efficiency of backtrack programs. *Mathematics of Computation*, *29*, pp. 122–136. https://doi.org/10.1090/S0025-5718-1975-0373371-6.

Larson, J., Johansson, M., & Carlsson, M. (2014). An integrated constraint programming approach to scheduling sports leagues with divisional and round-robin tournaments. In *Proceedings of the 11th International Conference on Integration of AI and OR Techniques in Constraint Programming (CPAIOR 2014)*, Vol. 8451 of *LNCS*, pp. 144–158. Springer. https://doi.org/10.1007/978-3-319-07046-9_11.

Le Charlier, B., Khong, M. T., Lecoutre, C., & Deville, Y. (2017). Automatic synthesis of smart table constraints by abstraction of table constraints. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence (IJCAI 2018)*, pp. 681–687. https://doi.org/10.24963/ijcai.2017/95.

Le Provost, T., & Wallace, M. (1992). Domain independent propagation. In *Proceedings of FGCS: International Conference on Fifth Generation Computer Systems*, pp. 1004–1011. IOS Press. http://eclipseclp.org/reports/corefgcs.pdf.

Lecoutre, C. (2011). STR2: optimized simple tabular reduction for table constraints. *Constraints*, *16*(4), pp. 341–371. https://doi.org/10.1007/s10601-011-9107-6.

Leo, K., Mears, C., Tack, G., & de la Banda, M. G. (2013). Globalizing constraint models. In *Proceedings of the 19th International Conference of the Principles and Practice of Constraint Programming (CP 2013)*, Vol. 8124 of *LNCS*, pp. 432–447. Springer. https://doi.org/10.1007/978-3-642-40627-0_34.

Leo, K., & Tack, G. (2015). Multi-pass high-level presolving. In *Proceedings of the 24th International Joint Conference on Artificial Intelligence (IJCAI 2015)*, pp. 346–352. https://www.ijcai.org/Abstract/15/055.

Löffler, S., Liu, K., & Hofstedt, P. (2020). The regularization of small sub-constraint satisfaction problems. In *Declarative Programming and Knowledge Management:*

INAP 2019, WLP 2019, WFLP 2019, Vol. 12057 of *LNCS*, pp. 106–115. Springer. https://doi.org/10.1007/978-3-030-46714-2_8.

Löffler, S., Becker, I., & Hofstedt, P. (2021). Ml-based decision support for csp modelling with regular membership and table constraints. In *Proceedings of the 13th International Conference on Agents and Artificial Intelligence - Volume 2: ICAART*, pp. 974–981. INSTICC, SciTePress. https://doi.org/10.5220/0010299109740981.

Mairy, J.-B., Deville, Y., & Lecoutre, C. (2015). The smart table constraint. In *Proceedings of the 12th International Conference on Integration of AI and OR Techniques in Constraint Programming (CPAIOR 2015)*, Vol. 9075 of *LNCS*, pp. 271–287. Springer. https://doi.org/10.1007/978-3-319-18008-3_19.

Mohr, R., & Masini, G. (1988). Good old discrete relaxation. In *Proceedings of ECAI 1988*, pp. 651–656. Pitman Publishing.

Nethercote, N., Stuckey, P. J., Becket, R., Brand, S., Duck, G. J., & Tack, G. (2007). MiniZinc: Towards a standard CP modelling language. In *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming (CP 2007)*, Vol. 4741 of *LNCS*, pp. 529–543. Springer. https://doi.org/10.1007/978-3-540-74970-7_38.

Nightingale, P. (2015). CSPLib problem 057: Killer sudoku. CSPLib: A problem library for constraints. https://www.csplib.org/Problems/prob057.

Nightingale, P. (2018). CSPLib problem 081: Black hole. CSPLib: A problem library for constraints. https://www.csplib.org/Problems/prob081/.

Nightingale, P. (2024). Savile Row manual. arXiv:2201.03472. https://doi.org/10.48550/arXiv.2201.03472.

Nightingale, P., Akgün, O., Gent, I. P., Jefferson, C., Miguel, I., & Spracklen, P. (2017). Automatically improving constraint models in Savile Row. *Artificial Intelligence*, *251*, pp. 35–61. https://doi.org/10.1016/j.artint.2017.07.001.

Nightingale, P., Gent, I. P., Jefferson, C., & Miguel, I. (2011). Exploiting short supports for generalised arc consistency for arbitrary constraints. In *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence (IJCAI 2011)*, pp. 623–628. https://www.ijcai.org/Abstract/11/111.

Nightingale, P., Gent, I. P., Jefferson, C., & Miguel, I. (2013). Short and long supports for constraint propagation. *Journal of Artificial Intelligence Research*, *46*, pp. 1–45. https://doi.org/10.1613/jair.3749.

Paterson, M. B., & Stinson, D. R. (2016). Combinatorial characterizations of algebraic manipulation detection codes involving generalized difference families. *Discrete Mathematics*, *339*(12), pp. 2891–2906. https://doi.org/10.1016/j.disc.2016.06.004.

Perron, L., & Didier, F. (2023). OR-Tools (CP-SAT v 9.7.2996).. https://developers.google.com/optimization/cp/cp_solver/.

Régin, J.-C. (1994). A filtering algorithm for constraints of difference in CSPs. In *Proceedings of AAAI 1994*, pp. 362–367. https://www.aaai.org/Papers/AAAI/1994/AAAI94-055.pdf.

Régin, J.-C. (1996). Generalized arc consistency for global cardinality constraint. In *Proceedings of AAAI 1996*, pp. 209–215. AAAI Press. https://www.aaai.org/Papers/AAAI/1996/AAAI96-031.pdf.

Régin, J.-C. (2011). Improving the expressiveness of table constraints. In *The 10th International Workshop on Constraint Modelling and Reformulation (ModRef 2011)*. http://www.constraint-programming.com/people/regin/papers/expressiveness.pdf.

Rendl, A., Miguel, I., Gent, I. P., & Jefferson, C. (2009). Automatically enhancing constraint model instances during tailoring. In *Eighth Symposium on Abstraction, Reformulation, and Approximation (SARA 2009)*, pp. 120–127. https://webdocs.cs.ualberta.ca/~nathanst/sara/papers/sara09_submission_35.pdf.

Roeder, O. (2017a). Is this bathroom occupied?. The Riddler. https://web.archive.org/web/20170804162423/https://fivethirtyeight.com/features/is-this-bathroom-occupied/.

Roeder, O. (2017b). Pick a number, any number. The Riddler. https://web.archive.org/web/20170728203913/https://fivethirtyeight.com/features/pick-a-number-any-number/.

Ross, K. A., & Knuth, D. E. (1989). A programming and problem solving seminar. Tech. rep. STAN-CS-89-1269, Stanford University, Stanford, CA, USA. http://infolab.stanford.edu/pub/cstr/reports/cs/tr/89/1269/CS-TR-89-1269.pdf.

Rossi, F., van Beek, P., & Walsh, T. (Eds.). (2006). *Handbook of Constraint Programming*. Elsevier.

Schneider, A., & Choueiry, B. Y. (2018). PW-CT: Extending compact-table to enforce pairwise consistency on table constraints. In *Proceedings of the 24th International Conference on Principles and Practice of Constraint Programming (CP 2018)*, Vol. 11008 of *LNCS*. Springer. https://doi.org/10.1007/978-3-319-98334-9_23.

Schwenk, A. J. (1991). Which rectangular chessboards have a knight's tour?. *Mathematics Magazine*, *64*(5), pp. 325–332. https://doi.org/10.1080/0025570X.1991.11977627.

Smith, B. (2015). CSPLib problem 032: Maximum density still life. CSPLib: A problem library for constraints. https://www.csplib.org/Problems/prob032.

Smith, B. M. (2002). A dual graph translation of a problem in 'Life'. In *Proceedings of the 8th International Conference on Principles and Practice of Constraint Programming (CP 2002)*, Vol. 2470 of *LNCS*, pp. 402–414. Springer. https://doi.org/10.1007/3-540-46135-3_27.

Smith, B. M., Petrie, K. E., & Gent, I. P. (2004). Models and symmetry breaking for 'peaceable armies of queens'. In *Proceedings of the First International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR 2004)*, Vol. 3011 of *LNCS*, pp. 271–286. Springer. https://doi.org/10.1007/978-3-540-24664-0_19.

Van Hentenryck, P., Michel, L., Perron, L., & Régin, J. (1999). Constraint programming in OPL. In *Proceedings of the International Conference on Principles and Practice of Declarative Programming (PPDP 1999)*, Vol. 1702 of *LNCS*, pp. 98–116. Springer. https://doi.org/10.1007/10704567_6.

Van Hentenryck, P. (1999). *The OPL Optimization Programming Language*. MIT Press, Cambridge, MA, USA.

Wang, R., & Yap, R. H. C. (2020). Bipartite encoding: A new binary encoding for solving non-binary CSPs. In *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence*, IJCAI'20, pp. 1184–1191, Yokohama, Yokohama, Japan. https://doi.org/10.24963/ijcai.2020/165.

Wang, R., & Yap, R. H. C. (2022). CNF Encodings of Binary Constraint Trees. In Solnon, C. (Ed.), *28th International Conference on Principles and Practice of Constraint Programming (CP 2022)*, Vol. 235 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pp. 40:1–40:19, Dagstuhl, Germany. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. https://doi.org/10.4230/LIPIcs.CP.2022.40.

Wang, R., & Yap, R. H. C. (2023). The Expressive Power of Ad-Hoc Constraints for Modelling CSPs. *Proceedings of the AAAI Conference on Artificial Intelligence*, *37*(4), pp. 4104–4114. https://doi.org/10.1609/aaai.v37i4.25526.

Xia, W., & Yap, R. H. (2013). Optimizing STR algorithms with tuple compression. In *Proceedings of the 19th International Conference on Principles and Practice of Constraint Programming (CP 2013)*, Vol. 8124 of *LNCS*, pp. 724–732. Springer. https://doi.org/10.1007/978-3-642-40627-0_53.