# Proof Logging for the Circuit Constraint

Matthew J. McIlree[1][0009−0005−5042−0876], Ciaran
McCreesh[1][0000−0002−6106−4871], and Jakob Nordström[2][0000−0002−2700−4285]

[1] University of Glasgow, Scotland
[2] University of Copenhagen, Denmark and Lund University, Sweden

**Abstract.** Proof logging in constraint programming is an approach to
certifying a conclusion reached by a solver. To allow for this, different
propagators must be augmented to produce justifications for any infer-
ences they make, so that an independent proof checker can certify cor-
rectness. The Circuit constraint is used to enforce a Hamiltonian cycle on
a set of vertices, e.g. for vehicle routing. Maintaining consistency for the
Circuit constraint is hard, so various ad-hoc propagation techniques have
been devised and implemented in solvers. We show that standard Circuit
constraint inference rules can be efficiently justified within a pseudo-
Boolean proof system, either by using a simple sequence of cutting planes
steps or through a conditional counting argument.

**Keywords:** Proof logging · Circuit · Constraint propagation.

## 1 Introduction

A constraint programming (CP) solver that implements *proof logging* is able to
provide a strong correctness guarantee for every result it produces. Alongside any
answer, it outputs a formal proof that rigorously demonstrates that the answer
is correct. This is already standard practice in the field of Boolean satisfiability
(SAT) solving [8, 32], and will, we believe, be crucial for the acceptance of CP in
safety-critical applications. Proof logging is achieved by having a solver output
justifications for all its reasoning steps in the language of a sound and complete
proof system. Previous work [10, 16, 25] has shown that it is possible to do this
efficiently for many important constraint propagation algorithms, by using a
*pseudo-Boolean* (PB) proof system that is based on *cutting planes* [3, 7] along
with further strengthening rules. These proofs are written in a machine-readable
format that can be independently verified using the *VeriPB* proof checker [2].

A common feature of all propagation algorithms that have been considered
before is that they enforce a strong level of local consistency among the vari-
ables in scope. For example, AllDifferent [10], SmartTable and Regular [25] enforce
domain consistency (DC), while LinearEquality [16] enforces bounds consistency
(BC). To show a proof logging procedure for these constraints is comprehensive,
it is sufficient to establish that any DC or BC inference can be justified, since this
demonstrates that certification is possible regardless of how that inference is ac-
tually computed. The same approach is no longer viable when dealing with more

(a) Valid assignment.          (b) Invalid assignment.          (c) Partial assignment.
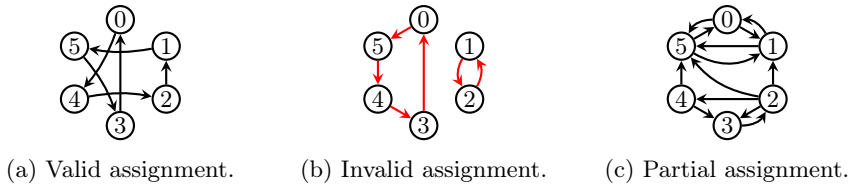
Figure 1: Interpretation of assignments for six variables constrained by Circuit.

complex propagators that have less clearly defined notions of consistency, as it is harder to capture clearly in advance what propagations are to be expected. If pseudo-Boolean proof logging is to be applicable to CP in general, it is important to show that these types of propagators do not present any fundamental barriers to its adoption.

In this paper we present proof logging for propagation of the (Hamiltonian) Circuit constraint. Enforcing domain consistency for Circuit is known to be NP-hard [19] and so it is generally propagated via ad-hoc propagation rules [4,12,30]. We therefore work from the simplest checking and basic lookahead inferences, up to more advanced propagation techniques based on depth-first search and identification of strongly connected components. In each case we briefly outline the situation in which the inference applies, and find that it can be justified either by a simple sequence of cutting planes steps, or through a conditional counting argument. The latter consists of identifying a vertex which cannot reach every other vertex under some conditions, and deriving PB constraints over auxiliary variables that establish the set of reachable vertices is too small. We have implemented and tested these techniques, building a complete certifying Circuit propagator comparable in propagation strength to well known open-source CP solver implementations [6,14,21], and have been able to produce and verify proofs using this for a variety of instance sizes.

## 2   Preliminaries

The Circuit constraint uses a successor representation to treat a set of variables $X_0 \ldots X_{n-1}$, each with domain $\{0, \ldots, n-1\}$ as the vertices of a directed graph. At any stage in the solving process, an edge $(i, j)$ is viewed as being present in the graph if and only if $j$ is still in the domain of the variable $X_i$. Circuit requires that any assignment represents a *Hamiltonian cycle*, with the value of $X_i$ representing the successor of $i$ in a tour that visits all vertices; see Figure 1. This is useful for modelling problems such as vehicle routing [22, 23], activity scheduling [9] and other graph problems [4, 13]. In CP solvers, propagation for a global Circuit constraint is generally achieved by first at least partially propagating an AllDifferent and then attempting further propagation based on the fact that there can be no sub-cycles. At a minimum the algorithm should check whether any sub-cycles are encoded by the current partial assignment and backtrack if so [4], but further lookahead and ad-hoc propagation rules are possible [12, 30].

### 2.1   Requirements for Proof Logging

To follow the CP proof logging methodology of previous work [16], we are required to compile any CP problem to a *pseudo-Boolean* (PB) format. This is a separate model of the problem that is only used for certification and should be kept independent of the solving process. In a PB model we are only allowed 0-1 variables (PB variables), and all constraints must be integer linear inequalities (PB constraints) over *literals*, where a literal $\ell$ is a PB variable or its negation $\bar{x} = 1 - x$. We also allow *reified* PB constraints of the form $\bigwedge r \implies \sum_{i=1}^{k} a_i \ell_i \geq A$, where $\bigwedge r$ is a conjunction of literals: these are syntactic sugar for $\sum K r + \sum_{i=1}^{k} a_i \ell_i \geq A$ for $K$ chosen to be sufficiently large. We can also reify the negation of a PB constraint on the negation of each of the literals in $r$, which allows us to define $\bigwedge r \iff \sum_{i=1}^{k} a_i \ell_i \geq A$.

   We can create a PB model from a CP problem by associating each integer variable $X$ with a set of bit variables $\{x_{b0}, x_{b1}, x_{b2}, \dots\}$ sufficient to represent every value in the variable's domain using a two's complement representation. We then encode restrictions on $X$ imposed by CP constraints by adding pseudo-Boolean constraints over these bit variables. Since this process is not verified in itself, we should choose simple encodings that establish a clear correspondence between satisfying assignments of the CP and PB models. To aid this, we can employ auxiliary PB variables $x[i]_{=j}$ which are defined through reification to be true precisely when the bit representation of the variable $X_i$ evaluates to the value $j$. For example if we have $k$ bits and want to define $x[4]_{=4}$ we would have PB constraints equivalent to $x[4]_{=3} \iff \sum_{i=0}^{k-1} 2^i x[4]_{bi} = 3$. This gives us flexibility for PB encodings since we can make use of either bitwise or direct representations for variables depending on what gives us the most straightforward encoding of each CP constraint. We might also define other auxiliary variables that are not directly tied to the values of CP variables for use as flags, selectors, or counters.

   For a Circuit constraint on $n$ variables $X_0, \dots, X_{n-1}$, we already know how to achieve proof-logging for the AllDifferent component [10]. A simple PB encoding of AllDifferent consists of constraints on each distinct pair of variables $(X_l, X_r)$ that enforce either $X_l < X_r$ or $X_r < X_l$ depending on a selector bit $f_{lr}$.

   We are then left with the task of defining PB constraints that encode the elimination of subcycles. For this we can take inspiration from known SAT encodings, since a logical clause such as $x \vee \bar{y} \vee \bar{z}$ is always equivalent to a PB constraint $(x + y + \bar{z} \geq 1)$, and so PB formulas can be viewed as a superset of conjunctive normal form. There are many possible options for such encodings [17], with different trade-offs, but since our chosen encoding will only be used for certification and not solving, compactness and obvious correctness is much more important than strong propagation properties. We make use of a PB encoding that is a simplified version of the SAT encoding given by Zhou [33, Sec. 4.2], and first define an additional set of auxiliary bit variables $\{p[i]_{b0}, p[i]_{b1}, \dots\}$ for each variable $X_i$, sufficient to represent the range of integers $0 \dots n-1$. We will use $P_i$ as a shorthand for the sum $\sum_j 2^j p[i]_{bj}$, and conceptually treat $P_i$ as a variable in itself, encoded with a sequence of PB bit variables. These bits are then constrained to represent the "position" of $X_i$ in the circuit relative to

---

**Algorithm 1** Procedure for constructing a PB encoding of a subcycle elimination constraint on variable $X_0, \ldots, X_{n-1}$

---

1: **define** $P_0 = 0$;
2: **for all** $i \in \{0, \ldots, n-1\}$ and $j \in \{1, \ldots, n-1\}$
3:    |    **define** $x[i]_{=j} \implies P_j - P_i \geq 1$
4:    |_    **define** $x[i]_{=j} \implies -P_j + P_i \geq -1$
5: **for all** $i \in \{1, \ldots, n-1\}$
6:    |    **define** $x[i]_{=0} \implies P_i \geq n - 1$
7:    |_    **define** $x[i]_{=0} \implies -P_i \geq -n + 1$

---

$X_0$, which is arbitrarily designated as the start vertex. We do this as shown in Algorithm 1: define $P_0 = 0$, and then require $P_j = P_i + 1$ whenever $x[i]_{=j}$ is true, unless $j = 0$, in which case require $P_i = n - 1$. A satisfying assignment to these PB constraints is only possible when the cycle obtained by following the successors starting from $X_0$ visits every vertex. The condition $P_i = j$ can then be interpreted as encoding the fact that "the vertex represented by $X_i$ is the $j_{th}$ vertex visited after vertex 0 in the Hamiltonian cycle".

Once a valid encoding has been produced, a proof logging CP solver can justify its reasoning steps by deriving further PB constraints, and recording them in a proof log file that can eventually be used certify whatever result it arrives at. Further explanation of how this can be achieved in general for a backtracking-based CP solver is given by Gocht et al. [16], but for our purposes the key idea is that whenever the solver backtracks it should be possible to derive a PB constraint that encodes the negated conjunction of the currently guessed assignments. For example, if the solver guesses $(X_0 = 2, X_1 = 3, X_2 = 1)$ and then discovers a contradiction before assigning the remaining variables it should derive the PB constraint

$$\overline{x[0]}_{=2} + \overline{x[1]}_{=3} + \overline{x[2]}_{=3} \geq 1. \tag{1}$$

As a shorthand we will sometimes denote the solver's guessed assignments by $\mathcal{G}$, and use $\bigwedge \mathcal{G}$ to denote the conjunction of pseudo-Boolean literals encoding them. So in general a backtracking justification is of the form $\bigwedge \mathcal{G} \implies 0 \geq 1$.

This is somewhat similar to the way *lazy clause generation* solvers work [27], except the "explaining" constraints do not inform the solving process, and they have to be formally derived from the model and previously derived constraints via *VeriPB's* sound and complete proof rules, rather than simply asserted. As mentioned, these rules are based on the cutting planes proof system: so we can derive linear combinations of PB constraints, divide with rounding, *saturate* constraints to minimise coefficients, and also use the axiom that any single literal is at least 0; see Buss and Nordström [3] for more details. Additionally, if any auxiliary PB variables required are not already defined in the original PB formula, *VeriPB* allows them to be introduced dynamically whenever needed as extension variables during the proof. This is an application of the *VeriPB's redundance-based strengthening* rule, and for our purposes is only needed to introduce fresh

variables reified on arbitrary constraints [2]. The backtracking justification (1) itself should be derived via a further rule: the *reverse unit propagation* (RUP) rule. This allows derivation of a PB constraint $D$ if the verifier can obtain a contradiction by iteratively enforcing bounds consistency on the negation of $D$ along with constraints in the original formula and any previously derived constraints. The iterative consistency process is the pseudo-Boolean generalisation of unit propagation from SAT, and can be performed efficiently by the verifier [15], allowing "obvious" facts to be made available when checking RUP derivations. In the example above, since (1) is a clause, the negation asserts that all the literals are false, i.e. $\overline{x[0]}_{=2} = \overline{x[1]}_{=3} = \overline{x[2]}_{=3} = 0$, and so for the RUP check to succeed we would need these assignments to trigger propagations that eventually lead the verifier to a contradiction when propagating over the constraints in the PB model along with those already derived in the proof log.

When the solver performs sophisticated reasoning via bespoke propagation algorithms, we are able to guarantee that deriving the backtracking clause in this way will be possible providing any inferences made by a CP propagator at the given level of search are also be available to the verifier via unit propagation when performing the RUP check. We can do this by ensuring that $\bigwedge \mathcal{G} \implies y \geq 1$ is in the proof log, where $y$ is a PB literal that encodes the propagator's inference. So if, under the sequence of guesses used in (1), and prior to backtracking, a Circuit propagator is able to infer say, $X_3 = 0$, we should somehow derive

$$x[0]_{=2} \wedge x[1]_{=3} \wedge x[2]_{=3} \implies x[3]_{=0} \geq 1; \tag{2}$$

$$\text{i.e.} \quad \overline{x[0]}_{=2} + \overline{x[1]}_{=3} + \overline{x[2]}_{=3} + x[3]_{=0} \geq 1. \tag{3}$$

These justifications are then interleaved with the backtracking clauses, resulting in the complete proof being essentially a description of the solver's backtracking search tree, expressed using RUP steps. What we show in this paper is that a range of standard Circuit propagation inferences can indeed be efficiently justified by deriving these intermediate pseudo-Boolean constraints.

## 3   Proof Logging for Simple Circuit Propagators

The minimum requirement for a Circuit sub-cycle elimination algorithm is that it is *checking*: it should return contradiction if a total assignment of the CP variables contains a small cycle. This requires no justification under our PB formula (as produced by Algorithm 1), since repeated bounds consistency (unit propagation) will immediately establish a contradiction. In particular, $P_0 = 0$ together with $X_0 = v_1$ will fix $P_{v_1} = 1$, and then this together with $X_{v_1} = v_2$ will fix $P_{v_2} = 2$, and so on. Since there must be a small cycle, say of length $m < n$, passing through $X_0$ (as we are assuming AllDifferent has been correctly enforced), at some point unit propagation will attempt to fix the value of $P_{v_m}$ when it has already been fixed to a smaller value, arriving at a contradiction.

A better checking propagator can also return contradiction on *partial* assignments, when they encode a small cycle. This is what Francis and Stuckey [12]

call *check*, and is a key component of the NoSubtour propagator of Pesant et al. [29] and the similar NoCycle propagator of Caseau and Laburthe [4]. Such solver reasoning does require some justification in the proof, since a small cycle encoded by the partial assignment might not set $X_0$, and the corresponding PB variables for this are required to set off the chain reaction of unit propagation and achieve the inconsistent setting of $p$ variables. To create such a justification we can use the cutting planes *addition rule* to add together all the corresponding constraints for the position variables in the cycle, allowing us to unit propagate a contradiction under the solver's guesses.

In particular, if a sequence of guesses $\mathcal{G}$ includes a small cycle of length $m < n$ not passing through 0 and consisting of vertices $(v_1, \ldots, v_m)$, we would add together each of the constraints of the form

$$x[v_i]_{=v_{i+1}} \implies P_{v_{i+1}} - P_{v_i} \geq 1 \tag{4}$$

(from line 3 in Algorithm 1) for each guessed assignment $(X_{v_i} = v_{i+1}) \in \mathcal{G}$ identified by the propagator as being part of the cycle. Recall from Section 2.1 that these reified PB constraints are actually represented as

$$K \cdot \overline{x[v_i]}_{=v_{i+1}} + P_{v_{i+1}} - P_{v_i} \geq 1 \tag{5}$$

for some sufficiently large $K$, and hence each successive addition cancels the previous $P_{v_i}$ value. This results in the constraint

$$K \cdot \overline{x[v_1]}_{=v_2} + \cdots + K \cdot \overline{x[v_m]}_{=v_1} - P_{v_1} + P_{v_2} - \ldots$$
$$\cdots - P_{v_{m-1}} + P_{v_m} - P_{v_m} + P_{v_1} \geq m, \tag{6}$$

which telescopes to

$$K \cdot \overline{x[v_1]}_{=v_2} + \cdots + K \cdot \overline{x[v_m]}_{=v_1} \geq m. \tag{7}$$

With this constraint present in the proof log, unit propagation of the small cycle in $\mathcal{G}$ will obviously lead to $0 \geq m$, a contradiction, and hence (7) is adequate to allow justification within the proof framework for the solver backtracking.

The above idea can be easily be extended to produce justifications for a basic lookahead version of the *check* propagator, called *prevent* by Francis and Stuckey [12], which is described in the literature [4, 29, 30]. This filters domains by disallowing any further assignments that would immediately complete a subcycle. So if a sequence of guesses $\mathcal{G}$ includes the encoding of a *chain* of vertices $(v_1, \ldots, v_m)$, *prevent* would remove $v_1$ from the domain of $X_{v_m}$, and this can be justified by first deriving (7) exactly as above, which then allows us to derive $\bigwedge \mathcal{G} \implies \overline{x[v_m]}_{=v_1}$ by RUP, as required.

## 4   Proof Logging for Stronger Propagation

There are several possibilities for stronger propagation for the Circuit constraint, although there is no general consensus between solvers on which forms are worthwhile in practice. This paper does not argue for one propagation strategy over any other; rather, our focus is to show that whatever propagator is chosen, it

should be feasible to implement a proof logging version of it. We will demonstrate that it is possible to provide pseudo-Boolean proof logging for Circuit propagators that make use of more complex reasoning by considering a further propagator and set of associated possible inferences. This algorithm is based on analysis of the depth-first spanning tree obtained during a search of the domain graph for *strongly connected components* (SCCs). Stuckey and Francis call it the SCC algorithm [12] and versions of it are implemented in the solvers *Gecode* [14], *Chuffed* [6], *JaCoP* [21], and *CP-SAT* [28] among others.

Let $G = (V, E)$ be a graph, and let $\mathcal{R}$ be the (directed) *reachability* relation on $G$ — for $v, w \in V, (v, w) \in \mathcal{R}$ if and only if there exists a path from $v$ to $w$. We will denote by $\text{REACH}(v)$ the set $\{w : (v, w) \in \mathcal{R}\}$, i.e. the set of all vertices in $G$ reachable from $v$. The core observation used by the SCC algorithm for Circuit propagation is that if the graph contains a Hamiltonian circuit, then it can only have a single strongly connected component, which means every vertex must be reachable from every other vertex. Thus, if we identify more than one strongly connected component in the graph induced by the current domains of variables in scope we can backtrack early, as no satisfying Circuit assignment is possible.

At any given point in the process of solving a constraint satisfaction problem involving a Circuit constraint on variables $X_0, \ldots, X_{n-1}$, let $G$ be the graph that has a directed edge $(v, w)$ whenever $w$ is still in the domain of $X_v$. To simplify the discussion of proof logging for the SCC algorithm, we will assume in what follows that if we can identify a vertex $v$ in this graph such that $|\text{REACH}(v)| < |G|$ then we can run a proving procedure `ReachTooSmall`$(v)$ that derives in the proof log a contradiction subject to the current sequence of guesses i.e. $\bigwedge \mathcal{G} \Rightarrow 0 \geq 1$. Furthermore, we will assume that if we have an additional "assumption" PB literal $\ell$ that encodes a further restriction on the graph so that $|\text{REACH}(v)| < |G|$ we can similarly run `ReachTooSmall`$(v)$ and derive $\bigwedge \mathcal{G} \wedge \ell \Rightarrow 0 \geq 1$. We will later outline in Section 4.2 and Section 4.3 how `ReachTooSmall` can construct this argument using proof steps recognised by *VeriPB*.

The SCC propagator is based on *Tarjan's algorithm* [31], which uses the fact that strongly connected components always form subtrees of a depth-first spanning forest of the graph. It initiates a depth-first search (DFS) from a chosen arbitrary vertex $v_0$, and immediately returns contradiction if any of its descendants are identified as the root of an SCC. To justify backtrack in this case we can run `ReachTooSmall`$(w)$, where $w$ is the root of the identified SCC. This will always prove contradiction as $v_0$ cannot possibly be reachable from $w$, otherwise $v_0$ would also be part of the SCC and hence $w$ would not be the SCC root.

A vertex can only be identified as the root of an SCC once all of its descendants have been visited during the DFS. So if none of $v_0$'s descendants are identified as SCC roots, it must be that all the vertices reachable from $v_0$ comprise a single SCC. In this case, either DFS has visited every vertex, in which case there is no contradiction for Circuit, or else there is some vertex not reachable from $v_0$ and the propagator returns a contradiction. The latter can clearly be justified by `ReachTooSmall`$(v_0)$.

Backtracking when the domain graph is disconnected or contains more than one SCC seems to be the most commonly implemented technique for SCC propagation, based on our examination of source code for open source solvers. Several solvers such as *Gecode* and *Chuffed* also implement further ad-hoc propagation opportunities when multiple distinct subtrees are explored below $v_0$. In each of the following cases we state a propagation rule applicable as part of the SCC algorithm and briefly indicate how `ReachTooSmall` can be used to justify these too. Figure 2 gives an illustration for each.

1. Prune any edge $(w, v_0)$ where $v_0$ is the starting vertex and $w$ is not in the *earliest* visited subtree [12]. To justify this we use $x[w]_{=v_0}$ as an assumption literal and run `ReachTooSmall`$(r)$, where $r$ is the root of a subtree visited earlier than the one containing $w$. Unit propagation of $x[w]_{=v_0}$ will force $x[w']_{=v_0} = 0$ for all $w' \neq w$ due to the encoding of AllDifferent, so the assumption excludes any edges from descendants of $r$ leading to $v_0$. Since vertices in this earlier subtree cannot have any edges leading to vertices in $w$'s subtree or later, otherwise they would have been traversed as part of the same subtree by DFS, it follows that $r$ cannot reach $w$. Hence, `ReachTooSmall`$(r)$ can be used to establish $\bigwedge \mathcal{G} \wedge x[w]_{=v_0} \Rightarrow 0 \geq 1$. See Figure 2a.

2. Prune any edge $(v_0, w)$ where $v_0$ is the starting vertex and $w$ is not in the latest visited subtree [30]. Similarly, we use $x[v_0]_{=w}$ as an assumption, and this time run `ReachTooSmall`$(w)$ to obtain a contradiction under the assumption. Since $w$ can only reach vertices in its own subtree or earlier, and $v_0$ no longer has edges to the later subtrees, it is clear than not everything can be reached from $w$. See Figure 2b.

3. Prune any edge $(v, w)$, where $w$ is $v$'s first child, and no edges from vertices in the subtree rooted at $w$ lead to vertices visited earlier in the DFS than $v$ [12]. Here we can run `ReachTooSmall`$(w)$ under the assumption $x[v]_{=w}$, since fixing the successor of $v$ to be $w$ eliminates any possibility of reaching any nodes visited earlier than $v$ from $w$. See Figure 2c.

4. Prune any edge $(v, w)$ that skips a subtree, that is, where $v$ is in the $i_{th}$ visited subtree and $w$ visited earlier than the root of the $(i-1)th$ subtree [30]. Intuitively this rule is sound because if the edge $(v, w)$ were used in the circuit we would have to visit the initial node $v_0$ between visiting $w$ and visiting the root $r$ of the $(i-1)th$ subtree, but also visit $v_0$ between visiting $r$ and visiting $v$, and both of these cannot be simultaneously true. A single assumption and `ReachTooSmall` argument is not always sufficient to justify this pruning, but our intuition *can* be encoded using two `ReachTooSmall` arguments and more complex assumptions. If $r$ is the root of the $(i-1)th$ subtree, we assume first that $r$ must be seen at some point *between* $w$ and $v_0$ (denoted as $w \prec r \prec v_0$) and then run `ReachTooSmall`$(v)$. This will derive a contradiction as every path from the subtree containing $w$ to the subtree rooted at $r$ must pass through $v_0$ and so there will be no way to reach $v_0$ without violating the assumption. We can similarly assume that $v$ must be seen between $r$ and $v_0$ ($r \prec v \prec v_0$) and establish a contradiction using `ReachTooSmall`$(v)$. Note that this establishes the negation of our two assumptions, namely that $v_0$
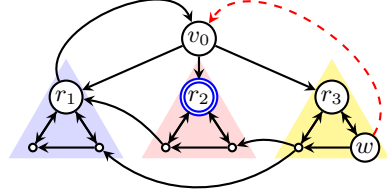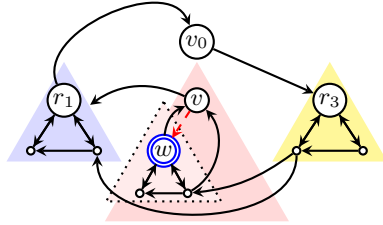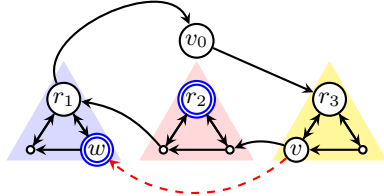
(a) Justifying the "prune skip to root" inference. If the dotted edge $(w, v_0)$ is used, $(r_1, v_0)$ is eliminated and so there is no way to reach $v_0$ from $r_2$.
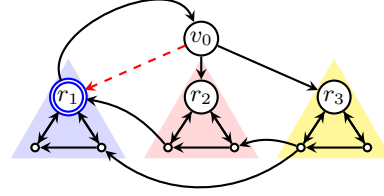
(b) Justifying the "prune root" inference. If the dotted edge $(v_0, r_1)$ is used, $(v_0, r_2)$ and $(v_0, r_3)$ are eliminated and so there is no way to reach e.g. $r_2$ from $r_1$.

(c) Justifying the "prune within" inference. If the dotted edge $(v, w)$ is used, $(v, r_1)$ is eliminated and so there is no way to reach e.g. $v_0$ from $w$.

(d) Justifying the "prune skip" inference. We can disprove the ordering assumption $w \prec r_2 \prec v_0$ with ReachTooSmall$(w)$ and disprove the ordering assumption $r_2 \prec v \prec v_0$ with ReachTooSmall$(r_2)$. These together imply that $(v, w)$ cannot be used.
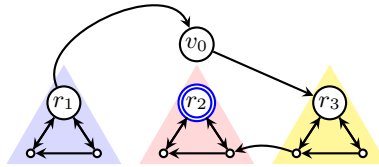
(e) Justifying "no backedges" contradiction. There are no backedges from the subtree rooted at $r_2$, and since "prune skip" inferences have already been made, there is then no way to reach any nodes earlier than $r_2$ from $r_2$.

Figure 2: Illustrations of how each SCC inference can be justified. Three distinct subtrees explored by a DFS of the domain graph starting at $v_0$ are indicated with triangles. The dashed edge is the one assumed to be used as part of the circuit (via the corresponding variable assignment), and the double ringed node is the one passed to the ReachTooSmall procedure.

must be seen both between $w$ and $r$ ($v \prec v_0 \prec r$) and between $r$ and $v$ ($r \prec v_0 \prec w$) which is impossible if $w$ is the immediate successor of $v$. So altogether, if we can encode these ordering assumptions in pseudo-Boolean form, and run `ReachTooSmall` subject to them, we should also be able to justify this pruning inference. See Figure 2d.

5. Return a contradiction if there are no *backedges* identified after exploring any subtree later than the first [30]. Backedges are edges from a node in the $i_{th}$ subtree to a node in the $(i-1)_{th}$. To justify contradiction in a case where a subtree rooted at $w$ has no backedges we can run `ReachTooSmall(w)`. Since any edges that skip subtrees have been removed at this point, by rule 3., and the only edges left leading to the initial node $v_0$ come from the earliest subtree, by rule 1., the only way to escape the subtree rooted at $w$ would be through a backedge, and so `ReachTooSmall(w)` will establish a contradiction. Similarly, if there is only a single backedge $(v, w)$ we can justify the fixing of $X_v = w$, by first assuming that it is not taken, i.e. $\overline{x[v]_{=w}}$, and running `ReachTooSmall(v)`. See Figure 2e.

These are all the inference rules we implemented in our prototype certifying Circuit propagator, as discussed in Section 5. We observe, however, that similar strategies may be used to introduce proof logging for other ad-hoc techniques. For example, if the algorithm is based on identifying *strong bridges* [18] and requiring them to be part of the solution, clearly a `ReachTooSmall` argument must be applicable if the bridge is assumed to be excluded. Another set of inferences can be applied if the Circuit is first relaxed to a path constraint [11], and the structural filtering of the reduced graph used here is essentially a generalisation of rule 4 ("prune skip") and so should be amenable to justification using `ReachTooSmall` and ordering assumptions.

### 4.1    Proving a Set Reachable from Vertex 0 is Too Small

We have shown in the previous section that all the inferences performed by a typical SCC propagator can be justified within a proof log if we are able to construct a sequence of PB steps `ReachTooSmall(v)` that establishes a contradiction for any vertex $v$ in the graph $G$ induced by the current domains of variables where $|\text{REACH}(v)| \leq |G|$. It needs to be possible to construct these steps subject to three kinds of assumption, namely, assuming an edge is required, assuming an edge is disallowed, and an "ordering assumption": assuming that a particular vertex must be seen between two other vertices. We now give a sketch for how such an argument can be constructed. First we will show, by way of example, how to construct it when running from the 0 index vertex, `ReachTooSmall(0)`, without assumptions, as this is the simplest case. We later show how this can be modified to work for an arbitrary vertex $v$, and then finally show how the assumptions can be taken into account.

The idea is to collect possible position values (as defined in Algorithm 1) in a breadth-first search from the starting node. We create auxiliary variables $p[i]_{=k}$ defined through reification to be true if and only if the bit sum $P_i$ is equal to

$k$, and we aim to derive sets of PB constraints enforcing AtLeast1 and AtMost1 requirements over all of the possible $i$ values for each $k \in \{0, \ldots, |\text{REACH}(0)|\}$. As an example, suppose the domain graph under a sequence of guesses $\mathcal{G}$ is as represented in Figure 1c. Clearly $\text{REACH}(0) = \{0, 1, 5\}$, which has fewer that 6 elements, so we should be able to run `ReachTooSmall`(0). In this particular case the procedure would derive constraints (8) to (11) which show that for each $k \in \{0, 1, 2, 3\}$ at least one of the vertices 0, 1, 5 must have position value $k$. It would then derive corresponding constraints (12) to (14) which express the fact that each vertex can have at most one position value. Note that these are all reified on the sequence of solver guesses, but the $\bigwedge \mathcal{G} \Rightarrow$ is omitted for compactness.

AtLeast1 constraints:

$$p[0]_{=0} \qquad\qquad\qquad \geq 1 \quad (8)$$
$$p[1]_{=1} + p[5]_{=1} \geq 1 \quad (9)$$
$$p[0]_{=2} + p[1]_{=2} + p[5]_{=2} \geq 1 \quad (10)$$
$$p[0]_{=3} + p[1]_{=3} + p[5]_{=3} \geq 1 \quad (11)$$

AtMost1 constraints:

$$-p[0]_{=0} \qquad -p[0]_{=2} - p[0]_{=3} \geq -1 \quad (12)$$
$$-p[1]_{=1} - p[1]_{=2} - p[1]_{=3} \geq -1 \quad (13)$$
$$-p[5]_{=1} - p[5]_{=2} - p[5]_{=3} \geq -1 \quad (14)$$

Using the addition rule the procedure can then derive the sum of all these constraints, and by construction everything on the left-hand side will cancel out, leaving $\mathcal{G} \Rightarrow 0 \geq 1$, as required. This process similar to how Hall violators for AllDifferent are derived by Elffers et al [10].

It remains to show how the AtLeast1 and AtMost1 constraints can be derived using *VeriPB* proof rules. The first AtLeast1 (8) can be introduced by RUP, since $p[0]_{=0}$ propagates directly from the encoding. Then, each subsequent constraint can be derived from the previous constraint by first deriving some intermediate reified constraints by RUP, adding them together, and applying the *saturation* rule [3], which reduces any unnecessarily large coefficients. For example to derive (10) from (9) we would use the following proof steps:

$$x[1]_{=0} + x[1]_{=5} \geq 1 \qquad\qquad\qquad (\text{RUP}) \qquad (15)$$
$$\overline{p[1]}_{=1} + \overline{x[1]}_{=0} + p[0]_{=2} \geq 1 \qquad\qquad (\text{RUP}) \qquad (16)$$
$$\overline{p[1]}_{=1} + \overline{x[1]}_{=5} + p[5]_{=2} \geq 1 \qquad\qquad (\text{RUP}) \qquad (17)$$
$$\overline{p[1]}_{=1} + p[0]_{=2} + p[5]_{=2} \geq 1 \quad ((15) + (16) + (17), \text{sat.}) \qquad (18)$$
$$\overline{p[5]}_{=1} + p[0]_{=2} + p[1]_{=2} \geq 1 \qquad\qquad (\text{similarly}) \qquad (19)$$
$$p[0]_{=2} + p[1]_{=2} + p[2]_{=2} \geq 1 \quad (9) + (18) + (19), \text{sat.}) \qquad (20)$$

To derive each of the AtMost1 constraints, we first introduce constraints $\overline{p[i]}_{=k} + \overline{p[i]}_{=l} \geq 1$ by RUP for each distinct pair of values $(l, k)$ values possible for $P_i$. We then add these together but divide by $j$ after adding the $j_{th}$ constraint to recover the required constraint.

## 4.2   Proving a Set Reachable from an Arbitrary Vertex is Too Small

The above example establishes the general structure of the `ReachTooSmall` procedure: we collect AtLeast1 constraints over auxiliary position variables until we have more values than variables, and then add recovered AtMost1 constraints to these to obtain contradiction. However, the specifics of deriving the AtLeast1 constraints depended on us starting from the 0 vertex, as this is required in the encoding to be 0. There is nothing particularly special about the 0 vertex, but without requiring some position label $P_i = 0$ there would be $n$ isomorphic solutions to the PB model for each arbitrary choice of starting vertex in a corresponding solution to the CP model. For our justifications from Section 4 to work, we need to be able to run `ReachTooSmall`$(v)$ from an arbitrary vertex, and so we need a way to start the breadth-first search for possible positions without necessarily knowing with the position of the first node might be.

The idea is to dynamically introduce a new set of position labels $\{q[r,i] : 1 \leq i \leq n\}$ for a given starting vertex $r$, that are tied to the value of the $P_i$ variables but represent what would be obtained if the value of each $P_i$ was *shifted back* modulo $n$ so that $P_r = 0$. Specifically we should have $q[r,i] = P_i - P_r \bmod n$. This preserves the useful property that if $X_i = j$ then $q[r,j] = q[r,i] + 1 \bmod n$, as is true for the $p$ variables. By construction we must have $q[r,r] = 0$, and so we should be able to collect sets of possible $q[r,i]$ variables for each subsequent value and use this to construct our `ReachTooSmall` argument as before.

As with the other auxiliary variables, flags for these $q$ variables can be introduced in the proof as needed using *VeriPB's* redundance-based strengthening rule. We do require some additional $d[r,i]$ flags to encode the definitions in pseudo-Boolean form, to correct for when the difference $P_i - P_r$ is less than 0. Specifically, whenever we require a variable $q[r,i]_{=k}$ we introduce the following.

$$d[r,i] \implies P_r - P_i \geq 1 \tag{21}$$

$$\overline{d[r,i]} \implies P_i - P_r \geq 1 \tag{22}$$

$$q[r,i]_{\geq k} \iff P_i - P_r + nd[r,i] \geq k \tag{23}$$

$$q[r,i]_{\geq k} \iff q[r,i]_{\geq k} + \overline{q[r,i]}_{\geq k+1} \geq 2 \tag{24}$$

These can each be introduced by redundance, and only need to be defined once for each combination of $r$, $i$, and $k$. One technicality is that for the constraint (22) we do require a subproof that establishes $P_i \neq P_r$ in order to apply redundance, but this is straightforward since we can pay a one-time cost to recover an AllDifferent constraint (AtLeast1 and AtMost1 constraints) over the $p$ variables at the very start of the proof.

With these in place, we can outline the general procedure for constructing a `ReachTooSmall` argument from an arbitrary vertex, which is shown in Algorithm 2. All of the statements marked with **derive** can be derived from the PB model and the previous statements either with a single RUP step, or by a sequence of cutting planes steps followed by a RUP step. For lines 10, 12, and 23 this is just adding up the defining constraints for the auxiliary variables involved so that any $p$ and $d$ variables cancel out—we omit the details for brevity.

**Algorithm 2** Procedure for constructing the proof, `ReachToSmall`$(r)$, showing that a sequence of guesses $\bigwedge \mathcal{G}$ imply contradiction.

---

1: **derive** $q[r,r]_{=0} \geq 0$
2: `reached` $\leftarrow \{r\}$;     `lastReached` $\leftarrow \{r\}$;     `valuesSeen`$[r] \leftarrow \{0\}$;     $k \leftarrow 1$
3: **while** $k \leq |\text{reached}|$
4:  |  `newReached` $\leftarrow \emptyset$
5:  |  **for** $i \in$ `lastReached`
6:  |  |  **for** $j \in \text{domain}(x[i])$
7:  |  |  |  `newReached` $\leftarrow$ `newReached` $\cup \{j\}$
8:  |  |  |  `valuesSeen`$[j] \leftarrow$ `valuesSeen`$[j] \cup \{k\}$
9:  |  |  |  **if** $j \neq r$
10: |  |  |  |  **derive** $q[r,i]_{=k-1} \wedge x[i]_{=j} \implies q[r,j]_{=k} \geq 1$   ▷ *Add defs. and RUP*
11: |  |  |  **else**
12: |  |  |  |  **derive** $q[r,i]_{=k-1} \wedge x[i]_{=j} \implies 0 \geq 1$          ▷ *Add defs. and RUP*

13: |  |  **derive** $\bigwedge \mathcal{G} \implies \sum_{j \in \text{domain}(x[i])} x[i]_{=j} \geq 1$                ▷ *RUP*

14: |  |  ▷ *Add up line 13 with all constraints last derived at lines 10 and 12*        ◁
15: |  |  **derive** $\bigwedge \mathcal{G} \wedge q[r,i]_{=k-1} \implies \sum_{j \in \text{domain}(x[i])} q[r,j]_{=k} \geq 1$

16: |  ▷ *AL1 constraint: Add up last AL1 constraint and all lines last derived at 15* ◁
17: |  **derive** $\bigwedge \mathcal{G} \implies \sum_{j \in \text{newReached}} q[r,j]_{=k} \geq 1$

18: |  `lastReached` $\leftarrow$ `newReached`;     `reached` $\leftarrow$ `reached` $\cup$ `newReached`
19: |  $k \leftarrow k + 1$

20: **for** $j \leftarrow 0$ to $n$
21: |  **if** `valuesSeen`$[i] \neq \emptyset$
22: |  |  **for** $(l,k) \in$ `valuesSeen`$[j] \times$ `valuesSeen`$[j]$ **where** $l < k$
23: |  |  |  **derive** $\overline{q[r,j]_{=l}} + \overline{q[r,j]_{=k}} \geq 1$ ▷ *NotBoth constraint: Add up definitions*

24: |  |  ▷ *AM1 constraint: Add up NotBoth constraints, dividing by i after each*
        *step adding the $i_{th}$ NothBoth constraint*        ◁
25: |  |  **derive** $\sum_{k \in \text{valuesSeen}[j]} -q[r,j]_{=k} \geq -1$

26: **derive** $\bigwedge \mathcal{G} \implies 0 \geq 1$                ▷ *Add up AL1 and AM1 constraints.*

---

### 4.3   Proving Reach is Too Small with Assumptions

The procedure outlined in Algorithm 2 requires minimal modification to work with assignment assumptions. Assuming $X_i = j$ or $X_i \neq j$ means including a PB variable encoding the assumption as an additional guess in $\bigwedge \mathcal{G}$, and skipping any domain values excluded by this assumption (either trivially or by AllDifferent) when iterating through the domains on line 5. This will allow the procedure to derive $\bigwedge \mathcal{G} \wedge x[i]_{=j} \implies 0 \geq 1$ or $\bigwedge \mathcal{G} \wedge \overline{x[i]}_{=j} \implies 0 \geq 1$, as required.

More care is required to encode and use ordering assumptions as discussed in Section 4. If we want to force the `ReachTooSmall`$(r)$ to assume that $r \prec a \prec b$,

that is, a vertex $a$ must be visited before $b$ when following a path from $r$, we first have to encode this assumption and reify it with its own flag. We can use auxiliary variables $d[i, j]$, defined as in (21) and (22) to do this:

$$a_{r \prec a \prec b} \iff \overline{d[r, a]} + \overline{d[a, b]} + \overline{d[b, r]} \geq 2 \tag{25}$$

We can then include $a_{r \prec a \prec b}$ as an additional "guess" and use it to exclude $q[r, b]_{=k}$ from any AtLeast1 constraint where, for $k' < k$, $q[r, a]_{=k'}$ was not part of a previous AtLeast1. This is achieved by deriving $q[r, a]_{\geq 1}$ after, line 1, and subsequently $q[r, a]_{\geq k}$ after line 15, and using these to derive $a_{r \prec a \prec b} \wedge q[r, i]_{=k+1} \wedge x[i]_{=b} \implies 0 \geq 1$ instead of line 10 whenever $j = b$. Once again these each amount to steps adding up definition constraints, followed by a RUP step, and we omit the details for brevity.

We note that the encoding of ordering assumptions (25) allows the justification of the prune skip inference with two conditional ReachTooSmall arguments, as discussed in Section 4. When ReachTooSmall arrives at a contradiction under an ordering assumption, the negation is established, and we can add up definitions for e.g. $\bar{a}_{j \prec r \prec v_0}$, $\bar{a}_{r \prec i \prec v_0}$, $x[i]_{=j}$ to cancel out $p$ and $d$ variables and arrive at a final contradiction for this inference.

## 5    Implementation and Evaluation

We have implemented proof logging versions of the *check*, *prevent* and *SCC* propagators (with all inference rules discussed in Section 5) using the techniques described in this paper as part of the auditable *Glasgow Constraint Solver* project [24], and we included in our implementation all the inference methods available in the Circuit propagators of *Gecode* [14]. We tested our implementation[3] by solving randomly generated travelling salesperson problems (TSPs), with graphs ranging in size from 3 to 40 vertices. The potential of proof logging as a powerful debugging and development tool was immediately apparent from

---

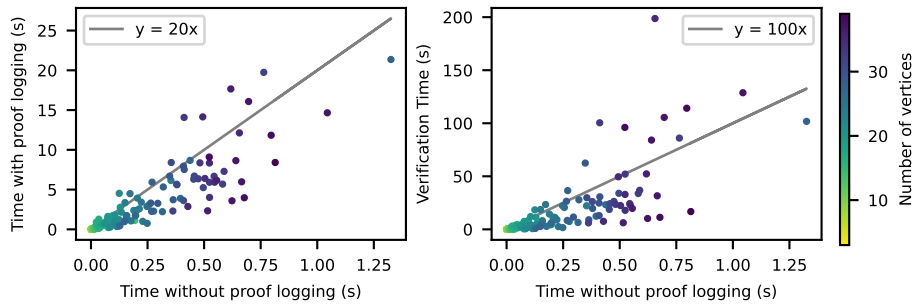[3] https://zenodo.org/records/10848992



Figure 3: Scatter plot of results of solving randomly generated TSP instances.

this, as initial proof failures immediately indicated bugs in our implementation such as *prevent* trying to disallow full circuits in certain situations, or *SCC* trying to apply an incorrect inference based on the structure of the graph. This aligns with the results of previous research projects, where the implementation of proof logging uncovered hard-to-find bugs in well-tested combinatorial solvers [1,5,20]. Once the bugs were addressed, all proofs were verified as correct using *VeriPB*. The performance data from our evaluation is shown in Figure 3.

There is clearly a cost in terms of overhead from enabling proof logging, although the exact slowdown is very dependent on hardware since we are writing to disk and using a non-optimised text-based proof format. What is clear is that the overhead is not unreasonable, with time to produce the proofs scaling roughly in proportion the time taken to solve without proof logging. This is what we would expect: our proof procedures for *check* and *prevent* output exactly one sequence of cutting planes steps for each subcycle prevented or disallowed, and so clearly are not doing significantly more work than the propagators themselves. Similarly, the `ReachTooSmall` procedure is called once for every inference (except the "prune skip" inference where it is called twice) and can at worst generate a number of proof steps proportional to $n \cdot E$ where $E$ is the number of edges currently encoded by the domains of variables. Since Tarjan's algorithm itself runs in $O(n + E)$, we are satisfied that we are roughly within a linear factor of the amount of work done by the propagator and that our proof logging procedure is practical and free from any exponential blow up.

Additionally, we tested the TSP instance from the *MiniCP* benchmark suite of Michel et al. [26]. This was created for testing CP solver speed, and took the *Glasgow Constraint Solver* 44.9407 seconds to solve without proof logging (using full Circuit and AllDifferent propagation, not the simple propagation used by MiniCP). With proof logging, it took 3603.84 seconds ($\sim$ 1 hour) to solve, and *VeriPB* needed 585893.41 seconds ($\sim$ 1 week) to verify the produced proof. This, together with previously implemented constraints [16], brings the *Glasgow Constraint Solver* in line with *MiniCP* in terms of propagators implemented and instance modelling capabilities.

## 6   Conclusion

We have exhibited the first certifying Circuit propagator using *VeriPB* proof logging, showing that ad-hoc inference rules with complicated notions of consistency can be included in an auditable constraint solver. In particular, we found that a range of standard inference types could make use of similar proof procedures, taking advantage of concepts such as connectedness and vertex ordering despite the proof system having no native representantions of these notions, or even of a graph. We expect that the core concepts exemplified here: such as counting reachable vertices under implications; creating shifted auxiliary labels; and running proof procedures under ordering assumptions will be useful for other constraints, and for proof logging combinatorial solving more generally.

# References

1. Berg, J., Bogaerts, B., Nordström, J., Oertel, A., Vandesande, D.: Certified core-guided MaxSAT solving. In: Pientka, B., Tinelli, C. (eds.) Automated Deduction - CADE 29 - 29th International Conference on Automated Deduction, Rome, Italy, July 1-4, 2023, Proceedings. Lecture Notes in Computer Science, vol. 14132, pp. 1–22. Springer (2023). https://doi.org/10.1007/978-3-031-38499-8_1, https://doi.org/10.1007/978-3-031-38499-8_1

2. Bogaerts, B., Gocht, S., McCreesh, C., Nordström, J.: Certified symmetry and dominance breaking for combinatorial optimisation. Journal of Artificial Intelligence Research **77**, 1539–1589 (Aug 2023), preliminary version in *AAAI '22*

3. Buss, S.R., Nordström, J.: Proof complexity and SAT solving. In: Biere, A., Heule, M.J.H., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 336, chap. 7, pp. 233–350. IOS Press, 2nd edn. (Feb 2021)

4. Caseau, Y., Laburthe, F.: Solving small TSPs with constraints. In: Naish, L. (ed.) Logic Programming, Proceedings of the Fourteenth International Conference on Logic Programming, Leuven, Belgium, July 8-11, 1997. pp. 316–330. MIT Press (1997)

5. Cheung, K.K.H., Gleixner, A.M., Steffy, D.E.: Verifying integer programming results. In: Eisenbrand, F., Könemann, J. (eds.) Integer Programming and Combinatorial Optimization - 19th International Conference, IPCO 2017, Waterloo, ON, Canada, June 26-28, 2017, Proceedings. Lecture Notes in Computer Science, vol. 10328, pp. 148–160. Springer (2017). https://doi.org/10.1007/978-3-319-59250-3_13, https://doi.org/10.1007/978-3-319-59250-3_13

6. Chu, G., Stuckey, P.J., Schutt, A., Ehlers, T., Gange, G., Francis, K.: Chuffed, a lazy clause generation solver (2023), https://github.com/chuffed/chuffed

7. Cook, W., Coullard, C.R., Turán, Gy.: On the complexity of cutting-plane proofs. Discrete Applied Mathematics **18**(1), 25–38 (Sep 1987). https://doi.org/10.1016/0166-218X(87)90039-4

8. Cruz-Filipe, L., Heule, M.J.H., Hunt Jr, W.A., Kaufmann, M., Schneider-Kamp, P.: Efficient Certified RAT Verification. In: de Moura, L. (ed.) Automated Deduction – CADE 26. pp. 220–236. Lecture Notes in Computer Science, Springer International Publishing, Cham (2017). https://doi.org/10.1007/978-3-319-63046-5_14

9. Di Gaspero, L., Urli, T.: A CP/LNS Approach for Multi-day Homecare Scheduling Problems. In: Blesa, M.J., Blum, C., Voß, S. (eds.) Hybrid Metaheuristics. pp. 1–15. Lecture Notes in Computer Science, Springer International Publishing, Cham (2014). https://doi.org/10.1007/978-3-319-07644-7_1

10. Elffers, J., Gocht, S., McCreesh, C., Nordström, J.: Justifying All Differences Using Pseudo-Boolean Reasoning. In: The Thirty-Fourth AAAI Conference on Artificial

Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020. pp. 1486–1494. AAAI Press (2020)

11. Fages, J.G., Lorca, X.: Improving the Asymmetric TSP by Considering Graph Structure (Jun 2012). https://doi.org/10.48550/arXiv.1206.3437

12. Francis, K.G., Stuckey, P.J.: Explaining circuit propagation. Constraints **19**(1), 1–29 (Jan 2014). https://doi.org/10.1007/s10601-013-9148-0

13. Gaspero, L.D., Rendl, A., Urli, T.: Balancing bike sharing systems with constraint programming. Constraints **21**(2), 318–348 (Apr 2016). https://doi.org/10.1007/s10601-015-9182-1

14. Gecode Team: Gecode: Generic constraint development environment (2023), http://www.gecode.org

15. Gocht, S.: Certifying Correctness for Combinatorial Algorithms: By Using Pseudo-Boolean Reasoning. Ph.D. thesis, Lund University, Sweden (2022)

16. Gocht, S., McCreesh, C., Nordström, J.: An Auditable Constraint Programming Solver. In: Solnon, C. (ed.) Proceeding of the 28th International Conference on Principles and Practice of Constraint Programming. Leibniz International Proceedings in Informatics (LIPIcs), vol. 235, pp. 25:1–25:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2022). https://doi.org/10.4230/LIPIcs.CP.2022.25

17. Heule, M.J.H.: Chinese Remainder Encoding for Hamiltonian Cycles. In: Li, C.M., Manyà, F. (eds.) Theory and Applications of Satisfiability Testing - SAT 2021 - 24th International Conference, Barcelona, Spain, July 5-9, 2021, Proceedings. Lecture Notes in Computer Science, vol. 12831, pp. 216–224. Springer (2021). https://doi.org/10.1007/978-3-030-80223-3_15

18. Italiano, G.F., Laura, L., Santaroni, F.: Finding strong bridges and strong articulation points in linear time. Theoretical Computer Science **447**, 74–84 (Aug 2012). https://doi.org/10.1016/j.tcs.2011.11.011

19. Karp, R.M.: Reducibility among Combinatorial Problems. In: Miller, R.E., Thatcher, J.W., Bohlinger, J.D. (eds.) Complexity of Computer Computations: Proceedings of a Symposium on the Complexity of Computer Computations, Held March 20–22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, and Sponsored by the Office of Naval Research, Mathematics Program, IBM World Trade Corporation, and the IBM Research Mathematical Sciences Department, pp. 85–103. The IBM Research Symposia Series, Springer US, Boston, MA (1972). https://doi.org/10.1007/978-1-4684-2001-2_9

20. Kraiczy, S., McCreesh, C.: Solving graph homomorphism and subgraph isomorphism problems faster through clique neighbourhood constraints. In: Zhou, Z. (ed.) Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI 2021, Virtual Event / Montreal, Canada, 19-27 August 2021. pp. 1396–1402. ijcai.org (2021). https://doi.org/10.24963/IJCAI.2021/193, https://doi.org/10.24963/ijcai.2021/193

21. Kuchcinski, K., Szymanek, R.: JaCoP - Java Constraint Programming Solver. In: CP Solvers: Modeling, Applications, Integration, and Standardization, co-located with the 19th International Conference on Principles and Practice of Constraint Programming (2013)

22. Lam, E., Van Hentenryck, P.: A branch-and-price-and-check model for the vehicle routing problem with location congestion. Constraints **21**(3), 394–412 (Jul 2016). https://doi.org/10.1007/s10601-016-9241-2

23. Lam, E., Van Hentenryck, P., Kilby, P.: Joint Vehicle and Crew Routing and Scheduling. In: Principles and Practice of Constraint Programming : 21st International Conference, CP 2015 Cork, Ireland, August 31 – September 4, 2015 Proceedings. pp. 654–670. Springer (2015). https://doi.org/10.1007/978-3-319-23219-5_45
24. McCreesh, C., McIlree, M.: The Glasgow Constraint Solver. GitHub repository (2023), https://github.com/ciaranm/glasgow-constraint-solver
25. McIlree, M.J., McCreesh, C.: Proof logging for smart extensional constraints. In: Yap, R.H.C. (ed.) 29th International Conference on Principles and Practice of Constraint Programming (CP 2023). Leibniz International Proceedings in Informatics (LIPIcs), vol. 280, pp. 26:1–26:17. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2023). https://doi.org/10.4230/LIPIcs.CP.2023.26
26. Michel, L.D., Schaus, P., Van Hentenryck, P.: MiniCP: a lightweight solver for constraint programming. Math. Program. Comput. **13**(1), 133–184 (2021). https://doi.org/10.1007/s12532-020-00190-7, https://doi.org/10.1007/s12532-020-00190-7
27. Ohrimenko, O., Stuckey, P.J., Codish, M.: Propagation = lazy clause generation. In: Bessiere, C. (ed.) Principles and Practice of Constraint Programming - CP 2007, 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007, Proceedings. Lecture Notes in Computer Science, vol. 4741, pp. 544–558. Springer (2007). https://doi.org/10.1007/978-3-540-74970-7_39, https://doi.org/10.1007/978-3-540-74970-7_39
28. Perron, L., Didier, F.: CP-SAT, https://developers.google.com/optimization/cp/cp_solver/
29. Pesant, G., Gendreau, M., Potvin, J.Y., Rousseau, J.M.: An Exact Constraint Logic Programming Algorithm for the Traveling Salesman Problem with Time Windows. Transportation Science **32**(1), 12–29 (Feb 1998). https://doi.org/10.1287/trsc.32.1.12
30. Schulte, C., Tack, G.: Weakly Monotonic Propagators. In: Gent, I.P. (ed.) Principles and Practice of Constraint Programming - CP 2009. pp. 723–730. Lecture Notes in Computer Science, Springer, Berlin, Heidelberg (2009). https://doi.org/10.1007/978-3-642-04244-7_56
31. Tarjan, R.: Depth-First Search and Linear Graph Algorithms. SIAM Journal on Computing **1**(2), 146–160 (Jun 1972). https://doi.org/10.1137/0201010
32. Wetzler, N., Heule, M.J.H., Hunt Jr, W.A.: DRAT-trim: Efficient Checking and Trimming Using Expressive Clausal Proofs. In: Sinz, C., Egly, U. (eds.) Theory and Applications of Satisfiability Testing – SAT 2014. pp. 422–429. Lecture Notes in Computer Science, Springer International Publishing, Cham (2014). https://doi.org/10.1007/978-3-319-09284-3_31
33. Zhou, N.F.: In Pursuit of an Efficient SAT Encoding for the Hamiltonian Cycle Problem. In: Simonis, H. (ed.) Principles and Practice of Constraint Programming. vol. 12333, pp. 585–602. Springer International Publishing, Cham (2020). https://doi.org/10.1007/978-3-030-58475-7_34