Pseudo-Boolean Reasoning About States and Transitions to Certify Dynamic Programming and Decision Diagram Algorithms

Emir Demirović
TU Delft, The Netherlands

Ciaran McCreesh
University of Glasgow, Scotland

Matthew J. McIlree
University of Glasgow, Scotland

Jakob Nordström
University of Copenhagen, Denmark
Lund University, Sweden

Andy Oertel
Lund University, Sweden
University of Copenhagen, Denmark

Konstantin Sidorov
TU Delft, The Netherlands

Abstract

Pseudo-Boolean proof logging has been used successfully to provide certificates of optimality from a variety of constraint- and satisifability-style solvers that combine reasoning with a backtracking or clause-learning search. Another paradigm, occurring in dynamic programming and decision diagram solving, instead reasons about partial states and possible transitions between them. We describe a framework for generating clean and efficient pseudo-Boolean proofs for these kinds of algorithm, and use it to produce certifying algorithms for knapsack, longest path, and interval scheduling. Because we use a common proof system, we can also reason about hybrid solving algorithms: we demonstrate this by providing proof logging for a dynamic programming based knapsack propagator inside a constraint programming solver.

2012 ACM Subject Classification Theory of computation → Logic and verification; Theory of computation → Discrete optimization

Keywords and phrases Proof logging, dynamic programming, decision diagrams

Digital Object Identifier 10.4230/LIPIcs.CP.2024.9

Supplementary Material *Software*: `https://doi.org/10.5281/zenodo.12574620`

Funding *Emir Demirović*: part of the XAIT lab funded by the Delft AI Labs programme.
Ciaran McCreesh: supported by a Royal Academy of Engineering research fellowship, and by the Engineering and Physical Sciences Research Council [grant number EP/X030032/1].
Jakob Nordström: supported by the Swedish Research Council grant 2016-00782 and the Independent Research Fund Denmark grant 9040-00389B.
Andy Oertel: supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.
Konstantin Sidorov: part of the XAIT lab funded by the Delft AI Labs programme.

Acknowledgements Part of this work was carried out while taking part in the semester program *Satisfiability: Theory, Practice, and Beyond* in 2021 at the Simons Institute for the Theory of Computing at UC Berkeley, and in the extended reunion of this semester program in the spring of 2023. This work has also benefited greatly from discussions during the Dagstuhl Seminars 22411 *Theory and Practice of SAT and Combinatorial Solving* and 23261 *SAT Encodings and Beyond*.

1 Introduction

It is sometimes vital that combinatorial solving algorithm implementations can be trusted to give correct answers. To this end, when claiming that a problem has no solution, Boolean satisfiability (SAT) solvers do not just assert unsatisfiability, but also provide an independently verifiable proof of this fact, in one of several standard formats such as *DRAT* [20, 19, 35], *LRAT* [10], or *VeriPB* [13]. The proof can then be inspected by a formally verified proof checker to assert its correctness. This means the algorithm is *certifying* [28]: while we still cannot trust that the implementation is correct, this does guarantee that if it ever gives an incorrect answer, then we can detect it.

Of the above proof formats, *VeriPB* is the most general-purpose: as well as supporting advanced SAT-solving techniques such as parity reasoning [18], symmetry and dominance breaking [4], and MaxSAT optimisation [1], it has also been used for subgraph-finding algorithms [16, 14, 15] and for constraint programming with a variety of global constraints [17, 29]. In these latter settings, a *VeriPB* proof resembles a description of a backtracking search tree, interleaved with justifications of facts obtained from inference algorithms or constraint propagation. However, the *VeriPB* proof format has no direct notion of a search tree. Instead, its underlying proof system is powerful enough to express implicational reasoning. In particular, constraints may be reified and dereified, and if some fact can be derived, it can also be derived under a sequence of guesses with (almost) no additional effort. This is in contrast to, e.g., the VIPR proof format [8], which was designed specifically for mixed integer programming and which has explicit notions of assumptions and closing branches that function independently from other proof rules. An advantage of a sufficiently powerful proof system that does not have a direct notion of search is that techniques like restarts [16] and autotabulation [17] can be encoded without needing additions to the proof system.

However, there are non-search-based ways of solving hard problems. Both dynamic programming and decision diagram algorithms can be viewed as working with partial states, and transitions between those states [22, 3]. In this work, we show that *VeriPB* can also be used for efficient proof logging for algorithms that work with states and transitions, rather than search, regardless of whether the algorithm uses memoisation, a matrix, or a layer-by-layer construction. This is primarily because the pseudo-Boolean constraints and extended cutting planes proof system underlying *VeriPB* makes it very clean to work with implications.

Using a common system, rather than inventing a new proof system for dynamic programming proofs, has several benefits: it allows us to reason about hybrid or nested solving strategies that use more than one kind of algorithm, it avoids the need to reinvent proof logging for various kinds of constraint and dominance reasoning, and it gives us immediate access to a suite of proof checking tools which would otherwise be expensive to recreate. To illustrate this, we have implemented proof logging for a knapsack constraint inside a constraint programming solver, whose propagator involves reasoning about paths through a dynamic programming table or decision diagram to detect loss of support for values in constraint programming variables [34].

2 Background

Before we can talk about proofs for dynamic programming problems, we give a brief overview of the *VeriPB* proof system, and outline how it has been used to generate proofs for backtracking search algorithms.

2.1 Pseudo-Boolean Preliminaries

Although designed to support many different kinds of solvers, the foundations of the *VeriPB* proof system are Boolean variables and pseudo-Boolean constraints. Let x_i be a set of Boolean variables ranging over 0 (false) and 1 (true). We write $\overline{x}_i$ to mean $1 - x_i$ (i.e. not x), and refer to x_i and $\overline{x}_i$ as *literals*. A pseudo-Boolean (PB) constraint over literals ℓ_i is an inequality in the form $\sum_i c_i\ell_i \bowtie A$, where $\bowtie$ is either $\geq$ or $\leq$ and c_i and A are integer constants. A PB constraint can always be rewritten in *normalised form* $\sum_i c_i\ell_i \geq A$ with all literals over distinct variables and all c_i and A non-negative, and when describing the proof system we will assume constraints are normalised. A PB optimisation problem is a set of PB constraints, together with an objective $\sum_i c_i\ell_i$ to be minimised.

Let $C = \sum_i c_i\ell_i \geq A$ be a PB constraint, and y and y_j be distinct literals. We define $\overline{C}$ to mean $\sum_i c_i\ell_i \leq A-1$; $\wedge_j y_j \Rightarrow C$ to mean $\sum_j K\overline{y}_j+\sum_i c_i\ell_i \geq A$ where $K = A-\sum_i \min(c_i, 0)$; and $y \Leftrightarrow C$ to mean the pair of PB constraints $y \Rightarrow C$ and $\overline{y} \Rightarrow \overline{C}$. It is easy to check that the constraints defined in this way have the meaning suggested by the notation used. Note how, unlike for Boolean formulae in conjunctive normal form (CNF), full reification of a pseudo-Boolean constraint by a literal requires only a pair of constraints.

2.2 The *VeriPB* Proof System

In a *VeriPB* proof, we begin with a set of pseudo-Boolean constraints as input – these are assumed, as axioms, and so they must accurately describe the high-level problem being solved. A proof is then a sequence of pseudo-Boolean constraints, where each new constraint follows either obviously or by explicit construction from the input and any other constraints already derived, in such a way that at least one optimal solution is always preserved.

When proof steps consist of explicit constructions, they are given as a sequence of *cutting planes* steps [7], as follows. For any literal ℓ_i, we may freely introduce a constraint $\ell_i \geq 0$. Given two constraints $\sum_i a_i\ell_i \geq A$ and $\sum_i b_i\ell_i \geq B$, we may add them together to derive $\sum_i(a_i + b_i)\ell_i \geq A + B$. We may also multiply by a positive integer constant c, to get $\sum_i ca_i\ell_i \geq cA$, or (assuming normalised form) divide to get $\sum_i \lceil\frac{a_i}{c}\rceil\ell_i \geq \lceil\frac{A}{c}\rceil$. Finally, we can *saturate*, turning (again assuming normalised form) $\sum_i a_i\ell_i \geq A$ into $\sum_i \min(a_i, A)\,\ell_i \geq A$.

A clausal constraint, or *clause*, is one of the form $\sum_i \ell_i \geq 1$. This corresponds naturally to a Boolean clause in CNF. By *resolution*, we mean deriving $\sum_i x_i + \sum_j y_j \geq 1$ from the clauses $r + \sum_i x_i \geq 1$ and $\overline{r} + \sum_j y_j \geq 1$; this may be achieved by adding the constraints and then saturating [21]. In particular, resolution allows us to take the clauses $r \Rightarrow \sum_i x_i \geq 1$ and $r + \sum_j y_j \geq 1$ and derive $\sum_i x_i + \sum_j y_j \geq 1$. Proof steps such as this that involve implications are generally straightforward in cutting planes: for example, if we have both $r \Rightarrow \sum_i a_i x_i \geq A$ and $s \Rightarrow r$, we may easily derive that $s \Rightarrow \sum_i a_i x_i \geq A$ by multiplication and then addition. As a special case of this, if we have established that the left hand side of an implication must be true, then we can dereify the implication and derive its right hand side unconditionally. Another useful fact, which we use repeatedly throughout this work, is that if we have a process for deriving a constraint D from a set of constraints C_i, then we can reuse this process to derive a reified version of D if we are given a set of reified constraints C'_i; we explain this in detail in the appendix.

An alternative to cutting planes steps is to allow the proof verifier to add constraints that are obvious enough that they do not require an explicit derivation. A constraint C follows by *reverse unit propagation* (RUP) if adding $\overline{C}$ to the existing set of constraints leads immediately to contradiction upon achieving integer bounds consistency for each constraint individually [9]. Obviously such constraints are implied, and this condition can be verified efficiently, so a RUP constraint may safely be added as a proof step. (The term *unit propagation* is used due to the SAT origins of proof logging [12]; if all constraints are clauses, integer bounds consistency and unit propagation are equivalent.) As with cutting planes proofs, RUP proof procedures can trivially be modified to work subject to reifications.

The *VeriPB* proof system also has a non-implicational *strengthening* rule [4]. We do not use the full generality of the rule in this paper, but will use it as an *extension* rule. An *extension variable* z reifying an arbitrary PB constraint C is a variable which has not previously been used, which is introduced in a proof alongside the pair of constraints $z \Leftrightarrow C$; the strengthening rule can be used to introduce an extension variable in this way. We will also use strengthening to implement *fusion resolution*: given $r \Rightarrow \sum_i a_i x_i \geq A$ and $\overline{r} \Rightarrow \sum_i a_i x_i \geq A'$, strengthening lets us derive that $\sum_i a_i x_i \geq \min(A, A')$.

A proof of unsatisfiability ends by deriving $0 \geq 1$. For an optimisation problem with objective expression $\sum_i c_i \ell_i$, a *VeriPB* proof will conclude by demonstrating that the objective lies between two integer lower and upper bounds – for an exact solution, these will be the same. To do this, a proof step may witness a solution by giving a partial assignment to variables. The proof checker verifies that this assignment unit propagates to a complete feasible assignment to all variables, and then introduces a new objective-improving constraint $\sum_i c_i \ell_i \leq A - 1$ where A is the calculated objective value from the assignment.

Finally, we may also delete derived constraints, under certain conditions. This will lower the amount of memory required to verify the proof, as well as potentially speeding up verification of RUP and strengthening steps. For soundness reasons, there are restrictions on when constraints may be deleted (e.g. to prevent us from deleting every constraint in the input and then claiming an optimal solution with zero cost) [4], but for the techniques used in this paper, the verifier will allow us to delete any constraint we introduce, as well as any extension variable by deleting its two defining constraints.

2.3 A Framework for Proofs for Backtracking Search

For a very simple backtracking search algorithm, a proof could consist of a RUP statement for every backtrack, asserting that at least one of the guessed assignments must be false. Alternatively, if we are using conflict-driven clause learning (CDCL), a proof consists of a RUP step for every learned clause in turn. This applies to proofs using either *DRAT* or *VeriPB*. However, this is only possible if every fact used by the search algorithm follows by integer bounds consistency on the PB representation of the problem (or, for *DRAT*, from unit propagation on the CNF representation). This would suffice, e.g. for conventional DPLL or CDCL SAT solvers, but does not work if we have stronger propagation or inference algorithms such as domain-consistent all-different. In this case, it is necessary to help the proof checker by interleaving additional steps within the proof [17]. The nature of these steps depends upon the inference being performed, and can involve additional RUP steps or (in *VeriPB* proofs only) explicit cutting planes steps. The aim here is to ensure that any fact “known” to the solving algorithm is also visible to the proof checker under unit propagation. Crucially, using PB proofs does *not* mean that the solving algorithm is in any way a PB solver, nor does it need to employ any cutting planes reasoning to be able to write cutting planes proof steps. Instead, most solvers that write *VeriPB* proofs are conventional algorithms that have subsequently been augmented with, effectively, template-based print statements.

Although variations on this technique are suitable for various forms of backtracking search, including with backjumping and restarts, this framework does not extend to being able to cover dynamic programming algorithms, which have a very different notion of a search space. The remainder of this paper explores a different framework, where the structure of *VeriPB* proofs represent how dynamic programming algorithms run.

3 Proofs Involving States and Transitions

The key idea we will use for the proofs in this paper is to introduce an extension variable for each entry in a dynamic programming matrix, or for each node in a memoised recursive search tree or a top-down decision diagram construction. Each of these extension variables will reify the conjunction of several other extension variables, representing different parts of the state. We will then build up implication constraints between these extension variables that reflect the way entries in the matrix are derived, the recursive call structure, or the edges in the decision diagram. We will additionally build up a series of at-least-one constraints, demonstrating that the structure we have created is complete. We finish by using the at-least-one constraint over the final row of the matrix, or the final non-terminal layer of the decision diagram, to prove the conclusion.

So far, this idea is not unique to *VeriPB* proofs. The *DRAT* proof system also has an extension rule, and indeed Sinz and Biere [31], Jussila et al. [23] and Bryant [6] have constructed *DRAT* proofs for binary decision diagram solvers using extension variables in a similar but more restricted way. However, using *VeriPB* has many theoretical and practical benefits when we look at more complex problems. For example, counting problems like pigeonhole have direct proofs in *VeriPB* that scale trivially to arbitrarily large numbers of pigeons, and do not require decision diagram structures for some semblance of efficiency. Similarly, cutting planes allows us to work efficiently with reified integer linear inequalities without requiring complex and inefficient adder and multiplier circuits. *VeriPB* also supports optimisation problems, whereas the *DRAT* proof system only guarantees that satisfiable instances cannot be made unsatisfiable, and would not be sound if used for optimisation problems. Since we are looking to bring proof logging to a broader range of algorithms that solve problems far beyond the reach of SAT solving, we will work exclusively with *VeriPB*.

3.1 Knapsack as a Dynamic Programming Problem

We will first illustrate how to create proofs for simple 0/1 knapsack problems. We are given n items with weights $\boldsymbol{w}_i$ and profits $\boldsymbol{p}_i$, and we want to maximise profit whilst not taking items with a combined weight more than some constant W. For simplicity, we assume that all weights and profits are non-negative integers. We can express this as the PB problem

$$x_i \in \{0,1\} \qquad i \in \{1,\dots,n\} \tag{1}$$

$$\text{minimise} \qquad \sum_{i=1}^{n} -\boldsymbol{p}_i x_i \tag{2}$$

$$\text{subject to} \qquad \sum_{i=1}^{n} \boldsymbol{w}_i x_i \le W, \tag{3}$$

recalling the convention that PB problems have an objective function to be minimised rather than maximised. Note already that this PB representation is extremely straightforward, and does not involve constructing adder and multiplier circuits as it would if we used a CNF encoding.

This problem has a recursive formulation. Letting $P(i, w)$ be the maximum profit obtainable after taking the first i items whilst having weight w still available to use, we have the properties

$$P(0, w) = 0 \tag{4}$$
$$P(i, w) = \max\{ \tag{5}$$
$$P(i-1, w), \tag{6}$$
$$P(i-1, w - \boldsymbol{w}_i) + \boldsymbol{p}_i \text{ if } \boldsymbol{w}_i \leq w\}. \tag{7}$$

Here, Equation (4) gives the initial condition that there is zero profit from taking no items, regardless of weight; Equation (6) describes the option where we do not take item i; Equation (7) describes the option where we do take item i if we are allowed to; and the max operator in Equation (5) says that if we have two partial sums over the first i items both using weight $W - w$ then we need only consider the one which gives us the better profit.

This relation does not directly give us an algorithm. However, there are several standard ways of turning such a recurrence relationship into an algorithm, including dynamic programming via a matrix built iteratively over weights; using recursion with memoisation; or constructing a decision diagram layer by layer from the root downwards [22, 32]. From an algorithm implementation perspective, the choice of methods can be very important; however, for proof logging, the approach we describe works equally well for all three methods. The important points are simply that

1. the algorithm somehow avoids calculating the same partial sums twice;
2. not all partial sums of weights and profits are necessarily calculated; and
3. there is some way of handling "dominated" states, such as the maximum operation in Equation (5).

For ease of explanation, and because it allows the widest range of techniques to be demonstrated, we will assume a layer-by-layer construction, starting by considering whether or not we take the first item, and then building this up to decide what combination of the first two items we will take, and then the first three items, and so on. Within layer i, we will consider every possible partial sum of the first i weights that does not already exceed our bound W, and associate that with the maximum possible partial sum of profits using exactly that weight. We call this information a *state*, no matter whether it is implemented as a node in a decision diagram, a memoised function call, or an entry in a matrix. We call partial sums of either weights or profits *partial* states, and view the full state as being the conjunction of partial weight and profit states.

The idea behind our *VeriPB* proof is that we will introduce an extension variable $S^i_{w,p}$ for each state on layer i with partial sum of weights w and partial sum of profits p. For convenience, we will also introduce these variables for states that will be ignored due to the maximum rule. Recall that an extension variable is introduced by reifying a constraint; in our case, this constraint will be

$$S^i_{w,p} \Leftrightarrow W^i_w + P^i_p \geq 2 \tag{8}$$

where W^i_w and P^i_p are themselves also extension variables,

$$W^i_w \Leftrightarrow \sum_{j=1}^{i} \boldsymbol{w}_j x_j \geq w \text{ and} \tag{9}$$

$$P^i_p \Leftrightarrow \sum_{j=1}^{i} \boldsymbol{p}_j x_j \leq p. \tag{10}$$

In other words, $S^i_{w,p}$ is defined to be true if and only if the sum of the taken weights for the first i items is *at least* w, *and* the sum of the taken profits for the first i items is *at most* p. The reason for this choice of inequalities will become evident when we look at the maximum rule.

Merely introducing extension variables tells us nothing about which states could actually occur. The remainder of the proof consists of deriving implicational relationships between extension variables (which correspond to edges in a decision diagram), and then in proving that each layer is complete (that is, that we have an extension variable for every possible state that has not been eliminated).

The first set of implications that we derive correspond to deciding not to take item x_i. We in turn derive

$$W^{i-1}_w \wedge \overline{x}_i \Rightarrow W^i_w \qquad \text{using a cutting planes addition rule, and then} \tag{11}$$
$$P^{i-1}_p \wedge \overline{x}_i \Rightarrow P^i_p \qquad \text{similarly, and finally} \tag{12}$$
$$S^{i-1}_{w,p} \wedge \overline{x}_i \Rightarrow S^i_{w,p} \qquad \text{follows by RUP.} \tag{13}$$

For the base case, the first part of the conjunction is trivially true and is instead omitted, whilst for subsequent layers we will already have created the earlier extension variables, either due to the algorithm's layer-by-layer construction, or iteration, or recursion.

Next, suppose we *cannot* take item i due to the partial sum of weights exceeding W (recalling that for simplicity, we are forbidding negative weights). If this is the case, we derive

$$W^{i-1}_w \Rightarrow \overline{x}_i \qquad \text{using cutting planes and RUP, and then} \tag{14}$$
$$S^{i-1}_{w,p} \Rightarrow \overline{x}_i \qquad \text{and} \tag{15}$$
$$S^{i-1}_{w,p} \Rightarrow S^i_{w,p} \qquad \text{both follow by RUP.} \tag{16}$$

This cutting planes addition step is between the forward implication constraint defining W^{i-1}_w, and the constraint giving the bound on W that is part of the input axiom. Because none of the remaining weight coefficients are negative, a simple bounds consistency calculation shows that if we have used too much weight already by layer i then there is no way of assigning the remaining x_i variables that will bring our weight sum back to be no more than W.

Finally, suppose we *can* take item i. Letting $w' = w + \boldsymbol{w}_i$ and $p' = p + \boldsymbol{p}_i$ be our new weights and profits respectively, we instead derive

$$W^{i-1}_w \wedge x_i \Rightarrow W^i_{w'} \qquad \text{using cutting planes, and} \tag{17}$$
$$P^{i-1}_p \wedge x_i \Rightarrow P^i_{p'} \qquad \text{similarly, then} \tag{18}$$
$$S^{i-1}_{w,p} \wedge x_i \Rightarrow S^i_{w',p'} \qquad \text{follows by RUP, as does} \tag{19}$$
$$S^{i-1}_{w,p} \Rightarrow S^i_{w,p} + S^i_{w',p'} \geq 1. \tag{20}$$

Until this point, we have been ignoring the maximum rule. If we have two states on the same layer with the same w, and one with profit p and another with profit $p' > p$, we will derive that

$$S^i_{w,p} \Rightarrow S^i_{w,p'}. \tag{21}$$

What this implication means is, "if there is an assignment to the first i x_i variables where the weight sums to at least w and the profit to no more than p, then there is an assignment where the weight sums to at least w and the profit sums to no more than some larger profit p'". This is almost vacuous, and can easily be proved in cutting planes by unwrapping the conjunctions.

In fact, in our proofs we can also do this for a distinct pair of states $S^i_{w,p} \Rightarrow S^i_{w',p'}$ where $w' \leq w$ and $p' \geq p$; this can be detected efficiently in a layer-by-layer algorithm, but not so easily with other approaches.

Now we have described the relationship between states on the same and subsequent layers. The last part of the structure of our proof consists in deriving an at-least-one constraint over the final layer, asserting that our diagram is complete. Again, we make use of an inductive argument, by first deriving at-least-one constraints over the first layer, then the second layer, and so on. This is a simple sequence of resolution steps: given

$$\sum_{(w,p) \text{ on layer } i-1} S^{i-1}_{w,p} \geq 1 \tag{22}$$

we may resolve every variable on

$$\begin{aligned} &S^{i-1}_{w,p} \Rightarrow S^i_{w,p} && \text{from Equation (16), or} \\ &S^{i-1}_{w,p} \Rightarrow S^i_{w,p} + S^i_{w',p'} \geq 1 && \text{from Equation (20)} \end{aligned}$$

to derive the desired

$$\sum_{(w,p) \text{ on layer } i} S^{i}_{w,p} \geq 1. \tag{23}$$

This sets us up to provide a conclusion for our proof. Our algorithm execution will have solved the problem at this point, so we know an optimal assignment with profit $P^\star$ that we can use to obtain a solution-improving constraint $\sum_i -\boldsymbol{p}_i x_i \leq -P^\star - 1$. This in turn contradicts each component of Equation (23), showing unsatisfiability.

To bring this together, we illustrate one way of implementing a proof-logging knapsack solving algorithm in Algorithm 1. We stress, however, that the techniques we have described are not in any way tied to this particular algorithm design. In particular, the same proof framework can be used for matrix-based dynamic programming where each weight is considered in turn, as well as for recursion with memoisation. For a matrix, more states will be created, both in the solving algorithm and in the proof, whilst for recursion the states will be constructed in an order corresponding to the recursive search execution, rather than layer by layer. Similarly, although we chose to apply (a more general version of) the maximum rule as a single pass at the end of constructing each layer, we could instead derive the appropriate implication whenever the maximum rule is used.

Until this point, we have not discussed deletions. To save memory, matrix and decision diagram approaches to dynamic programming sometimes need only keep the current and previous layers (or columns). We can do this in our proof too: when we start building layer $i \geq 3$, we can tell the proof verifier that we promise we will no longer need to access any constraint and extension variable defined in layer $i - 2$, and so these constraints may now be deleted. This will help the proof verifier use less memory, and can also speed up verification – proof steps using RUP or that introduce extension variables are not, strictly speaking, of constant complexity to verify in the worst case; we return to this in Section 4. With this caveat aside, the proofs we have written are efficient, in that we write effectively only a constant amount of data in the proof for each computation carried out by the algorithm.

3.2 A General Framework

In the same way that interleaving inference and backtrack constraints gives a general framework for proof logging for backtracking search algorithms, we are now in a position to describe how to generate proofs for dynamic programming and decision diagram algorithms. For a given problem and solving algorithm, we need to be able to do seven things.

Algorithm 1 One way of solving the knapsack problem, with proof logging, using a layer-by-layer decision diagram style construction.

```
S^0 ← {S^0_{0,0}}
for i ← 1...n do                                               // i.e. for each layer in turn
    for all S^i_{w,p} ∈ S^{i−1} do                   // i.e. for each state in the previous layer
        Extend W^i_w ⇔ ∑_{j=1}^i w_j x_j ≥ w, P^i_p ⇔ ∑_{j=1}^i p_j x_j ≤ p, and then
            S^i_{w,p} ⇔ W^i_w ∧ P^i_p if they do not already exist
        // Consider not taking item i
        S^i ← S^i ∪ {S^i_{w,p}}
        Derive W^{i−1}_w ∧ x̄_i ⇒ W^i_w and P^{i−1}_p ∧ x̄_i ⇒ P^i_p by cutting planes addition, then
            S^{i−1}_{w,p} ∧ x̄_i ⇒ S^i_{w,p} by RUP
        // Now see whether we could take item i
        if w + w_i > W then                                            // We cannot take item i
            Derive W^{i−1}_w ⇒ x̄_i by addition, then S^{i−1} ⇒ x̄_i and S^{i−1}_{w,p} ⇒ S^i_{w,p} by RUP
        else                                                            // We could take item i
            Let (w′, p′) = (w + w_i, p + p_i)
            Extend W^i_{w′} ⇔ ∑_{j=1}^i w_j x_j ≥ w′, P^i_{p′} ⇔ ∑_{j=1}^i p_j x_j ≤ p′, and then
                S^i_{w′,p′} ⇔ W^i_{w′} ∧ P^i_{p′} if they do not already exist
            S^i ← S^i ∪ {S^i_{w′,p′}}
            Derive W^{i−1}_w ∧ x_i ⇒ W^i_{w′} and P^{i−1}_p ∧ x_i ⇒ P^i_{p′} by addition, then
                S^{i−1}_{w,p} ∧ x_i ⇒ S^i_{w′,p′} and S^{i−1}_{w,p} ⇒ S^i_{w,p} ∨ S^i_{w′,p′} by RUP
    for all S^i_{w,p} ∈ S^i that is dominated by some other S^i_{w′,p′} do
        Derive S^i_{w,p} ⇒ S^i_{w′,p′} by unwrapping
        S^i ← S^i \ {S^i_{w,p}}
    Derive ∑ S^i ≥ 1 by resolving on each variable in ∑ S^{i−1} ≥ 1
    Delete every constraint created on layer S^{i−1}

if S^n is empty then
    Conclude infeasibility
else
    Log how we obtain the state with the best profit
    Derive that every S^n_{w,p} contradicts the solution-improving constraint
    Conclude optimality
```

1. Represent the problem as a set of PB inequalities and a PB objective to minimise.
2. Generate an extension variable for each new state, as it is encountered (whether that state is a node, a matrix entry, or a memoised recursive call). This is also done for infeasible states.
3. Generate an implication constraint $S' \wedge c \Rightarrow S$ linking each new state S to its predecessor S', showing that if we were in state S' and we choose a given condition c, then we arrive at this new state.
4. For any state S that is infeasible, generate a proof $S \Rightarrow \bot$ that being in this state implies contradiction. (In practice, this can sometimes be combined into the previous step instead, as we did in Equation (16).)
5. For any state S that is dominated, subsumed, or similar by a better state S', generate a proof that $S \Rightarrow S'$.
6. Show that we have considered every feasible state on a layer, or generated a complete column in a matrix, by creating an at-least-one constraint over the extension variables.
7. Derive a conclusion using the at-least-one constraint over the final layer or column.

The first requirement is generally straightforward, since the representation only needs to be correct, not useful for solving purposes. However, note that this means that our starting point is a problem, not an algorithm or a recurrence relation for solving that problem: we are certifying solutions that are found using dynamic programming, rather than specifically certifying the execution of a dynamic program. Ideally, this representation step should generally be carried out independently of how we then decide to go on and find a solution.

For the second requirement, we need to ask what kinds of state can be represented using extension variables in a *VeriPB* proof. For knapsack, the states represented a conjunction of pseudo-Boolean inequalities. However, this technique is much more general. For example, Bergman et al. [2] give an example of a decision diagram solver where states represent sets of vertices from a graph: these can be represented as conjunctions of Boolean variables, using a pair of reified inequalities to express a reified equality constraint. Similarly, we can reuse the encoding described by Gocht et al. [17] to represent anything that could be described in constraint programming terms using integer variables. It is not so obvious how to represent rational or real numbers in *VeriPB*, although in some circumstances these could be handled by scaling.

For the third requirement, if our conditions and states correspond cleanly to sets of Boolean variables then this is trivial: we are simply extending a set of inequalities by adding in additional fixed variables. For the fourth requirement, this may also be trivial, or we may need to reuse the constraint programming techniques of Gocht et al. [17] to show that a given partial state is infeasible. The sixth requirement needs only that we can show that we have indeed considered every possibility moving between layers or columns – for Boolean variables, this is immediate, whilst for encoded integer variables we can make use of the at-least-one constraint over each option. The seventh requirement comes down to showing that, given an optimal full state S and a suboptimal full state S', S' does not beat S – this should follow naturally from the objective function. For each of these requirements, we rely heavily upon the ability to cleanly wrap and unwrap reified constraints, and to reason as if reifications were not present using the technique described in Theorem 1 in the appendix. It is worth stressing that these properties, and the resulting ease of producing this kind of proof, are a specific characteristic of extended cutting planes, and they do not hold for many other proof systems.

This leaves the fifth requirement, being able to reason about dominated states. This potentially requires more creativity – and this should not be surprising, since alongside tracking states, merging states is the other feature which distinguishes dynamic programming style algorithms from backtracking search. Fortunately, the *VeriPB* proof system provides us with a suite of tools for these scenarios. In many cases, fusion resolution under implications (which, given $s \wedge r \Rightarrow \sum_i a_i x_i \geq A$ and $s \wedge \overline{r} \Rightarrow \sum_i a_i x_i \geq A'$ lets us infer that $s \Rightarrow \sum_i a_i x_i \geq \min(A, A')$ by resolving away the r) is sufficient, but *VeriPB*'s strengthening rule also allows sophisticated symmetry and dominance arguments [4].

At least so long as we are working with Booleans and integers, we have found this framework to be powerful enough for a wide range of problems. For example, weighted interval scheduling problems [25] have a natural recursive formulation using a maximum operation and sums, and dynamic programming gives a polynomial time solving algorithm. Proof logging for this problem is simpler than knapsack: the states are a simple sum, rather than a conjunction of sums.

Or, suppose we want to find the longest path in a directed acyclic graph. This also has a simple dynamic programming formulation, where nodes are visited in topological order. The longest path ending at a given node is then calculated by looking at each predecessor node and adding its longest path cost to the cost of its edge to our given node, and taking the maximum of these costs. In this case, our proof would use the costs as state variables, and rather than having two options at each transition, would be selecting between one option per incoming edge on the node. Note also that the proof process implicitly checks the correctness of the topological sort: if either the implementation were faulty, or the concept mathematically flawed (e.g. if we tried to do this in a graph with cycles), then the proof process would fail.

Of course, this does not mean that we can provide efficient proof logging for every dynamic programming or decision diagram algorithm that might ever be invented, just as it would not be reasonable to claim that efficient proof logging is definitely possible for every single backtracking search algorithm – for example, we do not yet know whether it is practically feasible to reason about real or floating point numbers in *VeriPB*. Nor does this automate the process of adding proof logging to a solver. However, in the same way that the framework of interleaving RUP backtracking steps with explicit derivations for reasoning has vastly simplified adding proof logging to a wide range of search algorithms, we can say that these techniques will vastly reduce the conceptual and implementation hurdles required to use proof logging for state- and transition-based algorithms.

3.3 Knapsack as a Constraint

We return now to knapsack, but in a more general setting. As well as being an interesting stand-alone problem, knapsack appears as a constraint in some constraint programming toolkits. Trick [34] describes a propagator for a single 0/1 integer linear inequality where the sum is a variable, whilst Fahle and Sellmann [11], Sellmann [30], Katriel et al. [24], Malitsky et al. [27], and Malitsky et al. [26] work on exactly two integer linear equalities that sum to two different variables, and do not restrict to 0/1 variables for the items. MiniZinc also defines the constraint this way [33], whilst XCSP^3 [5] allows for more than two inequalities. In all cases, the multiplier vector(s) are integer constants – sometimes these are required to be non-negative.

Propagators based upon Trick's approach can achieve either bounds or domain consistency on the sum variables, as well as domain consistency on the item variables. This is done by building a decision diagram, and then, by working from the final layer and moving backwards, deleting any nodes and edges that do not lead to a feasible state; what remains is a diagram where every path from the first layer to the final layer corresponds to a solution to the constraint. Once this is built, on some layers there may only be edges corresponding to the layer's item being accepted, or only edges corresponding to the layer's item being rejected; in this case, the associated item variable is forced.

Gocht et al. [17] described a framework for proof logging for constraint programming solvers using *VeriPB*. This framework supports integer variables, and a number of global constraints, including integer linear inequalities. To add a new constraint propagator to this framework, we must have two things. Firstly, we must be able to express the semantics of the constraint in PB form – this is trivial, because integer linear inequalities are already supported. Secondly, we must have a way of justifying all reasoning that can be carried out by its propagator. This will follow a similar pattern to proof logging for a standalone knapsack solver, but with different states and a more complicated conclusion.

For a standalone knapsack solver, recall that our states $S^i_{w,p}$ represented that the partial sum of the first i items has weight at least w, and profit at most p. For a constraint, we instead want to track states that have weight exactly w, and profit exactly p. To do this, we can introduce the four extension variables

$$W{\uparrow}^i_w \Leftrightarrow \sum_{j=1}^{i} \boldsymbol{w}_j x_j \geq w \qquad\qquad W{\downarrow}^i_w \Leftrightarrow \sum_{j=1}^{i} \boldsymbol{w}_j x_j \leq w \tag{24}$$

$$P{\uparrow}^i_p \Leftrightarrow \sum_{j=1}^{i} \boldsymbol{p}_j x_j \geq p \qquad\qquad P{\downarrow}^i_p \Leftrightarrow \sum_{j=1}^{i} \boldsymbol{p}_j x_j \leq p \tag{25}$$

which allow us to define

$$S^i_{w,p} \Leftrightarrow W{\uparrow}^i_w + W{\downarrow}^i_w + P{\uparrow}^i_p + P{\downarrow}^i_p \geq 4. \tag{26}$$

When building the structure of the proof, there are five differences.

1. We must construct implications for all four partial states, rather than just two.
2. We must bear in mind that we might be inside a backtracking search, and so some of the information we have about variables might be conditional. Fortunately this is not a concern: recall that any RUP or cutting planes proof can trivially and efficiently be extended to operate under assumptions.
3. We might be dealing with constraint programming variables whose domains are not 0/1. This means there may be more than two edges coming out of a state. To derive the implications for partial sums, we follow Gocht et al.'s approach of introducing direct variables as required, and then we use an additional cutting planes multiplication operation. We must also take care when deriving the at-least-one constraint over each layer, because this relies upon exhaustively branching. Again, this is dealt with by Gocht et al.'s framework, which allows us to obtain an at-most-one constraint for any constraint programming variable's values.
4. We may now only merge states with exact matches on weights and profits. This is true both algorithmically and in proof terms – reassuringly, if we were to forget this condition when implementing the propagation algorithm, we would quickly find it impossible to construct the appropriate implication steps in the proof.
5. We cannot delete intermediate layers as we go: we want to reason about the diagram as a whole, so it stands to reason that the structure of the diagram must remain in the proof. However, we can delete every intermediate constraint once the conclusions are derived.

Rather than establishing a proof of optimality, a knapsack propagator's proof aims to show lack of support for some variables' values. By looking at the possible weights and profits on the final layer of the decision diagram, we can recognise that either some bounds or some specific values are unsupported by the constraint; we can derive these facts inside a proof by resolving over the at-least-one constraint on the final layer. This gives us either bounds or domain consistency on the sum variables, as we prefer.

The backwards pass, which shows lack of support on the item values, is also straightforward – since our propagation algorithm works backwards from the final layer, eliminating infeasible nodes, it is sufficient to use RUP steps to show that the corresponding states must be false. Once this has been done, eliminating values from item variables also follows by RUP. This closely resembles the steps used by McIlree and McCreesh [29] to generate proofs from propagations for the regular language membership constraint.

4 Implementations and Evaluation

Before presenting the results of our empirical evaluation, it is important to ask what the purpose of such an evaluation should be. Rather than trying to implement the world's fastest dynamic programming algorithms or propagators, or even to tell you when to use these

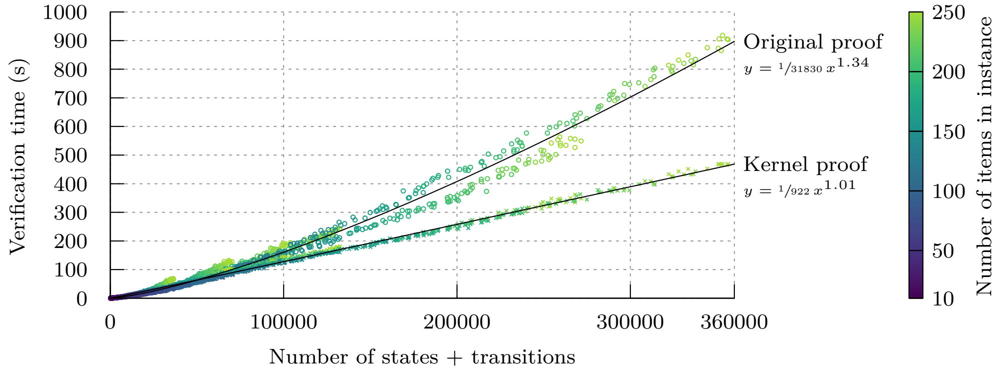

Figure 1 Verification times for knapsack problem instances with between 10 and 250 items (shown using colour). The power law fit lines show the original proof and the rewritten kernel proof times, plotted against the number of states plus transitions required to solve the instance.

techniques, the main aim of this paper is to demonstrate that *if* you choose to use these techniques, then certifying correctness using pseudo-Boolean proof logging is viable. To show this, we have implemented[1] stand-alone solvers for three problems: knapsack, longest path in a directed acyclic graph, and interval scheduling. For knapsack, we implemented both top-down and matrix-based algorithms, whilst for the other two problems we used only a matrix. With the aim of the paper in mind, our key measure of success from these implementations is that we were able to add proof logging to each solver simply by adding in statements to log information that was already present, without needing to extend or change the underlying algorithm. To validate our implementations, we tested them on a large number of randomly generated instances and were able to verify every proof produced.

Our proofs in each case are generated *efficiently*, having cost and length roughly linear in the amount of work done by the solver. However, the constant factor slowdown needed to write these proofs to disk is potentially large. Creating a new entry in a dynamic programming table for a problem such as knapsack can be extremely fast, requiring only a few additions, comparisons, and memory accesses. However, to justify an entry and the transition leading to it, we need to write several lines of text to a file. For an efficiently implemented algorithm, this can easily lead to more than an order of magnitude slowdown. This is much worse than for, e.g. SAT solving, because a CDCL solver does much more computation per proof step than a simple knapsack algorithm.

But what about proof verification time – is that also roughly linear in proof size? This turns out to be a more complex question. When using only explicit cutting planes derivations, we would expect the cost of verifying each proof step to depend only upon the number of operations. However, verifying reverse unit propagation or strengthening steps requires achieving bounds consistency over the active set of inequalities, which is not a constant-time operation. In the top line of Figure 1 we show the verification times required for 1,200 randomly generated knapsack problem instances with between 10 and 250 items, with random weights and profits both between 1 and 10, and a maximum weight of between 50 and 1000, solved using the top-down approach. (These parameters were selected to give instances where dynamic programming is a good choice of solving technique, so that we can measure the scalability of proof verification: we are trying to challenge the proof verifier, not the solver.)

[1] https://doi.org/10.5281/zenodo.12574620

We measure verification time as a function of the number of states plus transitions required to solve each instance, since this is in effect "the amount of work" the solver took to solve an instance. The fit line suggests that verification scales worse than linearly, but better than quadratically.

Similarly to how DRAT proofs can be converted to LRAT proofs, *VeriPB* is able to rewrite proofs into a simplified "kernel format" that does not require any propagations to verify: reverse unit propagation steps are rewritten to cutting planes derivations, and strengthening rule applications are also given explicit cutting planes subproofs for each proof goal [15]. Carrying out this simplification is not computationally more expensive than verifying the proof, and introduces only a small additional slowdown for outputting the rewritten proof to disk. In Figure 1 we also plot the time taken to verify these rewritten proofs, achieving the lower line. Now, the power law fit line suggests that verification time scales extremely close to linearly with proof size, with a verification rate of a little below a thousand states and transitions per second (which we expect to vary considerably based upon hardware and disk speeds). In principle, solvers could output these kernel proofs directly, avoiding the need for proof rewriting if an important concern is the initial proof verification time; however, this would require considerably more work from solver authors.

Finally, we have also implemented the knapsack constraint inside the Glasgow Constraint Solver, using a top-down construction. Our implementation supports arbitrarily many simultaneous inequalities, and is not restricted to 0/1 variables. It achieves domain consistency on every variable. Again, we were able to do this without having to restrict or alter the underlying propagation algorithm: *VeriPB* proofs are powerful enough to conveniently express the reasoning we wanted to carry out, and we did not have to design an algorithm specifically to make proof logging possible. To validate the implementation, we used the same system as other constraints in the Glasgow Subgraph Solver, where curated and randomly generated test data is combined with proof checking inside a continuous integration framework; we have successfully verified thousands of proofs in this manner. In terms of performance, any measurements are extremely sensitive to disk write speeds and to details of implementation, to the extent that using shorter variable names inside proofs can have a significant effect upon running times. However, to give indicative figures, verifying knapsack propagation proofs is typically between twenty and fifty times more expensive than producing them; this is somewhat more expensive than for some other propagators [17, 29], likely due to the large number of extension variables used in the proofs.

5 Conclusion

We have shown that the *VeriPB* proof system supports convenient and efficient proofs for a range of dynamic programming algorithms, and that it can do so regardless of whether the algorithms use a matrix, recursion and memoisation, or a top-down construction, and even when we are inside a dynamic programming propagator in a constraint programming toolkit. We saw that the cutting planes proof system makes it both natural and efficient to reason about reified linear inequalities, whilst extension variables give us the power to describe the logical relationships between states.

The knapsack propagation example showed how different conclusions could be inferred, depending upon how states were represented: when solving the knapsack problem directly, we tracked less information, thus allowing more states to be merged, whilst for constraint propagation our states were more expressive. This example could be extended further, e.g. to relaxed and restricted decision diagrams, where we are allowed to violate some constraints and only achieve a lower or upper bound rather than an exact solution. In such a setting, our ability to compose proofs and to run proofs conditional upon assumptions or guesses would be very helpful, since modern decision diagram based solvers can construct many decision diagrams during the solving process.

An interesting open question is how to extend this work to cover problems where we want to count solutions, rather than finding an optimal solution. Once a decision diagram or dynamic programming matrix has been constructed, solution counts are often easily accessible. However, this property does not immediately transfer through to proofs. In the same way that DRAT proofs can only be used to reason "without loss of satisfaction", *VeriPB* proofs establish "without loss of optimality". This means that solutions can be removed, so long it can be shown that another equally-good-or-better solution exists (for example, through symmetry or dominance breaking). We believe it is important to give solver authors the ability to write proofs that correspond precisely to the real-world problem being solved. As such, we would like to see an appropriate theoretical foundation that will allow solvers to produce proofs either for optimality reasoning or for counting, with only minimal changes that reflect the algorithmic differences needed in the two settings. We would also be interested to know whether *VeriPB* can reasonably be used to work with rational or real numbers, either by scaling or more advanced techniques.

References

1 Jeremias Berg, Bart Bogaerts, Jakob Nordström, Andy Oertel, and Dieter Vandesande. Certified core-guided MaxSAT solving. In Brigitte Pientka and Cesare Tinelli, editors, *Automated Deduction - CADE 29 - 29th International Conference on Automated Deduction, Rome, Italy, July 1-4, 2023, Proceedings*, volume 14132 of *Lecture Notes in Computer Science*, pages 1–22. Springer, 2023. doi:10.1007/978-3-031-38499-8_1.

2 David Bergman, André A. Ciré, Ashish Sabharwal, Horst Samulowitz, Vijay A. Saraswat, and Willem Jan van Hoeve. Parallel combinatorial optimization with decision diagrams. In Helmut Simonis, editor, *Integration of AI and OR Techniques in Constraint Programming - 11th International Conference, CPAIOR 2014, Cork, Ireland, May 19-23, 2014. Proceedings*, volume 8451 of *Lecture Notes in Computer Science*, pages 351–367. Springer, 2014. doi:10.1007/978-3-319-07046-9_25.

3 David Bergman, André A. Ciré, Willem-Jan van Hoeve, and John N. Hooker. *Decision Diagrams for Optimization.* Artificial Intelligence: Foundations, Theory, and Algorithms. Springer, 2016. doi:10.1007/978-3-319-42849-9.

4 Bart Bogaerts, Stephan Gocht, Ciaran McCreesh, and Jakob Nordström. Certified dominance and symmetry breaking for combinatorial optimisation. *J. Artif. Intell. Res.*, 77:1539–1589, 2023. doi:10.1613/JAIR.1.14296.

5 Frédéric Boussemart, Christophe Lecoutre, and Cédric Piette. XCSP3: an integrated format for benchmarking combinatorial constrained problems. *CoRR*, abs/1611.03398, 2016. arXiv:1611.03398.

6 Randal E. Bryant. Tbuddy: A proof-generating BDD package. In Alberto Griggio and Neha Rungta, editors, *22nd Formal Methods in Computer-Aided Design, FMCAD 2022, Trento, Italy, October 17-21, 2022*, pages 49–58. IEEE, 2022. doi:10.34727/2022/ISBN.978-3-85448-053-2_10.

7 Samuel R. Buss and Jakob Nordström. Proof complexity and SAT solving. In Armin Biere, Marijn J. H. Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, chapter 7, pages 233–350. IOS Press, 2nd edition, February 2021.

8 Kevin K. H. Cheung, Ambros M. Gleixner, and Daniel E. Steffy. Verifying integer programming results. In Friedrich Eisenbrand and Jochen Könemann, editors, *Integer Programming and Combinatorial Optimization - 19th International Conference, IPCO 2017, Waterloo, ON, Canada, June 26-28, 2017, Proceedings*, volume 10328 of *Lecture Notes in Computer Science*, pages 148–160. Springer, 2017. doi:10.1007/978-3-319-59250-3_13.

9 Chiu Wo Choi, Warwick Harvey, J. H. M. Lee, and Peter J. Stuckey. Finite domain bounds consistency revisited. In *AI 2006: Advances in Artificial Intelligence, 19th Australian Joint Conference on Artificial Intelligence, Hobart, Australia, December 4-8, 2006, Proceedings*, pages 49–58, 2006. doi:10.1007/11941439_9.

10 Luís Cruz-Filipe, Marijn J. H. Heule, Warren A. Hunt Jr., Matt Kaufmann, and Peter Schneider-Kamp. Efficient certified RAT verification. In Leonardo de Moura, editor, *Automated Deduction - CADE 26 - 26th International Conference on Automated Deduction, Gothenburg, Sweden, August 6-11, 2017, Proceedings*, volume 10395 of *Lecture Notes in Computer Science*, pages 220–236. Springer, 2017. doi:10.1007/978-3-319-63046-5_14.

11 Torsten Fahle and Meinolf Sellmann. Cost based filtering for the constrained knapsack problem. *Ann. Oper. Res.*, 115(1-4):73–93, 2002. doi:10.1023/A:1021193019522.

12 Allen Van Gelder. Verifying RUP proofs of propositional unsatisfiability. In *International Symposium on Artificial Intelligence and Mathematics, ISAIM 2008, Fort Lauderdale, Florida, USA, January 2-4, 2008*, 2008. URL: http://isaim2008.unl.edu/PAPERS/TechnicalProgram/ISAIM2008_0008_60a1f9b2fd607a61ec9e0feac3f438f8.pdf.

13 Stephan Gocht. *Certifying Correctness for Combinatorial Algorithms: by Using Pseudo-Boolean Reasoning*. PhD thesis, Lund University, Sweden, 2022. URL: https://lup.lub.lu.se/record/3550cb96-83d5-4fc7-9e62-190083a3c10a.

14 Stephan Gocht, Ross McBride, Ciaran McCreesh, Jakob Nordström, Patrick Prosser, and James Trimble. Certifying solvers for clique and maximum common (connected) subgraph problems. In Helmut Simonis, editor, *Principles and Practice of Constraint Programming - 26th International Conference, CP 2020, Louvain-la-Neuve, Belgium, September 7-11, 2020, Proceedings*, volume 12333 of *Lecture Notes in Computer Science*, pages 338–357. Springer, 2020. doi:10.1007/978-3-030-58475-7_20.

15 Stephan Gocht, Ciaran McCreesh, Magnus O. Myreen, Jakob Nordström, Andy Oertel, and Yong Kiam Tan. End-to-end verification for subgraph solving. In Michael J. Wooldridge, Jennifer G. Dy, and Sriraam Natarajan, editors, *Thirty-Eighth AAAI Conference on Artificial Intelligence, AAAI 2024, Thirty-Sixth Conference on Innovative Applications of Artificial Intelligence, IAAI 2024, Fourteenth Symposium on Educational Advances in Artificial Intelligence, EAAI 2014, February 20-27, 2024, Vancouver, Canada*, pages 8038–8047. AAAI Press, 2024. doi:10.1609/AAAI.V38I8.28642.

16 Stephan Gocht, Ciaran McCreesh, and Jakob Nordström. Subgraph isomorphism meets cutting planes: Solving with certified solutions. In Christian Bessiere, editor, *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020*, pages 1134–1140. ijcai.org, 2020. doi:10.24963/ijcai.2020/158.

17 Stephan Gocht, Ciaran McCreesh, and Jakob Nordström. An auditable constraint programming solver. In Christine Solnon, editor, *28th International Conference on Principles and Practice of Constraint Programming, CP 2022, July 31 to August 8, 2022, Haifa, Israel*, volume 235 of *LIPIcs*, pages 25:1–25:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPICS.CP.2022.25.

18 Stephan Gocht and Jakob Nordström. Certifying parity reasoning efficiently using pseudo-boolean proofs. In *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Thirty-Third Conference on Innovative Applications of Artificial Intelligence, IAAI 2021, The Eleventh Symposium on Educational Advances in Artificial Intelligence, EAAI 2021, Virtual Event, February 2-9, 2021*, pages 3768–3777. AAAI Press, 2021. doi:10.1609/AAAI.V35I5.16494.

19 Marijn Heule, Warren A. Hunt Jr., and Nathan Wetzler. Trimming while checking clausal proofs. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, pages 181–188. IEEE, 2013. URL: https://ieeexplore.ieee.org/document/6679408/.

20 Marijn Heule, Warren A. Hunt Jr., and Nathan Wetzler. Verifying refutations with extended resolution. In Maria Paola Bonacina, editor, *Automated Deduction - CADE-24 - 24th International Conference on Automated Deduction, Lake Placid, NY, USA, June 9-14, 2013. Proceedings*, volume 7898 of *Lecture Notes in Computer Science*, pages 345–359. Springer, 2013. doi:10.1007/978-3-642-38574-2_24.

21 John N. Hooker. Generalized resolution for 0-1 linear inequalities. *Ann. Math. Artif. Intell.*, 6(1-3):271–286, 1992. doi:10.1007/BF01531033.

22 John N. Hooker. Decision diagrams and dynamic programming. In Carla P. Gomes and Meinolf Sellmann, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, 10th International Conference, CPAIOR 2013, Yorktown Heights, NY, USA, May 18-22, 2013. Proceedings*, volume 7874 of *Lecture Notes in Computer Science*, pages 94–110. Springer, 2013. doi:10.1007/978-3-642-38171-3_7.

23 Toni Jussila, Carsten Sinz, and Armin Biere. Extended resolution proofs for symbolic SAT solving with quantification. In Armin Biere and Carla P. Gomes, editors, *Theory and Applications of Satisfiability Testing - SAT 2006, 9th International Conference, Seattle, WA, USA, August 12-15, 2006, Proceedings*, volume 4121 of *Lecture Notes in Computer Science*, pages 54–60. Springer, 2006. doi:10.1007/11814948_8.

24 Irit Katriel, Meinolf Sellmann, Eli Upfal, and Pascal Van Hentenryck. Propagating knapsack constraints in sublinear time. In *Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence, July 22-26, 2007, Vancouver, British Columbia, Canada*, pages 231–236. AAAI Press, 2007. URL: http://www.aaai.org/Library/AAAI/2007/aaai07-035.php.

25 Antoon W.J. Kolen, Jan Karel Lenstra, Christos H. Papadimitriou, and Frits C.R. Spieksma. Interval scheduling: A survey. *Naval Research Logistics (NRL)*, 54(5):530–543, 2007. doi:10.1002/nav.20231.

26 Yuri Malitsky, Meinolf Sellmann, and Radoslaw Szymanek. Filtering bounded knapsack constraints in expected sublinear time. In Maria Fox and David Poole, editors, *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2010, Atlanta, Georgia, USA, July 11-15, 2010*, pages 141–146. AAAI Press, 2010. doi:10.1609/AAAI.V24I1.7560.

27 Yuri Malitsky, Meinolf Sellmann, and Willem Jan van Hoeve. Length-lex bounds consistency for knapsack constraints. In Peter J. Stuckey, editor, *Principles and Practice of Constraint Programming, 14th International Conference, CP 2008, Sydney, Australia, September 14-18, 2008. Proceedings*, volume 5202 of *Lecture Notes in Computer Science*, pages 266–281. Springer, 2008. doi:10.1007/978-3-540-85958-1_18.

28 Ross M. McConnell, Kurt Mehlhorn, Stefan Näher, and Pascal Schweitzer. Certifying algorithms. *Comput. Sci. Rev.*, 5(2):119–161, 2011. doi:10.1016/J.COSREV.2010.09.009.

29 Matthew J. McIlree and Ciaran McCreesh. Proof logging for smart extensional constraints. In Roland H. C. Yap, editor, *29th International Conference on Principles and Practice of Constraint Programming, CP 2023, August 27-31, 2023, Toronto, Canada*, volume 280 of *LIPIcs*, pages 26:1–26:17. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPICS.CP.2023.26.

30 Meinolf Sellmann. Approximated consistency for knapsack constraints. In Francesca Rossi, editor, *Principles and Practice of Constraint Programming - CP 2003, 9th International Conference, CP 2003, Kinsale, Ireland, September 29 - October 3, 2003, Proceedings*, volume 2833 of *Lecture Notes in Computer Science*, pages 679–693. Springer, 2003. doi:10.1007/978-3-540-45193-8_46.

31 Carsten Sinz and Armin Biere. Extended resolution proofs for conjoining bdds. In Dima Grigoriev, John Harrison, and Edward A. Hirsch, editors, *Computer Science - Theory and Applications, First International Symposium on Computer Science in Russia, CSR 2006, St. Petersburg, Russia, June 8-12, 2006, Proceedings*, volume 3967 of *Lecture Notes in Computer Science*, pages 600–611. Springer, 2006. doi:10.1007/11753728_60.

32 Steven Skiena. *The Algorithm Design Manual, Third Edition.* Texts in Computer Science. Springer, 2020. doi:10.1007/978-3-030-54256-6.

33 Peter J. Stuckey, Kim Marriott, and Guido Tack. The MiniZinc handbook section 4.2.1: Global constraints, 2023. URL: https://www.minizinc.org/doc-2.5.3/en/lib-globals.html.
34 Michael A. Trick. A dynamic programming approach for consistency and propagation for knapsack constraints. *Ann. Oper. Res.*, 118(1-4):73–84, 2003. doi:10.1023/A:1021801522545.
35 Nathan Wetzler, Marijn Heule, and Warren A. Hunt Jr. DRAT-trim: Efficient checking and trimming using expressive clausal proofs. In Carsten Sinz and Uwe Egly, editors, *Theory and Applications of Satisfiability Testing - SAT 2014 - 17th International Conference, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, volume 8561 of *Lecture Notes in Computer Science*, pages 422–429. Springer, 2014. doi:10.1007/978-3-319-09284-3_31.

A Proofs Under Implications

In various pseudo-Boolean (PB) proof logging projects, it has been useful to rely on the assumption that if we have an efficient proof procedure for deriving a constraint D from a set of constraints F, then we can convert this into an efficient procedure for deriving $R \Rightarrow D$ from the set of constraints $\{R \Rightarrow C : C \in F\}$ for some conjunction of literals R. In this appendix we formalise and generalise this property, showing that efficient cutting-planes proofs can be "unrestricted" to construct analogous efficient proofs where the premises and conclusion are subject to (potentially different) conditions using reification.

A.1 Notation

A *(partial) assignment* is a (partial) function from variables to $\{0, 1\}$; we extend an assignment ρ from variables to literals in the natural way by respecting the meaning of negation, and for literals ℓ over variables x not in the domain of ρ, denoted $x \notin \mathrm{dom}(\rho)$, we use the convention $\rho(\ell) = \ell$. For notational convenience, we can also view ρ as the set of literals $\{\ell : \rho(\ell) = 1\}$ assigned true by ρ. Applying ρ to a constraint $C = \sum_i a_i \ell_i \geq K$ yields

$$C\restriction_\rho \doteq \sum_{\ell_i : \rho(\ell_i) = \ell_i} a_i \ell_i \geq K - \sum_{\ell_j \in \rho(\ell_j) = 1} a_j \qquad (27)$$

substituting literals as specified by ρ. We extend this notation to applying assignments to F in the natural way $F\restriction_\rho = \bigcup_{C \in F} C\restriction_\rho$.

We will write $\mathit{Vars}(C)$, $\mathit{Vars}(F)$, $\mathit{Lits}(C)$ and $\mathit{Lits}(F)$ to denote the sets of variables or literals appearing in a PB constraint C or formula F.

A.2 Constructing Proofs Under Implications

We can now state our main result in its general form.

▶ **Theorem 1.** *Let F be a PB formula over n variables, ρ be a partial assignment, and suppose that from $F\restriction_\rho$ we can derive a constraint D using a cutting planes and RUP derivation of length L. Then we can construct a derivation of length $\mathrm{O}(n \cdot L)$ from F of the constraint*

$$\bigwedge_{\ell \in \rho} \ell \Rightarrow D. \qquad (28)$$

In what follows, we assume all constraints are normalised. We will first show the following.

▶ **Lemma 2.** *For any PB constraint C and partial assignment ρ, we can always derive $\bigwedge_{\ell \in \rho} \ell \Rightarrow C\restriction_\rho$ from C using a cutting planes derivation of length $\mathrm{O}(|\mathit{Vars}(C)|)$.*

Proof. First, let us write C as

$$\sum_{\substack{\ell_i \in Lits(C):\\ \rho(\ell_i)=\ell}} a_i \ell_i + \sum_{\substack{\ell_j \in Lits(C):\\ \rho(\ell_j)=1}} b_j \ell_j + \sum_{\substack{\ell_k \in Lits(C):\\ \rho(\ell_k)=\ell}} c_k \ell_k \geq K. \tag{29}$$

Then, if we let $B = \sum_{\substack{\ell_j \in Lits(C):\\ \rho(\ell_j)=1}} b_j$, we note that $C\restriction_\rho$ is the constraint

$$\sum_{\substack{\ell_i \in Lits(C):\\ \rho(\ell_i)=\ell}} a_i \ell_i \geq K - B \tag{30}$$

and $\bigwedge_{\ell \in \rho} \ell \Rightarrow C\restriction_\rho$ is the constraint

$$\sum_{\substack{\ell_j \in Lits(C):\\ \rho(\ell_j)=1}} (K-B)\ell_j + \sum_{\substack{\ell_k \in Lits(C):\\ \rho(\ell_k)=0}} (K-B)\ell_k + \sum_{\substack{\ell_i \in Lits(C):\\ \rho(\ell_i)=\ell}} a_i \ell_i \geq K - B \tag{31}$$

To derive Equation (31) from Equation (29) we can proceed as follows.

1. For all j, add the literal axioms amounting to $b_j \bar{\ell}_j \geq 0$ to Equation (29) yielding

$$\sum_{\substack{\ell_k \in Lits(C):\\ \rho(\ell_k)=\ell}} c_k \ell_k + \sum_{\substack{\ell_i \in Lits(C):\\ \rho(\ell_i)=\ell}} a_i \ell_i \geq K - B \tag{32}$$

2. Saturate to ensure that for all k, $c_k \leq K - B$.
3. Add literal axioms $\ell_k \geq 0$ and $\bar{\ell}_j \geq 0$ as needed to obtain Equation (31).

This amounts to at most one weakening step per variable appearing in C, along with one saturation step, and hence has length $\mathrm{O}(|Vars(C)|)$. ◀

We are now able to prove the main result.

Proof. Let $\pi = (D_1, \ldots, D_L = D)$ be the derivation of D from $F\restriction_\rho$, and denote by π_s the set $\{D_1, \ldots, D_{s-1}\}$ of constraints prior to derivation step s. Each D_s is one of the following:

- An axiom (constraint in $F\restriction_\rho$).
- A literal axiom.
- The result of a cutting planes operation, with antecedents in π_s.
- A RUP constraint with respect to $F\restriction_\rho \cup \pi_s$.

We will proceed by structural induction on π and show that for any D_s we can construct a length $O(n \cdot s)$ derivation that $\bigwedge_{\ell \in \rho} \ell \Rightarrow D_s$ from F.

For the base cases, we consider an axiom $D_a \in F\restriction_\rho$. We must have some constraint $C \in F$ such that $C\restriction_\rho = D_a$. Hence we can derive C as an axiom, and then by Lemma 2 we can derive $\bigwedge_{\ell \in \rho} \ell \Rightarrow C\restriction_\rho$, i.e. $\bigwedge_{\ell \in \rho} \ell \Rightarrow D_a$, in $O(|Vars(C)|) \subseteq O(n)$ steps. Note that if D_a is instead a literal axiom then $\bigwedge_{\ell \in \rho} \ell \Rightarrow D_a$ is also a literal axiom, because the reification coefficients will all be zero.

Now assume for any non-axiom constraint D_s we have already constructed a derivation of length $O(n \cdot (s-1))$ deriving all the constraints in $\pi'_s = \left\{\bigwedge_{\ell \in \rho} \ell \Rightarrow D_i : D_i \in \pi_s\right\}$. We now consider different cases depending on how D_s was derived in π.

Case 1: D_s is the result of adding two constraints $D_i, D_j \in \pi_s$.

Then by assumption $\bigwedge_{\ell \in \rho} \ell \Rightarrow D_i$, and $\bigwedge_{\ell \in \rho} \ell \Rightarrow D_j$ have already been derived. If we let K_i and K_j be the degrees of D_i and D_j respectively, we can write these in the form

$$\sum_{\ell \in \rho} K_i \bar{\ell} + D_i \tag{33}$$

and

$$\sum_{\ell \in \rho} K_j \bar{\ell} + D_j, \tag{34}$$

and so adding these together yields

$$\sum_{\ell \in \rho} (K_i + K_j)\bar{\ell} + D_s. \tag{35}$$

If K_s is the degree of D_s, note that we must have $K_s \leq K_i + K_j$, since cancellation of matching literals when adding D_i and D_j can only reduce the degree of their sum. Hence if we apply saturation to Equation (35) we obtain $\sum_{\ell \in \rho} K_s \bar{\ell} + D_s$, i.e. $\bigwedge_{\ell \in \rho} \ell \Rightarrow D_s$, as required.

Case 2: D_s is result of multiplying a constraint $D_i \in \pi_s$ by a scalar λ.
Then by assumption $\bigwedge_{\ell \in \rho} \ell \Rightarrow D_i$ has already been derived, and again we can write this as

$$\sum_{\ell \in \rho} K_i \bar{\ell} + D_i \tag{36}$$

where K_i is the degree of K_i. If we multiply this by λ we obtain

$$\sum_{\ell \in \rho} \lambda K_i \bar{\ell} + \lambda D_i \tag{37}$$

which is precisely $\bigwedge_{\ell \in \rho} \ell \Rightarrow D_s$, as required.

Case 3: D_s is the result of dividing a constraint $D_i \in \pi_s$ by a scalar λ.
Then again by assumption $\bigwedge_{\ell \in \rho} \ell \Rightarrow D_i$ has already been derived, and this time we will write this in full as

$$\sum_{\ell \in \rho} K_i \bar{\ell} + \sum_j a_j \ell_j \geq K_i. \tag{38}$$

If we divide this by λ we obtain

$$\sum_{\ell \in \rho} \lceil (K_i/\lambda) \rceil \bar{\ell} + \sum_j \lceil a_j/\lambda \rceil \ell_j \geq \lceil (K_i/\lambda) \rceil, \tag{39}$$

which is precisely $\bigwedge_{\ell \in \rho} \ell \Rightarrow D_s$, as required.

Case 4: D_s is the result of applying saturation to a constraint $D_i \in \pi_s$.
Once again by assumption $\bigwedge_{\ell \in \rho} \ell \Rightarrow D_i$ has already been derived, and we can write this in full as above in Equation (38). After applying saturation to this we obtain

$$\sum_{\ell \in \rho} \min(K_i, K_i)\bar{\ell} + \sum_j \min(a_j, K_i)\ell_j \geq K_i. \tag{40}$$

which is precisely $\bigwedge_{\ell \in \rho} \ell \Rightarrow D_s$, as required.

Case 5: D_s is the result of applying weakening (adding literal axioms) to a constraint $D_i \in \pi_s$.
In this case we can view the added literal axioms as another degree-0 constraint D_j, which we can always derive, and so the fact we can obtain $\bigwedge_{\ell \in \rho} \ell \Rightarrow D_s$ follows immediately from Case 1.

Case 6: D_s is a RUP constraint.

Write $D_s = \sum_i a_i \ell_i \geq K$ and let $A = \sum_i a_i$. Then $\bigwedge_{\ell \in \rho} \ell \Rightarrow D_s$ is the constraint

$$\sum_{\ell \in \rho} K\bar{\ell} + \sum_i a_i \ell_i \geq K, \tag{41}$$

and its negation is

$$\sum_{\ell \in \rho} K\ell + \sum_i a_i \bar{\ell}_i \geq A + 1 + (|\rho| - 1)K. \tag{42}$$

We can see that for Equation (42) to be satisfied, all the reification literals $\ell \in \rho$ must be set to true. Recalling that all constraints in $\pi'_s = \left\{\bigwedge_{\ell \in \rho} \ell \Rightarrow D_i : D_i \in \pi_s\right\}$ are all assumed to have been previously derived, we can see that performing unit propagation will reduce constraints in $F \cup \pi'_s \cup \neg(\bigwedge_{\ell \in \rho} \ell \Rightarrow D)$ to be precisely the constraints in $F\lceil_\rho \cup \pi_s \cup \neg D$. Since by assumption deriving D_s from $F\lceil_\rho \cup \pi_s$ by RUP was a legitimate derivation step, continued unit propagation on the constraint database must result in a contradiction. Hence we can derive $\bigwedge_{\ell \in \rho} \ell \Rightarrow D$ from $F \cup \pi'_s$ as a single RUP step.

In all of these cases, we only need a constant number (at most two) proof steps, to derive $\bigwedge_{\ell \in \rho} \ell \Rightarrow D_s$, from what was assumed to already be derived, and so by starting from the axioms and applying induction we can construct a derivation which includes all of the constraints in $\pi'_L = \left\{\bigwedge_{\ell \in \rho} \ell \Rightarrow D_i : D_i \in \pi\right\}$ and in particular our desired $\bigwedge_{\ell \in \rho} \ell \Rightarrow D_L$.

Since each of the L constraints in π'_L requires at most $O(n)$ intermediate derivation steps, our constructed derivation has length at most $O(n \cdot L)$. ◀

With Theorem 1 established we easily obtain the following useful corollary.

▶ **Corollary 3.** *Let F be a PB formula over n variables and let R be a set of literals over distinct variables not appearing in F (i.e. for any $\ell \in R$, $\bar{\ell} \notin R$ and $\ell \notin Lits(F)$). Then let $R(F)$ be a set of reified constraints $\{R_C \Rightarrow C : C \in F\}$, where each reifying term R_C is a conjunction of literals in R.*

Then, if we can derive a constraint D from F using a cutting planes and RUP derivation of length L, we can construct a derivation of length $O(L \cdot n)$ of the constraint $\bigwedge_{C \in F} R_C \Rightarrow D$ from $R(F)$.

Proof. Take the partial assignment ρ setting $\ell = 1$ for each $\ell \in R$ and apply Theorem 1. ◀

Finally, we conclude with a closer look at when the $O(n \cdot L)$ worst case in Theorem 1 will actually occur.

▶ **Observation 4.** *In practice, we can often consider the length of the constructed derivation in Theorem 1 to be $O(L)$ rather than $O(n \cdot L)$. This is because the $O(n)$ overhead occurs only in the base case when transforming an axiom from the initial formula to the required form by adding literal axioms (n in the worst case) and saturating as described in Lemma 2. We can achieve the same transformation in $O(1)$ steps when a syntactic implication rule is implemented, as is the case for the VeriPB proof checker. This automatically checks that literal axioms can be added to a previously derived constraint to obtain a specified constraint.*