Pseudo-Boolean Proof Logging for Constraint Propagation Algorithms

Matthew John McIlree

Submitted in fulfilment of the requirements for the Degree of
Doctor of Philosophy

School of Computing Science
College of Science and Engineering
University of Glasgow

March 2026

Abstract

Proof logging is a way to increase trust in the conclusion reached by an algorithm. Alongside any answer, a proof logging algorithm should automatically output a mathematical certificate of correctness in a machine-checkable format. This can be of critical importance in areas where the results of algorithms have safety implications; are of academic mathematical significance; or simply have tangible effects on people's lives.

This thesis introduces new proof logging techniques in the area of *constraint programming* (CP). CP is a powerful algorithmic paradigm for modelling and solving problems expressed in terms of variables; possible values for those variables; and constraints on allowed combinations of values. The field has matured over the past several decades to the point where general-purpose solvers are routinely used to tackle large real-world problem instances efficiently. However, certification has remained a challenge and currently very few CP solvers implement proof logging. Part of the reason for this is the difficulty of capturing in a proof system the expressive, high-level reasoning techniques used by specialised *constraint propagation* algorithms. Expressivity and convenience need to be carefully balanced against trustworthy and efficient checking.

The main objective of this work is to investigate the extent to which *pseudo-Boolean* (PB) *reasoning* can be used to develop feasible proof-logging versions of state-of-the art constraint propagation algorithms. We use the *VeriPB* proof checker and its associated PB proof system based on 0-1 linear inequalities and cutting planes derivations, augmented with convenience and strengthening rules. *VeriPB* has previously been used in certification for a number of combinatorial solving paradigms, including for some constraint propagation and search techniques. We refine and build on this foundation, developing new PB proof logging methods for a number of important constraint propagators, with a particular focus on smart table, regular language membership, Hamiltonian circuit, and ternary multiplication constraints. Many of the techniques we develop can be generalised. They establish that, in principle, a wide variety of CP propagators are amenable to PB certification. We also implement our methods within a proof-logging CP solver, and present empirical evaluation in terms of logging overhead and checking time.

This work opens a clear avenue to a future where solvers are more trustworthy and auditable. It also gives us fresh insights into the applied capabilities of PB reasoning, adding new techniques to the PB justification repertoire.

Contents

List of Figures

Acknowledgements

First and foremost, I would like to thank my supervisor Ciaran McCreesh, for consistent support, advice, programming mentorship, and generally for introducing me to this topic and the world of research. Thank you also to my second supervisor David Manlove and the FATA research group at Glasgow for all the encouragement, and for creating an excellent environment to learn and work in. Special mention to Arthur Gontier and Philip Rodgers who provided many interesting discussions, technical support, and proofreading of my thesis draft.

Beyond Glasgow, I would like to thank Jakob Nordström for all the very enjoyable collaborations, and for affording me some amazing research opportunities and experiences in Lund, Copenhagen, at Dagstuhl, and in Banff. Thank you also to the many other collaborators involved in the various *VeriPB*-related projects, and in particular thank you to Adrian Rebola-Pardo and Wietze Koops for giving me very detailed expert feedback on some of my chapters.

To Peter Stuckey and Jip Dekker, and everyone I met while visiting Monash, thanks for a very fun eight weeks and for changing my perspectives on constraints and programming. Similarly, thanks to Tias Guns and friends in Belgium for hosting me on an excellent visit to Leuven.

Lastly on the academic side, thank you to my examiners Guido Tack and Michele Sevegnani for an enjoyable discussion in the viva, and for a thorough review of this document.

I am extremely fortunate to have been supported and encouraged throughout my PhD (and life in general) by my wonderful partner, family, and friends. To Mum, Dad, and Joseph, and of course to Sarah, thank you for everything. I can't hope to name everybody here that I'm grateful for, but a specific thank you also goes to Neel Mackinnon and John Hobrough for giving me their perspectives on PhD work and thesis writing from other areas of academia.

Finally, I would like to dedicate this thesis to the memory of Sam Myers (1999–2023): brilliant mathematician, organist, flatmate, and friend. Sam suggested before I started all this to think about whether I enjoy maths because it's easy, or because it's hard. I'm still not sure that I've figured out the answer.

Declaration

I certify that the thesis presented here for examination for a PhD degree of the University of Glasgow is solely my own work other than where I have clearly indicated that it is the work of others (in which case the extent of any work carried out jointly by me and any other person is clearly identified in it) and that the thesis has not been edited by a third party beyond what is permitted by the University's PGR Code of Practice.

I declare that the thesis does not include work forming part of a thesis presented successfully for another degree. I declare that this thesis has been produced in accordance with the University of Glasgow's Code of Good Practice in Research.

I acknowledge that if any issues are raised regarding good research practice based on review of the thesis, the examination may be postponed pending the outcome of any investigation of the issues.

Chapter 1

Introduction

> *"Suppose, for the moment, we did have such a powerful routine that when we gave the machine the Riemann Hypothesis it finally gave out some 500 pages of closely printed detail purporting to be a proof. Suppose further, that we have an independent routine called a 'theorem-proof checker.' Such a routine would be much easier to write than the first one. And suppose the alleged proof passed the second test. Would that constitute a proof of the theorem?"*
>
> — Richard Hamming, *The Mechanisation of Science*, 1961

1.1 Solving Problems with Constraints

Many important real-world tasks are naturally expressed as *constraint satisfaction* or *constraint optimisation* problems (CSPs/COPs). They can be understood in terms of a set of variables; a set of corresponding *domains* (possible values) for those variables; and a set of *constraints* restricting the combinations of values that the variables can simultaneously take. Examples include rostering [40], scheduling [100], routing [130], planning [186], configuration [69], placement [197], and resource allocation problems [114] — applications that can often be safety-critical and have a tangible effect on people's lives.

Take the task of designing a simplified 5-day shift roster for a group of four doctors. Each shift slot for each doctor might be either type 0 (off), type 1 (day shift) or type 2 (night shift). We could represent this problem as 20 *variables*, one for each shift slot each taking a value in the set $\{0, 1, 2\}$. For example, "$\mathsf{shift}_{1,3} = 2$" could represent doctor 1 being assigned to night shift on day 3.

It is easy to imagine the kind of requirements that might be imposed here:

- *Each day there must be at least 2 doctors working day shift.*
- *Each day there must be at least 1 doctor working night shift.*
- *A doctor may not work a night shift followed immediately by a day shift, or vice versa.*
- *A doctor may not work more than 3 consecutive days.*

	Day 1	Day 2	Day 3	Day 4	Day 5
Doctor 1	1	1	0	1	1
Doctor 2	2	0	1	1	1
Doctor 3	1	1	1	0	2
Doctor 4	0	2	2	2	0

Figure 1.1: Example valid shift schedule, where 1 = day, 2 = night, and 0 = off.

If we wrote down these rules as mathematical *constraints* on the shift_{ij} variables, then a valid roster would be characterised by an *assignment* to the variables that *satisfies* these constraints.

Now, in a small example such as this, the problem seems friendly enough that it could be solved by hand, similar to a kind of Sudoku puzzle. But if we scaled up to hundreds of doctors and shifts, and more numerous and complex constraints, it would quickly become painful. It would be particularly difficult to find an *optimal* solution with respect to some quantity to maximise or minimise, or determine that some combination of constraints are incompatible, excluding all possible solutions. For example, finding the solution that minimises number of night shifts worked by certain doctors, or determining that it is not possible for any doctor to have a certain number of days in a row off while respecting all other the constraints.

The discipline of *constraint programming* (CP) is concerned with *modelling* (precisely representing) problems such as this and solving them with general-purpose computer programs called *solvers*. It can be considered a subfield of artificial intelligence or automated reasoning, depending on how these terms are defined, since the solver generally receives a description of a problem, but not explicit instructions on *how* to solve it. The field has matured over the last few decades to the point where state-of-the-art software can efficiently decide satisfiability and optimality for an extensive variety of large-scale instances. This is despite theoretical computer science categorising the general (discrete, finite) constraint satisfaction problem as NP-*complete*. This means, very informally, that they are among the hardest in the class of computational problems where we know how to check a solution "quickly". In practice, solving is achieved through the use of advanced search techniques, as well as specialised constraint *propagation* algorithms that remove impossible values from variables' domains.

1.2 The Point of Writing Proofs

Unfortunately, the complexity of modern CP solvers can make their reasoning inscrutable. And concerningly, solvers will sometimes return the *wrong answer*. This is despite solving problems with a precise mathematical definition, using algorithms that have a well-defined notion of correctness. So a system might claim *unsatisfiable* when a solution exists; *optimality* of a solution when a *better solution* exists; or exhibit an assignment that *does not in fact satisfy the constraints*

as required. This fact is documented in academic publications, presentations, and as part of issue reports in solver repositories.

> *"There have been three wrong-answer bugs found in released versions of Minion since the introduction of metamorphic testing."* [4, pg. 8]
>
> *"CPLEX's solution to instance 3 is apparently incorrect."* [199, pg. 41]
>
> *"In MZC 2014 one third of the solvers provided at least an unsound answer."* [5, pg. 6]
>
> *"In MZNC 2015 sunny-cp checked HaifaCSP, [. . .] This allowed sunny-cp to detect 21 incorrect answers"* [6, pg. 8]
>
> *"Conversely, Opturion and OR-Tools solvers provided a lot of incorrect, and unfortunately unchecked, answers."* [6, pg. 9]
>
> *"the organisers enabled the solutions checking of G12/LazyFD, HaifaCSP, Mistral, Opturion, OR-Tools. This allowed sunny-cp to detect 19 incorrect answers."* [6, pg. 10]
>
> *"Experiments carried out using Choco JaCoP and MiniCP revealed the presence of numerous non-trivial bugs, no matter how carefully the test suites of these solvers have been engineered."* [85, pg. 1]
>
> *"In the 2021 MiniZinc challenge, at least 45 out of 3,500 claimed solutions were incorrect [. . .] and previous years saw similar rates."* [94, pg. 1]
>
> *"Out of all bugs discovered, 13 bugs were soundness bugs, 5 of which had their origin in backend solvers. In particular, we found 2 soundness bugs in the OR-tools solver and three in the MiniZinc system."* [192, pg. 13]
>
> See Appendix A for a sample of issue reports where open-source solvers apparently gave wrong answers.

This is not especially surprising to anyone who has experience developing software. Bugs in both algorithm design and implementation (and in interactions between the two) are a common occurrence. And even if they were not, computer hardware is itself not infallible, susceptible to design or degradation-related faults [167], and even interference from cosmic rays [202].

For constraint programming, occasional undetected wrong answers are often tolerable. The use of a solver is simply about getting a solution that is "good enough" or "approximately right". However, in certain applications, errors could be extremely problematic or even morally unacceptable. Examples include areas that have life-altering consequences, such as kidney exchange [57], or where absolute mathematical correctness is intrinsic to the task, such as searching for new classes of combinatorial objects [122]. There are also cases where a CP solver can be used as part of verifying the correctness of another software or hardware tool [48, 147]; and here, the possibility of incorrect results undermines the whole process. And when optimising

to obtain a good-enough result that is not necessarily proven optimal, even off-by-one errors in an internal subproblem could amplify to result in huge discrepancies in the final objective.

It is worth noting that finding case studies of when wrong answers have actually had a direct negative impact in industry is difficult, since CP systems are in practice often deeply integrated with other tools [69], among code that is proprietary and not publicly available. We would not necessarily expect companies to publish accounts of situations where commercial software making use of CP was incorrect and led to personal or financial consequence. But we can surmise from the kind of areas where CP is used, along with the available evidence of occasional solver inaccuracy, that this has been and continues to be a problem.

Regardless, even if a particular solver were always 100% correct, and its authors were completely confident of this, there would still be the challenge of convincing sceptical users that apparent "black-box" results are reliable.

The standard approach to this in software engineering is to perform extensive testing. There has been much research into how to effectively test CP solvers, including performing "metamorphic" or "fuzz" testing to randomly mutate solver inputs and check that certain properties are preserved after solving [4, 85, 192]. This can be very effective at finding bugs. Indeed, testing methodologies are responsible for identifying many of the wrong answers mentioned in the examples above.

But we should not be content with testing alone, as famously argued by Edsger Dijkstra [63]:

> *"Besides the notion of productivity, also that of quality control continues to be distorted by the reassuring illusion that what works with other devices works with programs as well. It is now two decades since it was pointed out that* ***program testing may convincingly demonstrate the presence of bugs, but can never demonstrate their absence****. After quoting this well-publicized remark devoutly, the software engineer returns to the order of the day and continues to refine his testing strategies, just like the alchemist of yore, who continued to refine his chrysocosmic purifications."*

To expand on this: testing only shows that software works for the inputs used *in the test cases*, under the specific conditions they were tested under, and not accounting for unpredictable events affecting hardware. This is particularly salient for CP systems, since unlike simpler computer programs, the space of possible inputs is huge, so there is absolutely no hope of testing every possible scenario in advance. For some users, a promise that fairly extensive testing has been performed might be satisfactory. But for others, stronger guarantees of software reliability would be highly desirable.

One way to get a stronger mathematical guarantee of correctness of a piece of software is to *formally verify* the source code itself. There are various ways of achieving this, within the broad field of *formal methods* [185]. At a high level, formal verification of software usually starts by defining a precise mathematical *specification* of how a program should behave. Then tools such as *model checkers* [47] or *theorem provers* [159, 26, 172] can be used to rigorously

prove that a concrete implementation meets this specification. This might involve verification of existing code by treating it as a mathematical object. Alternatively, formally verified code may be extracted from the verification process itself, generating an implementation that is "correct by construction" [135].

Unfortunately, writing code using a theorem prover can be extremely difficult and time-consuming. Although formal methods have delivered very promising results in other areas, they are yet to be able to directly tackle constraint problems with the full power of methods employed by modern solvers. There has been work on creating a formally verified CP solver by extracting executable code from specifications using the Coq (now Rocq) proof assistant [38, 66]. This has been successful at capturing the notions of variables and domains, and certain limited kinds of constraints and propagation. But it is still far from being able to certify the techniques and specialised propagators used by the state of the art, and far from achieving comparable performance.

So aside from testing or formal verification, what other approaches are available? Of course, a simple idea that can be effective in practice is running multiple solvers and seeing whether they agree. Doing this and getting differing answers demonstrates that at least one of the solvers must be wrong, and majority agreement among many independent solver implementations will increase some confidence in the result [6]. But this does not guard against multiple solvers being wrong in the same way, and it is not particularly useful when only one solver is actually capable of solving the problem.

An alternative approach, and the subject of this thesis, is *proof logging*. This is the approach where alongside any answer, a solver should output a *proof* of correctness that can be checked line-by-line by an external program. In a way, it generalises the method of verifying the output of one solver by running another one, by having the second "solver" use a detailed output from the first solver as its input. The proof should be produced incrementally during the solving process by *justifying* individual reasoning steps, hence "logging". This makes it an auditable record of the steps taken to obtain the result, as well as a verifiable certificate that the result is correct. In principle, proof logging will work providing that the proof language used is *sound* — it should be mathematically impossible to have a correct proof of an incorrect answer for a given input. And in practice, for maximum trust in the system, the proof-checking program should be simple enough to be trusted directly, or be itself amenable to formal verification.

This remedies several of the issues with the other approaches to reliability. Although it does not show that a solver is free of bugs, it does guarantee that a wrong answer can *always* be detected. This is regardless of whether it is due to a software bug, hardware failure, design flaw, mathematical error, or sporadic interference. Furthermore, correct answers can be forever accompanied by a persistent certificate. So even if a problem was later discovered with a solver, the answers remain trustworthy, and the certificate would in principle be available forever for sceptical users to independently check.

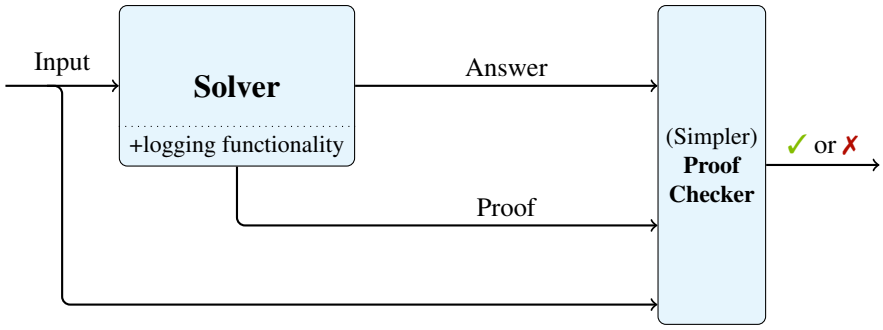


Figure 1.2: Proof logging at a high level.

Beyond certifying correctness, a proof log also provides a "trace" of execution of a solver, since it breaks down the process into reasoning steps following what the solver actually did. This is a useful source to mine for debugging and for insights into solver behaviour. The point at which an incorrect proof fails may lead to some insight into where the solver went wrong.

Proof logging falls within the framework of *certifying algorithms*: the philosophy that a procedure itself should be responsible for producing any additional information required to validate its output. This idea was first defined explicitly in 2011 by McConnell et al. [149], but arguably was understood in the 1990s via "certification trails" for fault-tolerant computing [182], and earlier still as a feature of foundational methods in operations research via "Farkas' certificates" [178]. Going back even further, certain mathematical algorithms defined long before the invention of computers, such as the extended Euclidean algorithm, could be argued to meet the criteria to be considered certifying.

Notwithstanding this, the current incarnation of certifying algorithms — as a method of verifying results from exact solvers — is mostly a development of the last two decades. It has been most prominent within the field of *Boolean satisfiability* (SAT), a specialised solving paradigm related to but distinct from general constraint programming. At the time of writing, SAT solvers can produce proofs as standard, and proof logging is seen as broadly successful. This same level of success is yet to be replicated in CP.

One reason for this is the difficulty of expressing higher-level CP constraints and reasoning mechanisms not present in SAT within a self-contained, usable proof language. A fundamental tradeoff in any proof logging system is between ease of logging and ease of verification. It is well understood how to instrument a CP solver to output *some* kind of "trace" file that closely documents exactly the steps it took, which can be useful for analysis and debugging purposes [3]. But to make this log a *certificate* of correctness by any reasonable definition, we require extremely strong mathematical guarantees from our independent checking program. This in turn will place restrictions on what kind of logging statements we can allow. Richer descriptions will require more complex checking logic, which is likely to be less trustworthy, less easy to write formally verified code for, and potentially slower per proof step. On the other hand, allowing only very simple operations as proof steps might make it very difficult or even

impossible to express sophisticated solver techniques within a log of manageable size. On top of this, it is (arguably) desirable for the logging format to use a *unified* proof language. This means that there should be a standard way of building representations of solver concepts, and general proof rules that operate on this standard representation. This ensures that new solver features should not require re-engineering of the proof system and checker, and different solvers that have different representations and features can still use the same standard format.

To put these points in their proper context, it will be useful to explain this area in some more detail and outline the development of proof logging over the last 25 years, particularly as it relates to SAT and CP.

1.3 A Short History of Proof Logging

In logic, *Boolean variables*, also called *propositional variables* or *truth variables*, are symbols that can only be associated with values *true* or *false*. We can construct statements about truth variables using logical connectives "**and**", "**or**" and "**not**". For example,

$$(x \textbf{ and } y) \textbf{ or } (z \textbf{ and not } x). \tag{1.1}$$

This expression is *satisfied* if we say $x =$ *false*; $y =$ *true*; and $z =$ *true*, because this respects the meaning of the logical connectives, i.e. "either both x and y are true, or z is true and x is false". A mapping of truth values to variables that is allowed by an expression in this way is called a *satisfying assignment*, and in general an expression is *satisfiable* if there exists at least one satisfying assignment, and *unsatisfiable* otherwise. It is standard to write $\vee$ instead of **or**; and $\wedge$ instead of **and**; and put a small bar above a variable x to denote negation, so $\bar{x}$, instead of "**not** x". This gives us a way to write compact *Boolean formulas*,

$$(x \wedge y) \vee (z \wedge \bar{x}) \tag{1.2}$$

Within a Boolean formula, variables x and their negations $\bar{x}$ are referred to collectively as *literals*. An expression of the form $\ell_1 \vee \cdots \vee \ell_n$, where $\ell_1, \ldots, \ell_n$ are literals, is called a *clause*, and it expresses a *disjunction*: at least one of these literals must be true. For example, $x \vee \bar{y} \vee p \vee \bar{q}$ says that at least one of ("x is *true*", "y is *false*", "p is *true*", "q is *false*") must be the case in a satisfying assignment. If we connect a collection of clauses with "$\wedge$", we create a "conjunction of disjunctions": a Boolean formula saying that *all* the clauses must be simultaneously satisfied. Such formulas are said to be in *conjunctive normal form*.

Determining whether a Boolean formula in conjunctive normal form (CNF) can be satisfied is the quintessential NP-complete problem [49]. This can be viewed as a very restricted type of constraint satisfaction problem, one where only 0-1 variables are allowed, and the only constraint type is clauses. It is known simply as the *Satisfiability* problem or "SAT" problem for short.

Although it appears at first glance to be a restricted case, it encompasses the general satisfiability problem for Boolean formulas. This is because an arbitrary formula can be converted to a CNF formula whose size is not unreasonably larger than the original (more formally, within a linear factor), while preserving a mapping from its solutions back to the original solutions [188].

Despite its computational hardness in theory, automated SAT solving has enjoyed considerable success in practice. The *Davis-Putnam-Logemann-Loveland* (DPLL) algorithm was created in 1960 by Davis et al. [54, 55], who also built one of the first implementations. This algorithm formed the basis for most practical SAT solving programs over the years, and the first SAT competition for these solvers took place in 1992 [125]. With the advent of the *conflict-driven clause learning* (CDCL) paradigm in 1996 [146], a multitude of substantial performance improvements allowed solvers to tackle progressively larger CNF formulas [125]. These could scale to hundreds, thousands, and even millions of variables and clauses, and hence were increasingly used as a component of automated problem-solving systems in massive industrial applications [145].

But improving performance meant increasingly complex software, so naturally there were concerns about the reliability of these solvers. By the early 2000s, with an annual international SAT competition established and a myriad of important applications in operation, a desire for verification had become apparent.

When a CNF formula is declared satisfiable, it is straightforward for a solver to produce a "witness" satisfying assignment that can be efficiently verified as such. It is simply a case of checking that in each clause, there is at least one Boolean literal set to true. In contrast, it is not so clear how to convince someone that a formula is *unsatisfiable* (UNSAT). Some kind of proof should be required.

One possible means to achieve this is to construct a sequence of clauses that are *implied* by the original formula. A clause C is implied by a formula F if every assignment that satisfies F also satisfies C. The idea is that the act of adding each implied clause should be a "*derivation*" that is easy to check, and then when a clause is added that is obviously, trivially unsatisfiable, this shows that the original formula had no solutions. It should be completely convincing to say that if a formula F and a formula G are *equisatisfiable* — either they are both satisfiable or both unsatisfiable — and G contains the trivially-false empty clause, then F simply must be unsatisfiable.

To make this possible, we need one or more *proof rules* for introducing clauses while preserving satisfiability. An example of such a rule, first introduced formally in 1965 [173], is the *resolution rule*. It can be applied if there are two clauses in a formula with a variable that appears *negated* (negatively) in the first and *not-negated* (positively) in the second. When this is the case, say for a variable x, a new clause called the "*resolvent*" can be derived by taking the disjunction of all literals, other than x and $\bar{x}$, appearing in either clause.

If C can be derived from F using the resolution rule, it must be the case that F *implies* C. Thus, satisfiability is preserved (F and $F \wedge C$ are equisatisfiable). Hence, a sequence of

resolvents, ending with the empty clause, provides a valid proof for an UNSAT claim. This is known as a *resolution refutation*.

In 2002, Van Gelder [79] described how a DPLL-based SAT solver could emit such refutations, demonstrating the first independently-checkable proof-logging methodology for SAT solvers. Following this, Zhang and Malik [200] implemented the first verifier for their proof-logging solver *Chaff* in 2003.

Resolution proofs were known to have theoretical limitations, however. In 1985 Haken [101] had shown that CNF encodings of the classic "Pigeonhole Principle" would require exponential-length proofs using standard resolution. Furthermore, the process of constructing each resolvent was thought to require maintaining a directed acyclic graph (DAG) of clauses, which could be difficult to implement and grow prohibitively large.

A parallel approach, outlined also in 2003 by Goldberg and Novikov [97], made use of the well understood notion of *unit propagation* from the DPLL algorithm. Unit propagation relies on the basic observation that if a clause (called a *unit clause*) contains only a single literal, that literal must be true in any satisfying assignment. So we can simplify a formula by iteratively setting any literals that appear in unit clauses to true until either a conflict is reached or no unit clauses remain.

It is clear that unit propagation produces equisatisfiable formulas. It can also be shown that it is *confluent*: it will always reach the same fixed point regardless of the order clauses are considered in. So when a conflict (the empty clause) is reached by performing unit propagation on a formula F, the formula has been shown to be UNSAT; and this can be externally checked by running the same polynomial-time procedure.

A key observation for turning this into a derivation rule is that if F is UNSAT, i.e. false under every assignment, then the logical negation $\neg F$ is a *tautology*, i.e. true under every assignment (we use $\neg$ to denote formula or clause negation to distinguish from literal negation). If we wish to show F and $F \wedge C$ are equisatisfiable, it is sufficient to show that $\neg F \vee C$ is a tautology, and hence sufficient to show we can reach a conflict on unit propagation from $\neg(\neg F \vee C)$, which is equivalent to $F \wedge \neg C$. This forms the basis of the *reverse unit propagation* (RUP) rule of inference. A clause is *RUP* with respect to a formula if adding its negation to the formula and running unit propagation results in a conflict.

For example, the clause $(\bar{x} \vee y)$ is RUP with respect to the formula

$$(y \vee \bar{z}) \wedge (\bar{x} \vee \bar{w}) \wedge (z \vee w). \tag{1.3}$$

To see this, observe that the negation $\neg(\bar{x} \vee y)$ is $x \wedge \bar{y}$, i.e. two unit clauses, and if we set x and

$\bar{y}$ to *true* (and hence $\bar{x}$ and y to false) in the formula we get

$$(\mathit{false} \vee \bar{z}) \wedge (\mathit{false} \vee \bar{w}) \wedge (z \vee w), \tag{1.4}$$

$$\text{which simplifies to } (\bar{z}) \wedge (\bar{w}) \wedge (z \vee w). \tag{1.5}$$

This now contains two further unit clauses: $\bar{z}$ and $\bar{w}$; so we continue unit propagation and set $\bar{z}$ and $\bar{w}$ to *true*, and hence z and w to *false*, to yield

$$(\mathit{true}) \wedge (\mathit{true}) \wedge (\mathit{false} \vee \mathit{false}). \tag{1.6}$$

This last formula contains the clause $(\mathit{false} \vee \mathit{false})$, which is equivalent to the empty clause — a contradiction.

Goldberg and Novikov demonstrated a proof logging methodology by which a SAT solver outputs a sequence of RUP clauses, ending with an assertion that the empty clause is RUP, and also implemented a checker containing an efficient unit propagation procedure. By 2008 a similar idea had been implemented by Biere in the checker program *TraceCheck* [28] for the proof-logging solver *PicoSAT* [29], while Van Gelder defined a general proof format, based on the same principles, simply called *RUP* [80, 81]. The methodology became known as *clausal* proof logging, as it allowed authors of CDCL-type solvers to "retrofit" verification by simply outputting learned clauses. In theory, any fact learned by a basic CDCL SAT solver could be expressed as a RUP clause, and so this was a complete proof logging method.

Even with these improvements, along with an increased enthusiasm for "certifying algorithms" more generally [149], many SAT solvers remained non-certifying by the time of 2011 SAT Competition. While there was a Certified UNSAT track, where most solvers emitted RUP-style clausal proofs, the main track did not require formal proofs of UNSAT claims, and uncertainty over incorrect answers remained a problem [124]. This was partly due to the fact that the sophistication of solvers was continuing to grow, with extensive *preprocessing* and even *inprocessing* techniques that were either difficult or even impossible to efficiently justify with RUP alone [190, 181]. Indeed, from a proof complexity perspective, RUP is no more powerful than resolution, and so also does not allow for short proofs of "pigeonhole"-type problems.

One way to make a proof system stronger, in terms of what can be refuted efficiently, is to introduce an *extension* rule. This essentially allows introducing statements of the form $y \Leftrightarrow F$ (meaning "y is true if and only if F is true") where F is an arbitrary Boolean formula and y is a "fresh" variable not appearing anywhere previously in the derivation or starting formula [36]. In the specific case of *extended resolution*, introducing three clauses of the form

$$\bar{x} \vee p \quad \bar{x} \vee q \quad \bar{p} \vee \bar{q} \vee x \tag{1.7}$$

is allowed, where p, q are literals and x is a new variable not appearing in the input formula.

These are equivalent to $x \Leftrightarrow p \wedge q$.

Extended resolution was introduced by Tseitin in 1968 [98], and shown to allow polynomial-length proofs of the pigeonhole principle encodings by Cook and Reckhow in 1979 [50], making it exponentially stronger than resolution. Whilst proof logging for SAT making use of this system had already been explored by Jussila et al. in 2006 [128], it was not until later that a more general methodology leveraging it was introduced.

In 2013 Heule et al. [106] proposed a proof system based on the *resolution asymmetric tautology* (RAT) property [126] which generalises both the RUP rule and the extension rule for resolution. Unlike resolution and RUP, a clause derived by this rule is not necessarily *implied* by the axioms and derived clauses. Instead, it is simply *redundant* with respect to them, meaning equisatisfiability is still preserved. Although this somewhat goes against the standard intuition of a mathematical proof being a series of logical consequences starting from an initial premise, it is sufficient to allow for a certificate of unsatisfiability, which was the main focus at the time.

Together with the addition of clause deletion information (previously introduced in a proof system called *DRUP*) [107, 112], the RAT rule became the foundation of the *deletion resolution asymmetric tautology* (DRAT) proof system. Not only was this able to justify most of the sophisticated techniques cutting-edge SAT solvers were using, it was relatively straightforward for authors to integrate into existing work, similar to the RUP system, and was efficient to check with a formally-verified verifier [196]. Part of the efficiency was due to *proof trimming*: reading a proof backwards and only checking the clauses actually needed for the conclusion. This led to it quickly becoming a *de facto* standard for SAT proof logging, and by 2014 proofs were mandatory for solvers tackling UNSAT instances in the SAT competition [21].

Since the widespread adoption of DRAT, trust in SAT solvers' claims of unsatisfiability has increased dramatically. In recent years, disqualifications from the SAT competition due to incorrect answers have dropped to negligible levels [103], and SAT solvers have even become accepted by the mathematical community as having settled certain unsolved conjectures [104, 109, 138].

Although at this point proof logging for SAT solving could be considered a mature field, the problem of formally justifying *all* techniques that modern SAT solvers employ had not been entirely resolved. While new proof formats, such as LRAT [52], GRIT [53], and FRAT [13] were proposed from 2017 onwards to decrease the unit propagation burden on the proof checker, and to aid formally verified checking, certain solver capabilities remained beyond the scope of any proof format. Foremost among these uncertified techniques was *symmetry breaking*: the process of exploiting known mathematical symmetries in a problem to bypass some amount of search [61]. Some restricted cases were known to be possible to prove within DRAT [108], but in general it was not clear whether general symmetry breaking for SAT solving would be expressible efficiently within this format. *Parity reasoning* over XOR constraints was another example where the viability of DRAT proofs was unclear [168].

Furthermore, the question of how to replicate the success of DRAT proof logs for combinatorial solving paradigms beyond classic Boolean satisfiability was gaining prominence. What if someone wished to maximise an *objective function* over the variables of the formula, as in the well studied domain of Maximum Satisfiability (MaxSAT) solving [12]? Or satisfy or optimise over *pseudo-Boolean* constraints, i.e. inequalities over linear sums of $0-1$ variables [33]? Or indeed, expand to non-Boolean variables and make use of general *constraint programming* techniques [175]?

The desire to certify constraint solving in a rigorous way has existed for a long time, but proposed solutions have consistently struggled when faced with the scale and complexity of the software. In 2010, Veksler and Strichman described a preliminary proof-producing CP solver that supported some constraints, but required a complex proof system with constraint-specific parametric inference rules for every different propagator [193]. They also did not provide a formally verified checker for their proof system. Later, Gange et al. [75] made a serious effort to develop a proof logging approach for a hybrid CP-SAT solver, using DRUP proofs as a "skeleton" and relying on external specialised checkers for constraint-specific reasoning, although this was never formally published. This, again, would require an extension of the proof framework for any new constraint types.

The difficulties in adapting proof logging for constraint programming were mirrored in the parallel efforts towards standardising proof production for "Satisfiability Modulo Theories" (SMT) solvers. These are yet another kind of solver, which use SAT methods in combination with solvers for specialised background "theories" such as linear real arithmetic [18, 17]. While there have been significant advances in proof production for SMT in recent years [116, 16, 113], standardisation and trusted checking remain challenging [15]. In particular, different checking mechanisms for different theory solvers are required and there are, as yet, no complete unified proof formats.

Going back to the fundamentals of proof logging philosophy, an ideal proof format:

- is simple enough for a feasible formally verified checker;
- has a flexible unified representation than can represent different aspects of a potentially heterogenous solver;
- is nevertheless based on a powerful proof system so that strong solver reasoning can be expressed using proofs of a reasonable length.

Starting from 2019, a new line of work towards developing a proof system aiming to satisfy all these criteria emerged.

An initial insight was centred on the *cutting planes* proof system [51] which operates on $0-1$ integer linear inequalities, a.k.a. *pseudo-Boolean* constraints. This had long been studied in proof complexity theory as an implicationally-complete proof system stronger than resolution but still relatively simple. Somewhat surprisingly, it appeared to be able to express compactly various kinds of solver reasoning relatively far removed from pseudo-Boolean representations, and in a

way that was efficiently checkable.

A first version of a proof system involving cutting planes augmented with generalised RUP steps was described by Elffers et al. [67] in 2020. This came with a verifier program called "*VeriPB*" [1], which defined a compact format for PB constraints based on the standard OPB file format, along with a format for cutting planes steps based on *reverse Polish notation*. The idea was that an OPB *model file* would describe a given combinatorial problem in *pseudo-Boolean form*, and then a *VeriPB proof file* would specify derivations of either RUP or cutting planes steps.

Elffers et al. demonstrated that the proof system could justify the reasoning performed for an AllDifferent CP constraint, which says the values taken by a set of variables must all be different [170]. In the same year Gocht et al. presented *VeriPB* justification methods for state-of-the-art *clique* [91] and *subgraph isomorphism* [92] solvers. Following this, in 2021, proof logging for the previously mentioned *parity reasoning* over XOR (exclusive-OR) constraints was addressed [89], although this required introduction of a new rule in *VeriPB* to allow for the use of extension variables. This rule generalised the formulation of *clause* redundancy for SAT formulas given in 2017 by Heule et al. [110, 111], and inspired including the notion of *substitution redundancy*, first proposed as a theoretical extension of DRAT by Buss and Thapen [37].

With this in place, it became possible to justify many kinds of strong reasoning efficiently within the *VeriPB* proof system. Another application demonstrated by Gocht et al. [93] in 2022 was showing how the same system of "cutting planes with RUP and redundancy" could certify CNF translations of pseudo-Boolean constraints. There was also the question of *optimality* proofs, which required a further modification to the proof system. PB optimisation, MaxSAT and COPs provided a strong motivation for a unified system to support this.

The primary changes required to turn a system for proofs of unsatisfiability into one for proofs of optimality are simple. First, the input for the proof needs to include a (pseudo-Boolean) objective function to minimise over some set of variables. Then, when a solution is found by a combinatorial algorithm it should be logged in the proof with a witness assignment. This is interpreted by the verifier as a prompt to both check that the claimed solution unit propagates to an actual solution for the PB model, and also to post a constraint that says any future solutions logged must evaluate to an even smaller objective value. When the solver eventually finds no better solution, it can proceed with a usual proof of unsatisfiability, making the whole proof a rigorous demonstration that "this is a solution, and nothing better is possible".

While this adaptation makes optimality proofs possible within *VeriPB*, the ultimate goal of developing a unified proof system for a diverse set of combinatorial optimisation techniques meant further changes were needed. For certified symmetry-breaking, Bogaerts et al. adapted the substitution redundancy rule into a "redundance-based strengthening" rule, and "dominance-based strengthening rule" [32]. They also specified further rules to make the proof system more efficient and usable, including conditions for deleting derived constraints (deletion) and working

with orders. This brings us (almost) to the version of pseudo-Boolean proofs (now known simply as "the *VeriPB* proof system") in use at the time of writing, and which will be described in detail in Chapter 2. Since it was fully defined, it has been successfully applied in a certified symmetry-breaking SAT solver [31]; in certified MaxSAT solving [191, 23, 123, 24]; certified classical planning [64], and from 2023 onwards has been used as an alternative official format and proof checker for the SAT Competition [7].

Of greatest interest from the point of view of this thesis is the 2022 "auditable constraint solver" methodology of Gocht et al. [94], which uses *VeriPB* to certify reasoning performed by a basic constraint programming solver. This was the first published work on proof logging for constraint programming since Veksler and Strichman's, and the first to demonstrate a methodology that was applicable to a variety of global constraints without requiring modification to the proof system.

Applications of *VeriPB* have benefited from the development of a formally-verified checker *CakePB* for a subset of the proof-format [7]. In 2023 the main *VeriPB* checker was adapted to *elaborate* to this "kernel" format — both checking the proof and converting any unsupported steps using supported rules. This allowed for *end-to-end verification* for subgraph solving [95], where a high level logic description of a graph problem was translated to PB constraints, and then a *VeriPB* proof for the conclusion was output by a specialised graph solver. Since the PB translation and proof checker were both formally verified, this allowed for formal guarantees for a high-level algorithm from the problem description all the way to a final answer.

Concurrent with the work presented for this thesis, another line of research on proof logging for constraint programming also emerged. This was published in 2024 as a "multi-stage proof logging framework" for CP solvers [70]. This involves optimisation of the logging process during solving by outputting just a proof "scaffold" and then post-processing this into a full proof for checking. In the published implementation, the scaffold is translated to pseudo-Boolean proofs using the methods of Gocht et al. and checked using *VeriPB*, but research into developing a specialised formally-verified CP proof checker is ongoing [184]. At the time of writing, the latter part the of work is still in its early stages, and supports checking for a very limited range of propagation algorithms. Hence, a direct comparison with PB-based methods is not yet possible.

1.4 Overview of This Thesis

The basic premise of this thesis is to investigate whether pseudo-Boolean reasoning can be used to support a proof logging CP solver. Expressed as a thesis statement:

> *Pseudo-Boolean proof logging provides a complete and practical means to certify and audit a wide range of modern constraint propagation techniques.*

We will explain fully what is meant by "pseudo-Boolean proof logging" in Chapters 2 and 3. For now, the fundamental point that allows it to be applied it to CP is that the input CSP/COP

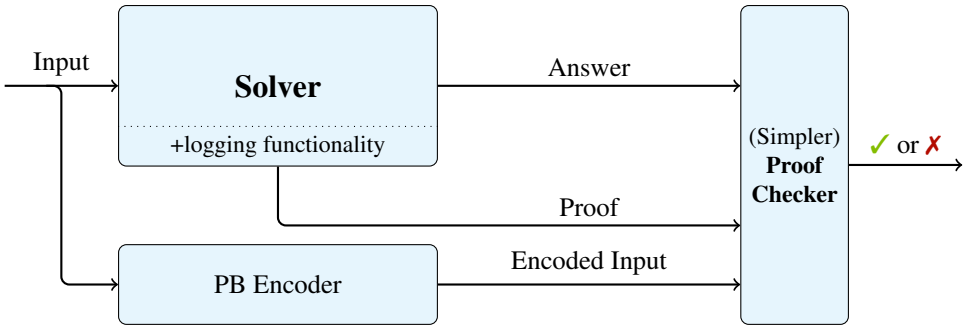


Figure 1.3: Adapted proof logging methodology for constraint programming.

should be first *compiled* or *encoded* to a pseudo-Boolean format. This means, essentially, that before logging any proof steps, we first produce a set of 0-1 linear constraints (and optional objective function) — a *pseudo-Boolean model* — that can be understood to faithfully represent *the same* problem as the one passed to the solver. Then, if we have a proof system able to formally certify conclusions (optimality, unsatisfiability), with respect to this model, we can treat these as applying to the original CP problem — providing, of course, we trust the encoding process. Similar to the philosophy of trusting a proof checker, we need an encoder that is simple enough to be directly trusted or else formally verified.

Importantly, none of this means that the solver itself must become a PB solver, or only use PB reasoning. It can still use its native CP solving methods; and only the proof logging functionality needs to depend on the encoding. As originally argued by Gocht et al. [94], a proof logging CP solver can create a complete proof of unsatisfiability or optimality providing it can *justify* the reasoning performed by constraint propagation algorithms using pseudo-Boolean proof steps.

This was shown to be possible for several fundamental propagators, but whether it is practically achievable for *all* constraint programming techniques is an open question. This is worth exploring, because if we are able to capture everything in a simple, unified system with a formally verified checker, we gain several important advantages for long term trustworthy solving. We avoid having to extend the checker and format for every new technique, and can focus on having a relatively small, efficiently-engineered verification pipeline. This in turns provides a better hope for a standardised format that can be adopted in different solvers, avoiding the situation of multiple competing heterogenous formats, as seen previously in the SMT community [116]. We would also expect a low-level proof logging format that uses a unified mathematical language to be ultimately more successful for reasoning that operates *globally* (i.e. over the whole problem representation), such as breaking symmetries during search [68] and automatic problem reformulation [158].

The range of constraint programming techniques is vast, even when restricted just to techniques for propagation. We therefore limit the scope of this thesis to the following research topics.

R1. Expanding and more rigorously defining the methodology of Gocht et al., making ex-

plicit the properties required from encodings and proof logging propagation algorithms to guarantee correct proof production.

R2. Developing proof logging methods for efficiently justifying important propagation algorithms for specific constraints types not covered by previous approaches.

R3. Extracting more general proof logging methods from ideas developed for R2 to show, in principle, that larger families of constraint types can be efficiently justified.

R4. Implementing proof logging methods within a proof-of-concept CP solver, and demonstrating empirical evidence that they work on a feasible timescale.

For the purposes of this thesis, a "correct" proof logging method is one which will *always* produce a proof that will be validated by the proof checker, and should do so in all cases of the propagation algorithms under consideration, regardless of the input parameters. There will therefore be a focus on proving, on paper, that proof logging methods always result in a correct proof, when implemented correctly with a correct solver. This will be complemented by experimental evidence that the proofs produced indeed pass checking.

An "efficient" proof logging method is somewhat more difficult to define. There will necessarily always be some overhead to adding proof logging to a solver, although the exact increase in solving time could be very implementation-dependent. Disk-write speeds, file formats, and logging mechanism could all conceivably have an impact on the exact overhead observed in practice. For this work, we will focus on designing methods that log a number of steps proportionate (by a linear or small polynomial factor) to the number of steps performed by the propagation algorithm. This will be made clear using explicit algorithm descriptions, and there will not be a focus on formal complexity or proof complexity theory via reductions or similar techniques.

Also not in scope will be methods for optimising the logging and checking processes themselves. We will not explore optimisations such as binary formats, or "scaffolding" and proof reconstruction as proposed by Flippo et al. [70] and Reeves et al. [169]. We will also not place focus on *proof trimming* — the process of reading a proof backwards from the conclusion and only checking steps that are actually needed. This idea has been key to the success of proof checkers for SAT formats [112], and very recently has been explored for *VeriPB* proofs too [2]. The work in this thesis should not conflict with any of these optimisations, and our contributed justification methods should hopefully be mutually beneficial with improved logging and checking mechanisms.

The primary contributions of this thesis will be structured as follows:

Part 1: A Certification Framework (Chapter 2 and Chapter 3) In this part we will review the necessary fundamentals of proofs based on pseudo-Boolean reasoning and how it can be applied to constraint programming. Most of Chapter 2 will be reviewing previous work, but we will specifically contribute explicit proofs and formal statements for various

properties that were previously "folklore" or referred to only in passing. In Chapter 3 we will contribute the expansion and explication of the framework of Gocht et al. [94], which we then use for the remainder of the thesis.

Part 2: General Justification Procedures In each of Chapters 4 and 5 we will begin with a particular recognised constraint type — *smart table* and *regular language membership* constraints respectively — and show how propagation for this constraint can be justified within our chosen proof framework. These are entirely novel, and not covered by any previous certification approaches. We will then argue that the proof logging methods developed for this constraint are somehow more broadly applicable, demonstrating for the first time a wider variety of constraints that can have certifying propagators. Each of these is accompanied by a reference implementation and experiments to demonstrate checking validity; and measure proof logging overhead and checking time.

Part 3: Specialised Justification Procedures In part 3, we will devise specialised justification methods for two constraint types that present particular challenges for the chosen proof framework. Chapter 6 is a detailed treatment of proof logging for the constraint enforcing that variables represent a *Hamiltonian circuit* in a graph, while Chapter 7 deals with fundamental arithmetic constraints of the form $X \times Y = Z$, where X, Y and Z are all variables.

Some of the work in Chapters 4 and 5 previously appeared in the conference publication

[151] 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)*, volume 280 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 26:1–26:17, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-95977-300-3. doi:10.4230/LIPIcs.CP.2023.26.

The work in Chapter 6 will be an expanded version of the conference publication

[153] Matthew J. McIlree, Ciaran McCreesh, and Jakob Nordström. Proof Logging for the Circuit Constraint. In Bistra Dilkina, editor, *Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 38–55, Cham, 2024. Springer Nature Switzerland. ISBN 978-3-031-60599-4. doi:10.1007/978-3-031-60599-4_3.

And the work in Chapter 7 will be an expanded version of the conference publication

[152] Matthew J. McIlree and Ciaran McCreesh. Certifying Bounds Propagation for Integer Multiplication Constraints. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pages 11309–11317, 2025.

The author of this thesis was the lead author and primary contributor for all of the above.

Additionally, some of the discussion and the worked example in the latter part of Chapter 5 is based on the conference publication

[59] Emir Demirović, Ciaran McCreesh, Matthew J. McIlree, Jakob Nordström, Andy Oertel, and Konstantin Sidorov. Pseudo-Boolean Reasoning About States and Transitions to Certify Dynamic Programming and Decision Diagram Algorithms. In Paul Shaw, editor, *30th International Conference on Principles and Practice of Constraint Programming, CP 2024, September 2-6, 2024, Girona, Spain*, volume 307 of *LIPIcs*, pages 9:1–9:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2024. doi:10.4230/LIPICS.CP.2024.9.

This was also co-authored by the author of this thesis, but not as the lead author or primary contributor. The work in the appendix of this paper was the primary contribution of the author of this thesis, and this is re-presented here as part of Chapter 2.

1.5 Notation and Prerequisites

In this thesis we are assuming a background (to an undergraduate level) in general computer science and discrete mathematics. In particular, we will assume familiarity with the basic notations of functions, sets and relations, and some fundamental graph-theoretic concepts such as vertices, directed and undirected edges, trees. We will also sometimes refer to very basic concepts from the study of computational complexity, such as *linear/polynomial/exponential* procedures, "big-O" notations.

We will not require knowledge of any particular programming languages, but we will make use of pseudocode with standard imperative programming constructs such as **if**, **while**, **for**, and **switch**, along with variables, tuples, sets, and functions. As part of this, we will use $\leftarrow$ to denote object naming/assignment, and occasionally use this multiple times for clarity. So

$$\mathsf{MyNamedObject} \leftarrow \sum a_i x_i \geq b \leftarrow \mathsf{MyFunction}() \tag{1.8}$$

means $\mathsf{MyFunction}()$ returns (some representation of) a linear inequality with coefficients a_i and left-hand side b, and this will henceforth be assigned the name $\mathsf{MyNamedObject}$.

Part I

A Certification Framework

Chapter 2

Pseudo-Boolean Reasoning

A *Boolean function* is a function of the form $f : \{0,1\}^n \to \{0,1\}$. If we instead allow the output to be a real number, we have a general *pseudo-Boolean* function with Boolean inputs and non-Boolean outputs. Any constraint which restricts the output of a pseudo-Boolean function could rightly be called a *pseudo-Boolean constraint*, but in practice we will reserve the term for constraints defined by inequalities such as

$$\sum_{i=1}^{n} a_i x_i \geq b, \qquad \text{where } a_1, \ldots, a_n, \text{ and } b \text{ are all integers.} \tag{2.1}$$

Even in this restricted case, pseudo-Boolean constraints are expressive and associated with powerful reasoning. This chapter will be dedicated to setting up the necessary definitions, proof rules, and properties involving constraints of this form that are required throughout the rest of the thesis. More details on the standard material in Section 2.1 can be found in Chapter 7 of the Handbook of Satisfiability (second edition) [36], while a formal description of the proof system outlined in Section 2.2 is given by Bogaerts et al. [32]. Section 2.3 introduces some common tricks and patterns that arise in pseudo-Boolean proofs and are generally known to experienced practitioners. Some of these have never been explicitly written down or formalised, and it will be noted when the presentation is original.

2.1 Basic Definitions

A *Boolean variable* is a symbol which can only be associated with the values 0 (*false*) or 1 (*true*). We will also refer to Boolean variables as "PB variables" in the context of pseudo-Boolean reasoning, and denote them as is standard using lowercase letters from near the end of the alphabet, possibly with subscripts. The logical *negation* of PB variable is usually denoted with a bar above the symbol, so $\overline{x} = 1 - x$, and PB variables and their negations are collectively referred to as *literals* (positive and negative literals respectively). The negation of a negated variable is the original variable again, $\overline{\overline{x}} = x$, hence we are free to write $\overline{\ell} = 1 - \ell$ and $\ell = 1 - \overline{\ell}$ for any literal ℓ.

2.1.1 Forms of Pseudo-Boolean Constraints

We will consider a pseudo-Boolean (PB) constraint to have the general form

$$\sum_{i=1}^{n} a_i \ell_i \geq b, \quad \text{where } a_1, \ldots, a_n, \text{ and } b \text{ are all integers; and } \ell_1, \ldots, \ell_n \text{ are literals.} \tag{2.2}$$

We may assume that the literals $\ell_1, \ldots, \ell_n$ are over distinct variables, since we can cancel opposite signed literals ($ax + b\overline{x} = (a - b)x + b$) and collect literals with the same sign.

It will often be convenient to consider PB constraints in equivalent *normalised* forms. If we replace any negated literals $\overline{x}$ with $(1 - x)$, we can rearrange any PB constraint to obtain a constraint in the form of (2.1) with only positive literals, and we call this its *variable normal form*. On the other hand, if we replace any literals ℓ with $(1 - \overline{\ell})$ whenever they have negative coefficients, and then increase a negative right-hand-side to 0 if necessary, we can rearrange to obtain a constraint in the form of (2.2) except with each $a_i \geq 0$ and $b \geq 0$, and we call this *coefficient normal form*. When we simply say a constraint is "normalised", we mean "in coefficient normal form" unless otherwise stated, and for a normalised constraint the non-negative integer on the right-hand side is the *degree of falsity*, or simply the *degree* of the constraint.

> **Example 2.1** (PB normal forms). *Let* $C := 3x - 2y + 5\overline{z} - w \geq 3$. *The coefficient normal form of* C *is* $3x + 2\overline{y} + 5\overline{z} + \overline{w} \geq 6$, *and so the degree of* C *is* 6, *while the variable normal form of* C *is* $3x - 2y - 5z - w \geq -2$.

A set of PB constraints is called a *formula*, which is to be interpreted as the logical conjunction of the constraints. The expressiveness of PB formulas stems from the fact that the individual constraints can express usual logical connectives like $\wedge$ and $\vee$, but also encompass (weighted) counting of *how many* variables are true or false. When all coefficients a_i of a constraint are equal to 1, we call it a *cardinality constraint*, and if the right-hand side of a cardinality constraint b is also 1 we call it a *clause*, as used in Boolean satisfiability (SAT). This makes sense, since a PB constraint such as $x + \overline{y} + z + \overline{w} \geq 1$ is equivalent to $x \vee \overline{y} \vee z \vee \overline{w}$. So every Boolean formula in *conjunctive normal form* (CNF) can be written as an equivalent PB formula. A cardinality constraint where the right-hand side is equal to the number of literals on the left is similarly easy to interpret as a conjunction of its literals. For example, $x + \overline{y} + z + \overline{w} \geq 4$ is equivalent to $x \wedge \overline{y} \wedge z \wedge \overline{w}$.

2.1.2 Assignments and Substitutions

In both PB proofs and PB solving, we require the notions of *assignments* and *substitutions* of the variables in a formula. A (partial) assignment is a (partial) mapping from PB variables to the values 0 or 1, while a substitution maps variables to 0, 1 or a literal. It is helpful to also treat variable assignments as applying to literals and write $\rho(\overline{x}) = 1 - \rho(x)$ for a variable x. We also

write $\rho(\ell) = \ell$ for any literal over a variable not mapped by a partial assignment ρ. Applying an assignment or substitution ρ to a PB constraint C consists of replacing any positive literals x with $\rho(x)$, and any negative literals $\overline{x}$ with $1 - \rho(x)$, and then rearranging to obtain a constraint in the form of (2.2) — note that the left-hand side is allowed to be 0, since this is considered to be the empty sum. We denote the result of this by $C\restriction_\rho$ and denote the result of applying ρ to every constraint in a formula F by $F\restriction_\rho$.

Example 2.2 (PB substitutions). *Let* $C := 3x - 2y + 5\overline{z} - w \geq 3$*, and suppose a substitution* ρ *is defined by* $y \mapsto 1, z \mapsto 0, w \mapsto y$*. Then*

$$C\restriction_\rho \;=\; 3x - 2(1) + 5(1-0) - (y) \geq 3 \;=\; 3x - y \geq 0$$

With this definition of assignments, standard notions from SAT extend to PB formulas as expected. An assignment is *total* if it maps all the variables in a formula, and a total assignment ρ satisfies a constraint C if the relation specified by $C\restriction_\rho$ holds over the integers. A total assignment that satisfies all the constraints is a *solution* for the formula. A normalised PB constraint $\sum_{i=1}^{n} a_i\ell_i \geq b$ is a *tautology* (it is satisfied by any assignment) if $b \leq 0$, whereas if $b > \sum_{i=1}^{n} a_i$ it's a *contradiction* (it can never be satisfied).

2.1.3 Unit Propagation

We can also extend the notion of *unit propagation* from SAT to the PB setting. The name is unfortunately less apt for PB constraints, since it is not just clauses that become "units" under a partial assignment that can propagate, and the same constraint can propagate more than once. But we will stick with this name to stay consistent with the literature. We say a PB constraint *propagates* a literal ℓ if it cannot possibly be satisfied unless $\ell = 1$. In a slight perversion of terminology, we might also phrase this as "ℓ propagates (due to the constraint)". Unit propagation of a PB formula F is then the process of starting with a (usually empty) assignment ρ; adding $\ell \mapsto 1$ (i.e. $x \mapsto 1$ for a positive literal x, or $x \mapsto 0$ for a negative literal $\overline{x}$) for any literals propagated by a constraint in the formula; and then repeating with $F\restriction_\rho$ until either a constraint becomes a contradiction or no further literals propagate. This can be implemented in polynomial time by making use of the *slack* of a PB constraint under an assignment ρ, defined as follows for a constraint in coefficient normal form.

$$\mathit{slack}\left(\sum_{i=1}^{n} a_i\ell_i \geq b, \rho\right) := \sum_{i:\rho(\ell_i)\neq 0} a_i - b \tag{2.3}$$

This function can be computed in at worst $O(n)$, and potentially recomputed even more efficiently if we knew a previous slack value. As we will see it tells us precisely what should propagate and

when the assignment conflicts with C.

Proposition 2.1 (Slack and propagation).
Let $C := \sum_{i=1}^{n} a_i \ell_i \geq b$ be a normalised PB constraint and ρ be a partial assignment. Then we have:

1. *$C\restriction_\rho$ is a contradiction if and only if $\mathit{slack}(C, \rho) < 0$;*
2. *For $1 \leq i \leq n$, C propagates the literal ℓ_i under ρ if and only if $\mathit{slack}(C, \rho) < a_i$.*

Proof.

1. Note that $C\restriction_\rho = \sum_{j:\rho(\ell_j)=\ell_j} a_j \ell_j \geq b - \sum_{k:\rho(\ell_k)=1} a_i$.
   If $\mathit{slack}(C, \rho) < 0$, then $\sum_{i:\rho(\ell_i)\neq 0} a_i = \sum_{j:\rho(\ell_j)=\ell_j} a_j + \sum_{k:\rho(\ell_k)=1} a_i < b$, and hence the relation cannot hold regardless of how the remaining unassigned literals ℓ_j are set. Conversely, suppose $\mathit{slack}(C, \rho) \geq 0$. Then $\sum_{j:\rho(\ell_j)=\ell_j} a_j + \sum_{k:\rho(\ell_k)=1} a_i \geq b$ and so extending ρ to a total assignment by setting each unassigned ℓ_j to 1 satisfies C — so $C\restriction_\rho$ is certainly not a contradiction.
2. Let $\rho' = \rho \cup \{\ell_i \mapsto 0\}$ and observe that $\mathit{slack}(C, \rho') = \mathit{slack}(C, \rho) - a_i$.
   We have: C propagates ℓ_i under $\rho \iff C\restriction_{\rho'}$ is a contradiction (by definition) $\iff$ $\mathit{slack}(C, \rho') < 0$ (by the previous part) $\iff$ $\mathit{slack}(C, \rho) < a_i$.

□

If we let N be the total number of literals (including repetitions) in the input formula F, a naïve algorithm for unit propagation will have worst-case $O(N^2)$ running time, as it simply iterates through all the constraints and recalculates the slack each time, adding at least one literal to the assignment at every iteration. In practice, however, much more efficient algorithms are possible, similar to the watched literal scheme for CNF unit propagation [60, 157, 155]. And as with unit propagation in SAT, PB unit propagation is *confluent*, so the same fixed point will always be reached regardless of the propagation order.

2.1.4 Convenience Notation

Finally, we can define some further syntactic sugar on top of pseudo-Boolean constraints to make their logical meaning clearer. First, the negation of a general PB constraint $C := \sum_{i=1}^{n} a_i \ell_i \geq A$ is given by

$$\neg C := \sum_{i=1}^{n} -a_i \ell_i \geq -A + 1, \tag{2.4}$$

which when normalised is

$$\sum_{i=1}^{n} a_i \bar{\ell}_i \quad \geq \sum_{i=1}^{n} a_i - A + 1. \tag{2.5}$$

Any assignment that satisfies C does not satisfy $\neg C$, and vice versa, as one would expect.

Example 2.3 (PB constraint negation). *Let* $C := 3x - 2y + 5\overline{z} - w \geq 3$. *Then* $\neg C$ *is* $-3x + 2y - 5\overline{z} + w \geq -2$.

We will also denote *integer linear combinations* with non-negative coefficients $\lambda_1, \ldots, \lambda_k$ of PB constraints $C_1, \ldots, C_k$, where $C_j := \sum_{i=1}^{n_j} a_{ji}\ell_{ji} \geq b_j$, as

$$\lambda_1 C_1 + \cdots + \lambda_k C_k \tag{2.6}$$

and this is understood to mean computing the 0-1 inequality

$$\sum_{i=1}^{n_1} \lambda_1 a_{1i}\ell_{1i} + \cdots \sum_{i=1}^{n_k} \lambda_k a_{ki}\ell_{ki} \geq \lambda_1 b_1 + \cdots + \lambda_k b_k \tag{2.7}$$

and rearranging — cancelling opposite literals and collecting like literals — to obtain a PB constraint again in the form of (2.2).

Example 2.4 (PB linear combination).
Let $C_1 := 3x - 2y + 5\overline{z} - w \geq 3$, *and* $C_2 := \overline{x} + 4y + z - 3\overline{w} \geq 8$.

$$\begin{aligned} \textit{Then } C_1 + 2C_2 &= (3x + 2\overline{x}) + (-2y + 8y) + (5\overline{z} + 2z) - (w + 6\overline{w}) \geq 3 + 16 \\ &= (x + 2) + 6y + (3\overline{z} + 2) - (5\overline{w} + 1) \geq 19 \\ &= x + 6y + 3\overline{z} - 5\overline{w} \geq 16. \end{aligned}$$

Note also that $C + \neg C = 0 \geq 1$ for any C.

Another important notation, which has a slightly less obvious native PB interpretation, represents *reified* constraints. Semantically, the reification of a PB constraint C with respect to a set of literals $r_1, \ldots, r_k$ is the constraint

$$r_1 \wedge \cdots \wedge r_k \Leftrightarrow \quad C, \tag{2.8}$$

which says that in any total assignment $r_i = 1$ for all $i \in \{1, \ldots, k\}$ if and only if C is satisfied. This can be equivalently be viewed as the $k + 1$ *half-reified* constraints

$$r_1 \wedge \cdots \wedge r_k \Rightarrow \quad C; \tag{2.9}$$

$$\overline{r}_1 \Rightarrow \neg C; \quad \ldots; \quad \overline{r}_k \Rightarrow \neg C. \tag{2.10}$$

which say, respectively, that if $r_i = 1$ for *all* $i \in \{1, \ldots, k\}$ in an assignment, then C must be

satisfied (2.9), and if $r_i = 0$ for *any* $i \in \{1, \dots, k\}$, then $\neg C$ must be satisfied (2.10). We can actually express each of these half-reified constraints with an equivalent native PB constraint, and so use the implication symbols as a syntactic sugar.

Proposition 2.2 (PB constraint half-reification). *Let* $C := \sum_{i=1}^{n} a_i \ell_i \geq b$ *be a normalised PB constraint and* $r_1, \dots, r_n$ *be a set of distinct literals. Then*

$$r_1 \wedge \cdots \wedge r_k \Rightarrow C \tag{2.11}$$

is equivalent to

$$b\overline{r}_1 + \cdots + b\overline{r}_k + \sum_{i=1}^{n} a_i \ell_i \geq b. \tag{2.12}$$

Proof. We need to show that the two constraints admit exactly the same solutions.

First let ρ be a total assignment that satisfies (2.11) as a logical expression. Suppose that $\rho(r_i) = 1$ for all $i \in \{1, \dots, n\}$. Then we must have $\sum_{i=1}^{n} a_i \rho(\ell_i) \geq b$ by the half-reification semantics, and so (2.12) is also satisfied by ρ since $\rho(\overline{r}_i) = 0$ for all $i \in \{1, \dots, n\}$. On the other hand if $\rho(r_i) = 0$ for some $i \in \{1, \dots, n\}$, then the left-hand side of (2.12) is already at least b regardless of the other value assignments, since all coefficients and b are non-negative, and hence (2.12) is again satisfied by ρ.

Now let ρ be a total assignment that does not satisfy (2.11). By the half-reification semantics, we must have $\rho(r_i) = 1$ (i.e. $\rho(\overline{r}_i) = 0$) for all $i \in \{1, \dots, n\}$, and nevertheless $\sum_{i=1}^{n} a_i \rho(\ell_i) < b$. So ρ does not satisfy (2.12). □

Even without Proposition 2.2, the correctness of a given reification representation is usually obvious on inspection, as we can see in the following example.

Example 2.5 (Full PB reification). *Let* $C := 3x - 2y + 5\overline{z} - w \geq 3$. *Then we can define a variable* r *with the semantics* $r \Leftrightarrow C$ *using the two PB constraints.*

$$6\overline{r} + 3x - 2y + 5\overline{z} - w \geq 3 \tag{2.13}$$

$$6r - 3x + 2y - 5\overline{z} + w \geq -2. \tag{2.14}$$

If $r = 1$ *then* C *is enforced by (2.13), and (2.14) becomes a tautology; and if* $r = 0$ *then* $\neg C$ *is enforced by (2.14), and (2.13) becomes a tautology. Recall that this value* 6 *is easily calculated from the normalised form of* C *(Example 2.1).*

Note that sometimes for a set of literals *Lits* we will often simply write *Lits* $\Rightarrow C$ as a further shorthand for $\bigwedge_{\ell \in Lits} \ell \Rightarrow C$.

2.2 A Pseudo-Boolean Proof System

In its most general setting, a *proof system* is simply a means of computing efficiently whether a given *string* (a sequence of symbols) is a member of some formal *language* (a set of strings). Mathematically speaking (see Buss and Nordström [36]), any proof system is specified by a predicate $\mathcal{P}(x, \pi)$ where the inputs x and π are both strings, and $\mathcal{P}$ is computable in polynomial time in the length of its inputs. We say π is a $\mathcal{P}$-proof of x if $\mathcal{P}(x, \pi)$ holds, and that $\mathcal{P}$ is a proof system for a language L if two important properties are respected:

- *Completeness:* If $x \in L$ there is some string π such that $\mathcal{P}(x, \pi)$ is true.
- *Soundness:* If $x \notin L$ there is no string π such that $\mathcal{P}(x, \pi)$ is true.

Usually proof systems are based on *derivation rules*, with a proof specifying a sequence of rule applications that derive facts from the input string and previously derived facts. Soundness and completeness can then be ensured as properties of the chosen rules.

2.2.1 Cutting Planes Rules

If we are interested in proving that a set of PB constraints is *unsatisfiable*, we should consider a proof system for the language consisting of all unsatisfiable PB formulas, using a set of rules that derive pseudo-Boolean constraints from a pseudo-Boolean input formula. A fundamental starting point is the well-known *cutting planes* proof system, which was developed from the method for solving integer linear programs of the same name, and was first analysed in the context of PB formulas by Cook et al. in 1987 [51]. The system has since been extensively studied as part of the discipline of *proof complexity* within theoretical computing science [133].

The rules in this system for deriving constraints from a formula F are specified below. We use a standard notation for proof rules where required antecedents are specified above an inference line, and the derived constraint is given underneath.

Rule 1 (Formula Axiom). *Constraints $C \in F$ are trivially derivable.*

Rule 2 (Literal Axiom). *Constraints of the form $\ell \geq 0$ are also trivially derivable and are called* literal axioms.

Rule 3 (Linear Combination). *A positive* linear combination *of constraints in F, as defined by (2.7), can be derived by specifying which constraints to add and their scalar coefficients $\lambda_1, \dots, \lambda_k \geq 0$.*

$$\frac{C_1, \dots, C_k}{\lambda_1 C_1 + \dots + \lambda_k C_k}. \tag{2.15}$$

This means in particular we have an addition *rule for two constraints, and a* scalar multiplication *rule for a single constraint.*

Rule 4 (Division). *We can apply a form of* division *to any constraint in the formula, dividing each coefficient and the degree in its normalised form through by a positive integer λ and rounding up.*

$$\frac{\sum_{i=1}^{n} a_i \ell_i \geq b}{\sum_{i=1}^{n} \lceil a_i/\lambda \rceil \ell_i \geq \lceil b/\lambda \rceil}. \tag{2.16}$$

Each of these rules derives a constraint D that is *implied* by the initial formula F, which means any solution to F also satisfies D. This is straightforward to show, although division requires slightly more care because it relies on the integrality of the 0-1 variables. The cutting planes proof system for unsatisfiable PB formulas then takes an input formula F, and requires the proof π to derive a sequence of constraints $D_1, \ldots, D_L$, where D_i must be derived by a specified cutting planes rule from PB constraints in $\{D_1, \ldots, D_{i-1}\}$. The proof is accepted if all the rule applications are valid and D_L is a contradiction.

The soundness of this proof system follows by an easy induction on the implicational property of the rules, since, naturally, a formula implies a contradicting constraint if and only if it is unsatisfiable. Completeness follows less obviously from more general results in the theory of rational polyhedra which show that cutting planes is *implicationally complete*, i.e. if F implies a constraint C then it is always possible to derive C from F, although this could require an exponential number of steps [177, 165].

Example 2.6 (Cutting planes proof).
Let F be the PB formula consisting of the constraints

$$\begin{array}{lll} C_1 := x_1 + x_2 \geq 1 & C_2 := y_1 + y_2 \geq 1 & C_3 := z_1 + z_2 \geq 1 \\ C_4 := \overline{x}_1 + \overline{y}_1 \geq 1 & C_5 := \quad \overline{y}_1 \geq 1 & C_6 := \overline{x}_1 + \overline{z}_1 \geq 1 \\ C_7 := \overline{x}_2 + \overline{y}_2 \geq 1 & C_8 := \quad \overline{y}_2 \geq 1 & C_9 := \overline{x}_2 + \overline{z}_2 \geq 1 \end{array}$$

One possible cutting planes proof of unsatisfiability of F is as follows:

$$\begin{aligned}
C_{10} &:= C_4 + C_5 + C_6 && = && 2\overline{x}_1 + 2\overline{y}_1 + \overline{z}_1 \geq 3; \\
C_{11} &:= C_{10}/2 && = \overline{x}_1 + \overline{y}_1 + \lceil 1/2 \rceil \overline{z}_1 \geq \lceil 3/2 \rceil = && \overline{x}_1 + \overline{y}_1 + \overline{z}_1 \geq 2; \\
C_{12} &:= C_7 + C_8 + C_9 && = && 2\overline{x}_2 + 2\overline{y}_2 + \overline{z}_2 \geq 3; \\
C_{13} &:= C_{12}/2 && = \overline{x}_2 + \overline{y}_2 + \lceil 1/2 \rceil \overline{z}_2 \geq \lceil 3/2 \rceil = && \overline{x}_2 + \overline{y}_2 + \overline{z}_2 \geq 2; \\
C_{14} &:= \textstyle\sum_{j \in \{1,2,3,11,13\}} C_j = && && 0 \geq 1.
\end{aligned}$$

The actual proof string π for cutting planes does not need to specify the inequality derived at each step, however, any verifier will need to compute them in order to check that the derivation is valid. Hence, following Buss and Nordström [36], we consider the *length* of a cutting planes derivation to be the total number of steps, or equivalently, the total number of constraints derived; while the *size* is the sum of the bit-lengths of all coefficients appearing in derived constraints. So the length of the derivation in Example 2.6 would be 5, while the size would be 25.

2.2.2 Additional VeriPB Rules

The pseudo-Boolean proof system *VeriPB* augments pure cutting planes with additional rules. First is *saturation*, which has been shown to allow shorter derivations for certain kinds of formula when added to the cutting planes rule set [90].

Rule 5 (Saturation). *For any (normalised) constraint in the formula we can derive a new constraint by reducing all coefficients until they are no larger than the degree. This is called* saturating *the constraint.*

$$\frac{\sum_{i=1}^{n} a_i \ell_i \geq b}{\sum_{i=1}^{n} \min\{a_i, b\} \cdot \ell_i \geq b} \tag{2.17}$$

Next, there is the *reverse unit propagation rule*, which provides a convenient way to add constraints that are "obviously" implied without writing down the details of the cutting planes derivation.

Rule 6 (Reverse Unit Propagation). *A constraint C can be derived from a formula F by* reverse unit propagation *(RUP) if applying unit propagation to $F \cup \{\neg C\}$ results in a contradiction. We call C a RUP constraint in this case.*

Directly from the definition of this rule, we have that if a partial assignment ρ falsifies any constraint in a formula F, then $\sum_{\ell \in \rho} \overline{\ell} \geq 1$ will be a RUP constraint with respect to F. This

means that the RUP rule on its own is in fact complete for unsatisfiable PB formulas (but not necessarily implicationally complete, see Example 2.16), since we can always write out a full search tree of all possible assignments as a sequence of RUP constraints. This will be discussed in more detail in the next chapter.

Another convenience rule for even more obviously implied constraints is the *syntactic implication* rule.

> **Rule 7** (Syntactic Implication). *A constraint C is syntactically implied by another constraint C' if C can be derived from the formula $\{C'\}$ using only addition of literal axioms and saturation steps. If C is syntactically implied by any constraint in the formula F, we allow derivation of C in a single step.*

This rule is often applied for convenience to (partially) *weaken* constraints, e.g. to replace a term $a\ell$ with its maximum contribution by adding literal axioms $\ell \geq 0$ or $\overline{\ell} \geq 0$, or to reduce the degree of a constraint by (conceptually) adding opposite literal axioms $\ell \geq 0 + \overline{\ell} \geq 0 = 0 \geq -1$. It is straightforward once again to show that these additional rules derive constraints implied by the original formula, and so including them with cutting planes is still sound and complete.

It is, however, not strictly necessary for a proof system for unsatisfiable PB formulas to only allow derivation of implied constraints. *Equisatisfiability* is a weaker property that can ensure soundness of a proof by an easy induction. Two formulas are equisatisfiable if they are either both satisfiable or both unsatisfiable. So a proof ending in contradiction will still be valid as long as any derived constraint D guarantees F and $F \cup \{D\}$ are equisatisfiable, where F is the conjunction of initial and previously derived constraints. Clearly, implied constraints already respect this property, but *VeriPB* has further rules for deriving constraints that are not necessarily implied, but can be derived "without loss of satisfaction".

For a PB formula F and constraint C, F and $F \wedge C$ are trivially equisatisfiable if F is unsatisfiable. So to show that the introduction of some constraint C preserves satisfiability, one approach is to assume F is satisfiable and use that fact to show C is also satisfiable. In particular, if F is satisfiable, there exists at least one satisfying total assignment ρ. If ρ also satisfies C we are done, so we are otherwise left with the task of showing that there exists at least one other assignment ρ' that satisfies both F and C. A key observation for turning this into the application of an efficiently-verifiable rule is that if ρ is a total assignment for $F \wedge C$ and ω is a *substitution* then $\rho \circ \omega$ is also a total assignment. So if it is possible to show that for any assignment ρ that satisfies F but *falsifies* C there exists a substitution ω such that ρ satisfies $\omega(F \wedge C)$, then we know $\rho' = \rho \circ \omega$ is an assignment satisfying both F and C and hence satisfiability is preserved. This forms the basis of the *redundance-based strengthening* rule, also called simply the "redundance" rule.

Rule 8 (Redundance-Based Strengthening). *A constraint C can be derived from a formula F by redundance-based strengthening if there exists a witness substitution ω such that $\omega(F \wedge C)$ can be derived (via the previous implicational rules) from $F \wedge \neg C$. Using $\vDash$ to represent an implicational derivation, we can notate this rule as follows.*

$$\frac{F \cup \{\neg C\} \vDash F\restriction_\omega \cup \{C\restriction_\omega\}}{C} \tag{2.18}$$

For this to be efficiently checkable, the proof should specify explicitly the witness, as well as the derivation of any non-trivial constraints in $\omega(F \wedge C)$ via a subproof – we call these proof "obligations".

Example 2.7 (Redundance with subproof). *Let F be the PB formula consisting of the constraints*

$$C_1 := 4\overline{z} + x_1 + 2x_2 - y_1 - 2y_2 \geq 1 \qquad C_2 := 4\overline{q} + x_1 + 2x_2 - y_1 - 2y_2 \geq 1$$
$$C_3 := 4\overline{w} - x_1 - 2x_2 + y_1 + 2y_2 \geq 1 \qquad C_4 := z + w \geq 1$$

then we can derive the (non-implied) constraint $D := 4q - x_1 - 2x_2 + y_1 + 2y_2 \geq 1$ via redundance-based strengthening using the substitution $\omega = \{q \mapsto 1\}$. Since C_1, C_3 and C_4 are unaffected by this substitution, the only proof obligations according to (2.18) are $F \cup \{\neg D\} \vDash C_2\restriction_\omega$ and $F \cup \{\neg D\} \vDash D\restriction_\omega$. Since $D\restriction_\omega = -x_1 - 2x_2 + y_1 + 2y_2 \geq -3$ is a tautology we only need a subproof for the former, which is as follows:

Begin Subproof: $(\neg D := -4q + x_1 + 2x_2 - y_1 - 2y_2 \geq 0)$

$$\begin{aligned}
D_1 &:= \neg D + C_3 + 4 \cdot (q \geq 0) = 4\overline{w} \geq 1; \\
D_2 &:= \mathit{saturate}(D_1) = \overline{w} \geq 1; \\
D_3 &:= 4 \cdot D_2 + C_1 + 4 \cdot C_4 + 4 \cdot (\overline{z} \geq 0) = x_1 + 2x_2 - y_1 - 2y_2 \geq 1 \\
&\qquad\qquad (= C_2\restriction_\omega.)
\end{aligned}$$

End Subproof.

Another non-implicational rule included in the *VeriPB* system is the *dominance-based strengthening* rule, but since the applicability conditions are more complex, and it is not strictly required for the work in this thesis, its description is omitted here — the interested reader can consult Bogaerts et al. [32] for full details.

Finally, we note that in practice for automatic verification it is useful to be able to *delete* constraints. This is effectively an annotation that tells the verifier it can henceforth disregard a constraint because it is no longer needed. This means it can no longer be used as a premise

for any rules, but any previously derived consequences of the deleted constraint remain valid unless they are also deleted. From a proof verification perspective, we can refer to the set of non-deleted constraints as the *current database*, and keeping this small is crucial to efficient checks, particularly for RUP and syntactic implication.

Notwithstanding the dominance rule, one way to ensure soundness of deletion is to simply decree that only derived constraints are allowed to be deleted, which is what we will assume in this thesis going forward. With dominance, more subtle issues can arise, leading to the concepts of *checked* and *unchecked* deletions, and *core* and *derived* constraint sets, as discussed by Bogaerts et al. [32]. *VeriPB* does in fact allow constraints in the original formula to be deleted in some circumstances, and it provides nuanced rules which we will not use for moving between proof "configurations" and modifying core and derived sets.

2.2.3 Proofs of Optimality

This system can also be adapted to prove the optimal value of some (linear) PB objective function to be minimised subject to the constraints of a PB formula. From a formal perspective, the input x to the system $\mathcal{P}(x, \pi)$ in this case needs to be split into three parts: the input formula F; the objective function f; and an *answer* string v; which is either a claimed minimum objective value, or the string `UNSAT`. If a suitable delimiter is used, we can then consider the proof system to be a quaternary predicate $\mathcal{P}(F, f, v, \pi)$. Soundness and completeness is then defined over the language of all possible conjunctions of PB formulas, objective functions, and correct answers for the given formula-objective pair. If $v = \texttt{UNSAT}$ then a valid proof is an unsatisfiability proof exactly as described above. Otherwise, a valid proof must have two parts: a derivation of the constraint $f \geq v$, and a witness assignment ρ^* such that $f\restriction_{\rho^*} = v$. For *VeriPB* the derivation can still use all the rules described above, along with an additional *solution improving rule* that can be used to lower a stored objective bound by specifying solutions that achieve a lower value.

> **Rule 9** (Solution Logging). *For a PB formula F and linear objective function f to be minimised, if a witness total assignment ρ can be provided such that ρ satisfies F and $f\restriction_{\rho'} = v$, then a constraint $-f \geq -v$ may be introduced.*

Completeness follows again from the implicational completeness of cutting planes, but soundness in optimisation proofs requires somewhat more care, since the rules now need to preserve at least one *optimal* solution, rather than simply any solution. This is not a problem for the implication rules, but, for example, the conditions of the redundance-based strengthening rule become

$$\frac{F \cup \{\neg C\} \vDash F\restriction_\omega \cup \{C\restriction_\omega\} \cup \{f\restriction_\omega \leq f\}}{C} \tag{2.19}$$

and further care is needed to handle dominance/deletions. Again, Bogaerts et al. [32] gives full

details.

2.2.4 Machine-Readable Syntax

A crucial aspect of this system is that, as well as being efficiently checkable, the rules are efficiently expressible in a machine-readable format. To emphasise this, we will examine some examples in the current syntax recognised by the *VeriPB* proof checker. All of these pass with version 3.0 of the implementation available at the time of writing on GitLab [1].

As mentioned, there is already a standard format for pseudo-Boolean problems, OPB, which *VeriPB* syntax builds upon [176]. Here PB constraints are expressed as strings of text with spaces separating alternating variable names and integer coefficients, followed by `>=` and an integer bound, and the tilde symbol (`~`) is used to represent negation.

Example 2.8 (OPB constraints). *The three forms of the constraint in Example 2.1:*

$$3x - 2y + 5\overline{z} - w \geq 3; \quad 3x + 2\overline{y} + 5\overline{z} + \overline{w} \geq 6; \quad 3x - 2y - 5z - w \geq -2;$$

can be written in the OPB format as

```
3 x0 -2 y0  5 ~z0 -1  w0 >=  3 ;
3 x0  2 ~y0  5 ~z0  1 ~w0 >=  6 ;
3 x0 -2 y0 -5  z0 -1  w0 >= -2 ;
```

Each variable identifier must be at least two characters, and cannot begin with a number.

An objective function can be specified similarly by prepending the string "`min:`" (or "`max:`" as a newer syntactic sugar) to an OPB expression. Hence, the input problem for any *VeriPB* proof can be completely specified as an OPB file. Comments, which are lines beginning with an asterisk (`*`) are also allowed, and are ignored by the verifier.

For the proof itself, some extension of the OPB format is needed. The *VeriPB* checker expects a proof to be in a separate file with a `.pbp` extension and this should begin with the header line "`pseudo-Boolean proof version 3.0`" (for the 3.0 format). This is followed by lines specifying how to derive all the constraints comprising the proof, ending with "`output`" and "`conclusion`" lines saying where contradiction or bounds were derived, and finally a footer "`end pseudo-Boolean proof`". Since asterisks have another meaning in this format, comments in a `pbp` file are instead prefixed with a percent sign (`%`).

Cutting planes and saturation derivations (Rules 2 to 5) can be expressed concisely as a sequence of operations in reverse Polish (postfix) notation, using `+`, `*`, `s`, `d` as operators, and integers or term names (for literal axioms) as operands. Note that each constraint present in the OPB file or derived in the proof is assigned a unique sequential integer ID by the verifier, and whether an operand is interpreted as an ID or a scalar is context dependent, but never ambiguous.

The sequence of reverse Polish operations is initiated with the string "pol" and terminated with a line break.

Example 2.9 (pol step syntax). *The following line in a* .pbp *file*

```
pol 1 2 + 3 * 5 + 4 d s x5 + ;
```

tells the verifier to add the constraints with IDs 1 *and* 2*, multiply the result by* 3*, then add the constraint with ID* 5*, then divide by* 4*, saturate, and add the literal axiom* $x_5 \geq 0$*.*

This is already sufficient syntax to allow us to rewrite Example 2.6 in the machine-checkable *VeriPB* format.

Example 2.10 (UNSAT *VeriPB* proof syntax).
The following two files are a complete description of a PB problem and a machine-checkable proof of unsatisfiability for that problem.

pol_proof.opb:

```
* Constraint IDs 1-10:
1  x1 1  x2 >= 1 ;
1  y1 1  y2 >= 1 ;
1  z1 1  z2 >= 1 ;
1 ~x1 1 ~y1 >= 1 ;
1 ~y1 >= 1 ;
1 ~x1 1 ~z1 >= 1 ;
1 ~x2 1 ~y2 >= 1 ;
1 ~y2 >= 1 ;
1 ~x2 1 ~z2 >= 1 ;
```

pol_proof.pbp:

```
pseudo-Boolean proof version 3.0
% Derive constraint id 10:
pol 4 5 + 6 + 2 d ;
% Derive constraint id 11:
pol 7 8 + 9 + 2 d ;
% Derive constraint id 12 (contradiction)
pol 1 2 + 3 + 10 + 11 + ;
output NONE ;
conclusion UNSAT : 12 ;
end pseudo-Boolean proof ;
```

For syntactically implied constraints (Rule 7), the full constraint to be derived must be written to the .pbp file, prepended with "ia" and terminated with a semicolon. Optionally, an ID can be appended to tell the verifier which previous constraint syntactically implies the derived constraint and save verification time. Similarly, for RUP constraints (Rule 6), the string "rup" is used, followed by the constraint that should follow by reverse unit propagation. For solution-improving constraints in optimisation proofs (Rule 9), we use the string "soli", and it is followed instead by a space separated list of literals specifying a partial assignment that propagates to a full solution.

Example 2.11 (rup, ia, step syntax).
The line

```
ia 3 x0 2 y0 1 z0 >= 1 : 3 ;
```

in a `.pbp` *file says that* $3x_0 + 2y_0 + z_0 \geq 1$ *can be derived from the constraint with ID* 3 *by syntactic implication, while*

```
rup 3 x0 2 y0 1 z0 >= 1 ;
```

means it follows from the verifier's current constraint database by RUP. The line

```
soli x0 ~z0 ;
```

means that running unit propagation on the current database starting with the assignment $\{x_0 \mapsto 1, z_0 \mapsto 0\}$ *should result in a full solution, and the upper bound on the objective function resulting from this solution should be added to the constraint database.*

The syntax for the redundance rule is more involved, so we will give one example (translating Example 2.7 to machine-readable form) to illustrate the idea and omit a complete description. Essentially, each non-trivial proof obligation should have its subproof enclosed inside a block denoted with `proofgoal` and the ID of the obligation. Inside this environment, temporary IDs are assigned to the negation of the constraint to be introduced and the negation of the proof goal, and it is sufficient to derive contradiction from these.

Example 2.12 (*VeriPB* `red` syntax).
Suppose the following are the first four lines of a `.opb` *file.*

```
4 ~z0 1 x1 2 x2 -1 y1 -2 y2 >= 1 ;
4 ~q0 1 x1 2 x2 -1 y1 -2 y2 >= 1 ;
4 ~w0 -1 x1 -2 x2 1 y1 2 y2 >= 1 ;
1 z0 1 w0 >= 1;
```

Then, in a `.pbp` *file, the lines*

```
red 4 q0 -1 x1 -2 x2 1 y1 2 y2 >= 1 : q0 -> 1 : subproof
    proofgoal 2
        pol -2 3 + q0 4 * + s 4 * 1 + 4 4 * + -1 +;
    qed : -1;
qed ;
```

are a valid VeriPB *derivation of the constraint* $4q_0 - x_1 - 2x_2 + y_1 + 2y_2 \geq 1$ *by redundance-based strengthening. The line* `proofgoal 2` *specifies that the subsequent lines (up to the* `qed` *delimiter) provides a proof that the constraint with ID 2 is implied under the given witness (*`q0 -> 1`*). The* `pol` *line then specifies the same derivation as Example 2.7, except an explicit contradiction is achieved by adding the negation of the proof goal constraint. This also uses the relative IDs* VeriPB *syntax feature – so* `-1` *means "one before the current ID".*

In some cases, and particularly for readability, it can be useful to specify a *label* for a

constraint that can be used in place of its ID. Labels can be any valid identifier, prefixed with an "@" symbol, and are written at the start of the line where the constraint to be labelled is defined or derived.

Example 2.13 (Labels syntax). *Suppose in a* `.opb` *file we define*

```
@foo 1 x1 1 x2 >= 1 ;
@bar 1 ~x2 3 x3 10 x4 >= 2 ;
1 ~x2 3 x3 10 x4 >= 2 ;
5 x2 2 ~x3 1 x5 >= 2 ;
@foobar 5 x2 2 ~x3 1 x5 >= 2 ;
```

The following line in a `.pbp` *file has the same effect as the one in Example 2.9, and the resulting derived constraint is labelled with* `@myresult`.

```
@myresult pol @foo @bar + 3 * @foobar + 4 d s x5 + ;
```

Full details of the current format of proof files recognised by *VeriPB* are available in the checker specification for the SAT competition 2025 [7].

2.3 Useful Derivation Patterns

Now that we have the basics of pseudo-Boolean representation and the core *VeriPB* rules, we are able to explore some useful derivation patterns that will be used in various places throughout this thesis.

2.3.1 Patterns Involving Implicational Rules

We begin with a general property of implicational derivations that relates to partial assignments and PB reification. Conceptually, if we *restrict* a PB formula F by applying a partial assignment ρ, and then derive a consequence D from this restricted formula using Rules 1 to 7, then intuitively we should be able to derive $\rho \Rightarrow D$ from F. As a more specific case, if we can derive D from a formula F not containing any literals in ρ, then we should be able to derive $\rho \Rightarrow D$ from $\rho \Rightarrow F$, the formula created by reifying every constraint in F by ρ. It turns out there is a straightforward procedure to reconstruct the unrestricted proof from the restricted one with a low overhead. This property was relied on implicitly in several proof logging works [94, 151], but it was not stated or proved explicitly until it was included as an appendix by Demirović et al. [59], which was contributed by the author of this thesis. For notational convenience in what follows, we will view an assignment ρ as the set of literals $\{\ell : \rho(\ell) = 1\}$ assigned true by ρ and assume all constraints are normalised. We use $\mathit{Lits}(C)$ to denote the set of literals with non-zero coefficients appearing in a constraint C.

Theorem 2.1 (Unrestricting proofs). *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 derivation of length L employing Rules 1 to 7. Then we can construct a derivation of length $O(L)$ from F of the constraint $\rho \Rightarrow D$ using precisely the same rules in sequence, except with some possible additional interleaved syntactic implication steps.*

To prove this we make use of the following lemma.

Lemma 2.1 (Unrestricting proofs base case). *For any PB constraint C and partial assignment ρ, we can always derive $\rho \Rightarrow C\restriction_\rho$ using derivation of length $O(1)$.*

Proof of Lemma 2.1. First, let us decompose C as

$$\sum_{\substack{\ell_i \in \mathit{Lits}(C): \\ \rho(\ell_i)=\ell_i}} a_i\ell_i + \sum_{\substack{\ell_j \in \mathit{Lits}(C): \\ \rho(\ell_j)=1}} b_j\ell_j + \sum_{\substack{\ell_k \in \mathit{Lits}(C): \\ \rho(\ell_k)=0}} c_k\ell_k \geq K. \tag{2.20}$$

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

$$\sum_{\substack{\ell_i \in \mathit{Lits}(C): \\ \rho(\ell_i)=\ell_i}} a_i\ell_i \geq K - B \tag{2.21}$$

and $\rho \Rightarrow C\restriction_\rho$ is the constraint

$$\sum_{\substack{\ell_j \in \mathit{Lits}(C): \\ \rho(\ell_j)=1}} (K-B)\overline{\ell}_j + \sum_{\substack{\ell_k \in \mathit{Lits}(C): \\ \rho(\ell_k)=0}} (K-B)\ell_k + \sum_{\substack{\ell_i \in \mathit{Lits}(C): \\ \rho(\ell_i)=\ell_i}} a_i\ell_i \geq K - B. \tag{2.22}$$

To derive (2.22) from (2.20) we can proceed as follows.

1. For all j, add the literal axioms amounting to $b_j\overline{\ell}_j \geq 0$ to (2.20), yielding

$$\sum_{\substack{\ell_k \in \mathit{Lits}(C): \\ \rho(\ell_k)=0}} c_k\ell_k + \sum_{\substack{\ell_i \in \mathit{Lits}(C): \\ \rho(\ell_i)=\ell_i}} a_i\ell_i \geq K - B. \tag{2.23}$$

2. Saturate to ensure that for all k, $c_k \leq K - B$.

3. Add multiples of literal axioms $\ell_i \geq 0$, $\ell_k \geq 0$ and $\overline{\ell}_j \geq 0$ as needed to obtain (2.22).

This amounts to one syntactic implication step and hence the derivation has constant length 1. □

We are now able to prove the main result.

Proof of Theorem 2.1. 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. Then each D_s is one of the following:

1. An axiom (constraint in $F\restriction_\rho$);
2. A literal axiom;
3. The result of one of Rules 3–5 or 7, with antecedents in π_s;
4. 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 $O(s)$ derivation that $\rho \Rightarrow D_s$ 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 2.1 we can derive $\rho \Rightarrow C\restriction_\rho$, i.e. $\rho \Rightarrow D_a$, in $O(1)$ steps. Note that if D_a is instead a literal axiom then $\rho \Rightarrow D_a$ 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 $O(s-1)$ deriving all the constraints in $\pi'_s = \{\rho \Rightarrow D_i : D_i \in \pi_s\}$. We now consider different cases depending on how D_s was derived in π.

Case 1: D_s is the result of taking a linear combination of constraints $D_{a_1}, \ldots, D_{a_k} \in \pi_s$ with coefficients $\lambda_1 \ldots \lambda_k$.

Then by assumption $\rho \Rightarrow D_{a_1}, \ldots, \rho \Rightarrow D_{a_k}$ have already been derived. If we let K_j be the degree of D_{a_j}, we can write each of these in the form

$$\sum_{\ell \in \rho} K_j \overline{\ell} + D_{a_j} \tag{2.24}$$

and so taking a linear combination with coefficients $\lambda_1, \ldots, \lambda_k$ yields

$$\sum_{\ell \in \rho} (\lambda_1 K_1 + \cdots + \lambda_k K_k)\overline{\ell} + \lambda_1 D_{a_1} + \cdots + \lambda_k D_{a_k}. \tag{2.25}$$

If K_s is the degree of D_s, note that we must have $K_s \leq \lambda_1 K_1 + \cdots + \lambda_k K_k$, since cancellation of matching literals in the combination can only reduce the overall degree. Hence, if we apply saturation to (2.25) we obtain $\sum_{\ell \in \rho} K_s \overline{\ell} + D'_s$, where D'_s is the saturated version of D_s. Then we can add literal axioms to obtain $\rho \Rightarrow D_s$, as required. Note that this saturation and literal axiom "fixing" amounts to a single syntactic implication step.

Case 2: D_s is the result of dividing a constraint $D_i \in \pi_s$ by a scalar λ.

Then again by assumption $\rho \Rightarrow D_i$ has already been derived, and this time we will write

this in full as

$$\sum_{\ell \in \rho} K_i \overline{\ell} + \sum_j a_j \ell_j \geq K_i. \tag{2.26}$$

If we divide this by λ we obtain

$$\sum_{\ell \in \rho} \lceil K_i/\lambda \rceil \, \overline{\ell} + \sum_j \lceil a_j/\lambda \rceil \, \ell_j \geq \lceil K_i/\lambda \rceil \,, \tag{2.27}$$

which is precisely $\rho \Rightarrow D_s$, as required.

Case 3: D_s is the result of applying saturation to a constraint $D_i \in \pi_s$.

Once again by assumption $\rho \Rightarrow D_i$ has already been derived, and we can write this in full as above in (2.26). After applying saturation to this we obtain

$$\sum_{\ell \in \rho} \min(K_i, K_i) \overline{\ell} + \sum_j \min(a_j, K_i) \ell_j \geq K_i, \tag{2.28}$$

which is precisely $\rho \Rightarrow D_s$, as required.

Case 4: D_s is syntactically implied by a constraint $D_i \in \pi_s$.

Then D_s is the result of adding literal axioms to D_i and possibly saturating, so combining the arguments in Case 1 and Case 3 tells us that $\rho \Rightarrow D_s$ is the result of adding literal axioms to $\rho \Rightarrow D_i$ and possibly saturating. So we can derive $\rho \Rightarrow D_s$ with one syntactic implication step.

Case 5: 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 $\rho \Rightarrow D_s$ is the constraint

$$\sum_{\ell \in \rho} K \overline{\ell} + \sum_i a_i \ell_i \geq K, \tag{2.29}$$

and its negation is

$$\sum_{\ell \in \rho} K \ell + \sum_i a_i \overline{\ell}_i \geq A + 1 + (|\rho| - 1)K. \tag{2.30}$$

We can see that for (2.30) to be satisfied, all the reification literals $\ell \in \rho$ must be set to true. Recalling that all the constraints in $\pi'_s = \{\rho \Rightarrow D_i : D_i \in \pi_s\}$ are all assumed to have been previously derived, we can see that performing unit propagation will reduce constraints in $F \cup \pi'_s \cup \neg(\rho \Rightarrow D_i)$ to be precisely the constraints in $F\restriction_\rho \cup \pi_s \cup \neg D$. Since by assumption deriving D_s from $F\restriction_\rho \cup \pi_s$ by RUP was a legitimate derivation step,

continued unit propagation on the constraint database must result in a contradiction. So we can derive $\rho \Rightarrow D_i$ 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 $\rho \Rightarrow D_s$, 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 the constraints in $\pi'_L = \{\rho \Rightarrow D_i : D_i \in \pi\}$ and in particular our desired $\rho \Rightarrow D_L$.

Since each of the L constraints in π'_L requires $O(1)$ intermediate derivation steps, our constructed derivation has length $O(L)$. □

With Theorem 2.1 established we easily obtain the following useful corollary.

Corollary 2.1 (Proofs Under Assumptions). *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 $\ell \in R$, we have $\overline{\ell} \notin R$, $\ell \notin \mathit{Lits}(F)$, and $\overline{\ell} \notin \mathit{Lits}(F)$).*
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 derivation of length L employing Rules 1 to 7, we can construct a derivation of length $O(L)$ of the constraint $\bigwedge_{C \in F} R_C \Rightarrow D$ from $R(F)$ using the same rules.

Proof. Take the partial assignment ρ setting $\ell = 1$ for each $\ell \in R$ and apply Theorem 2.1. □

A similar related property is that linear combinations play particularly nicely with partial assignments, even without requiring the reifying form.

Lemma 2.2 (Linear combinations with partial assignments). *Let $C_1 \dots C_k$ be PB constraints and let ρ be a partial assignment. If $D = \lambda_1 C_1 + \dots \lambda_k C_k$ for some positive integer coefficients $\lambda_1, \dots, \lambda_k$; then we must have $D\restriction_\rho = \lambda_1 C_1\restriction_\rho + \dots + \lambda_k C_k\restriction_\rho$.*

Proof. Let $x_1, \dots x_n$ be the all the variables appearing in any constraint C_i or assigned by ρ. We can write each C_i in its variable normal form (using 0 coefficients when necessary) as $\sum_j a_{ij} x_j \geq b_i$. So $D = \sum_j (\sum_i \lambda_i a_{ij}) x_j \geq (\sum_i \lambda_i b_i)$. Assuming without loss of generality that ρ assigns $x_1 \dots x_t$ for some $0 \leq t \leq n$, we have

$$\begin{aligned} D\restriction_\rho &= \textstyle\sum_{j=t+1}^{n} (\sum_i \lambda_i a_{ij}) x_j \geq (\sum_i \lambda_i b_i) - \sum_{j=0}^{t} (\sum_i \lambda_i a_{ij}) \rho(x_i) \\ &= \textstyle\sum_i \lambda_i (\sum_{j=t+1}^{n} a_{ij} x_j) \geq \sum_i \lambda_i (b_i - \sum_{j=0}^{t} a_{ij} \rho(x_i)) \\ &= \lambda_1 C_1\restriction_\rho + \dots + \lambda_k C_k\restriction_\rho. \end{aligned}$$

□

There are also some more specific derivation patterns involving Rules 1 to 7 that are worth noting here. First is the well known fact [118, 119] we can generalise the resolution procedure from CNF reasoning to PB constraints and implement this using cutting planes.

Theorem 2.2 (Generalised resolution). *Suppose the following two (normalised) PB constraints are present in a formula*

$$C_1 := a_j \ell_j + \sum_{i \neq j} a_i \ell_i \geq K_1 \qquad C_2 := b_j \bar{\ell}_j + \sum_{i \neq j} b_i \ell_i \geq K_2 \tag{2.31}$$

Then, letting $d = \gcd(a_j, b_j)$ *we can derive*

$$\sum_{i \neq j} \left(\frac{b_j a_i + a_j b_i}{d} \right) \ell_i \geq \frac{b_j K_1 + a_j K_2 - a_j b_j}{d} \tag{2.32}$$

using $O(1)$ *cutting planes steps (Rules 2 to 4).*

Proof. Straightforward application of addition and division, see Buss and Nordström [36, p.284]. □

The fact that cutting planes can implement resolution itself for clauses is an easy corollary of this: the right-hand side of (2.32) in this case is guaranteed to be 1, so we can saturate or divide to obtain the resolvent clause.

Another important but slightly lesser known fact is that we can efficiently recover cardinality constraints from a clique of 2-clauses using PB reasoning. This has been stated as an exercise in lecture notes on proof complexity [139], and discussed informally by McBride [148] and Gocht et al. [91], but the presentation here is original, as is the count for the number of steps.

Theorem 2.3 (Recovering cardinality constraints).
Let $\{\ell_1, \ldots, \ell_n\}$ *be a set of* $n \geq 2$ *literals and suppose we have the set of* $(n^2 - n)/2$ *constraints*

$$\{C_{ij} := \ell_i + \ell_j \geq 1 : i, j \in [n], i < j\} \tag{2.33}$$

expressing that no two of these literals can be false simultaneously. Then we can recover a cardinality constraint over the literals

$$\sum_{i=1}^{n} \ell_i \geq n - 1, \tag{2.34}$$

expressing that at most one can be false, using precisely $(n^2 + 3n - 10)/2$ *cutting planes steps (Rules 2 to 4).*

Proof. For any $k \in [n]$, let $D_k := \sum_{i=1}^{k} \ell_i \geq k-1$ and $p(k) := (k^2+3k-10)/2$, and note that $D_2 = C_{12}$ is already derived (and so requires $p(2) = 0$ steps).

Now suppose for $k \geq 3$ we have derived D_{k-1} in $p(k-1)$ steps. We can derive

$$\frac{(k-2)\cdot D_{k-1} + \sum_{i=1}^{k-1} C_{ik}}{k-1} \tag{2.35}$$

$$= \frac{\sum_{i=1}^{k-1}(k-2)\ell_i + \sum_{i=1}^{k-1}\ell_i + (k-1)\ell_k \geq (k-2)^2 + (k-1)}{k-1} \tag{2.36}$$

$$= \frac{\sum_{i=1}^{k}(k-1)\ell_k \geq (k-2)(k-1)+1}{k-1} = \sum_{i=1}^{k}\ell_k \geq \left\lceil k-2+\frac{1}{k-1}\right\rceil = D_k, \tag{2.37}$$

using an additional $k+1$ steps ($k-1$ additions, one multiplication and one division). Hence, we can derive D_k in $p(k-1)+k+1 = (k^2+3k-10)/2 = p(k)$ steps and the result follows by induction. □

Example 2.14 (Recovering cardinality constraints in *VeriPB* syntax).
The instantiation of Theorem 2.3 for $n = 4$ in VeriPB *syntax would be as follows.*

`recover_card.opb:`

```
* 4-clique of 2-clauses
1 ~x1 1 ~x2 >= 1 ;
1 ~x1 1 ~x3 >= 1 ;
1 ~x1 1 ~x4 >= 1 ;
1 ~x2 1 ~x3 >= 1 ;
1 ~x2 1 ~x4 >= 1 ;
1 ~x3 1 ~x4 >= 1 ;
```

`recover_card.pbp:`

```
pseudo-Boolean proof version 3.0
% Recover at-most-1 constraint:
pol 1 2 + 3 + 2 d 2 * 4 + 5 + 6 + 3 d ;
% Dummy conclusion
output NONE ;
conclusion NONE ;
end pseudo-Boolean proof ;
```

2.3.2 Patterns Involving Strengthening Rules

We next turn from pure cutting planes derivations to some tricks involving the redundance rule (Rule 8). The most important of these is the ability to introduce fresh variables, reified on a particular constraint. This means it can simulate what is often called an *extension* rule in proof complexity, which has been known since the rule's introduction [89].

Theorem 2.4 (Extension variables). *Suppose x is a fresh variable that does not appear anywhere in a PB formula or derivation. Then, for any PB constraint $C := \sum_i a_i \ell_i \geq b$*

(also not containing x or $\overline{x}$) we can introduce the constraints equivalent to $x \Leftrightarrow C$

$$D_1 := x \Rightarrow \sum_i a_i \ell_i \geq b, \quad \text{and} \quad D_2 := \overline{x} \Rightarrow -\sum_i a_i \ell_i \geq -b+1, \quad (\text{i.e., } \overline{x} \Rightarrow \neg C) \tag{2.38}$$

each with a single redundance-based strengthening step and an $O(1)$ length subproof.

Proof. For D_1 take the witness substitution $\omega_1 = \{x \mapsto 0\}$. Observe that $F\restriction_{\omega_1} = F$, since x is a fresh variable, and $D_1\restriction_{\omega_1} = \sum_i a_i \ell_i \geq 0$, both of which can be trivially derived, and hence the redundance rule is applicable.

Then for D_2 take the witness substitution $\omega_1 = \{x \mapsto 1\}$. Again $F\restriction_{\omega_2}$, and $D_2\restriction_{\omega_2}$ can be trivially derived, and we are left with the proof obligation $D_1\restriction_{\omega_2} = C$ which is syntactically implied by $\neg D_2 = -(\sum_i a_i - b + 1)x + \sum_i a_i \ell_i \geq b$. □

Note that these constraints (2.38) are not actually implied by the formula (x was conceptually unconstrained), and so they cannot not be derived using Rules 1 to 7.

Next, we note that redundance-based strengthening with an empty witness is essentially the same as a proof by contradiction, which can sometimes be easier to formulate than a direct proof. In the latest version of the *VeriPB* format, this can be treated as a separate rule, denoted `pbc`, for "proof by contradiction".

Rule 10 (Proof by contradiction). *A constraint C can be derived from a formula F if we can exhibit a subproof deriving contradiction from $F \cup \{\neg C\}$.*

An important application of this is to implement the *fusion resolution* rule, first highlighted by Buss and Nordström [36] (attributed to Stephan Gocht) as a potentially useful variation of the generalised resolution rule. As a motivating example, they consider the constraints

$$C_1 := 2x + 3y + 2z + w \geq 3 \qquad C_2 := 2\overline{x} + 3y + 2z + w \geq 3, \tag{2.39}$$

which can be seen to imply the conclusion

$$3y + 2z + w \geq 3 \tag{2.40}$$

by considering that since $x \in \{0, 1\}$, the corresponding x literal must disappear in at least one of C_1 and C_2. This can be implemented in constant length with a short proof-by-contradiction subproof, which was first observed by Demirović et al. [59] in a conference paper co-authored by the author of this thesis.

Theorem 2.5 (Fusion resolution). *Suppose we have the following two normalised PB constraints*

$$C_1 := a\overline{x} + \sum_i a_i\ell_i \geq m_1 \qquad C_2 := bx + \sum_i a_i\ell_i \geq m_2. \tag{2.41}$$

Then we can derive

$$D := \sum_i a_i\ell_i \geq \min\{m_1, m_2\} \tag{2.42}$$

with a single redundance step and a subproof of length $O(1)$.

Proof. Let $m = \min\{m_1, m_2\}$ and note that we can assume $m \geq 1$ or else the conclusion is trivial. We take the empty witness substitution (or use a "PBC" rule), and then we need to derive D from $\{C_1, C_2, \neg D\}$ to apply redundance.

To do this, we derive $\neg D + C_1$ which gives $a\overline{x} \geq m_1 - m + 1$, and then since $m_1 - m + 1 \geq 1$ we can divide by $\max\{a, m_1 - m + 1\}$ and multiply by b to get $b\overline{x} \geq b$. Adding this to C_2 and weakening the degree gives the required result. □

The same result that shows saturation is strictly stronger than division [90, Thm. 4.1] also implies that fusion resolution would always require a number of steps exponential in the bit-length of the largest reification coefficient to simulate with pure cutting planes with division (Rules 2 to 4); but it is an open question whether shorter derivations are possible using cutting planes with saturation (Rules 2 to 5).

2.3.3 Patterns Involving Unit Propagation

The last important derivation patterns we should cover at this point concern the reverse unit propagation rule, and rely on specific propagation properties of particular kinds of PB formulas. Generally speaking, one should be careful when relying on unit propagation in derivation procedures, since it can be surprisingly weak, leading even very simple and obvious conclusions to fail to follow by RUP (see Examples 2.15 and 2.16). Nevertheless, we can rely on propagation in certain restricted cases, first and foremost being the fact that the negation of a half-reified constraint propagates all the literals on its right-hand side.

Theorem 2.6 (Reverse unit propagation of reified constraints). *Let ρ be a conjunction of literals and $C := \rho \Rightarrow D$ be a reified constraint. Then $\neg C$ propagates $r_i = 1$ for every $r_i \in \rho$ and hence C is RUP with respect to a formula F precisely when D is RUP with respect to $F\restriction_\rho$.*

Proof. Let $D := \sum_i a_i\ell_i \geq b$. Then by Proposition 2.2, C can be written as $b\overline{r}_1 + \cdots + b\overline{r}_k +$

$\sum_i a_i\ell_i \geq b$. So $\neg C$ can be written as $-br_1 - \cdots - br_k - \sum_i a_i\ell_i \geq -b+1$ which has slack $b-1$, and so all literals in ρ propagate. □

Additionally, propagation properties specifically of sums of the form $\sum_i 2^i x_i$ are integral to the proof logging methodology discussed in the next chapter, and they can be shown to allow for specific cases of PB constraints that are guaranteed to unit propagate to contradiction. Again these are relied on implicitly in several published works [94, 151], and although they have been discussed in informal personal communications [87], to the author's knowledge they have not been explicitly written down anywhere.

Theorem 2.7 (Unit propagation of contradictory bounds on binary sums). *For any integers k, A, B where $k \geq 1$ and $A > B$ the contradictory PB constraints*

$$C_1 := \sum_{i=0}^{k-1} 2^i\ell_i \geq A; \quad \textit{and} \quad C_2 := \sum_{i=0}^{k-1} -2^i\ell_i \geq -B \quad \left(\textit{i.e.} \quad \sum_{i=0}^{k-1} 2^i\ell_i \leq B\right) \tag{2.43}$$

will always unit propagate to contradiction.

Proof. We proceed by induction on k.

For $k = 1$ we have $C_1 := \ell_i \geq A$ and $C_2 := -\ell_i \geq -B$. If $A \geq 1$ then either C_1 is a contradiction or $\ell_i = 1$ propagates, reducing C_2 to a contradiction ($0 \geq -B+1 > 0$). On the other hand, if $A \leq 0$ then $-B \geq -1$, so either C_2 is a contradiction or $\ell_i = 0$ propagates, reducing C_1 to a contradiction. This establishes the base case.

Now, for an arbitrary $k > 1$, assume the theorem holds for $k-1$. We have

$$\textit{slack}(C_1) = 2^k - 1 - A; \qquad \textit{slack}(C_2) = B; \tag{2.44}$$

so we can assume $0 \leq B < A \leq 2^k - 1$ otherwise one of the constraints is immediately contradictory. Now if $A \geq 2^{k-1}$ then $\textit{slack}(C_1) < 2^{k-1}$ and hence $\ell_{k-1} = 1$ propagates due to C_1, reducing C_1 and C_2 to

$$C_1 := \sum_{i=0}^{k-2} 2^i\ell_i \geq A - 2^{k-1}; \quad \text{and} \quad C_2 := \sum_{i=0}^{k-2} -2^i\ell_i \geq -B + 2^{k-1}; \tag{2.45}$$

which propagate to contradiction by assumption (since necessarily $A-1 > B-1$). On the other hand, if $A < 2^{k-1}$ then $B < 2^{k-1}$, so $\ell_i = 0$ propagates due to C_2, reducing C_1 and C_2 to

$$C_1 := \sum_{i=0}^{k-2} 2^i\ell_i \geq A; \quad \text{and} \quad C_2 := \sum_{i=0}^{k-2} -2^i\ell_i \geq -B \tag{2.46}$$

which also propagate to contradiction by assumption. □

Theorem 2.8 (Unit propagation of equality on binary sums). *For any integers k, A where $k \geq 1$ and $0 \leq A < 2^k$ the constraints*

$$C_1 := \sum_{i=0}^{k-1} 2^i \ell_i \geq A; \quad \text{and} \quad C_2 := \sum_{i=0}^{k-1} -2^i \ell_i \geq -A \quad \left(\text{i.e.} \quad \sum_{i=0}^{k-1} 2^i \ell_i \leq A \right) \tag{2.47}$$

will always unit propagate to a fixed assignment ρ where all underlying variables for $\ell_0 \ldots, \ell_{k-1}$ are consistently assigned, and $\sum_{i=0}^{k-1} 2^i \rho(\ell_i) = A$.

Proof. If $0 \leq A \leq 2^k$ then the constraints are not contradictory but an analogous argument to the proof of Theorem 2.7 shows that all literals must be eventually be set by unit propagation. □

The following somewhat delicate property can also be established; this was originally proved by the author of this thesis in discussion with Jakob Nordström, but the presentation here is based on a proof by Wietze Koops [131].

Theorem 2.9 (Contradictory constraints on binary sums). *For any positive integers m and n and integers A, B, C, with $A + B - C > 0$ and $B \in \{0, 1\}$ the contradictory constraints*

$$-2^{m-1} y_{m-1} + \sum_{i=0}^{m-2} 2^i y_i \geq \quad A; \tag{2.48}$$

$$2^{m-1} y_{m-1} - \sum_{i=0}^{m-2} 2^i y_i - 2^{n-1} x_{n-1} + \sum_{i=0}^{n-2} 2^i x_i \geq \quad B; \tag{2.49}$$

$$2^{n-1} x_{n-1} - \sum_{i=0}^{n-2} 2^i x_i \geq -C; \tag{2.50}$$

will always unit propagate to contradiction.

Proof. Normalising the constraints gives

$$C_1 := \quad 2^{m-1} \overline{y}_{m-1} + \sum_{i=0}^{m-2} 2^i y_i \geq \; A + 2^{m-1}; \tag{2.51}$$

$$C_2 := 2^{m-1} y_{m-1} + \sum_{i=0}^{m-2} 2^i \overline{y}_i + 2^{n-1} \overline{x}_{n-1} + \sum_{i=0}^{n-2} 2^i x_i \geq \; B + 2^{m-1} + 2^{n-1} - 1; \tag{2.52}$$

$$C_3 := \quad 2^{n-1} x_{n-1} + \sum_{i=0}^{n-2} 2^i \overline{x}_i \geq \; -C + 2^{n-1} - 1. \tag{2.53}$$

We will proceed in four steps. First we show that C_1 or C_2 directly unit propagate to

contradiction if n or m are too small. Secondly we analyse what happens to the variables y_n and x_m. Thirdly, we analyse the other variables with index at least $t = \min(m, n)$. Finally, we consider the other variables index by index.

Step 1: the cases $A \geq 2^{m-1}$ and $C < -2^{n-1}$.
Let ρ denote the empty assignment. If $A \geq 2^{m-1}$ then *slack*$(C_1, \rho) = 2^{m-1} - 1 - A \leq -1$ and we immediately have a contradiction. Similarly, if $C < -2^{n-1}$, then *slack*$(C_3, \rho) = 2^{n-1} + C \leq 0$ and we also have a contradiction.

We henceforth assume $A < 2^{m-1}$ and $C \geq 2^{n-1}$.

Step 2: the variables y_{n-1} and x_{m-1}.
Let ρ denote the empty assignment. If $A \geq 0$, then

slack$(C_1, \rho) = 2^{m-1} - 1 - A < 2^{m-1}$, so C_1 propagates $\overline{y}_{m-1}$ under ρ.

If $A < 0$ then, using the fact that $A + B - C > 0$, and $B \in \{0, 1\}$

slack$(C_3, \rho) = 2^{n-1} + C < 2^{n-1} + A + B \leq 2^{n-1}$, so C_3 propagates x_{n-1} under ρ.

Step 3: the variables with index at least $\min(m, n) - 1$.
We now show that for all $k \geq \min(m, n) - 1$, one of the literals x_k or $\overline{x}_k$ is propagated, and similarly for y_k or $\overline{y}_k$.

If $A \geq 0$ then as shown above C_1 propagates $\overline{y}_{m-1}$. Let ρ be the updated assignment i.e. $\rho = \{\overline{y}_{m-1}\}$. Then

$$\textit{slack}(C_2, \rho) = (2^{m-1} - 1) + 2^{n-1} + (2^{n-1} - 1) - B - 2^{m-1} - 2^{n-1} + 1 \tag{2.54}$$

$$= 2^{n-1} - 1 - B < 2^{n-1} \tag{2.55}$$

which shows that C_2 propagates $\overline{y}_k$ for $k \geq n - 1$ and also x_{n-1}. And then (after updating ρ), we have *slack*$(C_3, \rho) = C < A + B \leq 2^{m-1}$ and so C_3 propagates $\overline{x}_k$ for any $k \geq m - 1$.

The case $A < 0$ is analogous. As shown above, C_3 propagates x_{n-1}, so let ρ be the updated assignment, and now

$$\textit{slack}(C_2, \rho) = 2^m - 1 + 2^{n-1} - 1 - B - 2^{m-1} - 2^{n-1} + 1 \tag{2.56}$$

$$= 2^{m-1} - 1 - B < 2^{m-1} \tag{2.57}$$

which shows the constraint propagates x_k for any $k \geq m - 1$ and also y_{m-1}. Now, updating ρ again, we have *slack*$(C_1, \rho) = -A - 1 < B - C - 1 < 2^{n-1}$, so C_1 propagates y_k for any $k \geq n - 1$.

Step 4: the variables with index less than $\min(m, n) - 1$.
Let $t = \min(m, n) - 1$, and let the assignment reached after the previous step be ρ. We will now show by induction on $s \geq 0$ that under ρ, the constraints either propagate both x_{t-s} and y_{t-s} or $\overline{x}_{t-s}$ and $\overline{y}_{t-s}$ until there is a conflict in one of the constraints. The base case of the induction has already been completed in the previous step: if $A \geq 0$, $\overline{x}_t$ and $\overline{y}_t$ are propagated, and if $A < 0$, x_t

and y_t are propagated.

Now proceed with the inductive step, so we assume that for some s it holds that for each $s' \leq s$ that the inequalities either propagate both $x_{t-s'}$ and $y_{t-s'}$ or $\overline{x}_{t-s'}$ and $\overline{y}_{t-s'}$. Let ρ be the corresponding updated assignment.

Now we will compute the *sum* of the slacks of C_1 and C_3 under ρ, by first letting T be the sum of all literals with index at least t that are not falsified by ρ

$$T = 2^{m-1}\rho(\overline{y}_{m-1}) + 2^{n-1}\rho(x_{n-1}) + \sum_{\substack{t \leq k < n-1 \\ : \rho(\overline{x}_k)=1}} 2^k + \sum_{\substack{t \leq k < m-1 \\ : \rho(y_k)=1}} 2^k. \tag{2.58}$$

Based on the previous steps, there are 4 cases.

- If $A \geq 0$ and $m \geq n$, then $T = 2^{m-1} = 2^{\max(m,n)-1}$.
- If $A \geq 0$ and $m < n$, then $T = 2^{m-1} + (2^{m-1} + \cdots + 2^{n-2}) = 2^{n-1} = 2^{\max(m,n)-1}$.
- If $A < 0$ and $m \geq n$, then $T = 2^{n-1} + (2^{n-1} + \cdots + 2^{m-2}) = 2^{m-1} = 2^{\max(m,n)-1}$.
- If $A < 0$ and $m < n$, then $T = 2^{n-1} = 2^{\max(m,n)-1}$.

Next consider the sum of all literals with index $k \in \{t-s, \ldots, t-1\}$. By the inductive hypothesis, exactly one of y_k and $\overline{x}_k$ propagates, so they can only contribute 2^k to exactly one of the slacks.

So, we have

$$\textit{slack}(C_1, \rho) + \textit{slack}(C_3, \rho) \tag{2.59}$$

$$= 2^{\max(m,n)-1} + \sum_{k=t-s}^{t-1} 2^k + 2\sum_{k=0}^{t-s-1} 2^k - (A + 2^{m-1}) - (-C + 2^{n-1} - 1) \tag{2.60}$$

$$= 2^{\max(m,n)-1} + 2^t - 2^{t-s} + 2(2^{t-s} - 1) - 2^{m-1} - 2^{n-1} + C - A + 1. \tag{2.61}$$

And then recall $t = \min(m, n) - 1$, so $2^{\max(m,n)-1} + 2^t = 2^{n-1} + 2^{m-1}$. Hence

$$\textit{slack}(C_1, \rho) + \textit{slack}(C_3, \rho) \tag{2.62}$$

$$= 2^{t-s} - 1 + C - A < 2^{t-s} - 1 - B < 2^{t-s} - 1 \tag{2.63}$$

Now if either of these slack values are negative, we have already reached a contradiction by unit propagation. And if both are non-negative, at least one of them is at most $\lfloor \frac{1}{2}(2^{t-s} - 1) \rfloor = 2^{t-s-1} - 1$. So either $y_{t-(s+1)}$ or $\overline{x}_{t-(s+1)}$ must propagate. Add this to the assignment ρ.

We now compute the slack of C_2 under ρ. Note C_2 contains exactly all the opposite literals to those in C_1 and C_2, appearing with the same coefficients. The total contribution of all literals with index at least t added together is

$$T = (2^{\max(m,n)-1} + \cdots + 2^{\min(m,n)-1}) + 2^{\min(m,n)-1} = 2^{\max(m,n)} \tag{2.64}$$

so the contribution of those that appear in C_2 is exactly $2^{\max(m,n)-1}$.

So finally, we have

$$\begin{aligned}
&\mathit{slack}(C_2, \rho) && (2.65)\\
&= 2^{\max(m,n)-1} + \sum_{k=t-s-1}^{t-1} 2^k + 2\sum_{k=0}^{t-s-2} 2^k - B - 2^{m-1} - 2^{n-1} + 1 && (2.66)\\
&= 2^{\max(m,n)-1} + 2^t - 2^{t-s-1} + 2(2^{t-s-1} - 1) - B - 2^{m-1} - 2^{n-1} + 1 && (2.67)\\
&= 2^{t-s-1} - 1 - B < 2^{t-s-1} && (2.68)
\end{aligned}$$

and hence C_2 propagates $x_{t-(s+1)}$ or $\overline{y}_{t-(s+1)}$. This completes the induction. Now we see that we either reach a conflict in C_1 or C_3 during the process, or else we eventually set every literal by unit propagation, and since the inequalities are incompatible, this must in the end be a contradiction. □

Generalised versions of Theorem 2.9 do not always hold. For example, it seems at first plausible that the three inequalities would always unit propagate to contradiction for *any* $A + B - C > 0$, regardless of whether $B \in \{0, 1\}$. However, for specific values of A, B, C, m and n, this is not true.

Example 2.15 (Non-propagating contradictory constraints on binary sums).
The contradictory constraints

$$\begin{aligned}
C_1 &:= -8y_3 + 4y_2 + 2y_1 + y_0 && \geq -2 && (2.69)\\
C_2 &:= 8y_3 - 4y_2 - 2y_1 - y_0 - 8x_3 + 4x_2 + 2x_1 + x_0 && \geq 3 && (2.70)\\
C_3 &:= && 8x_3 - 4x_2 - 2x_1 - x_0 \geq 0 && (2.71)
\end{aligned}$$

do not unit propagate to contradiction. This can be seen by computing the slack values

$$\begin{aligned}
\mathit{slack}(C_1) &= 15 - 6 = 9 && (2.72)\\
\mathit{slack}(C_2) &= 30 - 12 = 18 && (2.73)\\
\mathit{slack}(C_3) &= 15 - 7 = 8 && (2.74)
\end{aligned}$$

which tell us (via Proposition 2.1) that no propagation occurs.

Of course, contradiction can easily be derived from these constraints by simply adding them up, but the point is that not all obviously contradictory constraints will unit propagate to contradiction, and likewise, even though any two of the above constraints imply the negation of the third, it will not be possible to introduce it purely by RUP.

Furthermore, even fundamental contradictions do not always unit propagate. For instance, $C \wedge \neg C$ is not guaranteed to unit propagate to contradiction, as the following example demonstrates.

Example 2.16 (Non-propagation of a constraint with its negation). *The constraints*

$$C_1 := x + y + z \geq 2 \quad \textit{and} \quad \neg C_1 := \overline{x} + \overline{y} + \overline{z} \geq 2 \tag{2.75}$$

do not together propagate to contradiction. In fact, since the slack of both constraints is equal to 1*, neither propagates anything at all.*

This can be easily extended to show that there does not exist a sequence of RUP steps deriving C_1 from itself, establishing that RUP is not implicationally complete. Suppose $C_2, \ldots, C_k$ is a sequence of RUP constraints derived from C_1, with $C_k = C_1$. Then by the implicational property of RUP, $C_1 \wedge \cdots \wedge C_{k-1}$ admits the same solutions as C_1, and so none of $\{x, y, z, \bar{x}, \bar{y}, \bar{z}\}$ can propagate from this, as none of these are implied by C_1. And since $\neg C_k$ alone does not propagate anything under the empty assignment, the only remaining option is for $C_1 \wedge \cdots \wedge C_{k-1}$ to unit propagate to contradiction, which would violate soundness.

What this means is that "RUP" should certainly not be understood as a synonym for "implied", or even "obviously implied", even though RUP constraints are always implied and usually obviously so.

2.4 Summary

The primary function of this chapter has been to establish groundwork to which the rest of the thesis can refer. We reviewed the fundamental concepts of pseudo-Boolean reasoning and how they can be applied within a proof system and machine-checkable logging format. We also explicated some useful derivation patterns that are crucial to the effectiveness of using a PB system to certify solving paradigms beyond $0-1$ linear inequalities. Of particular importance for this work is the behaviour of PB unit propagation on binary exponential sums as given by Theorems 2.7 to 2.9; and the fact that we can "unrestrict" proofs using Theorem 2.1.

Now that we have an understanding of the proof system, and some idea of its capabilities, we are ready to apply this specifically in the context of constraint programming representations and stronger reasoning.

Chapter 3

Proof Logging for Constraint Programming

We will now present a framework for using pseudo-Boolean proofs to create a certifying constraint programming solver. This is a refinement of the 2022 "Auditable Constraint Programming" methodology of Gocht et al. [94], and it has been informed by the author's participation in the continued development of the associated "*Glasgow Constraint Solver*" project [150]. We will incorporate the improved understanding that has evolved through the course of developing the justification procedures presented later in the thesis, and take advantage of the PB properties established in the previous chapter to explain the *Auditable CP* methodology more rigorously.

This chapter begins by defining constraint satisfaction and optimisation problems, and discussing at a high level how standard CP solvers work. We will then look at encoding CP problems to a PB format, showing how this can be done in such a way as to ensure the proof corresponds to the original input problem. With this in place, we can outline exactly what an auditable CP solver needs to log in order to guarantee a complete proof of any (correct) result it arrives at, and hence properly motivate the need for designing justification procedures for constraint propagation. Finally, we exhibit some simple justifications for fundamental constraints such as linear inequalities and show why they are correct according to this framework.

3.1 Constraint Programming Fundamentals

A constraint satisfaction problem (CSP) is traditionally defined by a set of *variables*; sets of *domain* values associated with each variable; and a set of *constraints*, each of which somehow restricts the combinations of values that the variables can take simultaneously. This definition is extremely general, but in practice we assume all of these sets are ordered and finite, and each variable is a named symbol such as X_i whose domain is a finite set of integers $\mathrm{dom}(X_i)$. This allows us to say that a valid *assignment* for a CSP is a mapping σ from variables to integers where $\sigma(X_i) \in \mathrm{dom}(X_i)$ for each variable X_i. We will assume each constraint C is also named

and specified over an ordered subset of the variables $\mathrm{scp}(\mathsf{C})$ which we call its *scope*, and call the number of variables in a constraint's scope its *arity*. As with PB constraints, we will use $\mathsf{Vars}(\mathcal{C}) := \bigcup_{\mathsf{C}\in\mathcal{C}} \mathrm{scp}(\mathsf{C})$ to denote all variables appearing in the scopes of set of constraints $\mathcal{C}$.

It will sometimes be useful to talk about *extending* and *restricting* assignments. This has the usual meaning in terms of mappings: if an assignment σ' is defined for a subset $\mathcal{S}$ of the variables that some another assignment σ is defined for; and $\sigma'(X) = \sigma(X)$ for all $X \in \mathcal{S}$; then σ' is a restriction of σ, and σ is an extension of σ'.

Finally, since domains are almost always modified throughout the process of solving a CSPs, we will often talk about a *domain state* at a particular point in time t. Formally, we say a *domain state* (or "state of domains") dom_t is just a function from variables to sets of values (domains). This allows us to keep notation and terminology consistent: $\mathrm{dom}_t(X)$ is the *domain* of X with respect to the *state of domains* at point t, and $\mathrm{dom}_0(X)$ is the initial domain of X with respect to the starting domain state as specified in the CSP/COP problem description. An assignment is *valid* for a particular domain state if for every variable X in the problem, $\rho(X_i) \in \mathrm{dom}_t(X_i)$. We can write $\mathsf{Valid}(\rho, \mathrm{dom}_t)$ for this.

In the literature (e.g. Schulte and Stuckey [179] and others [72, 180]), what we call "a domain state" is sometimes simply referred to as "a domain". In this thesis we will attempt to distinguish whenever possible between domains of individual variables and the *domain state* of *all* the variables in the problem at a particular point. We will however adopt similar relational notation for our domain states as is commonly used for domains in CP literature, and define a partial order on domain states defined over the same set of variables $\mathcal{X}$ by lifting the subset relation. Formally,

$$\mathrm{dom}_t \sqsubseteq \mathrm{dom}_u \iff \forall X \in \mathcal{X} \,.\, \mathrm{dom}_t(X) \subseteq \mathrm{dom}_u(X). \tag{3.1}$$

We say that dom_t is *stronger* than dom_u here, and that dom_u is *weaker* than dom_t.

3.1.1 Constraints and Solutions

It is debatable which mathematical object best encompasses the concept of a constraint, even in the finite integer setting. Often a constraint is said to be defined by relation: a subset of $\prod_{X_i\in\mathrm{scp}(\mathsf{C})} \mathrm{dom}(X_i)$ which specifies explicitly the list of allowed combinations of values for the variables in scope, sometimes called an *extensional* representation [141]. But this does not reflect the fact that solvers do not generally store this list anywhere (excepting Table constraints, see Example 3.1), and creates the annoyance that many constraints require an exponentially large relation, but do not require exponential space or time to check whether a given tuple is allowed. Another possibility is to say a constraint is simply a (computable, ideally efficiently computable) predicate, allowing a more syntactic view where mathematical expressions or programming language descriptions themselves define the constraint. This is the *intensional* view of constraints, which can be harder to work with in a formal setting but perhaps more accurately reflects real-

world use. It is also possible to forgo a general definition of constraints entirely, and instead require each constraint to be an instantiation of a *constraint type* from a predefined collection, such as linear and non-linear inequalities and non-equalities [179].

In any case, the important point is that given a tuple of domain values for variables in a constraint's scope, we can determine whether the constraint allows it. We say a valid assignment σ *satisfies* a constraint $\mathsf{C} \in \mathcal{C}$ if the tuple $\prod_{X_i \in \text{scp}(\mathsf{C})} \sigma(X_i)$ is permitted (i.e. the predicate is true, or it is contained in the relation). Overall the CSP is *satisfiable* if there exists at least one assignment, called a *solution*, that satisfies every constraint simultaneously, and deciding whether this is the case is one of the principal tasks of a CP solver. Another common task is to find a solution that maximises or minimises the value of a particular variable X_o: we call this solving a *constraint optimisation problem* (COP). The solution σ is *optimal* in a maximisation problem if for any other solution σ' we have $\sigma(X_o) \geq \sigma'(X_o)$ ($\leq$ respectively for minimisation problems).

3.1.2 Modelling Problems

Clearly these concepts generalise the NP-hard SAT and pseudo-Boolean solving paradigms. A CNF formula can be seen as a CSP where all variables have domain $\{0, 1\}$ and all constraints are clauses. Likewise, a PB formula is a CSP, and can be made into a COP by conceptually considering an additional objective variable and an additional constraint ensuring it is equal to the value of the PB objective function. *Integer linear programming* (ILP) problems [178] on bounded variables can similarly be thought of as a kind of COP, since they are specified by a set of integer linear equality and inequality constraints on integer variable, whose domains can be defined to be the set of integers within the range of each variable's bounds.

Theoretically speaking, the general CP setting imposes very few restrictions on the kind of constraints that can be introduced. In practice, however, we still assume a collection of allowed constraint types defined in some *constraint modelling language* or *toolkit*. For example, the *MiniZinc* language provides a way of expressing parameterised CSPs/COPs with a library of around 250 constraint types [156]. Other systems include *XCSP3* [35], *Essence* [73] and *CPMpy* [99], each with different syntax and features.

Common to all of these is the designation of *global constraints*. A global constraint type captures precise relational semantics over an arbitrary number of variables, in contrast to *primitive constraints* which usually enforce simple relations of a fixed arity.

Example 3.1 (Global constraints). *Some common global constraints include:*

- $\mathsf{AllDifferent}(X_1, \ldots, X_n)$*, which says that the values taken by* $X_1, \ldots, X_n$ *must all be distinct;*
- $\mathsf{Count}(X_1, \ldots, X_n, V, C)$*, which says that the number of variables taking the same value as* V *among variables in* $X_1, \ldots, X_n$ *must be equal to the value of* C*;*

- Element$(X_1, \ldots, X_n, Y, Z)$, *which says that* $Y = i$ *if and only if* $X_i = Z$*;*
- Max$(X_1, \ldots, X_n, M)$ *which says that* M *is equal to the max value among* $X_1, \ldots, X_n$*. Similarly, we have* Min$(X_1, \ldots, X_n, M)$*.*

There is also Table$(X_1, \ldots, X_n; \tau)$*, which is the most explicit extensional constraint.* τ *is a set of integer* n*-tuples (a "table") and the constraint requires that the tuple of values taken by* $X_1, \ldots, X_n$ *must be a member of* τ *(must match a "row" in the table).*
Any constraint with precise semantics can in theory be specified as a table constraint, although the size of the τ *may grow very large.*
A collection of $400+$ *global constraint patterns and variations used across academia and industry is currently available online in the form of the "global constraint catalogue" [20].*

We refer to a high-level description of a problem in a constraint modelling language as a *model* of the problem, although this term is sometimes used in formal logic contexts as a synonym for what we are here calling a "solution". The model may employ high-level constructs such as loops and quantifiers, but these are usually removed by the modelling system before passing to a solver by rewriting them in terms of fundamental supported constraints. In fact, one approach to solving CP problems is to translate all constraints entirely to a lower-level paradigm such as SAT or ILP. Our focus, however, is on native CP solvers, which generally have an interface that allows posting of primitive and global constraints on finite domain variables.

3.1.3 Solving Techniques

Once a supported model is passed to a solver, it can apply a range of techniques to find solutions or decide unsatisfiability. The discipline of devising and studying these techniques is expansive and has been developing over many decades, so we can only cover a surface-level overview here. A more detailed discussion of solving fundamentals can be found in the Handbook of Constraint Programming [175].

A core component of many solvers is *backtracking search*, an improvement on the obviously inefficient approach of generating all possible assignments and testing to see if the constraints are satisfied. With a complete backtracking algorithm, the solver creates subproblems with successively stronger domains that partition the space of all valid assignments, and solves them recursively. We can refer to the creation of each subproblem as the result of a *decision* made by the solver. If at any point a constraint is violated or a complete solution is found, the solver *backtracks* by undoing the last decision, restoring domain changes, and exploring another subproblem, or returning to the parent problem if all possibilities been tried. This is guaranteed to enumerate all solutions or determine if none exist, terminating once the solver is ready to backtrack from the initial decision point.

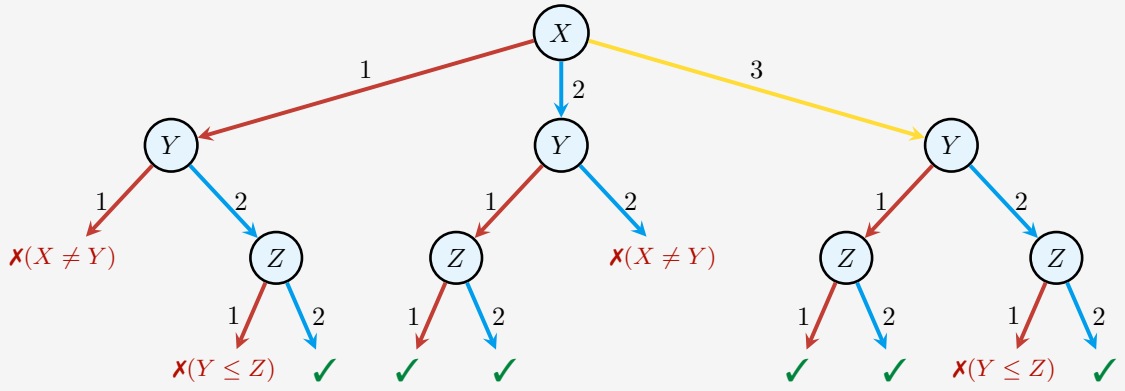
It is straightforward to visualise this as a search tree, with decision points as nodes and

decisions as branches, which is why the process of choosing variables and guessing values is often called *branching*. A common branching strategy is *k-way* branching: choosing a *decision variable* X, and "guessing" in turn each of the $|\operatorname{dom}_t(X)|$ possible assignments for that variable based on the current domain state dom_t. Other options are *two-way* branching: guessing $X = d$ and then $X \neq d$ respectively for some $d \in \operatorname{dom}_t(X)$; or some other form of *domain splitting*. See Chapter 4 of the Handbook of Constraint Programming [175] for further details.

Example 3.2 (Backtracking search).
Consider a simple CSP with variables $\{X, Y, Z\}$*, where*

$$\operatorname{dom}(X) = \{1, 2, 3\}; \quad \operatorname{dom}(Y) = \{1, 2\}, \quad \textit{and} \quad \operatorname{dom}(Z) = \{1, 2\};$$

and with constraints enforcing $X \neq Y$, $Y \leq Z$*. A possible search tree resulting from a pure backtracking algorithm with k-way branching is as follows.*



Regardless, although conventional backtracking is better than generating all possibilities, it can still be "grotesquely inefficient" [143] on even moderately-sized problem instances. Hence, most CP solvers also implement *inference* algorithms that attempt to remove values from variables' domains during search, reducing the possibilities for each decision and restoring the removed values upon backtracking. Conceptually, a value can be removed from a domain if there is no solution to the CSP instance where the variable takes that value. We call such a value *inconsistent* with the CSP and say that the inference method removing a set of inconsistent values is *enforcing* (some level of) *consistency*. In principle the strongest level of consistency that can be enforced is *global consistency*, where all inconsistent values are removed from the domains of variables. This would immediately determine the satisfiability of the problem, since in an unsatisfiable CSP all values are obviously inconsistent, and is therefore NP-hard to compute in general. So in practice solvers usually enforce some kind of *local consistency*, detecting values that are inconsistent with a single constraint or subset of constraints.

Given a variable X, domain state dom_t, value $v \in \operatorname{dom}_t(X)$, and a constraint C with

$X \in \mathrm{scp}(\mathsf{C})$, a *support* for (X, v) is a valid assignment of all variables in $\mathrm{scp}(\mathsf{C})$ that both includes the assignment $X = v$ and satisfies the constraint. If for every variable $X \in \mathrm{scp}(\mathsf{C})$ and every value $v \in \mathrm{dom}_t(X)$, the variable-value pair (X, v) has at least one support, we say that the constraint is *domain consistent*. This is also often referred to as *generalised-arc consistency* since the definition has its origins in the view of CSPs as constraint networks where are thought of edges and constraints as (hyper-)arcs [141].

Another notion is *bounds-consistency*, which comes in a variety of forms [42], but essentially only requires the maximum and minimum values (*bounds*) of the domain to have support, in addition to possibly relaxing the definition of support itself to include mapping to non-domain (known as bounds-$\mathbb{Z}$ consistency) or even noninteger values (known as bounds-$\mathbb{R}$ consistency) within the bounds of each variable.

Removing a set of values inconsistent with just one constraint, or else detecting infeasibility, is often called *propagation* of that constraint, with the procedure for doing so known as a constraint *propagator* or a *filtering algorithm*. Many propagators enforce domain consistency or bounds consistency, and a standard approach to inference in a solver is to run propagators for all constraints in some order until a fixed point is reached. Each constraint type usually requires a specialised propagator, although there are many different possible algorithms for each constraint and consistency property. Of course, for many constraints, computing domain-consistency is NP-hard even in itself and so ad-hoc propagators that remove just some of the inconsistent values may be employed.

In addition to search and propagation, solvers may also *learn* additional constraints during search. From a certain point of view, any backtracking solver learns a clause of dis-assignments whenever it backtracks, sometimes referred to as a "nogood". For optimisation problems, solvers also "learn" a bound constraint on the objective variable whenever a solution is discovered, which says the variable must get a "better" (larger or smaller) value in future solutions. "Lazy clause generation" (LCG) solvers extend these notions by learning more general nogoods using *conflict analysis* [160], employing hybrid CP-SAT technology and propagators that generate clausal *explanations* for their inferences.

These are the main solving techniques that are relevant from the point of view of proof logging, with the focus of this thesis being on justifying propagation algorithms. There are many other techniques, including local search, belief propagation, symmetry-breaking, streamliners, relaxations, tree decomposition etc., which we will not explore in order to keep the scope manageable, although we will touch upon *dynamic programming* techniques in Chapter 5. There are also many solver design choices such as heuristic selection, domain representation, how to restore state etc., which we do not need to cover, since, as a rule, proof logging is more concerned with *what* a solver does in terms of modifying domains, and not so much *how* it achieves this.

3.2 Representing Problems for Proofs

The philosophy of using pseudo-Boolean proof logging for constraint programming solvers hinges on the idea that, for any CSP or COP, we can produce a pseudo-Boolean encoding that can be reasonably understood as (and trusted to be) the *same* problem. Recall Figure 1.3.

Example 3.3 (PB encoding of a CP problem). *Consider the CSP with variables, X, Y, Z all with domain $\{0, 7\}$ and constraints $X = Y$, $Y \neq Z$, $X + Z \geq 3$. Intuitively, the PB formula*

$$x_0 + 2x_1 + 4x_2 - y_0 - 2y_1 - 4y_2 \geq 0; \tag{3.2}$$
$$-x_0 - 2x_1 - 4x_2 + y_0 + 2y_1 + 4y_2 \geq 0; \tag{3.3}$$
$$u \Rightarrow y_0 + 2y_1 + 4y_2 - z_0 - 2z_1 - 4z_2 \geq 1; \tag{3.4}$$
$$\overline{u} \Rightarrow -y_0 - 2y_1 - 4y_2 + z_0 + 2z_1 + 4z_2 \geq 1; \tag{3.5}$$
$$x_0 + 2x_1 + 4x_2 + z_0 + 2z_1 + 4z_2 \geq 3; \tag{3.6}$$

represents the same problem, since we can view each of the PB variables x_i, y_i, and z_i as representing the value of the i-th bit in the binary representation of the CP variables, and the "flag" variable u ensures that either $Y > Z$ or $Z > Y$.

In this section we expand on the intuition of the above example, and give a method (following Gocht et al. [94]) for producing a PB formula that is obviously equivalent to an input CSP, at least for some generalised definition of the word "obvious". For clarity, we will talk about transforming a "CP problem" into a "PB problem".

3.2.1 The Difficulty of Equivalence

To begin with, we must decide on what equivalence actually means in the context of CP proof logging, since conceptually transforming a CSP intended for a CP solver into PB is just a special case of transforming one CSP into another. There is no standard definition of this agreed upon in the constraint satisfaction community, with various interpretations proposed that differ in their restrictiveness. The Handbook of Constraint Programming [175, p.5] says the following.

> "It is difficult to define precisely what we mean when we say that a CSP represents a problem P. A possible definition is that: a CSP $M = \langle X, D, C \rangle$ represents a problem P, or M is a model of P, if every solution of C corresponds to a solution of P and every solution of P can be derived from at least one solution to C."

Rossi et al. [174] give an alternative definition based on *mutual reducibility*, and they contrast this with a much stricter, arguably more naïve, conception of equivalence that simply requires

two problems to share the same sets of variables, domains, and solutions. Other notions such as *viewpoints* [78] and variable *representations* [127] have also been explored.

We will not adopt any of these definitions exactly, since the ultimate goal of encoding for proof logging purposes is to trust that the process preserves the integrity of the *proof* with respect to the original problem, and potentially to formally verify this process in a theorem prover. So at one level, the only guarantees we really need are:

P1. The PB problem is satisfiable if and only if the CP problem is satisfiable;

P2. If there is an objective variable, the optimal value of the CP objective is equal to the optimal value of the PB objective.

However, to make it easier to obtain these guarantees, we should employ very restricted transformations with minimal reformulation, much less than one might use if encoding in order to use a PB or SAT system to *solve* the problem. To put it simply, we want our encodings to be as "dumb" as possible because we want to be extremely certain the two properties hold. So in practice, our working definition of equivalence is: "the PB problem is the result of applying Encoding Procedure 3.1 to the CP problem". We will define this procedure shortly.

If we are convinced that the transformation respects P1 and P2, we can be convinced that when we have an UNSAT proof for the PB problem, the CP problem is indeed UNSAT, and when we have a proof that the optimal value of the PB objective is a, then the optimal value of the CP problem is in fact a.

It is conceptually useful to view the encoding process in several stages, even if it is implemented in practice as a single pass. At each stage we can pay close attention to what is being preserved, and if we are convinced by the soundness of each stage, we should be convinced that the whole process is safe.

3.2.2 A Transformation Procedure

The basic idea of our transformation procedure is to first turn the CP problem into an equivalent problem with only reified linear constraints (an "intermediate ILP"). This still contains the original non-Boolean variables of the problem and so it is easy to argue equivalence by showing that the effect of the linear constraints is to restrict the variable's values in exactly the same way as the original CP constraints. We have the freedom to introduce auxiliary variables at this stage to assist in the linearisation (such as the u flag in Example 3.3). In particular, we will designate a special class of Boolean variable called "atomic constraint variables" (or simply "atomic variables") intended to represent basic relationships between variables and values. Our linearisations will be defined on a per-constraint basis, and then joined together to create the full equivalent problem. The soundness of this concatenation is ensured by requiring that only the original variables and the atomic variables are shared between linearisations, see Lemma 3.1.

We will assume we start with a general constraint satisfaction problem, with variable set $\mathcal{V}$, finite domains $\mathrm{dom}_0(X_i)$ for $X_i \in \mathcal{V}$, and constraints $\mathcal{C}$. All domains can be assumed to be ranges of integers without holes (since any missing values can be modelled with additional constraints if necessary). We also assume that the constraints are a flat list, without constructs such as loops that might be found in a higher-level modelling language.

The high level encoding process is then described by Encoding Procedure 3.1. The remainder of this section gives more details on each stage along with soundness proofs. Notation for various parts of the problem representation that will be used throughout this thesis will also be introduced here.

Encoding Procedure 3.1 *For a CSP with variables $\mathcal{V}$, initial domain state (without holes) dom_0, and constraints $\mathcal{C}$, we can produce a PB formula as follows.*

1: $\mathsf{Bnd}(\mathcal{V}) \leftarrow \bigcup_{X \in \mathcal{V}} \mathsf{Bnd}(X, \mathrm{dom}_0)$
2: $\mathcal{A} \leftarrow \bigcup_{\mathsf{C} \in \mathcal{C}} \mathsf{NeededAtomicVariables}(\mathsf{C}, \mathrm{dom}_0)$
3: $\mathsf{Def}(\mathcal{A}) \leftarrow \bigcup_{y \in \mathcal{A}} \mathsf{Def}(y)$
4: $\mathsf{Lin}(\mathcal{C}) \leftarrow \bigcup_{\mathsf{C} \in \mathcal{C}} \mathsf{Linearise}(\mathsf{C}, \mathrm{dom}_0)$
5: $F \leftarrow \mathsf{BinEnc}(\mathsf{Bnd}(\mathcal{V}) \cup \mathsf{Def}(\mathcal{A}) \cup \mathsf{Lin}(\mathcal{C}))$
6: $F \leftarrow \mathsf{Standardise}(F)$

Step 1: Create domain bound constraints.

For a CP variable X and domain state dom_0 let $\mathsf{Bnd}(X, \mathrm{dom}_0)$ be the two linear constraints

$$X \geq \min(\mathrm{dom}_0(X)) \quad \text{and} \quad X \leq \max(\mathrm{dom}_0(X)). \tag{3.7}$$

When it is clear from context, we can omit dom_0 and simply write $\mathsf{Bnd}(X)$ for these, and for a set of variables $\mathcal{X}$ we can represent the set of all bound constraints for variables in $\mathcal{X}$ by $\mathsf{Bnd}(\mathcal{X})$.

Clearly if we construct a CSP with variables $\mathcal{V}$, constraints $\mathcal{C} \cup \mathsf{Bnd}(\mathcal{V})$, and unrestricted integer domains (i.e. for all $X \in \mathcal{V}, \mathrm{dom}(X) = \mathbb{Z}$) then this has exactly the same solution set as our original CSP.

Steps 2 and 3: Introduce Atomic Constraint Variables.

We now designate our set of atomic constraint variables $\mathcal{A}$. These are fresh 0-1 variables that can be shared between linearisations and have a special meaning, closely related to "atomic constraints" as first discussed by Choi et al. [43]. For each supported constraint type C, we first need to know which atomic variables it requires, so the function $\mathsf{NeededAtomicVariables}$ should be defined on a per-constraint basis. Usually this will be immediately clear from the definition of $\mathsf{Linearise}(\mathsf{C}, \mathrm{dom}_0)$, as will be used in step 4.

Each atomic variable y requires a fully reified linear *definition* constraint $\mathsf{Def}(y)$ of the form $y \Leftrightarrow C$. We only allow two types of atomic variable definition, associated specifically with a CP variable $X_i \in \mathcal{V}$ and some integer v. For clarity, we will always use subscripted names for atomic variables suggestive of their definition.

1. A *bound variable* $x_{i\geq v}$ represents whether X_i must be at least v, and has definition constraint

$$\mathsf{Def}(x_{i\geq v}) := x_{i\geq v} \Leftrightarrow X_i \geq v \tag{3.8}$$

2. An *equality variable* $x_{i=v}$ represents whether X_i must be equal to v, and has definition constraint

$$\mathsf{Def}(x_{i=v}) := x_{i=v} \Leftrightarrow x_{i\geq v} + \overline{x}_{i\geq v+1} \geq 2 \tag{3.9}$$

   Naturally, if an equality variable is needed, the corresponding bound variables are also needed.

Because these are fully reified constraints on fresh variables they are entirely redundant: the values of the atomic variables are uniquely determined for any solution. This means that the solution set projected onto the original variables remains exactly the same, and in particular, adding them cannot possibly change satisfiability or the objective value.

Recalling 0-1 *literals* from Chapter 2, we can represent X_i being *at most* v and *not equal to* v by negating bound or equality variables. Collectively we will refer to literals of the form $x_{i\geq v}, \overline{x}_{i\geq v}, x_{i=v}$ and $\overline{x}_{i=v}$ as *atomic (constraint) literals*.

Step 4: Linearise Constraints

For each constraint C in the original constraint set, we next require a function definition $\mathsf{Linearise}(\mathsf{C}, \mathrm{dom}_0)$ that turns it into a set of reified linear constraints $\mathsf{Lin}(\mathsf{C})$. This can use the original variables, atomic literals, and potentially further auxiliary variables. Formally we can write $\mathsf{Vars}(\mathsf{Lin}(\mathsf{C}))$ for the set of variables appearing in the linearisation, and let $\mathsf{Aux}(\mathsf{C}) = \mathsf{Vars}(\mathsf{Lin}(\mathsf{C})) \setminus (\mathcal{V} \cup \mathcal{A})$ denote the set of auxiliary variables introduced. We can then require that the linearisation must respect the following properties.

L1. Each constraint in any $\mathsf{Lin}(\mathsf{C})$ fits one of two general forms: either

$$\sum_{j=0}^{n-1} a_j \cdot V_j \bowtie b \quad \text{or} \quad r_1 \wedge \cdots \wedge r_k \rightleftharpoons \sum_{j=0}^{n-1} a_j \cdot V_j \bowtie b \tag{3.10}$$

   where b and each a_i are integers; each r_k is a 0-1 literal; each V_j is either a 0-1 or an integer variable; $\rightleftharpoons$ is one of $\{\Leftarrow, \Rightarrow, \Leftrightarrow\}$; and $\bowtie$ is one of $\{\geq, \leq, =\}$.

L2. Any non-0-1 auxiliary integer variable $V_j \in \mathsf{Aux}(\mathsf{C})$ must be bounded with corresponding bound constraints $\mathsf{Bnd}(V_j)$. These bounds are considered part of the linearisation.

L3. Auxiliary variables must not be shared between linearisations, i.e.

$$\mathsf{Aux}(\mathsf{C}_1) \cap \mathsf{Aux}(\mathsf{C}_2) = \varnothing \qquad \text{for all } \mathsf{C}_1 \neq \mathsf{C}_2.$$

L4. Most importantly, $\mathsf{Lin}(\mathsf{C}) \cup \mathsf{Def}(\mathcal{A})$ must enforce the *same restriction* as C on the variables in $\mathcal{V}$. That is, a valid assignment $\sigma : \mathcal{V} \to \mathbb{Z}$ satisfies C if and only if it can be extended to a valid assignment $\sigma_+ : \mathsf{Vars}(\mathsf{Lin}(\mathsf{C}) \cup \mathsf{Def}(\mathcal{A})) \to \mathbb{Z}$ that satisfies $\mathsf{Lin}(\mathsf{C}) \cup \mathsf{Def}(\mathcal{A})$.

We will define Linearise concretely for several fundamental CP constraint types in the next section. Provided we are convinced that each linearisation is valid and respects the above properties, we can be sure that composing them to form a linear constraint problem maintains a strong correspondence with the original problem. We can state this formally as follows.

Lemma 3.1 (Composing linearisations). *Let P be a CSP with variables $\mathcal{V}$ and constraints $\mathcal{C}$, where all domains are integer ranges. After executing steps 1 to 4 of Encoding Procedure 3.1 let P' be the CSP with constraints $\mathcal{C}' := \mathsf{Lin}(\mathcal{C}) \cup \mathsf{Bnd}(V) \cup \mathsf{Def}(\mathcal{A})$, whose variables are precisely those occurring in $\mathcal{C}'$, and whose domain for every non-0-1 variable is $\mathbb{Z}$. Then we have:*

1. *Every solution σ of P can be* extended *to a solution σ' of P'.*
2. *Every solution σ' of P' can be* restricted *to a solution σ of P.*

Proof.

1. Let σ be a solution to P, and let $\mathsf{C}_1, \ldots, \mathsf{C}_n$ be the constraints of P. By the definition of validity σ satisfies all the constraints in $\mathsf{Bnd}(\mathcal{V})$. And by L4 above, σ can be extended to a solution of σ_1 of $\mathsf{Bnd}(\mathcal{V}) \cup \mathsf{Def}(\mathcal{A}) \cup \mathsf{Lin}(\mathsf{C}_1)$.
   Now assume we have extended σ to a solution σ_{k-1} of $\mathsf{Bnd}(\mathcal{V}) \cup \mathsf{Def}(\mathcal{A}) \cup \bigcup_{i=0}^{k-1} \mathsf{Lin}(\mathsf{C}_i)$. We know again by L4 that σ can at least be extended to a solution σ'_k to $\mathsf{Bnd}(\mathcal{V}) \cup \mathsf{Def}(\mathcal{A}) \cup \mathsf{C}_k$. So define a new solution assignment σ_k by

$$\begin{cases} \sigma_k(X) = \sigma(X) & \text{if } X \in \mathcal{V} \quad (3.11) \\ \sigma_k(\ell) = \sigma_{k-1}(\ell) & \text{if } \ell \in \mathcal{A} \quad (3.12) \\ \sigma_k(V) = \sigma_{k-1}(V) & \text{if otherwise } V \in \mathsf{Vars}(\cup_{i=0}^{k-1} \mathsf{Lin}(\mathsf{C}_i)) \quad (3.13) \\ \sigma_k(V) = \sigma'_k(V) & \text{if otherwise } V \in \mathsf{Vars}(\mathsf{Lin}(\mathsf{C}_k)) \quad (3.14) \end{cases}$$

   This is an extension of σ, and it must satisfy both $\mathsf{Lin}(\mathsf{C}_k)$ and $\mathsf{Bnd}(\mathcal{V}) \cup \mathsf{Def}(\mathcal{A}) \cup \bigwedge_{i=0}^{k-1} \mathsf{Lin}(\mathsf{C}_k)$ because

- $\sigma_k(X) = \sigma_{k-1}(X) = \sigma'_k(X)$ for any $X \in \mathcal{V}$;
- $\sigma_k(\ell) = \sigma_{k-1}(\ell) = \sigma'_k(\ell)$ for $\ell \in \mathcal{A}$; (since $\mathsf{Def}(\mathcal{A})$ uniquely determines the values of $\mathcal{A}$ for any assignment to $\mathcal{V}$)
- $\mathsf{Vars}(\cup_{i=0}^{k-1}\mathsf{Lin}(\mathsf{C}_i)) \bigcap \mathsf{Vars}(\mathsf{Lin}(\mathsf{C}_k)) \subseteq \mathcal{V} \cup \mathcal{A}$ by L3; which means σ_k is an extension of both σ_{k-1} and σ'_k.

The claim then follows by induction, as $P' = \mathsf{Bnd}(\mathcal{V}) \cup \mathsf{Def}(\mathcal{A}) \cup \bigwedge_{i=1}^{n} \mathsf{Lin}(\mathsf{C}_i)$

2. Let σ' be a solution to P'. There is only one possible restriction of σ' to a valid assignment to just $\mathcal{V}$, and by L4, since σ' satisfies $\mathsf{Lin}(\mathsf{C}_i) \cup \mathsf{Def}(\mathcal{A})$ for each $1 \leq i \leq n$; σ must satisfy each C_i and hence satisfy P. □

We will refer to the result of the previous three steps as the *intermediate ILP encoding*. In the linearisations presented in this thesis, the properties L1 to L4 should be verifiable by inspection.

Example 3.4 *An intermediate ILP encoding for the CSP in Example 3.3 could be*

$$X, Y, Z, u \in \mathbb{Z} \tag{3.15}$$

$$0 \leq X \leq 7; \quad 0 \leq Y \leq 7; \quad 0 \leq Z \leq 7; \quad 0 \leq u \leq 1; \tag{3.16}$$

$$X - Y = 0; \tag{3.17}$$

$$u \Rightarrow X - Y > 0; \quad \overline{u} \Rightarrow Y - X > 0; \tag{3.18}$$

$$X + Z \geq 3. \tag{3.19}$$

From this point, it is easy to turn the intermediate ILP into an "obviously correct" PB encoding by simply performing a few further transformations.

Step 5: Binary encoding of integer variables

The procedure BinEnc turns the intermediate ILP into a 0-1 ILP by replacing each occurrence of a non-Boolean variable with a binary exponential sum. This may be a poor encoding from a solving perspective, but it has the strong advantage of maintaining the form of the ILP, allowing us to "pretend" we have integer variables while still working honestly with native PB constraints.

Specifically, for any non-0-1 variable X, by construction it must have bound constraints $\mathsf{Bnd}(X)$ in the intermediate ILP. Let u and l denote the upper and lower bounds respectively.

If $l \geq 0$, let h be the least strictly positive integer such that $2^h \geq u + 1$ and define

$$\mathsf{BinEnc}(X) := \sum_{i=0}^{h-1} 2^i x_i, \tag{3.20}$$

i.e. the evaluation of the h-bit binary representation. Otherwise, if $l < 0$, i.e. X can be negative, instead let h be the least strictly positive number such that $2^{h-1} \geq \max\{|u| + 1, |\ell|\}$ and then

replace every occurrence of X with

$$\mathsf{BinEnc}(X) := -2^{h-1}x_{h-1} + \sum_{i=0}^{h-2} 2^i x_i, \tag{3.21}$$

i.e. the evaluation of the h-bit *two's-complement* representation.

We will also overload the BinEnc notation and write $\mathsf{BinEnc}(C)$ for the result of replacing every eligible variable in a constraint C; and $\mathsf{BinEnc}(F)$ in turn for applying the transformation to a set of constraints F.

Now the soundness of this replacement (in terms of preserving the required guarantees from Section 3.2.1) follows directly from the validity of binary and two's-complement encodings. There is a one-to-one correspondence between the solutions to the 0-1 ILP and the intermediate ILP, given precisely by the binary encoding or two's complement correspondence. We do of course require the property that h is always enough bits to uniquely represent every value in a variable's domain, which is straightforward to show.

Step 6: Standardise Constraints

Finally, we ensure all remaining constraints are in canonical form, yielding a PB formula consistent with our definitions in Chapter 2. This Standardise procedure simply needs to convert reifications to native PB constraints as specified by Proposition 2.2; replace "=" constraints with a pair of "$\geq$" and "$\leq$" constraints; and multiply through by -1 and/or add 1 as needed to ensure all resulting inequalities are of the "$\geq$" form. The soundness of this in the context of the whole process is trivial, as such transformations do not affect the solution set of the problem.

It is now straightforward to argue the result of the transformation process as a whole satisfies the required guarantee P1. If we set the PB objective function to be $\mathsf{BinEnc}(X_o)$ where X_o is the CP objective variable then we can easily argue that P2 is also satisfied.

Theorem 3.1 *Let P be a CSP with variables $\mathcal{V}$ and constraints $\mathcal{C}$, where all domains are integer ranges. Let F be the result of executing Encoding Procedure 3.1 on P. Additionally, if P has an associated objective variable $X_o \in \mathcal{V}$ to minimise/maximise let $\mathsf{BinEnc}(X_o)$ be the PB objective function. Then the properties P1 and P2 hold.*

Proof. Let P' be the intermediate ILP produced after steps 1-4 of Encoding Procedure 3.1. Lemma 3.1 implies there is a bijection between the solutions of P and the set of assignments to variables of $\mathcal{V}$ (projected from full assignments) that satisfy P'. Now let F^* be the 0-1 linear CSP obtained after step 5. We have

- Every solution to P' corresponds to a solution of F^*, by taking any value v assigned to a non-0-1 variable X and assigning the unique h-bit vector representing the binary or two's complement representation of v to the h variables in $\mathsf{BinEnc}(X)$.

- Every solution to F^* corresponds to a solution of P' by taking, for any non-Boolean variable $X \in \mathsf{Vars}(P')$, the values assigned to each variable in $\mathsf{BinEnc}(X)$ and decoding them as a binary or two's complement bit-vector to get a unique value v to assign to X.

The final PB formula F obtained after step 6 has exactly the same solutions as F^*, and so we have a bijection between the solutions of P' and F. So overall we have a bijection ϕ between the solutions of P and the solutions of F projected onto only the bit variables appearing in $\mathsf{BinEnc}(X)$ for some $X \in \mathcal{V}$. In particular, F is satisfiable $\iff$ P' is satisfiable $\iff$ P is satisfiable and hence P1 holds.

Furthermore, for all $X \in \mathcal{V}$ and all solutions σ to P we have by construction $\sigma(X) = \mathsf{BinEnc}(X)\restriction_{\phi(\sigma)}$ and in particular P2 holds. □

3.2.3 PB Encodings for Fundamental Constraints

To actually instantiate this encoding method for concrete CSPs, we simply need to exhibit "sufficiently obvious" linearisations producing an intermediate ILP for all supported non-linear constraints. Obviously, constraints already in one of the allowed forms (3.10), including (reified) linear (in)equalities, simple comparisons, clauses, and conjunctions do not require linearisation and can be immediately encoded to PB by applying BinEnc replacement and standardisation.

In this section, we restate the encodings given by Gocht et al. [94], as well as exhibit other simple linearisations used by the *Glasgow Constraint Solver* project for fundamental constraints. These linearisations can be viewed as implementations of the Linearise subprocedure from Encoding Procedure 3.1 for each supported constraint type. To help express them concisely, we will use the notation "**def** L" to indicate adding a (set of) linear constraint(s) L to the set of linearised constraints that will in the end be returned, rather than the somewhat cumbersome $\mathsf{Lin}(\mathsf{C}) \leftarrow \mathsf{Lin}(\mathsf{C}) \cup L$. We will not state the NeededAtomicVariables procedures explicitly, since this follows immediately from each Linearise definition. Any further auxiliary variables needed will be noted.

Encoding Procedure 3.2 (Not-Equals Constraint). *[94]*
Definition: $\mathsf{NotEquals}(X, Y) :=$ *X and Y take different values.*
Linearisation: **def** $u \Rightarrow X - Y > 0; \quad \overline{u} \Rightarrow Y - X > 0.$
Auxiliary Variables: $u \in \{0, 1\}$

Encoding Procedure 3.3 (Absolute Value Constraint).
Definition: $\mathsf{Abs}(X, Y) :=$ *The value of Y is equal to the absolute value of X.*
Linearisation: **def** $x_{\geq 0} \Rightarrow X - Y = 0; \quad \overline{x}_{\geq 0} \Rightarrow X + Y = 0.$

Encoding Procedure 3.4 (Element Constraint). *[94]*

Definition: $\mathsf{Element}(X_1, \ldots, X_n, Y, Z) :=$ *For some* $i \in \{1, \ldots, n\}$, Y *takes the value* i *and the value of* X_i *is equal to the value of* Z.

Linearisation:

1: **def** $Y \geq 1$
2: **def** $Y \leq n$
3: **for** $i \in \{1, \ldots, n\}$ **def** $y_{=i} \Rightarrow X_i = Z$.

Encoding Procedure 3.5 (Array Max Constraint).

Definition: $\mathsf{ArrayMax}(X_1, \ldots, X_n, Y) :=$ *The value taken by* Y *is equal to the maximum among values taken by* $X_1, \ldots, X_n$.

Linearisation:

1: **for** $i \in \{1, \ldots, n\}$ **def** $Y - X_i \geq 0$;
2: **for** $v \in \bigcup_i \mathrm{dom}(X_i)$ **def** $y_{=v} \Rightarrow \sum_i x_{i=v} \geq 1$.

Encoding Procedure 3.6 (Array Min Constraint).

Definition: $\mathsf{ArrayMin}(X_1, \ldots, X_n, Y) :=$ *The value taken by* Y *is equal to the minimum among values taken by* $X_1, \ldots, X_n$.

Linearisation:

1: **for** $i \in \{1, \ldots, n\}$ **def** $Y - X_i \leq 0$;
2: **for** $v \in \bigcup_i \mathrm{dom}(X_i)$ **def** $y_{=v} \Rightarrow \sum_i x_{i=v} \geq 1$.

Encoding Procedure 3.7 (Count Constraint).

Definition: $\mathsf{Count}(X_1, \ldots, X_n, Y, Z) :=$ *The value taken by* Z *is equal to the number of times the value of* Y *appears among values taken by* $X_1, \ldots, X_n$.

Linearisation:

1: **for** $i \in \{1, \ldots, n\}$
2: **def** $c_{i\leq} \Leftrightarrow X_i - Y \leq 0$;
3: **def** $c_{i\geq} \Leftrightarrow X_i - Y \geq 0$;
4: **def** $c_i \Leftrightarrow c_{i\leq} + c_{i\geq} \geq 2$;
5: **def** $\sum_i c_i = Z$

Auxiliary Variables: $\forall i; c_i, c_{i\geq}, c_{i\leq} \in \{0, 1\}$.

Encoding Procedure 3.8 (NValue Constraint).

Definition: $\mathsf{NValue}(X_1, \dots, X_n, Y) :=$ *The value of* Y *is equal to the number of distinct values among values taken by* $X_1, \dots, X_n$.

Linearisation:

1: **for** $v \in \bigcup_i \operatorname{dom}(X_i)$
2: $\quad$ **def** $w_v \Leftrightarrow \sum_i x_{i=v} \geq 1$;
3: **def** $\sum_{v \in \bigcup_i \operatorname{dom}(X_i)} w_v = Y$

Auxiliary Variables: $\forall v; w_v \in \{0, 1\}$.

Encoding Procedure 3.9 (All-Different Constraint).

Definition $\mathsf{AllDifferent}(X_1, \dots, X_n) := X_1, \dots, X_n$ *must all take different values.*

Linearisation:

1: **for** $i, j \in \{1, \dots, n\}$ **where** $i < j$
2: $\quad$ **def** $u_{ij} \Rightarrow X_i - X_j > 0$.
3: $\quad$ **def** $\overline{u}_{ij} \Rightarrow X_j - X_i > 0$

Auxiliary Variables; $\forall i \; \forall j$ *where* $i < j; u_{ij} \in \{0, 1\}$.

Encoding Procedure 3.10 (Table Constraint).

Definition $\mathsf{Table}(X_1, \dots, X_n, \tau) :=$ *The sequence of values taken by* $X_1, \dots, X_n$ *(as a tuple of integers) must be a member of the set* $\tau \subset \mathbb{Z}^n$.

Linearisation:

1: **for** $\tau_j \in \tau$
2: $\quad$ **def** $t_j \Leftrightarrow \sum_i x_{i=\tau_j[i]} \geq n$;
3: **def** $\sum_j t_j = 1$

Auxiliary Variables: $\forall j; t_j \in \{0, 1\}$.

It can be verified by inspection that these encodings satisfy L1-L4: they "obviously" enforce the restriction over the integer-valued variables that the natural language description of the constraints suggests.

One point worth emphasising here is that our chosen encodings are in no way optimised from a PB solving or ILP solving perspective. Indeed, they might have quite poor propagation properties compared to the actual CP algorithms. This is not a concern since, as will become clear in the next section, they are only used for logging subsequent proof statements and should not inform the native solving process in any way.

3.3 A Proof Logging Framework

We will begin from the assumption that we have used an encoding method following the steps of Encoding Procedure 3.1. So before logging anything in the proof, we have created a correct PB representation of an input CP problem in terms of binary-encoded CP variables, atomic literals defined for certain variables and values, and potentially some other auxiliary variables.

Based on this encoding method, we can formulate a proof logging framework based on maintaining some *invariants* relating the solver's explored domain states and the verifier's PB constraint database. Certain properties of atomic literals as defined by (3.8) and (3.9) are crucial for this setup, so we will demonstrate these first.

3.3.1 Properties of Atomic Literals

We know we can define atomic variables in the PB model. Recalling Theorem 2.4, we can also always freely introduce in the proof any further atomic variables $x_{i=v}$ and $x_{i\geq v}$ with definitions corresponding to (3.8) and (3.9), after binary encoding and standardisation. In full, these amount to the PB constraints

$$\mathsf{Def}_{\Rightarrow}(x_{i\geq v}) := x_{i\geq v} \Rightarrow \mathsf{BinEnc}(X_i) \geq v; \tag{3.22}$$

$$\mathsf{Def}_{\Leftarrow}(x_{i\geq v}) := \overline{x}_{i\geq v} \Rightarrow -\mathsf{BinEnc}(X_i) \geq -v+1; \tag{3.23}$$

$$\mathsf{Def}_{\Rightarrow}(x_{i=v}) := x_{i=v} \Rightarrow x_{i\geq v} + x_{i\geq v+1} \geq 2; \tag{3.24}$$

$$\mathsf{Def}_{\Leftarrow}(x_{i=v}) := \overline{x}_{i=v} \Rightarrow \overline{x}_{i\geq v} + \overline{x}_{i\geq v+1} \geq 1. \tag{3.25}$$

So we can safely assume in what follows (and throughout this thesis) that any PB literals that appear in a derivation with these suggestive names will be accompanied by the necessary definition constraints as above, either in the input formula or derived by redundance-based strengthening at an earlier point. For clarity, when we say that an atomic literal ℓ is *defined* at a particular point the proof we mean *both* directions of the reified definition for its underlying atomic variable either have appeared in the accumulated formula, and have not yet been deleted.

We can then express some useful properties.

Lemma 3.2 (Propagation of contradictory bound literals). *Suppose two inequality variables $x_{i\geq v}$ and $x_{i\geq w}$ are defined for some CP variable X_i and values $v \geq w$. Then if $x_{i\geq v}$ and $\overline{x}_{i\geq w}$ propagate, further unit propagation will result in a conflict.*

Proof. This follows directly from Theorem 2.7, since even if X_i requires a two's complement encoding, the definition constraints $\mathsf{Def}_{\Rightarrow}(x_{i\geq v})$ and $\mathsf{Def}_{\Leftarrow}(x_{i\geq w})$ are in the required form after substituting $x_{i\geq v} \mapsto 1, x_{i\geq w} \mapsto 0$. □

Corollary 3.1 (RUP Properties of Atomic Literals). *For two inequality variables $x_{i\geq v}$ and $x_{i\geq w}$ defined on some CP variable X_i and values $v \geq w$, the constraint*

$$x_{i\geq v} \Rightarrow x_{i\geq w} \geq 1 \tag{3.26}$$

will always follow by RUP.

These "linking" constraints allow for further propagation guarantees, providing that in any proof we always derive them for *all* currently defined bounds variables. If we let F_t be the accumulated set of constraints either in the initial PB encoding or derived up to some point t in a proof, we can phrase this requirement as the following invariant.

Inv1. For any two PB variables $x_{\geq u}$, $x_{\geq v}$ underlying bound literals on the same CP variable for values $u < v$, if the definition constraints $\mathsf{Def}(x_{\geq u})$, and $\mathsf{Def}(x_{\geq v})$ (3.8) for these literals appear (and are not deleted) in F_t, then either

- $x_{\geq w}$ is defined for some $u < w < v$
- The constraint $x_{\geq v} \Rightarrow x_{\geq u} \geq 1$ appears in F_t

From Corollary 3.1, Inv1 only requires two RUP steps for each bound literal used, so maintaining this is feasible. But more importantly, it is *worthwhile* to maintain this invariant, since it allows us to rely on the fact that any set of bounds and equality literals will now propagate all the encoded domain information throughout all other defined atomic variables. This avoids us needing to explicitly derive obvious facts such as $x_{\geq 3} \wedge \overline{x}_{=3} \Rightarrow x_{\geq 4}$.

To state this more precisely, we will define some further notation. Any set of atomic literals $\mathcal{L}$ defined on the same variable X_i represents a restriction of X_i (or the bit sum representing it) to some feasible set of values. We will call the values allowed by this restriction the *defined domain* for those atomic literals.

$$\begin{aligned}\mathrm{dom}_{\mathcal{L}}(X_i) := \{v : \forall x_{i\geq w} \in \mathcal{L} \,.\, v \geq w\} \cap \{\forall \overline{x}_{i\geq w} \in \mathcal{L} \,.\, v < w\} \\ \cap \{v : \forall x_{i=w} \in \mathcal{L} \,.\, v = w\} \cap \{v : \forall \overline{x}_{i=w} \in \mathcal{L} \,.\, v \neq w\}.\end{aligned} \tag{3.27}$$

Note that this set may be empty, or infinite (since e.g. $\mathrm{dom}_{\emptyset}(X_i) = \mathbb{Z}$).

Conversely, we will call the complete (potentially infinite) set of atomic literals that should be true for some domain $\mathrm{dom}_t(X_i)$ the literals *implied* by that domain.

$$\begin{aligned}\mathsf{ImpLits}(\mathrm{dom}_t(X_i)) := \ \{\overline{x}_{i=v} : v \notin \mathrm{dom}_t(X_i)\} \cup \{x_{i=v} : \mathrm{dom}_t(X_i) = \{v\}\} \\ \cup \{x_{i\geq v} : \forall u < v; u \notin \mathrm{dom}_t(X_i)\} \cup \{\overline{x}_{i\geq v} : \forall w \geq v; w \notin \mathrm{dom}_t(X_i)\}.\end{aligned} \tag{3.28}$$

Theorem 3.2 (Empty defined domain propagates to conflict). *Let F_t be a set of PB constraints either in the encoding of a CP problem (as produced by Encoding Procedure 3.1) or derived from it. Let $\mathcal{L}$ be the set of all literals over atomic variables defined for a particular CP variable X_i in F_t. Then if for some $\mathcal{R} \subseteq \mathcal{L}$, we have $\mathrm{dom}_{\mathcal{R}}(X_i) = \emptyset$, unit propagation on $F\restriction_{\mathcal{R}}$ must result in a conflict.*

Proof. Without loss of generality, we can replace any equality literals $x_{i=v} \in \mathcal{R}$ with $x_{i\geq v}$ and $\overline{x}_{i\geq v+1}$, since, unless conflict is somehow reached beforehand (in which case we already are done), these immediately propagate from the definition constraint $\mathsf{Def}_{\Rightarrow}(x_{i=v})$ (3.24).

Now since $\mathrm{dom}_{\mathcal{R}}(X_i) = \emptyset$, every integer must be excluded in (3.27) somehow. In other words, for each $v \in \mathbb{Z}$ we must have in $\mathcal{R}$ at least one of the atomic literals $\overline{x}_{=v}$; $x_{\geq u}$ for some $u > v$; or $\overline{x}_{\geq w}$ for some $w \leq v$. Let u be the *largest* value in $\mathbb{Z}$ such that $x_{\geq u}$ is either in $\mathcal{R}$ or propagates during unit propagation of $F\restriction_{\mathcal{R}}$. This must exist since $\mathcal{R}$ has finite size and so cannot have infinitely many literals defined to exclude the whole of $\mathbb{Z}$. Similarly, let w be the *smallest* value such that $\overline{x}_{\geq w}$ is in $\mathcal{R}$ or propagates. If $u \geq w$ then conflict follows by Lemma 3.2. Otherwise, $u < w$, but then to exclude the value u itself the only option is to have the negated equality literal $\overline{x}_{=u}$ in $\mathcal{R}$. This would imply that the definition constraint $\mathsf{Def}_{\Leftarrow}(x_{=u})$ propagates $x_{\geq u+1}$ under $\mathcal{R}$, contradicting the assumption that u is the smallest value for which this happens. □

Theorem 3.3 (Complete propagation of implied atomic literals). *Let F_t be a set of PB constraints either in the encoding of a CP problem (as produced by Encoding Procedure 3.1) or derived from it. Let $\mathcal{L}$ be the set of all literals over atomic variables defined for a particular CP variable X_i in F_t, and suppose Inv1 and has been respected for F_t. Then for any $\mathcal{R} \subseteq \mathcal{L}$, if $F\restriction_{\mathcal{R}}$ does not unit propagate to contradiction, then all literals in $\mathsf{ImpLits}(\mathrm{dom}_{\mathcal{R}}(X_i)) \cap \mathcal{L}$ propagate during unit propagation of $F\restriction_{\mathcal{R}}$ or were already present in $\mathcal{R}$.*

We first prove an easy lemma.

Lemma 3.3 *If Inv1 is respected, then for any X_i, if $x_{i\geq v}$ and $x_{i\geq u}$ are both defined in F_t for $v > u$ then if $x_{i\geq v}$ propagates then $x_{i\geq u}$ propagates, and if $\overline{x}_{i\geq u}$ propagates then $\overline{x}_{i\geq v}$ propagates.*

Proof. Let $w_1, > w_2 > \cdots > w_k$ be the finite sequence of $k \geq 2$ integers where $w_1 = v$ and $w_k = u$ and such that for each $j \in \{1, \ldots, k-1\}$, $x_{i\geq w_j}$ and $x_{i\geq w_{j+1}}$ are defined and $x_{i\geq a}$ is not defined for any $w_j > a > w_{j+1}$. Since there can only be finitely many bound literals defined in a finite formula, this sequence must exist. Now for each $1 \leq j \leq k$ Inv1 implies that the constraint $x_{i\geq w_j} \Rightarrow x_{i\geq w_{j+1}} \geq 1$ appears in F_t. So starting from the assumption that $x_{i\geq v}$ propagates, each

$x_{i\geq w_j}$ propagates in turn, and so eventually $x_{i\geq u}$ propagates. Likewise, from the assumption that $\overline{x}_{i\geq u}$ propagates, each $\overline{x}_{i\geq w_j}$ propagates in turn and so $\overline{x}_{i\geq v}$ propagates. □

We can now prove Theorem 3.3.

Proof. Again without loss generality, we can replace any equality literals $x_{i=v} \in \mathcal{R}$ with $x_{i\geq v}$ and $\overline{x}_{i\geq v+1}$. Also, we can assume propagation of $F\restriction_{\mathcal{R}}$ does not result in a contradiction (and hence assume $\mathrm{dom}_{\mathcal{R}}(X_i) \neq \emptyset$), since otherwise we are done.

Now consider a literal $\ell \in \mathsf{ImpLits}(\mathrm{dom}_{\mathcal{R}}(X_i)) \cap \mathcal{L}$. In light of (3.28) there are 4 cases.

Case 1: $\ell = x_{i\geq v}$ for some v where $u \notin \mathrm{dom}_{\mathcal{R}}(X_i)$ for all $u < v$. In particular this means that $\mathrm{dom}_{\mathcal{R}}(X_i)$ is bounded from below. Now let w be the largest value such that $x_{i\geq w}$ is either in $\mathcal{R}$ or propagates during unit propagation of $F\restriction_{\mathcal{R}}$. This must exist since $\mathcal{R}$ has finite size, and so cannot define a domain that is bounded from below without at least one non-negated bound literal. If $w > v$ then ℓ propagates by Lemma 3.3. Otherwise, $w \leq v$. Now if $w < v$ then the value w can only be excluded by an inequality literal $\overline{x}_{i=w} \in \mathcal{R}$. But then due to the definition constraint $\mathsf{Def}_{\Leftarrow}(x_{i=w})$ (3.25) $x_{i\geq w+1}$ propagates, contradicting the maximality of w. So $w = v$ in this case and thus $\ell \in \mathcal{R}$.

Case 2: $\ell = \overline{x}_{i\geq v}$. This is analogous to Case 1: $\mathrm{dom}_{\mathcal{R}}(X_i)$ is bounded from above, and we can let u be the *smallest value* such that $\overline{x}_{i\geq u}$ is either in $\mathcal{R}$ or propagates. We can show by a similar argument that either ℓ propagates or $\ell = \overline{x}_{i\geq u}$.

Case 3: $\ell = x_{i=v}$. Since ℓ is defined, $x_{i\geq v}$ is defined and is in $\mathcal{R}$ or propagates by Case 1, and $\overline{x}_{i\geq v+1}$ is defined and is in $\mathcal{R}$ or propagates by Case 2.

Case 4: $\ell = \overline{x}_{i=v}$. Again since ℓ is defined, both $x_{i\geq v}$ and $x_{i\geq v+1}$ are defined. If $\ell \notin \mathcal{R}$ then v must be excluded from $\mathrm{dom}_{\mathcal{R}}(X_i)$ by either $x_{i\geq w}$ for some $w > v$ or $\overline{x}_{i\geq u+1}$ for some $u < v$. In the first case $x_{i\geq v+1}$ propagates and in the second case $\overline{x}_{i\geq v}$ propagates, both by Lemma 3.3. Either way, $\overline{x}_{i=v}$ then propagates due to $\mathsf{Def}_{\Rightarrow}(x_{i=v})$ (3.24).

□

This result is similar to the SAT-based unit propagation property of the "*DOM*" encoding as stated by Ohrimenko et al. [160, Theorem 1] with the key difference being that we use the binary encoding to support lazy introduction of the linking constraints rather than having to assert them up front.

3.3.2 Unsatisfiability Proofs from Backtracking Search

Now we can finally explain how a CP solver can create a PB proof when it determines via standard decisions, backtracking, and constraint propagation that a problem is unsatisfiable.

In short, every time the solver backtracks, we require the solver to immediately log a PB constraint that encodes via atomic literals the negation of the sequence of decisions made prior to discovering a conflict. This can be thought of as a "*nogood* clause" from a solving perspective, but we will refer to it as a *backtracking justification* once it is written in the proof log.

Example 3.5 *Consider a backtracking search tree with the same structure as Example 3.2, but with conflicts at all leaf nodes.*
The backtracking justifications logged in the proof should be

$$\overline{x}_{=1} + \overline{y}_{=1} \geq 1; \tag{3.29}$$
$$\overline{x}_{=1} + \overline{y}_{=2} + \overline{z}_{=1} \geq 1; \tag{3.30}$$
$$\overline{x}_{=1} + \overline{y}_{=2} + \overline{z}_{=2} \geq 1; \tag{3.31}$$
$$\overline{x}_{=1} + \overline{y}_{=2} \geq 1 \tag{3.32}$$
$$\overline{x}_{=1} \geq 1 \tag{3.33}$$
$$\vdots$$
$$\vdots$$
$$\overline{x}_{=3} + \overline{y}_{=2} + \overline{z}_{=1} \geq 1; \tag{3.34}$$
$$\overline{x}_{=3} + \overline{y}_{=2} + \overline{z}_{=2} \geq 1; \tag{3.35}$$
$$\overline{x}_{=3} + \overline{y}_{=2} \geq 1; \tag{3.36}$$
$$\overline{x}_{=3} \geq 1 \tag{3.37}$$
$$0 \geq 1 \tag{3.38}$$

The solver is not restricted to only guessing assignments, but we do require that its branching decision can be expressed in terms of a single atomic literal and that there is a single decision variable for which all assignments are somehow partitioned. This is not a big restriction since it already encompasses the most common two-way, k-way and domain-splitting branching strategies as mentioned in Section 3.1.3. In general, if there are k decisions before a particular backtrack, and we let the decision variables be $X_0, \ldots, X_{k-1}$, the required backtracking justification at that point is

$$B_t := \sum_{i=0}^{k-1} \overline{d}_i \geq 1. \tag{3.39}$$

where each d_i is an atomic literal defined for X_k and represents the branching decision at level i of the current search path.

The complete proof can be viewed as a description of the solver's backtracking search tree. When the solver backtracks from the root decision level, the sequence of decisions is empty and (3.39) becomes simply $0 \geq 1$: precisely the trivial unsatisfiability conclusion we want to reach.

Of course, for these logged constraints to be efficiently checkable using *VeriPB*, they need to be the result of applying one of the recognised proof rules. The idea, going back to Elffers et al. [67], and stated more explicitly Gocht et al. [94, pg. 7] is to maintain an invariant in the proof so that all of these backtracking clauses will follow by RUP (Rule 6):

> "The core invariant we use is that at every backtrack, any variable-value deletion that is known to the CP solver (and thus part of the decision to backtrack) must be visible to the proof verifier either through unit propagation, or through reverse unit propagation of the backtrack clause."

In order to show that this works in general, we will state the required invariant slightly more formally, by defining what exactly is meant by "visible".

For any variable X in the input problem, and any point t in time immediately prior to backtracking or immediately before any domain changes, let $\mathrm{dom}_t(X)$ be the current domain of X as stored by the solver; let B_t be a backtracking justification with respect to the current sequence of decisions; and let F_t be the set of PB constraints either in the input PB encoding or derived in the proof up to the point t. We must have logged sufficient constraints in the proof so that:

Inv2. If the solver has detected contradiction (infeasibility) for the current state of domains $F_t \cup \{\neg B_t\}$ must unit propagate to contradiction.

Inv3. For every CP variable X, if $\mathcal{L}$ is the set of atomic literals defined for X that would propagate during unit propagation of $F_t \cup \{\neg B_t\}$, then unless contradiction is reached beforehand, we must have $\mathrm{dom}_{\mathcal{L}}(X) \subseteq \mathrm{dom}_t(X)$.

Theorem 3.4 *If the invariants Inv2 and Inv3 are respected, and a complete solver always logs backtracking justifications, then every backtracking justification B_t will follow by RUP.*

Proof. In a standard backtracking and propagation solver there are three cases in which the solver backtracks.

Case 1: A propagator has detected infeasibility. Then Inv2 implies B_t is RUP.

Case 2: A propagator has removed all the remaining values from the domain of some variable X. Then Inv3 implies that unit propagation on $F_t \cup \{\neg B_t\}$ obtains either a contradiction or at least propagates a set of literals $\mathcal{L}$ such that $\mathrm{dom}_{\mathcal{L}}(X) = \emptyset$. For the latter, Theorem 3.2 implies that further unit propagation of $(F_t \cup \{\neg B_t\})\restriction_{\mathcal{L}}$ must still result in contradiction.

Case 3: The solver has exhausted all the branching possibilities for its current branching variable X. Let $\mathrm{dom}_t(X)$ be the domain of X after constraint propagation at the current decision level and let $\mathcal{G}$ be the literals encoding the current sequence of decisions (not including decisions on X). Then the current branching choices must partition $\mathrm{dom}_t(X)$. Because this is not a leaf node, we can assume that backtracking justifications have already been logged in F_t for each of the k child branches. From (3.39) these must have the form

$\mathcal{G} \Rightarrow \overline{d}_i \geq 1$ where d_i represents the i_{th} branching decision on X at the current level. So each $\overline{d}_i$ propagates in $F_t \cup \{\neg B_t\}$.

Now suppose $F_t \cup \{\neg B_t\}$ does not unit propagate to contradiction and let $\mathcal{L}$ be the set of literals defined on X_i that propagate during unit propagation to fix point.

We must have $v \in \mathrm{dom}_{\mathcal{L}}(X_i)$, since otherwise $\mathrm{dom}_{\mathcal{L}}(X_i) = \emptyset$ and we would have reached a conflict by Theorem 3.2. But then by invariant Inv3, $v \in \mathrm{dom}_t(X_i)$, and this implies that v is not excluded by any $\overline{d}_i$, contradicting the completeness of the branching partition. □

The natural question is then how to maintain these invariants. If the CP solver happened to be performing inferences that are no stronger than unit propagation on the input PB formula, they would be trivially maintained. But we already established that the encoding should not inform the solving process, and that the eventual goal here is to enable proof logging for any features of a modern CP solver.

So we will require some further kinds of justification constraints to be logged in the proof. To define these we will make use of *reasons*, which for CP proof logging are just literal sets $\mathcal{R}$ that are used to reify intermediate facts to be written in the proof log. We will say a reason $\mathcal{R}$ is *valid* for a particular domain state dom_t at time t if every $\ell \in \mathcal{R}$ is guaranteed to propagate under $F_t \cup \{\neg B_t\}$, if contradiction is not reached first.

This enables us to require some more specific behaviours from a proof logging CP solver. Whenever the solver detects contradiction from the current domain state, it should somehow derive in the proof a PB constraint of the form

$$\mathcal{R} \Rightarrow 0 \geq 1 \tag{3.40}$$

where $\mathcal{R}$ is a reason valid at the point of conflict. We will refer to this as a *conflict justification* or *infeasibility justification*. Similarly, whenever a constraint propagator makes an inference it should derive a *propagation justification*

$$\mathcal{R} \Rightarrow \ell \geq 1 \tag{3.41}$$

where ℓ is an equality or bound literal describing the inference, and $\mathcal{R}$ is a reason valid immediately before making the domain change encoded by the inference.

There are several options for selecting a valid reason. The atomic literals representing the current decisions $\mathcal{G}$ are always valid for the current subtree as they are by definition propagated by $\{\neg B_t\}$. This is the main approach proposed by Gocht et al. [94]. Alternatively, a valid reason can always be a set of atomic literals that is a subset of $\bigcup_i \mathsf{ImpLits}(\mathrm{dom}_t(X_i))$, as shown by the following lemma.

Lemma 3.4 *If $\mathcal{R}$ is a (finite) subset of $\bigcup_i \mathsf{ImpLits}(\mathrm{dom}_t(X_i))$ that is defined in F_t, and Inv1 and Inv3 are respected, then $F \cup \{\neg B_t\}$ either propagates to contradiction or propagates every literal in $\mathcal{R}$.*

Proof. Let ℓ be an atomic literal in $\mathcal{R}$ and let X_i be the CP variable ℓ is defined for, i.e. $\ell \in \mathsf{ImpLits}(\mathrm{dom}_t(X_i))$. Suppose $F \cup \{\neg B_t\}$ does not propagate to contradiction (otherwise we are done), and let $\mathcal{L}$ be the atomic literals that propagate due to $F \cup \{\neg B_t\}$ for X_i guaranteed by Inv3. Since we do not reach conflict, we must have $\mathrm{dom}_{\mathcal{L}}(X_i) \neq \emptyset$, by Theorem 3.2. Then since $\mathrm{dom}_{\mathcal{L}}(X_i) \subseteq \mathrm{dom}_t(X)$, we must have $\mathsf{ImpLits}(\mathrm{dom}_{\mathcal{L}}(X_i)) \supseteq \mathsf{ImpLits}(\mathrm{dom}_t(X_i))$ from the definition (3.28). Thus, Theorem 3.3 tells us that ℓ propagates. □

Example 3.6 *If a problem has a constraint $Y = 3X$, and the solver knows $X = 1$, $Y = 4$, and so infers infeasibility, a possible conflict justification would be*

$$x_{=1} \wedge y_{=4} \Rightarrow 0 \geq 1. \tag{3.42}$$

If on the other hand the solver only knows $X \geq 2$, and so infers $Y \geq 6$, a possible propagation justification would be

$$x_{\geq 2} \Rightarrow y_{\geq 6} \geq 1. \tag{3.43}$$

These justifications therefore have a significant overlap with the *reason clauses* from the lazy-clause-generation solving paradigm, which are directly implied by the constraint propagator definition expressed in terms of propagation rules [160]. It can be intuitive for our proof logging justifications to be similarly implied by a single constraint definition, particularly if it is straightforward to come up with an "explaining" set of literals. However, our definition of reasons is more permissive, and we retain the ability in the proof framework to use, for example, auxiliary variables from the PB encoding, or even fresh variables introduced by redundance as part of the reason.

To ensure that justifications are actually sufficient to maintain the invariants, we also require the solver to introduce by redundance all the atomic literals for the initial bounds at the start of the proof, if they are not already present, and then derive by RUP the unit constraints stating that they are true at the root decision level. If we let l and u be the initial upper and lower bounds of a variable X_i, these are

$$x_{i \geq l} \geq 1 \quad \text{and} \quad \overline{x}_{i \geq u+1} \geq 1. \tag{3.44}$$

These follow by RUP by the same argument as the linking constraints (Corollary 3.1) since the

bounds constraints $\mathsf{Bnd}(X_i)$ are assumed to be present.

Theorem 3.5 *A correct proof logging CP solver that always logs propagation and conflict justifications, maintains Inv1 and initially introduces bound literals will always maintain Inv2 and Inv3.*

Proof. Let t be any point in the proof immediately before a backtrack or domain change; let B_t be a backtracking justification with respect to the current sequence of decisions; and let F_t be the set of PB constraints either in the input PB encoding or derived in the proof up to the point t.

Assume that Inv3 has been maintained at all relevant points prior to point t. Suppose the solver has detected infeasibility of the current domain state. Then it will have just logged a conflict justification $\mathcal{R} \Rightarrow 0 \geq 1$. Then tells us that reverse unit propagation of the backtracking justification leads either to contradiction, or at least propagates every literal in $\mathcal{R}$. Either way, a contradiction is reached due to the conflict justification and hence Inv2 is respected.

Suppose instead a constraint propagator has just made an inference that changed a domain. Then the solver should have logged a propagation justification of the form $\mathcal{R} \Rightarrow \ell \geq 1$, where ℓ is a literal describing the domain change. The reason is valid at the point directly before the inference was made, and so every literal in $\mathcal{R}$ and hence ℓ propagates under reverse unit propagation of the backtracking justification. This means the domain restriction represented by ℓ is incorporated into the defined domain for the set of literals propagated under the backtracking justification and hence Inv3 continues to be respected.

It only remains for us to argue that there is a starting point in the proof where the Inv3 is respected. Consider the very first conflict or propagation inference (including backtracking due to domain wipeout). The only changes to domains that could have happened at this point (again assuming the solver is correct) are those due to initial decisions, which by definition propagate on reverse unit propagation of the backtracking justification, and the initial domain bounds, which by assumption we have introduced as units. Hence, Inv3 must be initially respected. □

Our proof logging framework for producing an unsatisfiability for a standard backtracking and propagation solving procedure may be summarised as follows.

1. Encode the problem using the methods using Encoding Procedure 3.1 from Section 3.2.

2. At the start of the proof, introduce by RUP $x_{\geq l} \geq 1$ and $\overline{x}_{\geq u+1} \geq 1$ for the bounds $l..u$ on each variable X.

3. Any time the solver backtracks derive by RUP a *backtracking justification* $\mathcal{G} \Rightarrow 0 \geq 1$, where $\mathcal{G}$ is a set of equality literals representing the sequence of decisions prior to backtracking.

4. Any time the solver detects infeasibility derive a *conflict justification* $\mathcal{R} \Rightarrow 0 \geq 1$, where $\mathcal{R}$ is a *reason*, usually a set of atomic literals implied by the current state of variable domains.

5. Any time a constraint propagator makes an inference, derive a *propagation justification* $\mathcal{R} \Rightarrow \ell \geq 1$ where $\mathcal{R}$ is a reason and ℓ is a literal encoding the inference.

6. Any atomic variables needed for any of the above should be defined in the proof (if not already defined in the input model) with the corresponding *definition* constraints (3.9) and (3.8), and all defined bound literals should have appropriate linking constraints $x_{\geq v} \Rightarrow x_{\geq u} \geq 1$ introduced by RUP in the proof.

Despite requiring a somewhat delicate set-up in terms of variable encodings and propagation properties, this method allows the proof to in the end have a relatively intuitive structure. Other than the definition and linking constraints for inequality and equality literals, it is essentially getting the solver to simply write down all the facts that it learned, at the points that it learned them. As Gocht et al. [94] observe, this makes proof logging particularly useful for *auditability*, as the proof becomes a record of at least all the domain-altering steps that the solver took, and if any individual change was unsound then the proof should fail, even if the solver eventually arrived at the correct answer. Of course, in some situations, the invariants are more stringent than strictly required, since it might be that certain justifications are not actually needed in order to derive the required conclusions by reverse unit propagation.

Finally, it is evident that the methodology hinges on us being able to derive propagation and conflict justifications efficiently using the *VeriPB* proof system. There is no reason to expect that these will always be RUP for any constraint and any associated inference procedure. Indeed, it is not even immediately clear whether the rest of the PB proof rules set out in Chapter 2 will be sufficient to capture all the reasoning methods of widely-used constraint propagators. Before we tackle this issue however, we will briefly note how the same framework can be easily adapted to allow for proofs beyond unsatisfiability.

3.3.3 Optimality Proofs from Solution-Improving Search

There is not a huge conceptual difference between a CP solver performing complete backtracking search to establish unsatisfiability, and performing so-called "solution improving" search to determine the optimal solution with respect to a set of constraints.

The only changes required from the proof's point of view is that whenever the solver finds a solution and introduces a new objective bound, it should use the "`soli`" rule (Rule 9) to witness a solution (see "Partial Solution Witnessing" below for some details), introducing a constraint requiring objective improvement, and define the corresponding inequality literal, and introduce the constraint setting it to true by RUP.

Example 3.7 *Take again the CSP from Example 3.2, and suppose there is an objective to maximise Y. Then, when the first solution ($X = 1, Y = 2, Z = 2$) is found, the proof logging solver should derive (by witnessing the solution with* `soli`*) the constraint* $\mathsf{BinEnc}(Y) \geq 3$*, after which it can derive $y_{\geq 3} \geq 1$ by RUP.*

In general, for an objective variable W and attained objective value o, a solution witness will introduce the constraint

$$\mathsf{BinEnc}(W) \geq o + 1 \text{ for a maximisation problem or}$$
$$-\mathsf{BinEnc}(W) \geq -o - 1 \text{ for a minimisation problem,}$$

followed by a corresponding literal definition and RUP constraint $w_{\geq o+1} \geq 1$ or $\overline{w}_{\geq o} \geq 1$.

This fits neatly into the above framework, as the RUP constraint derived after a solution is found can be seen simply as a propagation justification with $\mathcal{R} = \emptyset$, as if the objective-improving constraint had always existed as part of the model. It has the effect of maintaining Inv3 by ensuring the bound restriction on the objective variable domain propagates. If the CP solver is performing a complete search to establish a globally optimal solution, the final part of the proof should then just be a proof of unsatisfiability stemming from an unachievable objective-improving constraint. This, together with the final witness solution (which will have been checked against the correctly encoded PB model), constitutes a proof of the optimal solution and objective value for the input PB problem and hence (recall Theorem 3.1) for the original CP problem.

A Requirement for Partial Solution Witnessing

One technicality here is that for solution witnessing to be compact and user-friendly, we would ideally hope to only require witnessing a single equality literal for each of the variable-value pairs in the found satisfying assignment to the original CP variables. So in the above example we would like to write `soli` followed by just the literals $x_{=1}, y_{=2}, z_{=2}$, and in particular not have to write out the complete corresponding solution setting all PB variables appearing the encoding (bit variables, other atomic literals, and other auxiliary variables). Morally this should be acceptable, given the correspondence we established in Theorem 3.1. However, for a partial assignment to be verifiable as solution witnesses in the current version of the *VeriPB* proof checker, it is required that the given assignment *unit propagates* to a complete solution of the PB problem. We can guarantee this by additionally having our encodings respect the property that any partial assignment that sets all the bit variables $x_i \in \mathsf{BinEnc}(X)$ for every $X \in \mathcal{V}$ is guaranteed to propagate to a total assignment or contradiction. This is similar to the notion of "arc-consistency" in SAT-encoding literature [82].

For the linearisations we have presented so far, this property is straightforward to ascertain by observing that any variable that is fully reified on a constraint involving only bit variables will be propagated either true or false by an assignment to those variables. This already includes

for example the atomic bound variables. Then recursively, any variable that is fully reified on a constraint that only contains variables which we already know propagate is also guaranteed to propagate, unless contradiction is reached first. It can be verified by inspection of the Linearise procedures that all auxiliary variables are fully constrained by reification in at least one constraint in this manner.

3.3.4 Enumeration Proofs from Solution Excluding Search

At the time of writing, the *VeriPB* proof system did not strictly support enumeration proofs: certifying that the solver has found *all* solutions to a satisfaction problem. However, very recent work [96], still under review at the time of the latest revisions of this thesis, indicates that this should be a workable extension to the proof system.

When a solver performs complete solution excluding search — adding a nogood eliminating each solution once found — this type of conclusion could be dealt with in a manner conceptually similar to proofs of optimality. We would witness each solution (relying on unit propagation to get complete solutions) with a proof rule "`solx`" that introduces the negation of the conjunction of literals encoding the solution. Then, when the solver finally establishes unsatisfiability we would be able to conclude that "the problem has no solutions other than the solutions we have witnessed" and thus have an enumeration certificate. So our basic framework can in principle still be applied, and the *Glasgow Constraint Solver* implementation includes the facility to make use of this.

Unfortunately, we do have to be careful here, since as discussed in Chapter 2, the proof system is not necessarily implicational and certain rules such as redundance and dominance can add or remove solutions. To get a formal enumeration conclusion, we would require an additional guarantee that our set of bit variables corresponding to the original CP variables are "protected" or "preserved", and we are not allowed to change the projected solution set for these variables. This is the main extension of the current proof system required.

The required properties from the encoding would also need to be expanded:

P3. There is a one-to-one correspondence between the solutions to the CP problem, and the solutions to the PB problem projected onto a preserved set.

Further conclusion types are areas of ongoing work and research. For example, *VeriPB* could also in principle support certifying what the *number* of solutions is for a given problem, or certifying that the solver has found *all optimal solutions*.

3.4 Justification Procedures

So the final remaining task to accomplish before this framework can be implemented in an actual solver is to figure out how to construct propagation and conflict justifications for all the inferences

that could possibly be made by the propagators included in that solver. This is no small challenge. As noted in Example 3.1, there are a huge number of potential constraint types, and even though no CP solver implements propagators for all of them, many solvers support an extensive library of global constraints and propagators enforcing various levels of consistency by many different methods.

The good news is that we can immediately show that a number of constraints are easy to justify using PB reasoning, giving us strong reason to hope this framework will be suitable for a wide range of constraint propagation algorithms (see again the thesis statement in Section 1.4). In fact, all the constraints we provided encodings for in Section 3.2.3 fall into this category — with the possible exception of AllDifferent, depending on the definition of "easy".

3.4.1 Notation and Presentation

The important point to convey is that our justification procedures are both correct in terms of *VeriPB* proof rules, but also efficient and easily implementable. We will therefore adopt a presentation format for them that is a hybrid between algorithmic pseudocode and mathematical theorems. Specifically, each numbered "Justification Procedure" will have three parts: a description of preconditions detailing assumptions about constraints, inferences, and encodings; followed by pseudocode for a proof logging procedure; and then finally a *correctness proof* to show rigorously that if the preconditions hold, the steps logged by the procedure will be verified according to the proof rules described in Chapter 2. We will only give arguments in this section for the correctness of non-obvious proof steps, such as arguing a constraint will always be RUP, and omit derivations that should be verifiable by definition. Sometimes, there will be implicit preconditions that we miss out from the assumption lists, particularly if we are defining multiple procedures with similar preconditions. It will be made clear from the surrounding commentary when this is the case.

In addition to standard programming control keywords **if**, **for**, **proc** etc. we will adopt in our pseudocode some *VeriPB*-specific syntax for writing and recording proof steps with the following meaning:

1. **get** C : Obtain the PB constraint C, so it can be used in subsequent derivations. This means C must be an axiom from the input formula (equivalent to Rule 1), or must have been previously derived (by some precondition).
2. **cut** S: Write a sequence of cutting planes steps corresponding to the expression S (includes saturation via a "sat" function). (Rules 2 to 5).
3. **rup** D : Write a RUP step deriving D (Rule 6).
4. **imp** D **from** C : Write a syntactic implication step deriving D from C (Rule 7)
5. **red** D **with witness** ω : Write a redundance-based strengthening step deriving D (Rule 8) using the witness substitution ω (Rule 8).

6. **pbc** D **with witness** ω : Write a proof by contradiction step deriving D, (Rule 10, equivalent to Rule 8 with an empty witness).
7. **ext** $y \Leftrightarrow D$: Write two redundance-based strengthening steps deriving the two PB constraints that introduce a fresh variable y fully reified on the PB constraint C (Rule 8, via Theorem 2.4).
8. **subproof of** C : Write a directive to begin a new subproof for the proof obligation constraint C. Any nested statements following this that write to the proof are considered part of the subproof.

For the keywords above, we will assume that any operations deriving constraints return identifiers that can be used at future points in the procedure. For example, we might write pseudocode as follows.

$D_1 \leftarrow$ **rup** $x_1 + x_2 \geq 1$
$D_2 \leftarrow$ **get** $\overline{x}_1 + x_2 \geq 1$
$D_3 \leftarrow$ **cut** $\mathsf{sat}(D_1 + D_2)$

This can be concretely implemented either by using the *VeriPB* labels feature, or by keeping track of derived constraint IDs in the proof logging code itself. Furthermore, whenever atomic literals appear in a derived constraint this should be read as implicitly ensuring they are defined at an earlier point. This can be implemented efficiently with a map Defined to keep track of which literals are defined. The implicit pseudocode included before an atomic literal ℓ is used is then as follows.

if not $\mathsf{Defined}(\ell)$
    $\mathsf{Def}_{\Rightarrow}(\ell), \mathsf{Def}_{\Leftarrow}(\ell) \leftarrow$ **ext** $\ell \Leftrightarrow \mathsf{Def}(\ell)$
    $\mathsf{Defined} \leftarrow \mathsf{Defined} \cup (\ell, \mathit{true})$

Finally, it will often be useful to make use of a *generic reason* over a set of variables $\mathcal{X}$. This is a set of literals sufficient to encompass all the domain information for those variables in X at a point t. Specifically we will make use of a GenericR function as follows.

$$\begin{aligned} \mathsf{GenericR}(X, D) &= \{x_{\geq \min(D)}, \overline{x}_{\geq \max(D)+1}\} \\ &\quad \cup \{\overline{x}_{=v} : \min(D) < v < \max(D), v \notin D\} \end{aligned} \tag{3.45}$$

$$\mathsf{GenericR}_t(X) = \mathsf{GenericR}(X, \mathrm{dom}_t(X)) \tag{3.46}$$

$$\mathsf{GenericR}_t(\mathcal{X}) = \bigcup_{X \in \mathcal{X}} \mathsf{GenericR}_t(X) \tag{3.47}$$

3.4.2 Justifications using a single RUP step

We will start by dealing with constraints where the propagation or conflict justifications themselves are just RUP with respect to the encoding and literal definitions.

Justifying Not-Equals

The only filtering possible for a constraint $X \neq Y$ is to remove v from the domain of Y whenever $\mathrm{dom}(X) = \{v\}$ for some $v \in \mathbb{Z}$, and vice versa; and the only infeasibility that can be detected occurs when $\mathrm{dom}(X) = \mathrm{dom}(X) = \{v\}$.

Noting that $x_{=v} \Rightarrow \overline{y}_{=v} \geq 1$ and $x_{=v} \wedge y_{=v} \Rightarrow 0 \geq 1$ are just different ways of writing the same constraint, and that we can always swap X and Y without loss of generality, the following justification procedure can be used for any NotEquals inference.

Justification Procedure 3.1 (Not-Equals).
Preconditions: $X \neq Y$ *encoded as per Encoding Procedure 3.2; and* $v \in \mathbb{Z}$.
Procedure: **rup** $x_{=v} \wedge y_{=v} \Rightarrow 0 \geq 1$

Correctness Proof. The negation obviously propagates $x_{=v}$ and $y_{=v}$, which then from their definition constraints propagate $x_{\geq v}, \overline{x}_{\geq v+1}, y_{\geq v}$, and $\overline{y}_{\geq v+1}$. Theorem 2.8 then tells us that this sets all the bits in $\mathsf{BinEnc}(X) = \mathsf{BinEnc}(Y) = v$, which then propagates contradictory literals for the encoding flag u. □

Justifying Simple Comparisons

A binary comparison constraint is a special case of a linear inequality. It is of the form $X \geq Y$ or $X > Y$ for variables X, Y. Because bounds-consistency and domain-consistency are equivalent for this constraint, the only kind of inference we need to be able to justify is $y_{\geq v} \wedge x_{\geq u} \Rightarrow 0 \geq 1$ where $u \leq v$ (or $u < v$ in the second case).

Justification Procedure 3.2 (Comparison).
Preconditions: $B \in \{0, 1\}$*; and* $X - Y \geq B$ *encoded using* BinEnc *replacement.*
Procedure: **rup** $y_{\geq v} \wedge \overline{x}_{\geq u} \Rightarrow 0 \geq 1$

Correctness Proof. After propagating the negated justification, we will have active in the accumulated PB formula the three constraints

$$-\mathsf{BinEnc}(X) \geq -u + 1; \quad \mathsf{BinEnc}(X) - \mathsf{BinEnc}(Y) \geq B; \quad \mathsf{BinEnc}(Y) \geq v. \tag{3.48}$$

If X and Y are both encoded with two-complement bit strings, these meet exactly the conditions for Theorem 2.9 and hence unit propagate to contradiction. If either or both are not encoded with two's complement bit strings, we are in the same the situation as would have been reached after propagating the most significant bit in the proof of Theorem 2.9. So the same argument applies. □

Justifying Table Constraints

There are many algorithms for efficiently propagating table constraints [189, 140, 58]. Fortunately, we can simply demonstrate that any individual domain-consistent pruning will always be RUP by using the generic reason, without having to go into the details of any particular algorithm.

Justification Procedure 3.3 (Table propagation).
Preconditions: $\mathsf{Table}(X_1, \ldots, X_n; \tau)$ *encoded as in Encoding Procedure 3.10 for a set of tuples* τ*;* $k \in \{1, \ldots, n\}$*;* $v \in \mathbb{Z}$*; and* $X_k = v$ *is not domain-consistent.*
Procedure:

1: $\mathcal{R} \leftarrow \mathsf{GenericR}_t(X_1, \ldots, X_n)$
2: **rup** $\mathcal{R} \Rightarrow \overline{x}_{k=v} \geq 1$

Proof. By the definition of domain-consistency $X_k = v$ must have no support among the possible solutions to the table constraint, i.e. there does not exist a tuple $\tau_j \in \tau$ where $\tau_j[k] = v$ and $\tau_j[i] \in \mathrm{dom}(X_i)$ for all $j \in \{1, \ldots, n\} \setminus \{k\}$. We argue that the propagation justification follows by RUP by showing that all selector variables t_j must propagate to 0 under its negation, contradicting the final constraint from line 3 of the encoding.

Consider the part of the Encoding Procedure 3.10 reified on the j_{th} selector variable

$$t_j \Leftrightarrow \sum_i x_{i=\tau_j[i]} \geq n \tag{3.49}$$

If $\tau_j[k] \neq v$ then the constraint can never be satisfied unless $t_j = 0$, so $\overline{t}_j$ propagates. On the other hand, if $\tau_j[k] = v$, then by the above there must exist an $i \neq k$ where $\tau_j[i] \notin \mathrm{dom}(X_i)$, and so under $\mathsf{GenericR}_t(X_1, \ldots, X_n)$, $\overline{x}_{i=\tau_j[i]}$ propagates and hence $\overline{t}_j$ propagates. □

A very similar argument shows that if no values in the current domain have support then $\mathsf{GenericR}_t(X_1, \ldots, X_n) \Rightarrow 0 \geq 1$ (conflict justification) must also be RUP.

Justification Procedure 3.4 (Table infeasibility).
Preconditions: $\mathsf{Table}(X_1, \ldots, X_n; \tau)$ *encoded as in Encoding Procedure 3.10 for a set of tuples* τ*; and the domain state* dom_t *is infeasible.*
Procedure:

1: $\mathcal{R} \leftarrow \mathsf{GenericR}_t(X_1, \ldots, X_n)$
2: **rup** $\mathcal{R} \Rightarrow 0 \geq 1$

Justifying Array Min/Max

We will only discuss how to justify the ArrayMin and assert that the ArrayMax case is analogous.

Propagation of this constraint can be understood as a set of filtering rules which we will consider in turn. These rules, when enforced iteratively until a fixed point, give *domain-consistency* (or conflict in the case of infeasible domains), however we will not prove this, as we will prove as part of the generic justification method in Chapter 4 that any domain-consistent inference for this constraint can always be justified in general.

In each of the following we assume we have a global constraint $\mathsf{ArrayMin}(X_1, \dots, X_n, Y)$, encoded as in Encoding Procedure 3.6.

Justification Procedure 3.5 (Upper bounding for Array Max).
Preconditions: $i \in \{1, \dots, n\}$; $u := \max(\mathrm{dom}_t(X_i))$; *and hence a propagator can infer* $Y \leq u$.
Procedure: **rup** $\overline{x}_{i \geq u+1} \Rightarrow \overline{y}_{\geq u+1} \geq 1$

Correctness Proof. Immediate from Theorem 2.6 and Theorem 2.9. □

Justification Procedure 3.6 (Lower bounding for Array Max).
Preconditions: $i \in \{1, \dots, n\}$; $l := \min(\mathrm{dom}(Y))$; *and hence a propagator can infer* $X_i \geq l$.
Procedure: **rup** $y_{\geq l} \Rightarrow x_{i \geq l} \geq 1$

Correctness Proof. Again immediate from Theorem 2.6 and Theorem 2.9. □

Justification Procedure 3.7 (Missing value for Array Max).
Preconditions: $v \in \mathrm{dom}_t(Y)$; $v \notin \cup_i \mathrm{dom}(X_i)$; *and hence a propagator can infer* $Y \neq v$.
Procedure: **rup** $\bigwedge_i \overline{x}_{i=v} \Rightarrow \overline{y}_{=v} \geq 1$

Correctness Proof. From the negation of the propagation justification, the literals $\overline{x}_{i=v}$ for each i and also $y_{=v}$ propagates. This will directly contradict the constraint $y_{=v} \Rightarrow \sum_i x_{i=v} \geq 1$, which we have from Encoding Procedure 3.6. □

Justification Procedure 3.8 (Forced value for Array Max).

Preconditions: $w \notin \mathrm{dom}_t(Y)$; *for every* $v \in \mathrm{dom}_t(Y)$, *we have* $v \in \mathrm{dom}(X_i)$ *for some* $i \in \{1, \ldots, n\}$ *and* $v \notin \mathrm{dom}(X_j)$ *for every* $j \neq i$; *and hence a propagator can infer* $X_i \neq w$.

Procedure:

1: $\mathcal{R} \leftarrow \{\overline{x}_{j=v} : j \in \{1, \ldots, n\} \setminus \{i\},\ v \in \mathrm{dom}(Y)\}$
2: **rup** $\mathcal{R} \Rightarrow \overline{x}_{i=w}$

Correctness Proof. From Encoding Procedure 3.6 we have, for each $v \in \mathrm{dom}_0(Y)$

$$y_{=v} \Rightarrow \sum_i x_{i=v} \geq 1. \tag{3.50}$$

Now under propagation of the negation of the propagation justification this is reduced to

$$y_{=v} \Rightarrow 0 \geq 1, \tag{3.51}$$

so $\overline{y}_{=v}$ propagates. But then $\overline{y}_{=v}$ propagates for all $v \in \mathrm{dom}_0(Y)$ which we know by Theorem 3.2 leads to a contradiction. □

3.4.3 Justifications using multiple RUP steps

Other constraints do not immediately allow for simple RUP justifications, but yield with some additional RUP derivations for each inference.

Justifying Element

Similar to Array Min/Max, we can understand the Element propagator as a collection of filtering rules, and skip for now the proof that these encompass *any* domain-consistent inference. In each of the following, we assume we have a global constraint $\mathsf{Element}(X_1, \ldots, X_n, Y, Z)$ encoded as in Encoding Procedure 3.4.

Justification Procedure 3.9 (Empty intersection for Element).

Preconditions: $v \in \mathrm{dom}_t(Y)$; *and* $\mathrm{dom}_t(X_v) \cap \mathrm{dom}_t(Z) := \emptyset$.

Procedure:

1: $\mathcal{R} \leftarrow \mathsf{GenericR}_t(X_1, \ldots, X_n, Y, Z)$
2: **for** $w \in \mathrm{dom}_t(X_v)$
3: ⌊ **rup** $\mathcal{R} \Rightarrow \overline{y}_{=v} + \overline{x}_{v=w} \geq 1$
4: **rup** $\mathcal{R} \Rightarrow \overline{y}_{=v} \geq 1$

Correctness Proof. We first show that each constraint on line 3 will be RUP. The negation of this constraint propagates all the literals in $\mathsf{GenericR}_t(\dots)$ along with $y_{=v}$ and $x_{v=w}$. By Theorem 2.8, this means all the bits in $\mathsf{BinEnc}(X_v)$ are set so that $\mathsf{BinEnc}(X_v) = w$, which means the two constraints representing the corresponding reified equality from Encoding Procedure 3.4 are reduced to

$$Z \geq v \quad \text{and} \quad -Z \geq -v. \tag{3.52}$$

Theorem 2.8 then implies propagation setting $\mathsf{BinEnc}(Z) = v$. Since we know $Z \neq v$ in the current domain state, this must be sufficient to cause contradiction under $\mathsf{GenericR}_t$.

Now assuming all the constraints from line 3 are derived for each $w \in \mathrm{dom}(X_v)$, it is immediate that the final propagation justification will be RUP, since the negation of this now propagates $\overline{x}_{v=w}$ for every $w \in \mathrm{dom}(X_v)$, once again triggering a domain-wipeout contradiction as per Theorem 3.2. □

Justification Procedure 3.10 (Missing value for Element).
Preconditions: $v \in \mathrm{dom}_t(Z)$; $v \notin \cup_{i \in \mathrm{dom}_t(Y)} \mathrm{dom}_t(X_i)$; *and hence a propagator can infer* $Z \neq v$.
Procedure:

1: $\mathcal{R} \leftarrow \mathsf{GenericR}_t(X_1, \dots, X_n, Y, Z)$
2: **for** $i \in \mathrm{dom}_t(Y)$
3: $\quad$ **rup** $\mathcal{R} \Rightarrow \overline{z}_{=v} + \overline{y}_{=i} \geq 1$
4: **rup** $\mathcal{R} \Rightarrow \overline{y}_{=i} \geq 1$

Correctness Proof. Analogous to the proof for Justification Procedure 3.9. □

Justification Procedure 3.11 (Single value for Element).
Preconditions: $\mathrm{dom}(Y) := \{i\}$; $v \in \mathrm{dom}(X_i) \setminus \mathrm{dom}_t(Z)$; *and hence a propagator can infer* $X \neq v$.
Procedure:

1: $\mathcal{R} \leftarrow \mathsf{GenericR}_t(X_1, \dots, X_n, Y, Z)$
2: **rup** $\mathcal{R} \Rightarrow \overline{x}_{i=v} \geq 1$

Correctness Proof. Propagation of the literals in $\mathcal{R}$ must propagate $y_{=i}$. Theorem 2.8 implies that propagation of $x_{i=v}$ sets $\mathsf{BinEnc}(X) = v$. These together cause Encoding Procedure 3.4 to propagate literals sufficient to set $\mathsf{BinEnc}(Z) = v$, which then must cause contradiction under $\mathcal{R}$. □

Justifying Equality

Filtering for simple equality constraints $X = Y$ is also a single RUP step, but infeasibility requires slightly more work. The domains are infeasible if and only if $\mathrm{dom}(X) \cap \mathrm{dom}(Y) = \emptyset$, and the only pruning possible is to remove any values v from either domain where $v \notin \mathrm{dom}(X) \cap \mathrm{dom}(Y)$.

Justification Procedure 3.12 (Equality propagation).
Preconditions: $X = Y$ *encoded via* BinEnc *replacement.*
Procedure: **rup** $\overline{y}_{=y} \Rightarrow \overline{x}_{=v} \geq 1$

Correctness Proof. The negation of the propagation justification propagates $\overline{x}_{=v}$ and $y_{=v}$, the latter of which sets $\mathsf{BinEnc}(Y) = v$, by Theorem 2.8. But then the two PB constraints encoding $X = Y$ set $\mathsf{BinEnc}(X) = v$, also by Theorem 2.8, which directly contradicts the definition of $\overline{x}_{=v}$. □

Justification Procedure 3.13 (Equality infeasibility).
Preconditions: $X = Y$ *encoded via* BinEnc *replacement;* $l := \min(\mathrm{dom}_t(X))$*;* $u := \max(\mathrm{dom}(X))$*;* $R := \{l, ..., u\}$*; and* dom_t *is infeasible with respect to* $X = Y$.
Procedure:

1: **for** $v \in R \cap \mathrm{dom}_t(X)$
2: **rup** $\overline{y}_{=v} \Rightarrow \overline{x}_{=v} \geq 1$
3: $\mathcal{R} \leftarrow \mathsf{GenericR}_t(X)$
4: $\mathcal{R} \leftarrow \mathcal{R} \cup \{\overline{y}_{=v} : v \in R \cap \mathrm{dom}_t(X)\}$
5: **rup** $\mathcal{R} \Rightarrow 0 \geq 1$

Correctness Proof. The same argument as in the proof Justification Procedure 3.12 shows each constraint on line 2 is RUP. After these are in place, the infeasibility justification is RUP, because for any $v \in \mathrm{dom}_0(X)$, v is excluded by a literal either directly in $\mathsf{GenericR}_t(X)$, or indirectly after propagating $\overline{y}_{=v}$ and then $\overline{x}_{=v}$. Theorem 3.2 tells us that this causes a domain-wipeout contradiction. □

3.4.4 Justifications using Cutting Planes

RUP derivations have the advantage of being very straightforward for a solver to write out, placing somewhat more onus on the proof checker to figure out the "details" of why a constraint introduction is sound. But as mentioned, the unit propagation properties of our encodings from

Section 3.2 can be quite weak — recall again the non-propagating contradictory constraints in Example 2.15. For some constraints it makes more sense to try to take advantage of the more powerful rules in the *VeriPB* proof system and write out explicit cutting planes derivations.

Justifying Linear Inequalities

Consider first a (non-reified) linear inequality constraint on CP variables $X_1, \ldots, X_n$, with coefficients $a_1, \ldots, a_n$ and right-hand-side b.

Without loss of generality, we can assume this has the form

$$\mathsf{L} := a_1 X_1 + a_2 X_2 + \ldots a_n X_n \geq b \tag{3.53}$$

since the $\leq, <$, and $>$ cases can all be rearranged to match this. Furthermore, we only need to consider bounds-consistency, since bounds and domain consistency are equivalent for this constraint [198]. In what follows we assume we have in the input PB formula the BinEnc replacement for the linear constraint L.

Justification Procedure 3.14 (Linear inequality infeasibility).

Preconditions: *dom_t is infeasible with respect to L; and for each $i \in \{1, \ldots, n\}$* $l_i := \min(\mathrm{dom}_t(X_i))$ *and* $u_i := \max(\mathrm{dom}_t(X_i))$.

Procedure:

1: $\mathcal{L}_1 \leftarrow \{(a_i, x_{i \geq l_i}) : i \in \{1, \ldots, n\}, a_i < 0\}$

2: $\mathcal{L}_2 \leftarrow \{(a_i, \overline{x}_{i \geq u_i + 1}) : i \in \{1, \ldots, n\}, a_i \geq 0\}$

3: $\mathcal{R} \leftarrow \{\ell : (a_i, \ell) \in \mathcal{L}_1 \cup \mathcal{L}_2\}$

4: $C \leftarrow$ **get** $a_1 \cdot \mathsf{BinEnc}(X_1) + \cdots + a_n \cdot \mathsf{BinEnc}(X_n) \geq b$

5: $D \leftarrow$ **cut** $C + \sum_{(a_i, \ell) \in \mathcal{L}_1} a_i \cdot \mathsf{Def}_{\Rightarrow}(\ell) + \sum_{(a_i, \ell) \in \mathcal{L}_2} a_i \cdot \mathsf{Def}_{\Leftarrow}(\ell)$

6: **imp** $\mathcal{R} \Rightarrow 0 \geq 1$ **from** D

Correctness Proof. Let us show that the linear combination on line 5 will derive $0 \geq A$ for positive A, from $C\restriction_{\mathcal{R}} \cup \mathsf{Def}(\mathcal{R})\restriction_{\mathcal{R}}$, with a view to applying Lemma 2.2. Denote the cutting planes linear combination on line 5 by S.

First notice that the choice of literals in $\mathcal{R}$ ensures that all the variables in $C\restriction_{\mathcal{R}} = C$ would be cancelled out after computing $S\restriction_{\mathcal{R}}$ and we would be left with

$$0 \geq b - \sum_{i : a_i \geq 0} a_i \cdot u_i - \sum_{i : a_i < 0} a_i \cdot l_i \tag{3.54}$$

Because dom_t is infeasible, we also know that

$$\max\left\{\sum_{i=1}^{n} a_i\rho(X_i) : \mathsf{Valid}(\rho, \mathrm{dom}_t)\right\} < b \tag{3.55}$$

and hence

$$\sum_{i=1}^{n} \max\{a_i\rho(X_i) : \mathsf{Valid}(\rho, \mathrm{dom}_t)\} < b. \tag{3.56}$$

Now, if $a_i \geq 0$, we have $\max\{a_i\rho(X_i) : \mathsf{Valid}(\rho, \mathrm{dom}_t)\} = a_i \cdot u_i$ and if $a_i < 0$, we have $\max\{a_i\rho(X_i) : \mathsf{Valid}(\rho, \mathrm{dom}_t)\} = a_i \cdot l_i$. Hence,

$$\sum_{i:a_i\geq 0} a_i \cdot u_i + \sum_{i:a_i<0} a_i \cdot l_i < b \tag{3.57}$$

and so the right-hand side in (3.54) is positive.

This means, by Lemma 2.2 that if D is the result of computing S, then $D\restriction_{\mathcal{R}} = 0 \geq A$ for positive A. So by Lemma 2.1 we can derive $\mathcal{R} \Rightarrow 0 \geq A$ using saturation and syntactic implication steps. Hence, the final conflict justification will indeed be syntactically implied. □

Propagation justifications are then a straightforward corollary of this.

Justification Procedure 3.15 (Linear inequality propagation).
Preconditions: $c \in \{1, -1\}$*; and from* dom_t*, a propagator for* L *can infer* $c \cdot X_i \geq c \cdot v$.
Procedure:

$\mathcal{L}_1 \leftarrow \{(a_i, x_{i\geq l_i}) : i \in \{1, \ldots, n\}, a_i < 0\}$
$\mathcal{L}_2 \leftarrow \{(a_i, \overline{x}_{i\geq u_i+1}) : i \in \{1, \ldots, n\}, a_i \geq 0\}$
if $c = -1$
    $\mathcal{L}_1 \leftarrow \mathcal{L}_1 \setminus \{(a_i, x_{i\geq l_i})\}$
    $\mathcal{L}_1 \leftarrow \mathcal{L}_1 \cup \{(a_i, x_{i\geq v+1}\}$
else if $c = 1$
    $\mathcal{L}_2 = \mathcal{L}_2 \backslash \{(a_i, \overline{x}_{i\geq u_i+1})\}$
    $\mathcal{L}_2 = \mathcal{L}_2 \cup \{(a_i, \overline{x}_{i\geq v})\}$
$\mathcal{R} \leftarrow \{\ell : (a_i, \ell) \in \mathcal{L}_1 \cup \mathcal{L}_2\}$
$C \leftarrow$ **get** $a_1 \cdot \mathsf{BinEnc}(X_1) + \cdots + a_n \cdot \mathsf{BinEnc}(X_n) \geq b$
$D \leftarrow$ **cut** $C + \sum_{(a_i,\ell)\in\mathcal{L}_1} a_i \cdot \mathsf{Def}_{\Rightarrow}(\ell) + \sum_{(a_i,\ell)\in\mathcal{L}_2} a_i \cdot \mathsf{Def}_{\Leftarrow}(\ell)$
imp $\mathcal{R} \Rightarrow 0 \geq 1$ **from** D

Correctness Proof. If from dom_t, L implies $X \geq v$, then modifying dom_t with $X < v$ must result in an infeasible domain. Likewise, for $X < v$: modifying dom_t with $X \geq v$ must be

infeasible. The correctness of the procedure is then immediate from the correctness of Justification Procedure 3.15. Note that $\mathcal{R} \Rightarrow 0 \geq 1$ is a propagation justification since $\mathcal{R}$ includes the negation of the literal encoding the domain change. □

The following example rewrites the original sketch of how to justify inequalities from Gocht et al. [94], using the procedure above.

Example 3.8 *Suppose we have a constraint $2X+3Y+4Z \leq 42$, with all three variables having initial domain $\{0,\dots,8\}$; but in a particular domain state a CP solver knows $X \geq 5$ and $Z \geq 3$. Then a linear inequality propagator will infer that $Y \leq 6$.*
We can justify this as follows. First, introduce with definitions the inequality literals $x_{\geq 5}$, $y_{\geq 7}$, $z_{\geq 3}$ if they do not already exist, so we have the constraints

$$x_{\geq 5} \Rightarrow \sum_{i=0}^{3} 2^i x_i \geq 5; \qquad y_{\geq 7} \Rightarrow \sum_{i=0}^{3} 2^i y_i \geq 7; \qquad z_{\geq 3} \Rightarrow \sum_{i=0}^{3} 2^i z_i \geq 3; \tag{3.58}$$

and from the encoding of the linear inequality (via BinEnc*) we have*

$$-\sum_{i=0}^{3} 2 \cdot 2^i x_i - \sum_{i=0}^{3} 3 \cdot 2^i y_i - \sum_{i=0}^{3} 4 \cdot 2^i y_i \geq -42. \tag{3.59}$$

Now using cutting planes steps, we can multiply the constraints in (3.58) by $2, 3$ and 4 respectively (the coefficients of the corresponding variables in the original linear inequality). We then add these to the encoding constraint (3.59) obtaining

$$25\overline{x}_{\geq 5} + 49\overline{y}_{\geq 7} + 9\overline{z}_{\geq 3} \geq (10 + 21 + 12 - 42) = 1 \tag{3.60}$$

Then saturating this constraint gives us precisely the propagation justification

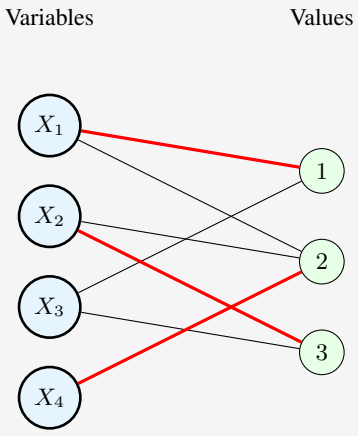
$$x_{\geq 5} \wedge z_{\geq 3} \Rightarrow \overline{y}_{\geq 7} \geq 1. \tag{3.61}$$

Justifying All-Different

The AllDifferent was the first global constraint shown to be amenable to pseudo-Boolean justification. Unlike the constraints we have considered so far, this does require some more detailed consideration of a specific domain-consistent propagation algorithm and so might not fall so obviously into the category of "easy to justify". However, the original explanation by Elffers et al. [67] was mostly by example, only used sequences of guesses as reasons, and used a different encoding to what is now used in the *Glasgow Constraint Solver*. So it is worth giving this an updated treatment here.

First, to justify infeasibility, we can take advantage of the fact that the classical domain-consistent propagation algorithm for AllDifferent [171] involves solving a matching problem on a directed bipartite graph (sometimes called the *value graph*) where variables $\{X_1, \ldots, X_n\}$ and domain values $\{v_1, \ldots, v_m\}$ form the left and right vertex sets, and the edges are precisely all the pairs (X_i, v_j) where $v_j \in \mathrm{dom}_t(X)$. We don't need to cover all the details of the algorithm here: it is sufficient to note that it will efficiently compute a *maximum cardinality matching*, i.e. a set of edges where no vertex has more than one incident edge of largest possible size. It can be observed that for the AllDifferent constraint to be feasible, every maximum cardinality matching must be *left-saturating*: each variable vertex on the left must have at least one incident edge. The propagation algorithm then detects infeasibility by using an *augmenting-paths* algorithm finding a maximum cardinality matching that is *not* left-saturating.

Example 3.9 *Suppose we have an* AllDifferent *constraint and for a given domain state* dom_t *we have:* $\mathrm{dom}_t(X_1) = \{1, 2\}; \mathrm{dom}_t(X_2) = \{2, 3\}; \mathrm{dom}_t(X_3) = \{1, 3\}; \mathrm{dom}_t(X_4) = \{2\}$.
Then the value graph has a maximum cardinality matching as shown that is not left-saturating and hence the constraint is infeasible.



We can exploit this fact, and a connection to *Hall's Marriage Theorem* [102] to derive conflict justifications for AllDifferent. The key insight is that a *Hall violator* can be extracted from the augmenting paths algorithm for finding a maximum cardinality matching and this immediately leads to an easy PB derivation. To briefly review terminology, given a bipartite graph (V, E) and matching $M \subseteq E$, we have:

- A vertex is *matched* if it is incident to an edge in M, and *unmatched* otherwise.
- An *alternating path* is a sequence of edges $P \subseteq E$ beginning with an unmatched vertex such that consecutive edges alternate between being in the matching M and not being in M.

- An *augmenting path* is an alternating path that also ends with an unmatched vertex.
- A *Hall violator* is a set of left vertices $W \subseteq V$ for which $|W| > |\mathsf{Nh}(W)|$, where $\mathsf{Nh}(W) := \{v \in V : \exists w \in W, (w, v) \in E\}$ denotes the set of right vertices adjacent to vertices in W.

A maximum cardinality matching will be identified once a matching is found M that does not permit any augmenting paths [170]. Any remaining alternating paths are therefore not augmenting, and if M is not left-saturating, there must always exist at least one of these. This allows us the following lemma.

Lemma 3.5 *Let $G = (V_L \cup V_R, E)$ be a bipartite graph and M be a maximum cardinality matching that is not left saturating. Then a Hall violator W can always be constructed by finding all vertices reachable via an alternating path from an unmatched vertex v.*

Proof. Since M is not left-saturating, there exists at least one unmatched left vertex $v \in V_L$. Consider the set W of all left vertices reachable from v via alternating paths (paths that start with an unmatched edge and then alternate between edges not in M and edges in M). Let Z be the set of right vertices reachable from v via such paths.

By construction, every vertex in Z is matched in M; otherwise, there would exist an augmenting path, contradicting the maximality of M. Therefore, for each vertex in Z, there is at least one corresponding vertex in W, plus the original unmatched vertex v, so $|W| \geq |Z| + 1$. Furthermore, Z is exactly the set of right vertices adjacent to W, i.e., $Z = \mathsf{Nh}(W)$. This is because for any edge (w, z) with $w \in W$, if (w, z) is in M, then w must have been reached via z, and thus $z \in Z$; and if (w, z) is not in M, then z can be reached by extending the alternating path through the matching edge to w.

Thus, we have $|W| \geq |\mathsf{Nh}(W)| + 1$, so W is a Hall violator. □

Given this lemma, we can assume we have a procedure $\mathsf{FindHallViolator}(M)$ which computes a Hall violator given a maximum-cardinality matching that is not left-saturating. Recalling Theorem 2.3 we can also assume we have a procedure RecoverCardinality that derives an at-most-one constraint from a clique of n "not-both" constraints.

Justification Procedure 3.16 (All-Different infeasibility).
Preconditions: $\mathcal{X} := \{X_1, \ldots, X_n\}$; $\mathsf{AllDifferent}(\mathcal{X})$ *encoded as per Encoding Procedure 3.9;* dom_t *is infeasible with respect to* $\mathsf{AllDifferent}(\mathcal{X})$; $\mathcal{D} := \bigcup_i \mathrm{dom}_t(X_i)$; *and* $M \subseteq \mathcal{X} \times \mathcal{D}$ *is a maximum-cardinality matching that is not left-saturating.*
Procedure:

1: $W \leftarrow \mathsf{FindHallViolator}(M)$

2: **proc** SumAtLeastAtMost(W)
3: $l, u \leftarrow \min \mathsf{Nh}(W), \max \mathsf{Nh}(W)$ $\quad \mathcal{R} \leftarrow \emptyset$
4: **for** $X_i \in W$
5: $\mathcal{R}_i \leftarrow \{x_{i \geq l}, \overline{x}_{i \geq u+1}\} \cup \{\overline{x}_{i=v} : v \in \{l, \dots, u\} \setminus \mathsf{Nh}(W)\}$
6: $\mathcal{R} \leftarrow \mathcal{R} \cup \mathcal{R}_i$
7: $L_i \leftarrow$ **rup** $\mathcal{R}_i \Rightarrow \sum_{v \in \mathsf{Nh}(W)} x_{i=v} \geq 1$
8: **for** $v \in \mathsf{Nh}(W)$
9: $\mathcal{N} \leftarrow \emptyset$
10: **for** $(X_i, X_j) \in W \times W$ **where** $i < j$
11: **rup** $\overline{x}_{i=v} + \overline{x}_{j=v} \geq 1$
12: $M_v \leftarrow \mathsf{RecoverCardinality}(\mathcal{N})$
13: **return** $(\mathcal{R},$ **cut** $\sum_{X_i \in W} L_i + \sum_{v \in \mathsf{Nh}(W)} M_v)$
14: $\mathcal{R}, D \leftarrow \mathsf{SumAtLeastAtMost}(W)$
15: **imp** $\mathcal{R} \Rightarrow 0 \geq 1$ **from** D

Correctness Proof. Each of the constraints on line 7 are RUP by Theorem 3.2. And each of the constraints on line 11 are RUP by the same argument in the proof of Justification Procedure 3.1 (since we have $X_j \neq X_i$ encoded in the same way as Encoding Procedure 3.2). Theorem 2.3 implies that the constraint we recover on line 12 is of the form

$$M_v := \sum_{i : X_i \in W} \overline{x}_{i=v} \geq 1. \tag{3.62}$$

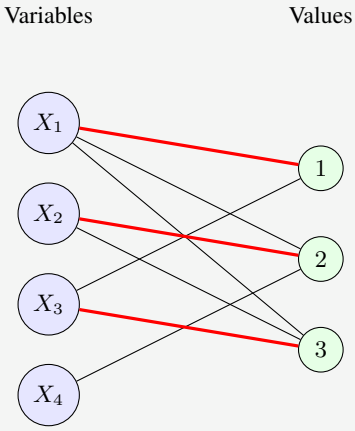
Now consider the at-least-one L_i and at-most-one constraints M_v restricted by the reason $\mathcal{R}$, and note that the sets of PB constraints

$$\{L_i{\restriction_{\mathcal{R}}} : X_i \in W\} \quad \text{and} \quad \{M_v{\restriction_{\mathcal{R}}} : v \in \mathsf{Nh}(W)\}$$

contains all the same variables with opposite signs. So summing these gives 0 on the left-hand side and $|W| - |\mathsf{Nh}(W)| \geq 1$ (since W is a Hall violator) on the right-hand side. This means that due to Lemma 2.2 we can be sure $(\sum_i L_i + \sum_v M_v){\restriction_{\mathcal{R}}} = 0 \geq b$ for some $b > 0$, and hence Lemma 2.1 tells us we can apply syntactic implication and saturation to the result of $(\sum_i L_i + \sum_v M_v)$ to yield $\mathcal{R} \Rightarrow 0 \geq 1$. □

Elffers et al. [67] and Gocht [88] both provide worked examples for this Hall violator argument, but we will take the opportunity here to rewrite an example with the slightly different reason literals we used above.

Example 3.10 *Suppose we have an* AllDifferent *constraint on variables* X_1, X_2, X_3, X_4*, all with initial domains* $\{1, \ldots, 4\}$ *and a CP solver reaches a domain state where* $\mathrm{dom}_t(X_1) = \{1, 2, 3\}$*,* $\mathrm{dom}_t(X_2) = \{2, 3\}$*,* $\mathrm{dom}_t(X_3) = \{1, 3\}$ *and* $\mathrm{dom}_t(X_4) = \{2\}$*. The* AllDifferent *is therefore infeasible, and a propagator might infer this by identifying the following maximum matching in the value graph, with unmatched vertex* X_4*.*



Following all possible alternating paths from X_4 *will yield the Hall violator*

$$\{X_1, X_2, X_3, X_4\} \text{ with neighbourhood } \{1, 2, 3\}.$$

So in the proof we first derive by RUP reified at-least-1 constraints

$$x_{1\geq 1} \wedge \overline{x}_{1\geq 4} \Rightarrow x_{1=1} + x_{1=2} + x_{1=3} \geq 1; \tag{3.63}$$

$$x_{2\geq 1} \wedge \overline{x}_{2\geq 4} \Rightarrow x_{2=1} + x_{2=2} + x_{2=3} \geq 1; \tag{3.64}$$

$$x_{3\geq 1} \wedge \overline{x}_{3\geq 4} \Rightarrow x_{3=1} + x_{3=2} + x_{3=3} \geq 1; \tag{3.65}$$

$$x_{4\geq 1} \wedge \overline{x}_{4\geq 4} \Rightarrow x_{4=1} + x_{4=2} + x_{4=3} \geq 1; \tag{3.66}$$

followed by the "not-both" constraints

$$-x_{1=1} - x_{2=1} \geq -1; \qquad -x_{1=1} - x_{3=1} \geq -1;$$
$$-x_{1=1} - x_{4=1} \geq -1; \qquad -x_{2=1} - x_{3=1} \geq -1;$$
$$-x_{2=1} - x_{4=1} \geq -1; \qquad -x_{3=1} - x_{4=1} \geq -1;$$

to recover the at-most-1 constraint

$$-x_{1=1} - x_{2=1} - x_{3=1} - x_{4=1} \geq -1; \tag{3.67}$$

then

$$-x_{1=2} - x_{2=2} \geq -1; \qquad -x_{1=2} - x_{3=2} \geq -1;$$
$$-x_{1=2} - x_{4=2} \geq -1; \qquad -x_{2=2} - x_{3=2} \geq -1;$$
$$-x_{2=2} - x_{4=2} \geq -1; \qquad -x_{3=2} - x_{4=2} \geq -1;$$

to recover the at-most-1 constraint

$$-x_{1=1} - x_{2=1} - x_{3=1} - x_{4=1} \geq -1; \tag{3.68}$$

and finally

$$\begin{aligned} -x_{1=3} - x_{2=3} &\geq -1; & -x_{1=3} - x_{3=3} &\geq -1; \\ -x_{1=3} - x_{4=3} &\geq -1; & -x_{2=3} - x_{3=3} &\geq -1; \\ -x_{2=3} - x_{4=3} &\geq -1; & -x_{3=3} - x_{4=3} &\geq -1; \end{aligned}$$

to recover the at-most-1 constraint

$$-x_{1=1} - x_{2=1} - x_{3=1} - x_{4=1} \geq -1. \tag{3.69}$$

Note that if the AllDifferent *had propagated before, the at-most-1 constraints (or stronger versions of them) might already have been recovered in the proof and then the not-both RUP steps would be unnecessary. Regardless, summing the* 4 *at-least-1 and* 3 *at-most-1 constraints gives*

$$x_{1\geq 1} \wedge \overline{x}_{1\geq 4} \wedge x_{2\geq 1} \wedge \overline{x}_{2\geq 4} \wedge x_{3\geq 1} \wedge \overline{x}_{3\geq 4} \wedge x_{4\geq 1} \wedge \overline{x}_{4\geq 4} \wedge \Rightarrow 0 \geq 1. \tag{3.70}$$

If an AllDifferent constraint is not infeasible for a domain state, the propagator has the opportunity to make inferences that remove unsupported values. This is also based on matchings in the value graph: filtering occurs once the propagator has identified a maximum cardinality that *is* left-saturating. To find the values that can be filtered, a *residual* graph and its strongly-connected components is calculated from the value graph based on the matching M. There are multiple possible ways of defining this, but we will use the definition from Gent et al. [83, Definition 2.2], except with the edge direction reversed, as it simplifies the description of the inference procedure.

For an undirected bipartite graph $G = (V_l \cup V_R, E)$ and matching M, the *residual graph* is the directed graph $R(G) = (V', E')$ where $V' = V_l \cup V_R \cup \{t\}$ are the vertices of the original graph, plus an additional "sink" vertex t, and the edge set $E' = E_1 \cup E_2 \cup E_3 \cup E_4$ is defined as follows:

- Direct the matching edges *left-to-right*:
  $E_1 := \{(w, v) : (v, w) \in M\}$.
- Direct the non-matching edges *right-to-left*:
  $E_2 := \{(v, w) : (v, w) \in E \setminus M\}$.
- Connect the sink to matched right vertices:
  $E_3 := \{(t, w) : \exists v \in V_l \text{ s.t. } (v, w) \in M\}$.
- Connect unmatched right vertices to the sink:

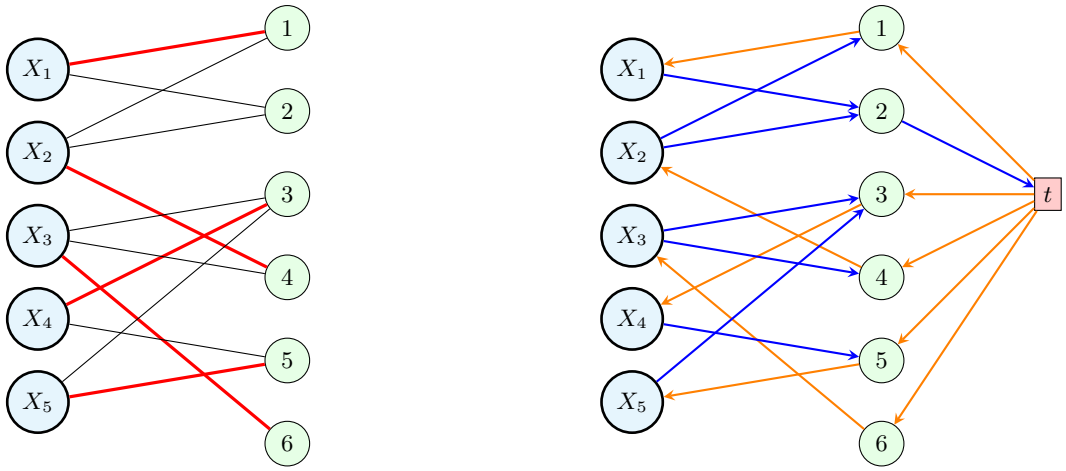


Figure 3.1: Left-saturating matching and its residual graph

$$E_4 := \{(w, t) : \nexists v \in V_R \text{ s.t. } (v, w) \in M\}.$$

As with any directed graph, the *strongly-connected components* of the residual graph for a given maximum-cardinality matching can be found efficiently, e.g. via Tarjan's algorithm [183]. Recall that a strongly-connected component of a directed graph is a maximal subset of vertices where every vertex is reachable from every other vertex.

It turns out that for the value graph, a variable-pair (X_i, v) has no support precisely when the corresponding edge is not part of the matching, and in the residual graph begins and ends in *different* strongly-connected components [171].

The justification procedure for these deletions is based on finding *Hall sets* as opposed to Hall violators. For a bipartite graph with vertices $V_L \cup V_R$, a *Hall set* is a set of left vertices $W \subseteq V$ for which $|W| = |\mathsf{Nh}(W)|$. We will make use of the following lemma.

Lemma 3.6 *Let $G = (V_L \cup V_R, E)$ be a bipartite graph and M be a left-saturating maximum-cardinality matching. If there exists an edge $(X, w) \in V_L \times V_R$ in the residual graph $R(G)$ that is not part of the matching and begins and ends in* different *strongly connected components, then the set of left vertices H reachable from w forms a Hall set.*

Proof. First X must be matched with another vertex v, since M is left-saturating. Hence, w cannot be unmatched, otherwise there would be a cycle in $R(G)$ $(X, w), (w, t), (t, v), (v, X)$ contradicting X and w being in different SCCs. In fact, t cannot be reachable at all from w for this same reason.

So w is matched with a vertex $Y \in V_L$, i.e. there is at least one vertex in H. Now consider all vertices D reachable from w in $R(G)$. Since t is not reachable, aside from $w \in N(Y)$ itself, these can only be reached via some unmatched edge from a vertex in H, so $Z \subseteq N(H)$. On the other hand, for any $u \in N(H)$ there exists an edge (K, u) for $K \in H$ and if this is not in M then u is reachable from w via K, and if it is in M, then u is reachable from w as K must have been

reached through u. So $N(H) \subseteq Z$ and hence $N(H) = Z$.

Finally, note that since every vertex in $H \cup Z$ is matched, there is a perfect matching in the corresponding induced subgraph, which is only possible if $|H| = |Z| = |N(Z)|$. Hence, H is a Hall set. □

The justification procedure is then very similar to the conflict justifications using Hall violators. The same Hall set can justify multiple deletions, and so to save space in the proof, the summing argument can be formulated so that it only needs to be done once for each Hall set. So we will assume this time that we have a procedure FindHallSet based on Lemma 3.6, and also GetDeletionEdges that return the set of deletion edges $D \subseteq \mathcal{X} \times \mathcal{D}$ associated with the hall set. We will also reuse the subprocedure SumAtLeastAtMost from Justification Procedure 3.16.

Justification Procedure 3.17 (All-Different propagation).
Preconditions: $\mathcal{X} := \{X_1, \ldots, X_n\}$; AllDifferent$(\mathcal{X})$ *encoded as per Encoding Procedure 3.9;*
$\mathcal{D} := \bigcup_i \mathrm{dom}_t(X_i)$*; and* $M \subseteq \mathcal{X} \times \mathcal{D}$ *is a left-saturating maximum-cardinality matching*
Procedure:

```
1: H ← FindHallSet(M)
2: E ← GetDeletionEdges(H, dom_t)
3: R, D ← SumAtLeastAtMost(H)
4: for (X_i, v) ∈ E
5:     rup R ⇒ x̄_{i=v} ≥ 1
```

Correctness Proof. The same arguments from the proof of Justification Procedure 3.16 apply to tell us we can derive the at-least-one and at-most-one constraints $\{L_i\}$ and $\{M_v\}$. Note that in this case these again contain almost exactly opposite literals, except for each $(X_k, d) \in E$, the literals $x_{k=d}$ appear in M_d but not in any L_i, since by assumption $X_k \notin H$. So, this time the result of $(\sum_i L_i + \sum_v M_v)\restriction_{\mathcal{R}}$ will be

$$\sum_{(X_k,d)\in D} -x_{k=d} \geq |H| - |N(H)| = 0. \tag{3.71}$$

Since each $\overline{x}_{k=d}$ is RUP with respect to this, Theorem 2.6 tells us that each $\mathcal{R} \Rightarrow \overline{x}_{k=d}$ will be RUP once we have computed $(\sum_i L_i + \sum_v M_v)$. □

It is an immediate corollary of this and Lemma 3.6 that we can efficiently justify any domain-consistent deletion for the AllDifferent constraint by first computing the SCCs of the residual graphs, and then the Hall sets, and then identifying which deletions are associated with each Hall set.

3.4.5 Justifying Reified Constraints

Finally, it is worth noting that if we know how to derive conflict and propagation justifications for a particular constraint, then we are also able to derive justifications for the half-reified version of that constraint.

If C is a constraint on variables $X_1, \ldots, X_n$ then we can denote by $Y \Rightarrow \mathsf{C}$ the half-reified version of C. This means $\mathrm{scp}(\mathsf{C}) = Y, X_1, \ldots, X_n$ and $Y \Rightarrow \mathsf{C}$ enforces that $Y \in \{0, 1\}$, along with C conditionally if $Y = 1$.

If we know how to encode C with a PB formula E, we can encode $Y \Rightarrow \mathsf{C}$ using $E' := \{y_{=1} \Rightarrow C : C \in E\} \cup \{y_{=0} + y_{=1} \geq 1\}$. Furthermore, a propagator for $Y \Rightarrow \mathsf{C}$ can easily be implemented using a propagator for C as follows:

1. If $\mathrm{dom}_t(Y) = \{1\}$ propagate C as normal.
2. If $\mathrm{dom}_t(Y) = \{0\}$ do nothing (i.e. disable the propagator until backtrack).
3. If $\mathrm{dom}_t(Y) = \{0, 1\}$ and C is infeasible, remove 1 from the domain of Y.

Justifying this is very straightforward. Suppose we are in the first or third case. Then $E'{\restriction_{\{y_{=1}\}}}$ must contain E, so for any conflict justification $\mathcal{R} \Rightarrow 0 \geq 1$ that we can derive from E, Theorem 2.1 tells us we can derive $\mathcal{R} \wedge y_{=1} \Rightarrow 0 \geq 1$ from E'. Likewise for any propagation justification $\mathcal{R} \Rightarrow \ell \geq 1$ we can certainly derive $\mathcal{R} \wedge \overline{y}_{=1} \Rightarrow \ell \geq 1$.

3.5 Summary

This chapter, similarly to the previous chapter, has been simultaneously a review of background material and the presentation of some novel contributions. We have reviewed some fundamentals of constraint programming, and then dedicated a significant amount of space to deconstructing in detail the ideas underpinning the "Auditable" *Glasgow Constraint Solver* (GCS). In particular, we have specified precisely for the first time what is required from a PB representation of a CP problem in order to be usable for a *VeriPB* proof logging approach. We have also brought the concepts of atomic literals, and defined domains, familiar from the area of lazy clause generation into the proof logging context, and proved the important properties describing how these literals behave in conjunction with binary variable encodings in the context of the PB proof framework. Finally, we have standardised a notation for presenting justification procedures, and explained how proof logging can be added to a number of fundamental constraint propagators. In some cases, this is the first place outside the source code of the GCS where this has been written down explicitly, and in all cases it is the first time the procedures have been proved correct on paper. The work is not purely descriptive: it has led to concrete redesigns within the Glasgow Constraint Solver and informed its continued development. As a result, the current GCS implementation should be understood as distinct from the system originally presented by Gocht et al. [94].

It is worth emphasising that understanding the proof system and logging framework in this level of detail is not necessarily required for a solver author to implement pseudo-Boolean proof logging. There has been a great deal of effort made to set up definitions and properties very carefully, so that we can prove that justification procedures will always work based on this. But from an implementation perspective, once a justification procedure is designed, the process of adding proof logging can be reduced to little-more than adding template-based print statements to propagation algorithms.

Part II

Generally Applicable Justification Procedures

Chapter 4

Smart Extensional Constraints

In the previous chapter we evidenced a number of fundamental constraints that can be justified within a pseudo-Boolean proof logging framework using specialised derivation procedures. But designing these procedures can require some measure of creativity, as well as knowledge of the proof system and its "tricks". If proof logging is to be widely applicable for a range of CP solvers, it would be useful to have some generic methods for adding proof logging to propagation algorithms.

As a basic starting point, we know that every constraint can theoretically be represented extensionally as a collection of allowed tuples and hence by a Table constraint.

Example 4.1 *Let* $\mathrm{dom}(X) = \mathrm{dom}(Y) = \mathrm{dom}(Z) = \{1, 2, 3\}$ *and consider the constraint* $X - Y + Z = 2$*. The corresponding table is*

X	Y	Z
1	1	2
1	2	3
2	1	1
2	2	2
2	3	3
3	2	1
3	3	2

If we are happy to trust this tabulation, we can simply use this to represent the constraint in the PB encoding using Encoding Procedure 3.10, and then all propagation and conflict justifications follow by RUP as per Justification Procedure 3.3. Of course, the tabulation algorithm may itself be prone to bugs, and so we might want to derive the table in the proof from a more trusted encoding. This can be achieved with an *auto-tabulation proof* as a pre-processing step [94]. But

a more fundamental problem is that for some constraints, the tabulation can be exponentially large.

Take for example the *Lexicographical ordering* constraint

$$\mathsf{LexGreater}(X_1, \ldots, X_n, Y_1, \ldots, Y_n),$$

which says that the sequence of values taken by $X_1, \ldots, X_n$ comes lexicographically after the sequence taken by $Y_1, \ldots, Y_n$. If all these variables have d values in their domains, there would be $(d^{2n} - d^n)/2$ solutions to this constraint, making writing out a full table intractable.

We can, however, write a LexGreater compactly using a decomposition into $(n^2 + n)/2$ simple binary constraints:

$$\begin{aligned}
&(X_1 > Y_1) \\
\vee&(X_1 = Y_1 \wedge X_2 > Y_2\ ) \\
\vee&(X_1 = Y_1 \wedge X_2 = Y_2 \wedge \quad X_3 > Y_3\ ) \\
&\vdots \\
\vee&(X_1 = Y_1 \wedge X_2 = Y_2\ \wedge \ldots \qquad \ldots \wedge X_n > Y_n)
\end{aligned} \tag{4.1}$$

It turns out that this is a logical combination of constraints that can be efficiently propagated to maintain domain-consistency, by considering it to be a *smart table* — essentially a Table constraint where the entries are allowed to be simple constraints rather than just values. If we've seen how to justify all the constituent constraint types, it is a natural question whether this somehow gives us a way of justifying consequences of some logical combination, and hence a way of justifying any domain-consistent inference for the LexGreater constraint.

In this chapter we will first consider SmartTable as a global constraint in its own right, showing that we can justify any domain-consistent inference for a restricted set of entry constraints. From this we settle the open question of whether efficient PB proof logging is possible for a number of global constraints for which this was previously unknown, including LexGreater, NotAllEqual, and AtMostOne.

We then generalise the procedure to arbitrary entry constraints, i.e. general disjunctions of conjunctions with certain restrictions on constraint scopes, expanding even further the range of constraint types that can in principle be justified efficiently.

Finally, we briefly discuss the implementation of a proof logging SmartTable propagator, and present some experimental evaluation of this.

4.1 Smart Table Constraints

A *smart table* constraint generalises the idea of a Table constraint to allow wildcards and comparisons, allowing for compact representation of a much larger set of relations, while still retaining an efficient domain-consistent propagator [144, 194, 195]. There are several ways to define a global constraint SmartTable (also called HybridTable), depending on which restriction types are allowed within tuples [34]. To begin with, we restrict our focus to unary comparisons, binary comparisons, and set membership, and define SmartTable based on this.

Let $\mathcal{X}$ be a sequence of finite-domain variables $X_1, \ldots, X_n$. A *smart entry* constraint is a unary or binary constraint in one of three possible forms:

1. <var> <op> a
2. <var> $\in S$ or <var> $\notin S$
3. <var> <op> <var>

where a is an integer constant, S is a set of integer constants, <op> denotes an operator in the set $\{<, \leq, =, \neq, \geq, >\}$, and <var> denotes a variable in $\mathcal{X}$. A *smart tuple* is a set of these *smart entry* constraints, and a *smart table* is a set of smart tuples. When a variable does not appear in a given tuple, it is implicitly unrestricted (equivalent to a wildcard entry).

For a given smart table T, where the scope of every smart entry constraint is a subset of $\mathcal{X}$, a smart table constraint $\mathsf{SmartTable}(\mathcal{X}, T)$ requires that there is at least one smart tuple in T where every smart entry constraint holds. It can hence be thought of as a special case of a disjunction of conjunctions of simple constraints.

4.1.1 Encoding Smart Table Constraints

As we have seen, our first task in adding pseudo-Boolean proof logging to a propagator is to establish a trustworthy encoding procedure. In particular, we need a linearisation of SmartTable that fits into the framework of Section 3.2.

Let T be a smart table with smart tuples $\sigma_1, \ldots, \sigma_k$. The encoding idea is similar to how we dealt with basic table constraints (Encoding Procedure 3.10). For each smart tuple σ_i in the table we will have a selector PB variable s_i that controls whether the tuple is active (i.e. not yet infeasible). Then for each smart entry $\mathsf{C}_j \in \sigma$, we will have a PB variable e_{C_j} that is set to 1 if and only if the constraint C_j is satisfied.

Since smart entries are themselves simple binary and unary constraints, all with PB encodings that we have already demonstrated in Section 3.2, the process of correctly constraining this e_{C_j} variable is straightforward. We can simply *reify* part of the encoding of the constraint and the encoding of its negation, making use of newly-defined auxiliary variables when necessary. This is easiest to see via an example.

Example 4.2 *Suppose we have a smart table constraint on three variables, X, Y, Z, all with domains $\{1, 2, 3\}$, defined by the following table T with two tuples:*

$$T := \{\sigma_1, \sigma_2\}; \qquad \sigma_1 = \{(X < Y), (X \in \{1, 2\}), (Z = 3)\}, \tag{4.2}$$
$$\sigma_2 = \{(X = Y), (X \neq 1), (Y \geq Z)\}. \tag{4.3}$$

The first entry constraint $X < Y$ can be encoded with the single constraint

$$-\mathsf{BinEnc}(X) + \mathsf{BinEnc}(Y) \geq 1 \tag{4.4}$$

and hence both directions of the entry flag can be constrained with

$$e_{X<Y} \Leftrightarrow -\mathsf{BinEnc}(X) + \mathsf{BinEnc}(Y) \geq -1. \tag{4.5}$$

The situation is similar for $Y \geq Z$. For the $\in$ and $\notin$ entries it is simply a case of imposing conjunction constraints on flags disallowing the missing values, similar to how the generic reasons (3.45) in justifications are represented, i.e.,

$$e_{X\in\{1,2\}} \Rightarrow \overline{x}_{=3} \tag{4.6}$$
$$\overline{e}_{X\in\{1,2\}} \Rightarrow x_{\geq 3} \tag{4.7}$$

Finally, for $=$ and $\neq$ we can make direct use of the $x_{=v}$ literal if the right-hand side is a value, as is done for Table *constraints, so $e_{X\neq 1}$ is replaced with $\overline{x}_{=1}$. If the right-hand side is instead another variable we make use of further intermediate flags to express the relation in terms of strict or non-strict inequalities, so:*

$$e_{X=Y} \Rightarrow e_{X\geq Y} + \overline{e}_{X\geq Y+1} \geq 2 \tag{4.8}$$
$$\overline{e}_{X=Y} \Rightarrow e_{X\geq Y} + \overline{e}_{X\geq Y+1} \geq 1 \tag{4.9}$$

where the $\geq$ flags are also defined similarly to (4.5).
This allows the SmartTable *with tuples T to be encoded as*

$$s_1 \Leftrightarrow e_{X<Y} + e_{X\in\{1,2\}} + z_{=3} \geq 3 \tag{4.10}$$
$$s_2 \Leftrightarrow \quad e_{X=Y} + \overline{x}_{=1} + e_{Y\geq Z} \geq 3. \tag{4.11}$$

For completeness, the full SmartTable encoding procedure can be written as follows, using an encoding subprocedure EntryEnc as defined in Encoding Procedure 4.2.

Encoding Procedure 4.1 (Smart Table). *A linearisation for the* SmartTable *constraint with tuples T is given by the following procedure.*

allEntryFlags $\leftarrow \emptyset$
for $\sigma_i \in T$:
  flagsForTuple $\leftarrow \emptyset$
  for C $\in \sigma_i$:
    flagsForTuple $\leftarrow$ flagsForTuple $\cup$ {EntryEnc(C, allEntryFlags)}
  def $s_i \Leftrightarrow \sum_{\ell \in \text{flagsForTuple}} \ell \geq |\sigma_i|$
def $s_1 + s_2 + \cdots + s_{|T|} \geq 1$

Auxiliary variables are all 0-1*, and are given by* allEntryFlags*, along with* $s_1, \ldots, s_{|T|}$.

Encoding Procedure 4.2 (Entries for a Smart Table).

proc EntryEnc(C, allEntryFlags)
  if $e_{\mathsf{C}} \in$ allEntryFlags
    return e_{C}
  allEntryFlags $\leftarrow$ allEntryFlags $\cup \{e_{\mathsf{C}}\}$
  switch C
    case $X_i \geq X_j$
      def $e_{X_i \geq X_j} \Leftrightarrow X_i - X_j \geq 0$
    case $X_i < X_j$
      $e_{X_i \geq X_j} \leftarrow$ EntryEnc$(X_i \geq X_j,$ allEntryFlags$)$
      return $\overline{e}_{X_i \geq X_j}$
    case $X_i = X_j$
      $e_{X_i \geq X_j} \leftarrow$ EntryEnc$(X_i \geq X_j,$ allEntryFlags$)$
      $e_{X_j \geq X_i} \leftarrow$ EntryEnc$(X_j \geq X_i,$ allEntryFlags$)$
      def $e_{X=Y} \Leftrightarrow e_{X_i \geq X_j} + e_{X_j \geq X_i} \geq 2$
    case $X_i \neq X_j$
      $e_{X_i = X_j} \leftarrow$ EntryEnc$(X_i = X_j,$ allEntryFlags$)$
      return $\overline{e}_{X_i = X_j}$
    case $X_i \in D$
      $\mathcal{L} \leftarrow$ GenericR(X_i, D)
      def $e_{\mathsf{C}} \Leftrightarrow \sum_{\ell \in \mathcal{L}} \ell \geq |\mathcal{L}|$
      return e_{C}
    case $X_i \notin D$
      $\mathcal{L} \leftarrow$ GenericR$(X_i, \text{dom}_0(X_i) \setminus D)$
      def $e_{\mathsf{C}} \Leftrightarrow \sum_{\ell \in \mathcal{L}} \ell \geq |\mathcal{L}|$
      return e_{C}
    case else
      return AtomicLiteralFor(C)

4.1.2 Justifying Smart Table Propagation

As mentioned, the SmartTable constraint is a special case of a disjunction of conjunctions of constraints, but it is worth considering in its own right first because it has efficient domain-consistent propagators with low polynomial overheads [194, 195].

Variable-value assignments have no support if they have no support in any of the tuples. Taking again the conjunctions (4.2) from Example 4.2, it is easy to see by inspection that the assignment literal $Y = 1$ does not have any support on this constraint as it would contradict $X < Y$ on the first tuple and $X = Y, X \neq 1$ together on the second. Therefore, a propagation algorithm that achieves domain consistency should immediately remove the value 1 from the domain of Y.

The original propagation algorithm described by Mairy et al. [144] uses a "Simple Tabular Reduction" strategy to eliminate such unsupported values. Essentially, it initialises a set sl (for "supportless") with every variable value/pair that is possible given the current domains of variables, and then iterates through the tuples, removing any values that they in turn support.

The literals that are left in sl are not supported by any tuple and hence the ones that should be eliminated. A key observation is that each smart tuple σ, as a conjunction of simple constraints, can be thought of as a small CSP P_σ in its own right, and so the supported assignments for a tuple are precisely those appearing in solutions to this problem. Mairy et al. show that as long as the constraint graph of P_σ (where variables are nodes, and constraint scopes are edges) is *acyclic* it can be effectively filtered as a collection of tree CSPs via a two pass filtering process. Due to the restricted structure of the smart tuples, the result of this is a collection of copies of the domains where the only values present are those that appear in some solution (global consistency for P_σ), and so this can be used to remove values from sl in polynomial time.

For proof logging, of course, the main concern is how to *justify* the elimination of these unsupported values. In particular, for a suitable reason $\mathcal{R}$ and newly derived unsupported assignment $X \neq v$, we would like to log the PB constraint

$$\mathcal{R} \Rightarrow \bar{x}_{=v} \geq 1. \tag{4.12}$$

In some situations this might follow directly by reverse unit propagation, as is always the case for proof-logging the classical table constraint as shown in Justification Procedure 3.3.

For smart tables, however, we can show that more work is required in general. Going back to the previous example and following Encoding Procedure 4.1, we note that if we try to derive $\overline{y}_{=1} \geq 1$ by reverse unit propagation, we do not reach a contradiction after propagating $y_{=1}$. This is despite $Y = 1$ lacking support in the table. To see why this is the case, note that the only way unit propagation of a non-auxiliary literal such as $y_{=1}$ could falsify the whole model would be for it to eventually result in all the selectors s_i being falsified. In turn, the only way that could happen is for there to be at least one e_{C} with $\mathsf{C} \in \sigma_i$ falsified for every $\sigma_i \in T$. But this does

not happen on propagation of $y_{=1}$ alone. The smart entry constraints that are responsible for eliminating $(Y = 1)$'s support are $X < Y$ in the first tuple and both $X = Y, X \neq 1$ in the second. From Encoding Procedure 4.1, the first of these is present in the PB model in the form of two constraints:

$$e_{X<Y} \Rightarrow \quad y_0 + 2y_1 - x_0 - 2x_1 \geq 1; \tag{4.13}$$

$$\overline{e}_{X<Y} \Rightarrow -y_0 - 2y_1 + x_0 + 2x_1 \geq 0. \tag{4.14}$$

Assuming the equality literals are correctly defined, we know that propagating $y_{=1}$ should propagate y_0 and $\overline{y}_1$, meaning (4.13) is reduced to

$$e_{X<Y} \Rightarrow 1 - x_0 - 2x_1 \geq 1. \tag{4.15}$$

Now we might hope that $\overline{e}_{X<Y}$ would then propagate, as we know $x_0 + 2x_1$ is at least 1 due to the domain constraints of the variable X and so the right-hand side of the implication in (4.15) must be false. However, because the falsity does not follow from a contradiction intrinsic to the PB constraint itself — in isolation it can be satisfied by having both x_0 and x_1 equal to 0 — unit propagation alone is not strong enough to determine that $\overline{e}_{X<Y}$ must follow. A similar problem holds for the other tuple, where the contradiction arises due to two constraints being incompatible rather than just one being falsified.

Fortunately, we can ensure that the desired constraints do follow by RUP by first explicitly deriving some intermediate proof steps. For a suitable reason $\mathcal{R}$ and newly derived unsupported assignment $X \neq v$, instead of simply logging the PB constraint (4.12) as above, we first log

$$\textbf{rup} \;\; \mathcal{R} \Rightarrow \bar{x}_{=v} + \bar{s}_k \geq 1 \tag{4.16}$$

for each $k \in \{1, \ldots, |T|\}$, i.e. first show by RUP that no tuple selector can be set to true without contradiction, and then derive from these the desired constraint (4.12). So in our running example, we would log

$$\textbf{rup} \;\; \mathcal{R} \Rightarrow \overline{y}_{=1} + \overline{s}_1 \geq 1; \qquad \textbf{rup} \;\; \mathcal{R} \Rightarrow \overline{y}_{=1} + \overline{s}_2 \geq 1; \qquad \textbf{rup} \;\; \mathcal{R} \Rightarrow \overline{y}_{=1} \geq 1. \tag{4.17}$$

This is now sufficient for this and similar examples. However, it is still possible to construct situations where even these constraints in the form of (4.16) do not follow by RUP.

Example 4.3 *Suppose we have four variables W, X, Y, Z, all with domain $\{-2, -1, 0\}$, and then a single smart tuple of the form*

$$(W < X), (X < Y), (Y < Z). \tag{4.18}$$

Clearly this is unsatisfiable, and so in particular $W = -2$ should be unsupported by the tuple. But the negation of the constraint

$$\overline{w}_{=-2} + \overline{s}_1 \geq 1 \tag{4.19}$$

does not unit propagate to contradiction. This can be seen by considering the PB encoding of the first smart entry constraint,

$$e_{W<X} \Leftrightarrow -2x_{neg} + x_0 + 2w_{neg} - w_0 \geq 1. \tag{4.20}$$

The negation of (4.19) will lead to $e_{W<X}, w_{neg}$, and $\overline{w}_0$ being propagated, which would mean the two PB constraints encoding both implication directions in (4.20) are reduced to

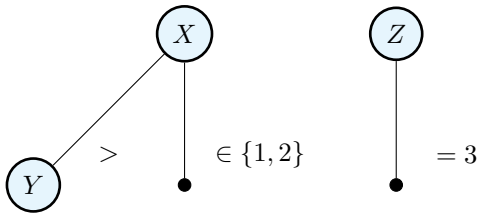
$$-2x_{neg} + x_0 \geq -1; \qquad \textit{and} \qquad 2x_{neg} - x_0 \geq -1. \tag{4.21}$$

But then no further propagation results from these constraints, which can be seen either by calculating the slack values, or simply by observing that the value tuples $(0,0)$, $(1,1)$, $(0,1)$ would each satisfy both constraints if assigned to $\{x_{neg}, x_0\}$ and so both 0 and 1 are supported possibilities for each variable. None of the other constraints contain bit variables for W and so no propagation will result from these either, and hence no contradiction will be reached.

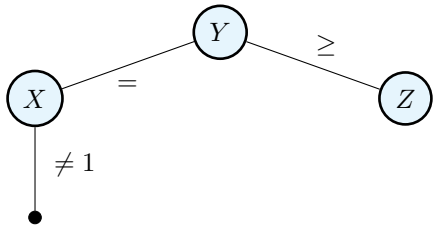
This weak propagation is not surprising given what we have seen in Chapter 3, where binary encodings frequently fail to automatically propagate what we want even for very "obvious" inferences. Fortunately in this case, the missing intermediate inference steps are being made in the solver as part of the smart table propagator anyway at the tree filtering stage, and so the desired constraints in the form of (4.16) and then (4.12) can still be logged providing we have first logged the inferences for binary constraints explicitly. The only difference in justifying these and justifying simple equality and inequality constraints as in Chapter 3 is that each inference will be conditional on the tuple selector. Or by another way of thinking, the tuple selector becomes part of the "reason" in the justification process. For example, if a variable Y has domain with maximum value u and the filtering process for $X < Y$ updates the domain copy for X so that $X < u$, then we would write

$$\textbf{rup} \;\; \overline{y}_{\geq u+1} \wedge s_1 \Rightarrow \overline{x}_{\geq u+1} \geq 1. \tag{4.22}$$

This is now enough for a complete proof logging procedure. We know from Justification Procedure 3.1; Justification Procedure 3.2; and Justification Procedure 3.12 that propagation justifications for all the by simple binary constraints such as (4.22) will always be RUP, providing they are logged with suitable reasons.



(a) The graph for P_{σ_1}, which consists of two trees.



(b) The graph for P_{σ_2}, which consists of a single tree.

Figure 4.1: The constraint graphs for the two sub-CSPs P_{σ_1} and P_{σ_2} represented by the smart tuples σ_1 and σ_2. Both graphs are acyclic and thus composed of trees.

Then, any unsupported value must be unsupported in every tuple, and hence removed in the filtering process for some smart entry constraint. So if we take as the overall reason $\mathcal{R}$ either the current sequence of decisions or the generic reason over all the variables in the SmartTable, the negation of the removal must eventually propagate the negation of each selector s_i and hence reach contradiction. We will formalise this argument within a more general justification procedure in Section 4.2.

4.1.3 A Worked Example for Smart Table Propagation

First, let us consider a worked example for justifying an inference from the basic SmartTable propagator at the top of search.

We will continue to use the running example from Example 4.2. As required for the efficient domain-consistent propagator, these tuples here do indeed form acyclic graphs and so the procedure is applicable. This can be seen in Figure 4.1, where unary constraints are represented as edges to a dummy node. The first tuple is made up of two small trees, and the second consists of a single tree.

Initially all variables have $\{1, 2, 3\}$ in their domains and the set of unsupported values sl is

$$\mathsf{sl} = \{X : 1, 2, 3; \quad Y : 1, 2, 3; \quad Z : 1, 2, 3\}. \tag{4.23}$$

To find the set of values supported by P_{σ_1} we find the values supported by each tree comprising the constraint graph. This involves two filtering passes over copies of the domain state, working first from the root outwards, and then back again.

Taking the left tree first, on the first pass 3 is removed from the domain copy for X and 1 from the domain of Y due to $X < Y$. So we would log

$$\textbf{rup} \;\; s_1 \wedge \overline{y}_{\geq 4} \Rightarrow \overline{x}_{\geq 3} \geq 1; \qquad\qquad \textbf{rup} \;\; s_1 \wedge x_{\geq 1} \Rightarrow y_{\geq 2} \geq 1; \tag{4.24}$$

as these inferences were made due to a binary constraint. No further inferences are made on the second pass. This leaves the values

$$\{X : 1, 2; \quad Y : 2, 3\} \tag{4.25}$$

supported by the first tree, and so these are removed from sl. Filtering the other tree simply reduces the domain copy for Z to $\{3\}$, and so $Z = 3$ is removed from sl. No inferences need to be logged here as this is a unary constraint. We now have

$$\mathsf{sl} = \{X : 3; \quad Y : 1; \quad Z : 1, 2\}. \tag{4.26}$$

Moving on to the second tuple, no inference occurs on the first pass due to $X = Y$ nor $Y \geq Z$, but 1 is removed from the (fresh) domain copy for X due to the unary $X \neq 1$ entry. Then on the second pass, 1 is now removed from the domain copy for Y due to $X = Y$, and so we log

$$s_2 \wedge \overline{x}_{=1} \Rightarrow \overline{y}_{=1} \geq 1; \tag{4.27}$$

and then no further inference occurs. So the values occurring in some solution for this tree are

$$\{X : 2, 3; \quad Y : 2, 3; \quad Z : 1, 2, 3\}, \tag{4.28}$$

and hence these can be removed from sl, leaving

$$\mathsf{sl} = \{Y : 1\}. \tag{4.29}$$

Ultimately, $Y = 1$ is the only assignment lacking support in the table after this initial execution of the domain-consistent propagator and the only value to be removed from an actual variable domain. We are then able to log by RUP the constraints from (4.17), which with expanded reasons are

$$\textbf{rup} \;\; x_{\geq 1} \wedge \overline{x}_{\geq 4} \wedge y_{\geq 1} \wedge \overline{y}_{\geq 4} \wedge z_{\geq 1} \wedge \overline{z}_{\geq 4} \wedge s_1 \Rightarrow \overline{y}_{=1} \geq 1; \tag{4.30}$$

$$\textbf{rup} \;\; x_{\geq 1} \wedge \overline{x}_{\geq 4} \wedge y_{\geq 1} \wedge \overline{y}_{\geq 4} \wedge z_{\geq 1} \wedge \overline{z}_{\geq 4} \wedge s_2 \Rightarrow \overline{y}_{=1} \geq 1; \tag{4.31}$$

$$\tag{4.32}$$

This allows us to successfully derive the propagation justification

$$\textbf{rup}\ \ x_{\geq 1} \wedge \overline{x}_{\geq 4} \wedge y_{\geq 1} \wedge \overline{y}_{\geq 4} \wedge z_{\geq 1} \wedge \overline{z}_{\geq 4} \Rightarrow \overline{y}_{=1} \geq 1. \tag{4.33}$$

4.1.4 To Which Constraints Does This Apply?

As well as being a potentially useful constraint in itself, having a certifying propagator for SmartTable immediately expands the set of constraints that can be supported in a proof logging constraint solver. For any constraint that has a valid SmartTable decomposition we can always install a SmartTable propagator and use it for both propagation and proofs. We also have the option of running the SmartTable propagator purely as a proof procedure for domain-consistent inferences discovered by a different algorithm.

This allows us to argue that, for example, the LexGreater constraint has an efficient PB proof procedure, by relying on the smart table decomposition (4.1) given in the introduction to this chapter. Related constraints such as LexGreaterEqual, LexLess etc. are of course covered by analogous decompositions.

We also have the following compact smart table representations.

Encoding Procedure 4.3 (At most one).

Definition: $\mathsf{AtMostOne}(X_1, \ldots, X_n, Y)$:= *At most one of the variables* $X_1, \ldots, X_n$ *takes a value equal to the value taken by* Y.

Smart Table Decomposition:

$$\bigvee_{i=1}^{n} \left(\bigwedge_{\substack{j \in \{1,\ldots,n\} \\ j \neq i}} (X_j \neq Y) \right) \tag{4.34}$$

Encoding Procedure 4.4 (Not all equal).

Definition: $\mathsf{NotAllEqual}(X_1, \ldots, X_n)$:= *At least one variable in* $X_1, \ldots, X_n$ *takes a different value to the rest.*

Smart Table Decomposition:

$$(X_2 \neq X_1) \vee (X_3 \neq X_1) \vee \ldots (X_n \neq X_1) \tag{4.35}$$

Encoding Procedure 4.5 (Value precede).

Definition: ValuePrecede$(X_1, \ldots, X_n; s, t)$:= *For fixed integers s and t, if $X_j = t$ for some $j \in \{1, \ldots, n\}$ then $X_i = s$ for some $i < j$.*

Smart Table Decomposition:

$$\bigvee_{i=1}^{n} \tau_i \tag{4.36}$$

where τ_i is given by

$$(X_1 = s); \tag{4.37}$$

τ_i with $i \in \{2, \ldots, n-1\}$ is given by

$$(X_1 \neq t \wedge \cdots \wedge X_{i-1} \neq t \wedge X_i = s); \tag{4.38}$$

and τ_n is given by

$$(X_1 \neq t \wedge \ldots X_n \neq t). \tag{4.39}$$

Encoding Procedure 4.6 (Increasing).

Definition: Increasing$(X_1, \ldots, X_n)$:= *The values taken by $X_1, \ldots, X_n$ are (non-strictly) increasing.*

Smart Table Decomposition:

$$(X_1 \leq X_2) \wedge (X_2 \leq X_3) \wedge \cdots \wedge (X_{n-1} \leq X_n) \tag{4.40}$$

Analogous decompositions exist for StrictlyIncreasing*;* Decreasing*; and* StrictlyDecreasing.

Encoding Procedure 4.7 (Element).

Definition: Element$(X_1, \ldots, X_n, Y, Z)$:= *For some $i \in \{1, \ldots, n\}$, Y takes the value i and the value of X_i is equal to the value of Z.*

Smart Table Decomposition:

$$\bigvee_{i=1}^{n} (Y = i \wedge X_i = Z) \tag{4.41}$$

Encoding Procedure 4.8 (Array Max).
Definition: ArrayMax$(X_1, \ldots, X_n, Y)$:= *The value taken by* Y *is equal to the maximum among values taken by* $X_1, \ldots, X_n$.
Smart Table Decomposition:

$$\bigvee_{i=1}^{n} \tau_i \tag{4.42}$$

where τ_i *is given by*

$$(Y = X_i) \wedge \bigwedge_{\substack{j \in \{1,\ldots,n\} \\ j \neq i}} (X_j \leq X_i) \tag{4.43}$$

ArrayMin *is analogous.*

For these last two — Element and ArrayMax — we already discussed linearisations and PB proof logging procedures in Chapter 3. Interestingly, the result of applying Encoding Procedure 4.1 for Element to the smart table decomposition given here is extremely similar to what we obtain directly using Encoding Procedure 3.4. Effectively, the only difference is that the latter unifies the $y_{=i}$ variables with the tuple selector variables τ_i using the fact that the value of Y uniquely identifies each tuple. Justification Procedure 3.11 and Justification Procedure 3.10 are then directly comparable to smart-table justifications: for an unsupported value we first prove it is unsupported in each tuple, allowing the final justification to be RUP.

In contrast, the array-min/max decomposition actually yields a more compact encoding than the one specified by Encoding Procedure 3.5, and avoids having to define explicit $x_{i=j}$ for each possible value of Y. However, the smart table justification method would require more steps for each propagation here, as the derivation methods such as Justification Procedure 3.7 only require a single RUP step for each inference.

4.2 General Disjunctions of Conjunctions

We have shown by example how to augment the tree-based filtering in the Mairy et al. [144] propagator with conditional RUP justifications in order to derive justifications for SmartTable with our chosen entry constraint set. But what if we wanted to expand the set of allowable entry types, for example, allowing constraints of the form $X = Y + a$, or even arbitrary linear constraints such as $2X + 3Y \geq Z$? We know from Chapter 3 that these require more than a single RUP step to derive their justifications.

Essentially the same procedure can be extended to deal with these more general types of smart table, albeit requiring somewhat more care.

4.2.1 Propagating Disjunctions of Conjunctions

Consider allowing an arbitrary set of constraint types for our entries. In the basic SmartTable, the constraint graph of each tuple of entries is required to be acyclic. This property makes sense for binary and unary constraints, and is what allows the efficient tree-based filtering method to find all unsupported values. Constraints of arbitrary arity do not by default allow for the same guarantees. However, the algorithm of Mairy et al. [144] is, as the authors acknowledge, a special case of the method for propagating logical combinations of constraints by Bacchus and Walsh [11]. This method does work with arbitrary constraints, albeit only guaranteeing domain-consistency under certain conditions.

Specifically, if we assume we have:

- a conjunction of constraints $\bigwedge_j \mathsf{C}_j$;
- a procedure Inc for computing a set of inconsistent variable-value pairs for each C_j;
- and a given domain state dom_t;

then a procedure for computing a set of inconsistent variable-value pairs can be written as follows (c.f. [11, Table 1]).

```
1: proc GetUnsupportedByConjunction(⋀_j C_j, dom_t)
2:     unsupported ← ∅
3:     dom_c ← dom_t
4:     repeat
5:         newUnsupported ← ⋃_i Inc(C_j, dom_c)
6:         dom_c ← dom_c \ newUnsupported
7:         unsupported ← unsupported ∪ newUnsupported
8:     until newUnsupported = ∅
9:     return unsupported
```

It has been shown that this procedure achieves domain-consistency for a given conjunction if each Inc computes a complete set of inconsistent variable value pairs, and for any i, j, $|\operatorname{scp}(\mathsf{C}_i) \cap \operatorname{scp}(\mathsf{C}_j)| \leq 1$ and the constraint-scope (hyper)-graph $(\mathcal{X}, E)$ of $\bigwedge_j \mathsf{C}_j$ has an *acyclic tree decomposition* [11, Theorem 3]. A graph has an acyclic tree decomposition if there exists a tree T, such that:

- there exists a bijection $\mathit{label} : T \to E$ between the vertices of the tree and the hyper-edges (constraint scopes); and
- for each $X \in \mathcal{X}$, the set $\{t : X \in \mathit{label}(T)\}$ forms a subtree of T.

So a generalised smart table can, at least in theory, be propagated by collecting inconsistent variable-value pairs for each tuple, which should respect the above conditions, and then filtering

those which are unsupported in every tuple.

1: **proc** GetUnsupportedByDisjunctionOfConjunctions($\bigvee_j(\bigwedge_j \mathsf{C}_{ij}), \mathrm{dom}_t$)
2: **return** $\bigcap_j$ GetUnsupportedByConjunction($\bigwedge_j C_{ij}, \mathrm{dom}_t$)

This already allow us to say that we can efficiently justify any constraint that can be decomposed into a short conjunction of simpler constraints, providing we have an efficient certifying propagator for each, and the conjunction respects the required restrictions on constraint scopes. We simply run the Inc procedures iteratively until fix-point, writing propagation justifications as we go, and we are guaranteed to have covered any domain-consistent inference.

We can then go further and say that any constraint that can be expressed concisely as a *disjunction* of such decomposable constraints — in effect, having generalised smart table representation — can also be efficiently justified. This relies on us generalising the EntryEnc procedure for encodings, which can be slightly subtle, as we will see in the next subsection.

4.2.2 Justifying Disjunctions of Conjunctions

Let us view a smart table constraint as a disjunction of conjunctions of smart entry constraints $\vee_i(\wedge_j \mathsf{C}_{ij})$. If we look again at the examples from the previous section, we can ask: what was fundamentally required here for each C_{ij} to get the justification methods to work? We needed to write reified versions of each entry constraint for the encoding, as implemented by the EntryEnc procedure. Then we needed to derive justifications for any individual entry constraint, additionally reified on the selector variable. Less obviously, we needed complete propagation of entry and selector flags or contradiction to result from any assignment setting all the bit variables arising from BinEnc replacement of the original CP variables. This ensured the solution witnessing property as discussed in Section 3.3.3.

Generalising this, we can say that for each entry type C_{ij} we need to be able to produce a special linearisation ReifLin(C_{ij}, e_{ij}) using an auxiliary flag e_{ij} such that the properties E1 and E2 below hold.

E1. ReifLin(C_{ij}, e_{ij}) is a linearisation of $e_{ij} \iff \mathsf{C}_{ij}$ satisfying the properties L1 – L4 as discussed in Section 3.2.

E2. Let $\mathcal{X} := \mathrm{scp}(\mathsf{C}_{ij})$. For any valid assignment ρ to $\mathcal{X}$, let ϕ be the corresponding assignment to the bit variables in BinEnc($\mathcal{X}$). If ρ satisfies C_{ij} then ReifLin(C_{ij}, e_{ij})$\restriction_\phi$ propagates e_{ij}; and if ρ does not satisfy C_{ij} then ReifLin(C_{ij}, e_{ij})$\restriction_\phi$ propagates $\overline{e}_{ij}$.

The overall linearisation is then straightforward to express.

Encoding Procedure 4.9 *For a table T of tuples representing a disjunction of conjunctions of constraints C_{ij} for which a reified linearisation $\mathsf{ReifLin}(\mathsf{C}_{ij}, e_{ij})$ satisfying E1 is known we can produce a linearisation of the overall constraint described by T as follows.*

1: **for** $\sigma_i \in T$:
2: **for** $\mathsf{C}_{ij} \in \sigma_i$:
3: **def** $\mathsf{ReifLin}(\mathsf{C}_{ij}, e_{ij})$
4: **def** $s_i \Leftrightarrow \sum_{\mathsf{C}_{ij} \in \sigma_i} e_{ij} \geq |\sigma_i|$
5: **def** $s_1 + s_2 + \cdots + s_{|T|} \geq 1$

Property E1 ensures this is semantically correct, and a valid linearisation $\vee_i \wedge_j \mathsf{C}_{ij}$ for the purposes of proof logging. Property E2 ensures that a complete assignment to the bit variables sets all the e_{ij} variables by unit propagation, and hence sets all the s_i variables by unit propagation. This maintains the requirement needed for partial solution witnessing with equality literals.

Then, for propagation, we are assuming we have a procedure Inc for each constraint type as described in the previous section. For proof logging we need a corresponding procedure JustifyConditionally that takes an entry constraint C_{ij}, and a set of unsupported variable value pairs, U with respect to a domain state dom_c, and ensures that the following property holds.

E3. Let $\mathcal{L}$ be any set of atomic literals such that $\mathrm{dom}_{\mathcal{L}} \sqsubseteq \mathrm{dom}_c$. Then, after executing $\mathsf{JustifyConditionally}(\mathsf{C}_{ij}, U, \mathrm{dom}_c)$, sufficient constraints should have been logged in the proof so that propagation of $s_i \wedge \mathcal{L}$ either results in contradiction or in propagation of a set of literals $\mathcal{L}'$ such that $\mathcal{L}' \sqsubseteq \mathrm{dom}_c \setminus U$.

Correctness Proof for Justification Procedure 4.1. The final $\mathcal{R}$ is constructed so that propagation of each $\mathcal{R} \wedge s_i$ propagates all the literals returned by each $\mathsf{JustifyConditionally}(\dots)$. This is because $\mathrm{dom}_{\mathcal{R}} = \mathrm{dom}_t$, which is initially the same as dom_c. So we can see that, from E3, each execution of JustifyConditionally ensures that a set of literals required for the next JustifyConditionally also propagate. This in turn ensures that for each i, propagation of $\mathcal{R} \wedge s_i$ propagates a set of literals $\mathcal{L}$ that excludes all unsupported values from the defined domain state,

$$\text{i.e } \mathrm{dom}_{\mathcal{L}} \sqsubseteq \mathrm{dom}_{\emptyset} \setminus \mathsf{unsupportedByTuple}.$$

Then, since (X, v) must be unsupported in every tuple σ_i, it is guaranteed by Theorem 3.2 that each constraint from line 26 will be RUP, as $(X, v) \notin \mathrm{dom}_{\mathcal{L}}$. Likewise, if the disjunction of conjunctions is infeasible, we must propagate a set of literals describing an empty domain, and hence the constraints from line Line 21 are RUP by Theorem 3.2.

Once these are derived, in either case the final constraint is certainly RUP, since $\overline{s}_i$ will propagate for every i, contradicting the constraint from the final line of Encoding Procedure 4.9.

Justification Procedure 4.1 (General disjunctions of conjunctions).

Preconditions: *A disjunction of conjunctions of linearisable constraints* $\bigvee_j(\bigwedge_j \mathsf{C}_{ij})$*, encoded as in Encoding Procedure 4.9;* $\mathcal{X} \leftarrow \cup_{i,j} \mathrm{scp}(\mathsf{C}_{ij})$*; and for each* C_{ij} *there exists a procedure JustifyConditionally such that property E3 is satisfied.*

Procedure:

unsupported $\leftarrow \emptyset$
$\mathcal{R} \leftarrow \emptyset$
infeasible $\leftarrow$ *true*
for $\sigma_i \in T$
  unsupportedByTuple $\leftarrow \emptyset$
  newUnsupported $\leftarrow \emptyset$
  $\mathrm{dom}_c \leftarrow \mathrm{dom}_t$
  repeat
    for $\mathsf{C}_{ij} \in \sigma_i$
      newUnsupported $\leftarrow$ newUnsupported $\cup$ Inc$(\mathsf{C}_{ij}, \mathrm{dom}_c)$
      unsupportedByTuple $\leftarrow$ unsupportedByTuple $\cup$ newUnsupported
      JustifyConditionally$(\mathsf{C}_{ij}, \mathrm{dom}_c,$ newUnsupported$)$
      $\mathrm{dom}_c \leftarrow \mathrm{dom}_c \setminus$newUnsupported
  until newUnsupported $= \emptyset$
  unsupported $\leftarrow$ unsupported $\cap$ unsupportedByTuple
  if $\forall X \in \mathcal{X}, (X, \emptyset) \notin \mathrm{dom}_c$
    infeasible $\leftarrow$ *false*
$\mathcal{R} \leftarrow$ GenericR$(\mathcal{X}, \mathrm{dom}_t)$

if infeasible
  for $\sigma_i \in T$
    rup $\mathcal{R} \wedge s_i \Rightarrow 0 \geq 1$
  rup $\mathcal{R} \Rightarrow 0 \geq 1$
else
  for $(X, v) \in$ unsupported
    for $\sigma_i \in T$
      rup $\mathcal{R} \wedge s_i \Rightarrow \overline{x}_{=v} \geq 1$
    rup $\mathcal{R} \Rightarrow \overline{x}_{=v} \geq 1$

□

Obviously this justification procedure depends heavily on the properties E1, E2, and E3 in order to work. However, if the constraint is one for which we already know how to achieve unconditional proof logging, we can argue it is straightforward to devise the ReifLin and Justify-Conditionally procedures.

Devising Reified Linearisations

First, for ReifLin, the EntryEnc procedure already gives examples of how this can be achieved for a number of simple constraints. We can see by inspection that properties E1 and E2 are upheld here.

More generally, if we already know how to linearise an entry constraint C_{ij}, and it happens that $\mathsf{Lin}(\mathsf{C}_{ij})$ does not contain any auxiliary variables, then we can simply implement ReifLin as shown in Encoding Procedure 4.10.

Encoding Procedure 4.10 (Naïve conditional linearisation).

proc ReifLinBasic(C_{ij}, e_{ij})
  if $|\mathsf{Lin}(\mathsf{C}_{ij})| = 1$
    def $e_{ij} \Leftrightarrow \mathsf{Lin}(\mathsf{C}_{ij})$
  else
    for $C_k \in \mathsf{Lin}(\mathsf{C}_{ij})$
      def $e_{ij} \Rightarrow C_k$
      def $\overline{e}_{ijk} \Rightarrow \neg C_k$
    def $\overline{e}_{ij} \Leftrightarrow \sum_k \overline{e}_{ijk} \geq 1$

E1 is respected by this, since the constraints enforce every $C_k \in \mathsf{Lin}(\mathsf{C}_{ij})$ when e_{ij} is true, and at least one $\neg C_k$ where $C_k \in \mathsf{Lin}(\mathsf{C}_{ij})$ when e_{ij} is false. As a technicality, if we are reifying the negation of an already-reified constraint in the intermediate ILP format here i.e. $y \Rightarrow \neg(x \Rightarrow C)$ then we can avoid nesting by observing this is equivalent to the two constraints $y \Rightarrow x$ and $y \Rightarrow \neg C$.

If each C_k in the PB encoding resulting from this linearisation contains only bit variables or atomic variables, then every e_{ijk} and e_{ij} will be uniquely determined by unit propagation for any CP-variable assignment as required by E2. So this ReifLinBasic procedure already allows us to handle, for example, comparisons with scaling/translation e.g. $3X - 1 \geq 2$ or linear (in)equalities in the disjunction.

Unfortunately, the property E1 can be violated by ReifLinBasic when auxiliary variables are involved, as shown by the following example.

Example 4.4 *A linearisation* $\mathsf{Lin}(X \neq Y)$ *of a not-equals constraint on* X *and* Y *is given by*

$$u \Rightarrow X - Y \geq 1 \quad \textit{and} \quad \overline{u} \Rightarrow Y - X \geq 1 \tag{4.44}$$

for a fresh auxiliary variable u*. However, the result of* $\mathsf{ReifLinBasic}(X \neq Y, e)$ *based on this would be*

$$e \Rightarrow (u \Rightarrow X - Y \geq 1); \tag{4.45}$$
$$e \Rightarrow (\overline{u} \Rightarrow Y - X \geq 1); \tag{4.46}$$
$$\overline{e}_1 \Rightarrow u \geq 1 \tag{4.47}$$
$$\overline{e}_1 \Rightarrow -X + Y \geq 0 \tag{4.48}$$
$$\overline{e}_2 \Rightarrow \overline{u} \geq 1 \tag{4.49}$$
$$\overline{e}_2 \Rightarrow -Y + X \geq 0 \tag{4.50}$$
$$e \Leftrightarrow e_1 + e_2 \geq 2; \tag{4.51}$$

And if we encode this to PB, then the corresponding PB assignment for a CP assignment satisfying $X \neq Y$ *may not propagate* e*. This can be seen if we take* $X = 4$ *and* $Y = 3$*, as this reduces the formula to PB constraints equivalent to*

$$e \Rightarrow (u \Rightarrow 0 \geq 0); \tag{4.52}$$
$$e \Rightarrow (\overline{u} \Rightarrow 0 \geq 1); \tag{4.53}$$
$$\overline{e}_1 \Rightarrow u \geq 1; \tag{4.54}$$
$$\overline{e}_1 \Rightarrow 0 \geq 1; \tag{4.55}$$
$$\overline{e}_2 \Rightarrow \overline{u} \geq 1; \tag{4.56}$$
$$\overline{e}_2 \Rightarrow 0 \geq -1; \tag{4.57}$$
$$\overline{e} \Rightarrow \overline{e}_1 + \overline{e}_2 \geq 1. \tag{4.58}$$

Although e_1 *propagates here, no further propagation occurs after this, since all remaining constraints can be satisfied with assignments in which all literals appear both positively and negatively.*

We got around this problem for not-equals constraints in our original EntryEnc procedure, by using separate flags for the $X > Y$ and $Y > X$ cases, and then only reifying the constraint saying at least one of these must hold. For many of the linearisations we have presented in this thesis we can achieve a similar workaround — reify only part of the constraint to get both the correct semantics and required propagation properties. For example, for Table or even SmartTable constraints themselves, ReifLin can be achieved by reifying just the final at-least-one constraint

over the selector variables.

$$e \Leftrightarrow s_1 + \cdots + s_k \geq 1. \tag{4.59}$$

Since each s_i is guaranteed to be set by unit propagation on a complete assignment to the CP variables, e would then be propagated as required.

Another possibility would be to abandon the E2 requirement entirely, and then use ReifLinBasic for any constraint type. Recall that this was only needed for partial solution witnessing in optimisation proofs, and to pave the way for future work on enumeration proofs as discussed at the end of Section 3.3.3. This would then necessitate witnessing the correct values for all tuple selectors along with the CP variables when writing a `soli` line, e.g. `soli xeq2 yeq2 zeq1 ~s1 s2`.

Devising Conditional Justifications

If we have a reified encoding to use for a particular constraint C_{ij} type based on a linearisation for which we already know a proof logging procedure, then devising a corresponding JustifyConditionally procedure should also be straightforward. This is because we can rely on Theorem 2.1. Assuming that we have $\mathsf{ReifLin}(\mathsf{C}_{ij}, e_{ij})\restriction_{e_{ij}} \supseteq \mathsf{Lin}(\mathsf{C}_{ij})$ (as is the case for ReifLinBasic above), then we can use the fact that we know how to derive propagation justifications of the form $\mathcal{R} \Rightarrow \ell \geq 1$ to obtain a procedure for deriving $\mathcal{R} \wedge e_{ij} \Rightarrow \ell \geq 1$ as required. This is effectively what we are doing in the worked example in Section 4.1.3. All the basic smart entry types have linearisations for which we have RUP justifications, or require no justification at all. Then the constructive proof of Theorem 2.1 (case 5) tells us that making these justifications conditional on the tuple selector s_i is also a single RUP step.

Extending this to linear inequalities as smart entries would involve a single cutting planes operation followed by a RUP or syntactic implication steps. And in general, we simply need to insert syntactic implication steps where necessary into a known justification procedure to implement JustifyConditionally.

4.2.3 Further Constraints Where This Applies

Even just extending smart tables to allow linear inequalities as entries already increases the suite of global constraints that can be represented as a disjunction of conjunctions with an acyclic tree decomposition, and hence can have propagation justified by the above methods. As an example (following Mairy et al. [144]) we could justify the following.

Encoding Procedure 4.11 (Two non-overlapping orthotopes:).
Definition: TwoOrthNoOverlap$(X_1, \ldots, X_k, Y_1, \ldots, Y_k, D_1, \ldots, D_k, H_1, \ldots, H_k)$:= *The two k-dimensional boxes with origins (in $\mathbb{R}^m$) at $(X_1, ..., X_k)$ and $Y_1, \ldots, Y_k)$ and lengths given by $(D_1, \ldots, D_k)$ and $(H_1, \ldots, H_k)$ respectively do not overlap.*
Smart Table Decomposition:

$$\bigvee_{i=1}^{k} (X_i - Y_i - H_i \geq 0) \vee \bigvee_{i=1}^{k} (Y_i - X_i - D_i \geq 0) \tag{4.60}$$

We can also now justify certain constraints with decompositions involving auxiliary variables and linear inequalities, such as the following.

Encoding Procedure 4.12 (At most k in values).
Definition: AtMostInValues$(X_1, \ldots, X_n; k, \mathcal{V})$:= *At most k of the variables in $X_1, \ldots, X_k$ take a value in $\mathcal{V}$.*
Smart Table Decomposition:

$$\bigwedge_{i=1}^{n} C_i \in \{0, 1\} \wedge \bigwedge_{i=1}^{n} C_i \Leftrightarrow X_i \in \mathcal{V} \wedge \sum_{i} C_i \leq k \tag{4.61}$$

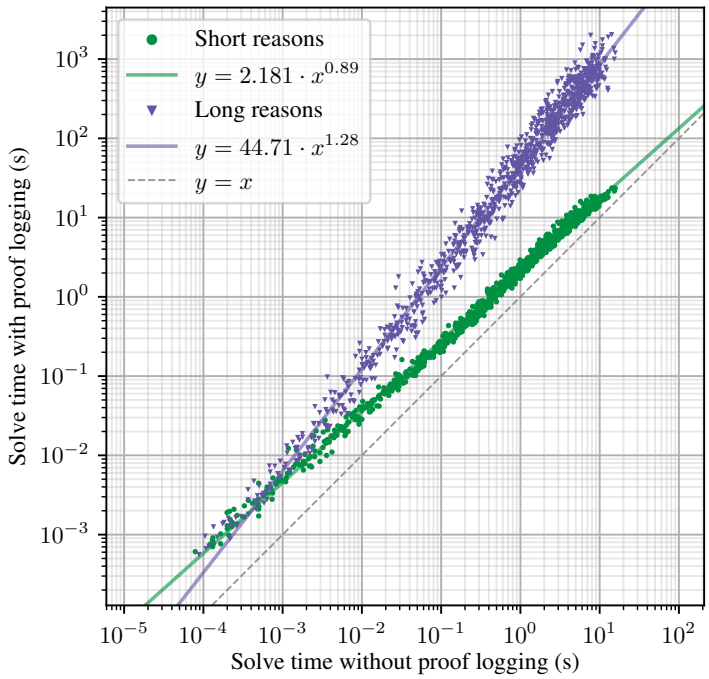
AtLeastInValues *is similar.*

4.3 Implementation and Experiments

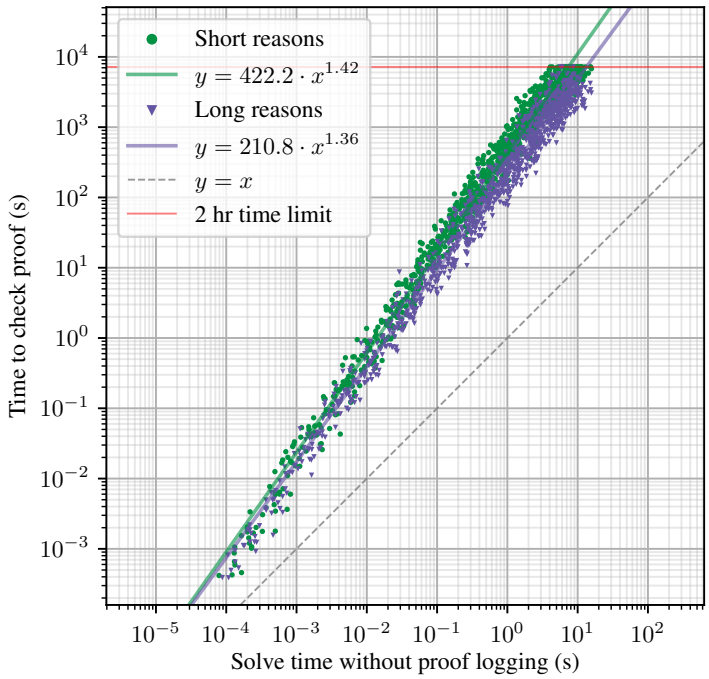
As discussed in the introduction to this thesis, one objective of the overall research project was to concretely implement the new justification procedures to show that they can indeed lead to a CP solver producing valid proofs within a feasible timeframe. However, our aim here is not to devise new constraint propagation algorithms or better implementations, nor to judge which propagation methods should be used in practice. So we will not go into many details of the software design and engineering choices here.

A prototypical implementation of the ideas in this chapter was achieved by adding a SmartTable propagator to the *Glasgow Constraint Solver* (GCS) project. This supports filtering of conjunctions of disjunctions of smart entries (of the restricted kind discussed in Section 4.1.2) with RUP justifications as exemplified in Section 4.1.3. The propagation algorithm was a straightforward implementation of the one described by Mairy et al. [144], and the justification procedures were embedded directly into the solver, in keeping with the rest of the GCS framework.

The correctness proof for Justification Procedure 4.1 gives us assurance that this should always produce correct certificates, but to validate the implementations and get rough indication

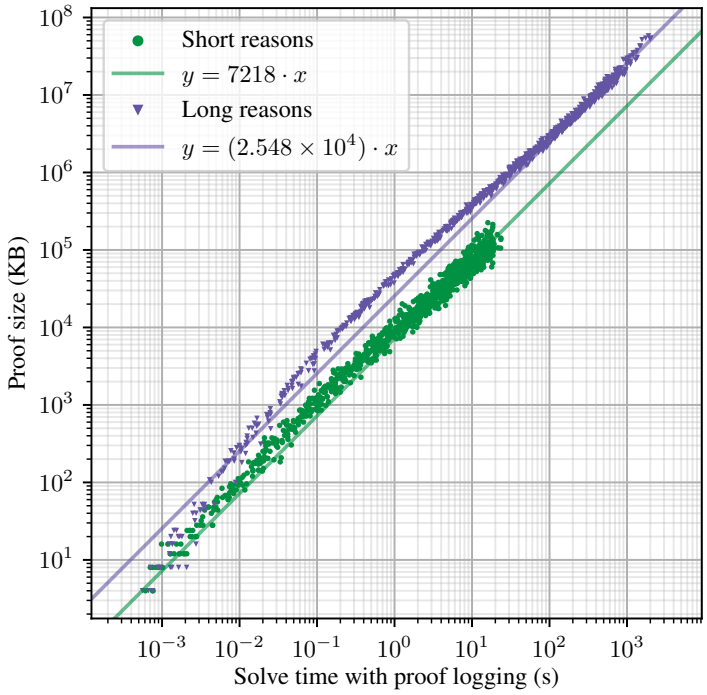


(a) Overhead of proof logging during solving.

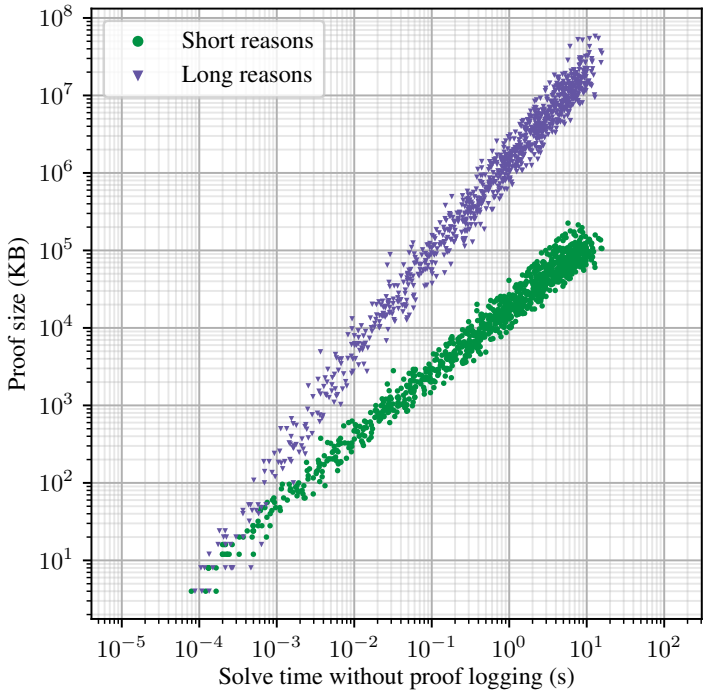


(b) Proof checking time relative to solve time.

Figure 4.2: Timing results for random SmartTable instances.



(a) Proof size relative to time to produce the proof.



(b) Proof size relative to solve time.

Figure 4.3: Proof size results for random SmartTable instances.

of overheads in practice we formulated some simple benchmarks and examples, to be run with and without proof logging. We checked the results using the 3.0 version of the *VeriPB* proof checker.

Since we are primarily interested in the overhead of proof logging for this specific propagator, we formulated (seeded) randomised instances containing of only a single smart table constraint, over increasing large domain sizes and number of variables. We generated random trees using Prüfer sequences to ensure the required acyclic property of tuples, and chose randomised tuple lengths up to number of variables. These might not be particularly interesting instances from a solving perspective, but they give a reasonable indication of overhead since we guarantee all non-search-related proof logging is entirely due to SmartTable.

One detail that is worth mentioning is a simple optimisation to proof writing that turns out to have a significant impact on logging overhead in practice. Our procedure, based on Justification Procedure 4.1, makes use of the *generic reason* for all intermediate reified RUP steps. In some cases, particularly with large numbers of variables, and large domains with many holes, the length of this reason can get quite long, and the ASCII text representation of a mathematical constraint such as

$$\mathsf{GenericR}_t \Rightarrow \overline{y}_{=3} \geq 1 \tag{4.62}$$

might look something like the following.

```
rup 1 ~i19_e17 1 ~i0_eminus1 1 ~i1_eminus1 1 ~i2_gminus1 1 ~i3_gminus1
1 i3_g21 1 ~i4_gminus1 1 i4_g21 1 ~i5_gminus1 1 i5_g21 1 i5_e10
1 ~i6_gminus1 1 i6_g21 1 i6_e18 1 ~i7_g0 1 i7_g21 1 ~i8_gminus1
1 i8_g21 1 ~i9_gminus1 1 i9_g21 1 ~i10_e1 1 ~i11_gminus1 1 i11_g21
1 ~i12_gminus1 1 i12_g20 1 ~i13_gminus1 1 i13_g21 1 ~i14_g0 1 i14_g21
1 ~i15_gminus1 1 i15_g3 1 ~i16_gminus1 1 i16_g21 1 ~i17_g0 1 i17_g21
1 ~i18_gminus1 1 i18_g21 i2_g3 >= 1 ;
```

If this is repeated many times — as can happen in the final loops of Justification Procedure 4.1 — the size of the proof and amount of data that needs to be written to disk can grow significantly.

From a correctness point of view, it makes no difference to the justification procedure if we instead introduce a new auxiliary literal *fully reifying* a reason we want to use repeatedly.

$$\textbf{ext} \;\; \ell_{\mathcal{R}} \Leftrightarrow \sum_{\ell_i \in \mathcal{R}} \ell_i \geq |\mathcal{R}|. \tag{4.63}$$

Propagation of $\ell_{\mathcal{R}}$ here propagates all the literals in $\mathcal{R}$, and so satisfies the same propagation properties that we are relying upon for $\mathcal{R}$. So we can safely use the "short reason" $\ell_{\mathcal{R}}$ in place of the "long reason" $\mathcal{R}$ throughout each execution of the justification process.

We therefore tested the random smart table instances, each without and without proof logging (with the same random seed), and with and without the short-reasons optimisation. Figures 4.2 and 4.3 show the results of this via parallel jobs on a cluster of compute nodes with dual AMD EPYC 7643 processors, 2 TBytes of RAM, and local solid state hard drives, running Ubuntu 25.10. 80 processes were run in parallel on each node, and each process (both solving and then checking) was limited to two hours of total running time and 8GB of memory.

We can see that in either case proof logging incurs a significant but proportionate overhead, with the solve time with proof logging remaining within a constant factor of the solve time without proof logging as the difficulty of the problem grows. All the produced proofs were validated by the *VeriPB* proof checker, except those which reached the time limit, which were partially verified. As can be seen in Figure 4.2, checking the proof is significantly slower than solving the original problem without proof logging, and exhibits at least linear scaling as the problem difficulty increases. This might at first seem surprising, but it is in line with other empirical results in different solving paradigms with the current version of the proof checker [123, 115]. Improvements to checking efficiency are out of scope for this work, but would likely be provided by a binary proof representation and a backwards-checking/proof-trimming implementation.

We can also see that the short reasons optimisation provides a significant benefit for logging overhead, at a cost of modest constant factor to checking time. For larger instances it decreases the size of proof and checking time by about two orders of magnitude. This reinforces the general intuition that having different constant factors for proof logging can make a very noticeable in practice, even with justification procedures that are mathematically equivalent. We would expect similar improvements to the logging overhead with further optimisations such as using shorter variable names or a proof binary format; or improved engineering of the means of writing formatted strings to disk.

4.4 Conclusions

This chapter has introduced proof logging for propagation of "smart" extensional representations of constraints. We have illustrated a RUP-based justification method for SmartTable, and shown why this works more generally for general disjunctions of conjunctions. This effectively hinged on Theorem 2.1, which allowed us to say that if we know a justification procedure for a constraint then we can construct *conditional justifications* that fit into a table-like justification framework. With the property of Bacchus and Walsh [11] for ensuring domain consistency on disjunctions of conjunctions, our methods significantly expand the set of constraints for which we can say, in principle, pseudo-Boolean proof logging is applicable.

Our empirical results also suggest that the methods are workable in practice, although clearly further engineering would be required for a production-ready solver proof-producing solver and corresponding checker. Additionally, we can provide some anecdotal evidence of the value of

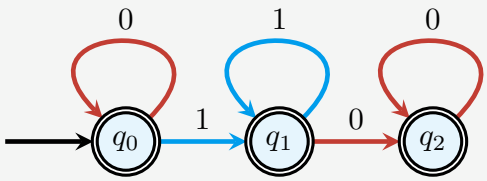
proofs for debugging during propagator development. Simple mistakes, such as getting "$>$" and "$<$" the wrong way round, or mixing up variable names, caused proofs to fail almost immediately on small instances (such as the one from Example 4.2). These are likely bugs that would have been hard to spot, since in some cases the solver was still arriving at correct solutions, but making unsound inferences along the way.

Chapter 5

Constraints Based On States and Transitions

Continuing with our aim to show that large categories of general constraint reasoning can be justified using pseudo-Boolean proofs, we now turn to propagators based on *states and transitions*. These take inspiration from automata theory: a state is just collection of quantities describing a system at a particular point in time, and a transition is a rule describing when moving from one state to another is allowed. This is a natural next step after the smart extensional constraints we dealt with in the previous chapter. Several constraints that are propagated using states and transitions such as the regular language membership constraint Regular or the *multivalued decision diagram* constraint MDD are in effect compressed extensional representations of all allowed solutions. And similar to SmartTable they can be used to represent a large variety of commonly used global constraint patterns.

Example 5.1 (Global contiguity constraint as an automaton).
A GlobalContiguity *constraint enforces that a sequence of* 0-1 *variables must have all the "*1*" values appearing contiguously.*
The sequences of values allowed by this constraint form precisely the regular language accepted by the following deterministic finite automaton.



In this chapter we will primarily focus on the regular language membership constraint. We will see that this constraint has a direct, intuitive PB representation, and if this representation is used for proofs it is possible to efficiently justify all the inferences made by the original domain-

consistency Regular propagator given by Pesant [163]. We will then observe that this very same justification mechanism requires essentially no modification to work with more general decision diagram constraints. This will lead to a brief discussion of how to adapt very similar justification techniques for constraints that are not fundamentally represented by states and transitions, but nevertheless use them as part of the propagation algorithm. This latter section is connected to a project that the author was involved with but not the main contributor and so it will be more expository, with Demirović et al. [59] referred to for more details.

Finally, as with SmartTable in the previous chapter, the ideas have been realised through an implementation of a Regular constraint in the Glasgow Constraint Solver, which we will briefly discuss.

5.1 Regular Language Membership

We can define the Regular constraint as follows. Let $\mathcal{X}$ be a sequence of finite-domain variables $X_0, \ldots, X_{n-1}$ and let $M = (Q, \Sigma, \delta, q_0, F)$ denote a deterministic finite automaton (DFA), where

- Q is a set of states;
- $q_0 \in Q$ is the initial state and $F \subseteq Q$ is the set of accepting states;
- Σ is a set of symbols having the domain of every variable in $\mathcal{X}$ as a subset; and
- $\delta : (Q \times \Sigma) \to Q$ is the transition function.

A regular language membership constraint $\mathsf{Regular}(\mathcal{X}, M)$ requires that any sequence of values taken by the variables of $\mathcal{X}$ must belong to the regular language recognised by M.

We can allow the transition function to be partial, and define any missing value/state pairs as being an implicit rejection by the DFA.

5.1.1 Encoding Regular Language Membership Constraints

Given what we have seen in previous constraint linearisations — particularly the use of auxiliary variables and reifications — a very natural encoding of the DFA underpinning the Regular constraint is apparent.

We can use $n \times |Q|$ *state flags*, which are 0-1 variables $s_{i=q}$ each representing whether the state of the DFA after processing i values is q. So if we simplify notation and identify the states by numbers $0, \ldots, m-1$, with 0 being the initial state, then we should enforce $s_{0=0} \geq 1$ to ensure the DFA begins in the initial state and $\sum_{r \in F} s_{n=r} \geq 1$ to ensure it ends in an accepting state.

This allows us to encode the transition function directly by simply writing out the relation for every variable-value pair as an inequality:

$$s_{i=q} \wedge x_{i=v} \Rightarrow s_{i+1=\delta(q,v)} \geq 1. \tag{5.1}$$

If any value-state pairs for a position i are missing from the transition function we can also disallow these in the natural way

$$s_{i=q} \wedge x_{i+1=v} \Rightarrow 0 \geq 1. \tag{5.2}$$

Finally, we require that exactly one of the state flags is true for each position i in the sequence. This results in the following linearisation.

Encoding Procedure 5.1 (Regular Language Membership). *Let $M = (Q, \Sigma, \delta, q_0, F)$ denote a deterministic finite automaton (DFA). A linearisation / PB encoding of regular language membership constraint* Regular$(\mathcal{X}, M)$*, where $\mathcal{X} = \langle X_1, \ldots, X_n \rangle$ is given by the following procedure.*

1: **for** $i \in \{0, \ldots, n\}$
2: ⌊ **def** $\sum_{q \in Q} s_{i=q} = 1$
3: allowedTransitions$(\delta) \leftarrow \{(q, v) \in Q \times \Sigma \ : \ \delta(q, v)$ is defined$\}$
4: disallowedTransitions$(\delta) \leftarrow (Q \times \Sigma) \setminus$ allowedTransitions(δ)
5: **for** $i \in \{0, \ldots, n-1\}$
6: ⌊ **for** $(q, v) \in$ allowedTransitions(δ)
7: ⌊ **def** $s_{i=q} \wedge x_{i=v} \Rightarrow s_{i+1=\delta(q,v)} \geq 1$
8: ⌊ **for** $(q, v) \in$ disallowedTransitions(δ)
9: ⌊ **def** $s_{i=q} \wedge x_{i=v} \Rightarrow 0 \geq 1$
10: **def** $s_{0=0} \geq 1$
11: **def** $\sum_{f \in F} s_{n=f} \geq 1$

Auxiliary variables are simply all the $s_{i=q}$ variables, for each $i \in \{0, \ldots, n\}$ and $q \in Q$.

Note that, aside from the at-most-one constraint as part of line 2, this encoding is almost entirely clausal. Given the fact that encoding a Regular constraint as a set of clauses for a SAT solver has already been explored extensively in several works [56, 142], we might wonder whether we should be using one of these more optimised encodings for the PB model. It would be tempting, for example, to adopt the encoding by Bacchus [10], for which (SAT-based) unit propagation alone can be used to ensure domain consistency. This is similar to the above encoding, employing Boolean variables equivalent to our $x_{i=j}$ and $s_{i=q}$ PB variables, but it uses an additional set of *transition* variables in order to correctly enforce the constraint. If we used the equivalent of this in our linearisation, every inference made by any domain-consistent propagator would require no justification at all, since Inv3 would already be satisfied by unit propagation on the encoding.

Encoding Procedure 5.2 (Regular Language Membership — Alternative). *For a regular constraint* Regular$(\mathcal{X}, M)$ *as defined for Encoding Procedure 5.1, an alternative encoding, following Bacchus [10] is as follows.*

allowedTransitions$(\delta) \leftarrow \{(q, v) \in Q \times \Sigma \ : \ \delta(q, v)$ is defined$\}$
for $i \in \{0, \dots, n-1\}$
  for $q \in Q$
    for $v \in \Sigma$ s.t. $(q, v) \in$ allowedTransitions(δ)
      def $t^i_{q\langle v\rangle} \Rightarrow x_{i=v} + s_{i=q} + s_{i+1=\delta(q,v)} \geq 3$
    if $i > 0$
      inTransitions $\leftarrow \{t^{i-1}_{p\langle v\rangle} : (p, v) \in$ allowedTransitions and $\delta(p, v) = q\}$
      def $s_{i=q} \Rightarrow \sum_{t^{i-1}_{p\langle v\rangle} \in \text{inTransitions}} t^{i-1}_{p\langle v\rangle} \geq 1$
    outTransitions $\leftarrow \{t^i_{q\langle v\rangle} : (q, v) \in$ allowedTransitions$\}$
    def $s_{i=q} \Rightarrow \sum_{t^i_{p\langle v\rangle} \in \text{outTransitions}} t^i_{p\langle v\rangle} \geq 1$
  for $v \in \Sigma$
    def $x_{i=v} \Rightarrow \sum_{q:(q,v)\in \text{allowedTransitions}} t^i_{q\langle v\rangle} \geq 1$
def $\sum_{q \in Q \setminus \{0\}} \overline{s}_{0=q} \geq |Q| - 1$
def $\sum_{i \in Q \setminus F} \overline{s}_{n=i} \geq |Q \setminus F|$

Auxiliary variables are the $s_{i=q}$ variables, for each $i \in \{1, \dots, n\}$ and $q \in Q$, along with further variables $t^i_{q\langle v\rangle}$ for each $q \in Q$, $v \in \Sigma$, and $i \in \{0, \dots, n\}$.

But once again, the aim in this work is to choose a representation for proof logging not based on propagation strength but based on how easily we can agree on the required correspondence with the constraint's definition — recall L4 from Section 3.2. We can argue that Encoding Procedure 5.1 is both more compact and better fits this aim. Moreover, if we show that inferences made by a Regular propagator can be justified without "cheating" and relying on a strong encoding, we better demonstrate the capabilities of the PB proof system to explicitly capture this kind of reasoning, which, as we will see later in the chapter, becomes useful for other constraint propagators.

5.1.2 Justifying Regular Language Membership Propagation

Once again, our main goal is to derive, for some suitable reason $\mathcal{R}$

$$\mathcal{R} \Rightarrow \overline{x}_{i=v} \geq 1 \tag{5.3}$$

whenever $X_i = v$ is found to be an assignment that is not supported by the regular language membership constraint. In particular, $X_i = v$ is not supported when there is no sequence of symbols belonging to the language recognised by M that has v as the i_{th} symbol.

Such justifications will not in general follow by reverse unit propagation, which is unsurprising given the relative simplicity of the encoding. As we have now seen several times, we will have to log additional facts during the execution of a propagation algorithm, and we will articulate this using some justification procedures and subprocedures.

We will consider here the original domain-consistent Regular propagator by Pesant [163], for a constraint over a sequence of n variables $\mathcal{X} = \langle X_1 \dots, X_n \rangle$ and associated with a DFA $M = (Q, \Sigma, \delta, q_0, F)$. This works by maintaining a layered, directed multigraph that has $n + 1$ layers, with the i_{th} layer having $|Q|$ nodes $q^i_0, \dots q^i_{|Q|-1}$. The edges in this graph are labelled with values in Σ, and are required to respect several properties that are satisfied from the outset and then maintained through propagation. These are:

1. Edges can only appear between nodes in consecutive layers.

2. An edge labelled with the value v can only be included between nodes q^i_k and nodes q^{i+1}_l if there exists a transition between the states numbered k and l in the DFA that is valid for v, and v is currently in the domain of X_i.

3. Furthermore, every edge included must appear in a path from the node q^0_0 (representing the initial state) to a node representing a final state in the last layer.

The graph is constructed at the start of the solving process, and then updated incrementally during propagation to reflect changes in the variable domains and preserve the required properties. Once this is achieved, an assignment $X_i = v$ loses support with respect to the Regular constraint exactly when there are no edges labelled with v between nodes in the i_{th} and $(i + 1)_{th}$ layers. Examples of what this looks like are shown later in Figures 5.2 and 5.6.

The strategy for proof logging is then as follows. Every time an edge (q^i_k, q^{i+1}_l) labelled with the symbol v is removed by the propagation algorithm, we should aim to derive a PB constraint

$$\mathcal{R} \Rightarrow \overline{s}_{i-1=k} + \overline{x}_{i=v} \geq 1, \tag{5.4}$$

for some suitable reason $\mathcal{R}$. This can be interpreted as saying: "given this domain state, we can't both be in state k after we've processed the first $i - 1$ symbols, and have the i_{th} symbol be v". If we do this, then when an assignment $X_i = v$ loses support, we will have logged in the proof a constraint in the form of (5.4) for all $s_{i=k}$ where $\delta(q, v)$ is an allowed transition. This ensures the desired justification (5.3) will be RUP.

We can formalise the above requirement as precondition for a justification procedure consisting of a single RUP step. A pair (q^i_k, q^{i+1}_l) of state nodes in the Regular propagator's multigraph will be called "*explicitly removed edge*" if

- there exists a transition in M between states k and l labelled with w; and
- $w \in \mathrm{dom}_t(X_i)$; but
- (q^i_k, q^{i+1}_l) is nevertheless no longer an edge in the multigraph.

All justification (sub)procedures in this section will have as implicit preconditions that we have a domain state dom_t; that $\mathsf{Regular}(\mathcal{X}, M)$ is a regular language membership constraint on a sequence of variables $\mathcal{X} = \langle X_0, \ldots, X_{n-1} \rangle$ encoded as in Encoding Procedure 5.1; and that we can access the current state of the multigraph representation valid for dom_t, as outlined above.

Justification Procedure 5.1 (Regular language membership propagation).
Preconditions: (X_i, v) *is a variable-value pair that has no support with respect to the constraint and* dom_t *(and thus will be removed by the* Regular *propagator);* E *is the set of explicitly removed edges at time* t*; and for all* $e := (q_k^i, q_l^{i+1}) \in E$ *we have already derived an "edge deletion justification"* $\mathcal{R}_e \Rightarrow s_{i=k} + \overline{x}_{i=v} \geq 1$.
Procedure:

1: $\mathcal{L} \leftarrow \cup_{e \in E} \mathcal{R}_e$
2: $\mathcal{R} \leftarrow \mathsf{GenericR}(\mathcal{X}, \mathrm{dom}_{\mathcal{L}})$
3: **rup** $\mathcal{R} \Rightarrow \overline{x}_{i=v} \geq 1$

Correctness Proof. The negation of the RUP constraint propagates all the literals in $\mathcal{R} \cup \{x_{i=v}\}$. Since $X_i = v$ has no support, there are no edges beginning in layer i labelled with v. We now argue that for every $k \in Q$, $\overline{s}_{i=k}$ must be propagated under $\mathcal{R} \wedge x_{i=v}$, contradicting the at-least-one constraint from line 2 of Encoding Procedure 5.1.

First if $\delta(k, v)$ is not an allowed transition, $\overline{s}_{i=k}$ is directly propagated by a constraint from line 9 in Encoding Procedure 5.1. Otherwise, $\delta(k, v) = l$ for some state $l \in Q$, and so (q_k^i, q_l^{i+1}) must be an explicitly removed edge, and hence by assumption $\mathcal{R}_e \Rightarrow \overline{x}_{i=v} + \overline{s}_{i=k} \geq 1$ has already been derived in the proof. Since by construction $\mathrm{dom}_{\mathcal{R}} \sqsubseteq \mathrm{dom}_{\mathcal{R}_e}$, Theorem 3.3 tells us that every literal in R_e propagates, meaning $\overline{s}_{i=k}$ is certainly propagated. □

Justification Procedure 5.2 (Regular language membership infeasibility).
Preconditions: dom_t *is a domain state that is* infeasible *with respect to the given* Regular *constraint;* E *is the set of explicitly removed edges at time* t*; and for all* $e := (q_k^i, q_l^{i+1}) \in E$ *we have already derived an "edge deletion justification"* $\mathcal{R}_e \Rightarrow s_{i=k} + \overline{x}_{i=v} \geq 1$.
Procedure:

1: $\mathcal{L} \leftarrow \cup_{e \in E} \mathcal{R}_e$
2: $\mathcal{R} \leftarrow \mathsf{GenericR}(\mathcal{X}, \mathrm{dom}_{\mathcal{L}})$
3: **rup** $\mathcal{R} \Rightarrow 0 \geq 1$

Correctness Proof. The negation of the RUP constraint propagates all the literals in $\mathcal{R}$. Since the regular constraint is infeasible, there can be no edges left in the multigraph, while respecting

the required properties and in particular, no edges left in the layer 0. Since $s_{0=0}$ propagates by definition, this means $x_{0=v}$ must propagate due to the edge deletion justifications for every $v \in \mathrm{dom}_0(X_0)$, and hence we certainly reach a domain-wipeout contradiction as per Theorem 3.2. □

The problem of constructing justifications for Regular is therefore reduced to that of deriving the constraints in the form of (5.4). It turns out that if we use the generic reason $\mathsf{GenericR}_t$ for the variables in scope of the Regular constraint, these will either follow immediately by RUP, or else will only require a small amount of additional justification, depending on how the corresponding edge elimination is inferred by the propagator.

Edges are removed by the Regular propagator in one of four ways. Firstly, when an assignment $X_i = v$ has already been removed from a domain state, either because another value has been guessed or because it has been eliminated by a propagator for another constraint, all the edges extending from the i_{th} layer labelled with value v are also removed. These removals are not "explicit" as required for Justification Procedure 5.1 and so nothing additional needs to be logged.

Secondly, when a particular node loses all of its outgoing edges, none of its incoming edges can be part of a valid path and so should be removed. These removals occur recursively in a backward pass, c.f. Algorithm 3 from Pesant [163]. So if a removed incoming edge was the last outgoing edge of a node in a previous layer, then the incoming edges of that previous node are also removed, and so on. In this case, the constraints in the form of (5.4) follow by RUP, providing any previous explicit edge removals have also been justified. We can articulate this as a justification subprocedure with appropriate preconditions.

Justification Subprocedure 5.3 (Remove incoming edges for Regular).

Preconditions: *In the multigraph associated with* Regular *all the outgoing edges for a node q_l^i (where $1 \leq i \leq n-1$) have been removed; (q_k^{i-1}, q_l^i) is an edge labelled with the value v, which therefore becomes explicitly removed; and for every previous explicitly removed outgoing edge we have already logged an edge removal justification*

$$\mathsf{GenericR}_{t'}(\mathcal{X}) \Rightarrow \overline{s}_{i=l} + \overline{x}_{i=w} \geq 1 \tag{5.5}$$

where t' is the time the removal occurred.

Procedure:

1: $\mathcal{R} \leftarrow \mathsf{GenericR}_t(\mathcal{X})$

2: **rup** $\mathcal{R} \Rightarrow \overline{x}_{i-1=v} + \overline{s}_{i-1=k} \geq 1$

Correctness Proof. The negation of the RUP constraint propagates all the literals in

$$\mathsf{GenericR}_t(\mathcal{X}) \cup \{s_{i-1=k}\} \cup \{x_{i-1=v}\}. \tag{5.6}$$

Then, since (q_k^{i-1}, q_l^i) has become an explicitly removed edge, a constraint $s_{i-1=k} \wedge x_{i-1=v} \Rightarrow s_{i=l} \geq 1$ must be defined by line 5 of Encoding Procedure 5.1 and will propagate $s_{i=l} \geq 1$.

Now since all outgoing edges from the node q_l^i have been excluded, for every value $w \in \mathrm{dom}_t(X_i)$ either the constraint $\overline{s}_{l=i} + \overline{x}_{i=w} \geq 1$ is present (from line 9 in Encoding Procedure 5.1), and so $\overline{x}_{i=w}$ propagates, or else we have logged $\mathcal{R}' \Rightarrow \overline{x}_{i=w} + \overline{s}_{i=l} \geq 1$ with $\mathcal{R}' = \mathsf{GenericR}_{t'}(\mathcal{X})$, at an earlier point t' in the current search subtree, by assumption. In the latter case, it must be the case that $\mathrm{dom}_{\mathcal{R}} \sqsubseteq \mathrm{dom}_{\mathcal{R}'}$ since both reasons fully describe the respective domain states on the same search path, and constraint propagation is monotonic. Hence, all the literals in $\mathsf{GenericR}_{t'}$ must propagate and so $\overline{x}_{i=w}$ propagates in this case too.

So in the end we exclude every value from the domain of X_i with literals, leading to contradiction as per Theorem 3.2. □

Thirdly, in the other direction, when a particular node loses all of its *incoming* edges, similarly none of its outgoing edges can be part of a valid path and so should be removed. These removals also occur recursively, this time in a forward pass, c.f. Algorithm 4 from Pesant [163]. In contrast to the previous two cases, this removal requires more justification as the desired constraint will not always follow by RUP, as is the case for (5.16) in the worked example below. The additional steps required are given as part of the following justification procedure.

Justification Subprocedure 5.4 (Remove outgoing edges for regular).

Preconditions: *In the multigraph associated with* Regular *all the incoming edges for a node q_k^i (where $1 \leq i \leq n-1$) have been removed, and hence any outgoing edges (q_k^i, q_l^{i+1}) will be explicitly removed; and for every explicitly removed incoming edge, we have already logged an edge removal justification of the form*

$$\mathsf{GenericR}_{t'}(\mathcal{X}) \Rightarrow \overline{x}_{i-1=w} + \overline{s}_{i-1=h} \geq 1; \tag{5.7}$$

where t' is the time the removal occurred.

Procedure:

1: $\mathcal{R} \leftarrow \mathsf{GenericR}_t(\mathcal{X})$
2: **for** $h \in Q$
3: **rup** $\mathcal{R} \Rightarrow \overline{s}_{i-1=h} + \overline{s}_{i=k} \geq 1$
4: **rup** $\mathcal{R} \Rightarrow \overline{x}_{i=v} + \overline{s}_{i=k} \geq 1$

Correctness Proof. We first show that each intermediate step from line 3 will be RUP. First, the negation propagates all the literals in $\{\mathsf{GenericR}_t(\mathcal{X}) \cup s_{i-1=h} \cup s_{i=k}\}$. Then, for each value $w \in \mathrm{dom}_t(X_{i-1})$, either it is already excluded by $\mathsf{GenericR}_t$; or $\delta(h, w)$ is a disallowed transition

and so there is a constraint in the form

$$\overline{x}_{i-1=w} + \overline{s}_{i-1=h} \geq 1 \tag{5.8}$$

propagating $\overline{x}_{i-1=w}$ from line 9 of Encoding Procedure 5.1; or else this is an explicitly removed edge, and so

$$\mathsf{GenericR}_{t'} \Rightarrow \overline{x}_{i-1=w} + \overline{s}_{i-1=h} \geq 1 \tag{5.9}$$

must have been previously derived, by assumption. In the last case, similar to the proof of Justification Subprocedure 5.3, $\mathsf{GenericR}_t(\mathcal{X})$ must propagate all the literals in $\mathsf{GenericR}_{t'}(\mathcal{X})$, and hence $x_{i-1=w}$ propagates. So all values of X_{i-1} are excluded by propagation, and we reach a contradiction again by Theorem 3.2.

Once these constraints are derived on line 3, it is immediate that the final constraint is RUP, as $\overline{s}_{i-1=h}$ must be propagated for every $h \in Q$, contradicting line 2 of Encoding Procedure 5.1. □

Finally, at the top of search we should remove edges outgoing from the first layer when they do not start in the designated initial node q_0^0 and incoming to the final layer when they do not end in a designated final state note $\{q_n^f : f \in F\}$. These are trivial to justify by RUP.

Justification Procedure 5.5 (Remove edges from the first or last layer for Regular).
Preconditions: (q_i^l, q_{i+1}^k) *is an edge labelled with the value* v *to be explicitly removed where either:* $i = 0$ *and* $l \neq 0$*; or* $i = n - 1$ *and* $k \notin F$
Procedure: **rup** $\overline{s}_{i=l} + \overline{x}_{i=v} \geq 1$

Correctness Proof. In the first case, where $i = 0$, the negation of the RUP constraint immediately reaches contradiction, since $\overline{s}_{i=l}$ propagates due to Lines 2 and 13 of Encoding Procedure 5.1.

Otherwise, if $i = n - 1$, since (q_i^l, q_{i+1}^k) is to be explicitly removed, we must have $s_{i=l} \wedge x_{i=v} \Rightarrow s_{i+1,k} \geq 1$ defined from line 5 of Encoding Procedure 5.1, and hence the negation of the RUP constraint will cause this to propagate $s_{i+1=k}$, which will in turn propagate $\overline{s}_{i+1=f}$ for every $f \in F$, due to line line 2, and hence contradict the constraint on line 14 □

With these three procedures, we can be confident we can justify any domain-consistent inference made by a Regular propagator, providing we maintain the required intermediate steps and keep track of the state of the multigraph. Then because edge removal occurs recursively on each layer of the graph, either in a forwards or backwards pass, we can always ensure that all edge removals from a previous layer are justified before justifying removals in the current layer.

The number of steps logged is within a linear factor of the amount of work performed by the Regular propagator. Each backward pass removal requires just one RUP step, and each forward pass removal requires $O(|Q|)$ steps.

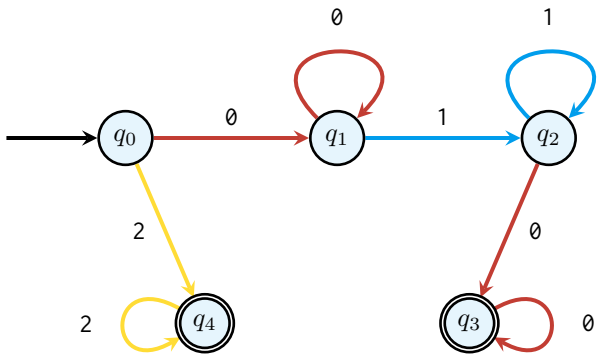


Figure 5.1: A DFA M recognising the regular expression `00*11*00*|2*`. Double circles indicate accepting states.

5.1.3 A Worked Example for Regular Propagation

Once again, to demonstrate how the proof logging procedure works more concretely, we will now show a worked example, including the initial building of the graph and a round of domain-consistent propagation. We will adapt the simple Example 1 from Pesant [163]. This has a regular language membership constraint on five variables $X_0, \ldots, X_4$ each with domain $\{0, 1, 2\}$, and uses the DFA M as shown in Figure 5.1.

For a PB model of this, we will as usual omit the constraints encoding the domains and atomic literals and focus on the constraints present due to Encoding Procedure 5.1. We can see that we have five states numbered $0 \ldots 4$, and five variables in the sequence, so we would first define:

$$s_{0=0} + \cdots + s_{0=4} \geq 1; \quad \ldots \quad s_{4=0} + \cdots + s_{4=4} \geq 1; \tag{5.10}$$

$$-s_{0=0} - \cdots - s_{0=4} \geq -1; \quad \ldots \quad -s_{4=0} - \cdots - s_{4=4} \geq -1; \tag{5.11}$$

saying that we have to have exactly one of the state-position flags set for each position. Next for each position $i \in \{0, \ldots, 4\}$ we would define eight PB constraints corresponding to the eight valid transitions:

$$\begin{aligned} &\overline{s}_{i=0} + \overline{x}_{i=0} + s_{i+1=1} \geq 1; \quad \overline{s}_{i=0} + \overline{x}_{i=2} + s_{i+1=4} \geq 1; \quad \overline{s}_{i=1} + \overline{x}_{i=0} + s_{i+1=1} \geq 1; \\ &\overline{s}_{i=1} + \overline{x}_{i=1} + s_{i+1=2} \geq 1; \quad \overline{s}_{i=2} + \overline{x}_{i=0} + s_{i+1=3} \geq 1; \quad \overline{s}_{i=2} + \overline{x}_{i=1} + s_{i+1=2} \geq 1; \\ &\overline{s}_{i=3} + \overline{x}_{i=0} + s_{i+1=3} \geq 1; \quad \overline{s}_{i=4} + \overline{x}_{i=2} + s_{i+1=4} \geq 1; \end{aligned} \tag{5.12}$$

along with constraints of the form

$$\overline{s}_{i=q} + \overline{x}_{i=v} \geq 1; \tag{5.13}$$

for the remaining seventeen invalid transitions. Finally, we would have constraints saying that we

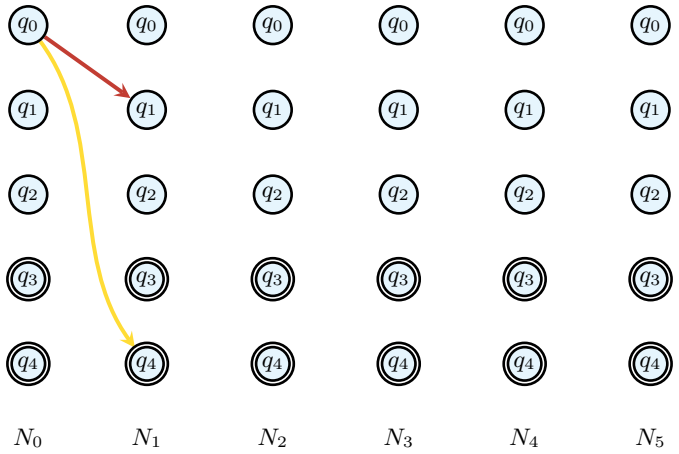


Figure 5.2: The first traversal in the forward pass for building the layered multigraph used in propagation of $\mathsf{Regular}(X_1, \ldots, X_5, M)$.

have to start in an initial state and end in a final state:

$$s_{0=0} \geq 1; \qquad s_{5=3} + s_{5=4} \geq 1. \tag{5.14}$$

Propagation depends on us having first built a valid graph, with nodes as shown in Figure 5.2. As mentioned, this occurs in two passes, with the forward pass collecting nodes that can be reached from the initial state. This corresponds to removing outgoing edges only reachable from nodes other than the initial state. So to begin with, we traverse the two edges incident to node q_0^0, reaching nodes q_1^1 and q_4^1 and thereby eliminating all other edges. These are the two edges shown in Figure 5.2.

Since we are eliminating outgoing edges from the first layer, we can apply Justification Procedure 5.5. This tells us that we can log justifications for each of the six eliminated edges:

$$\begin{aligned} &\overline{s}_{0=1} + \bar{x}_{0=0} \geq 1; \quad \overline{s}_{0=1} + \bar{x}_{0=1} \geq 1; \quad \overline{s}_{0=2} + \bar{x}_{0=0} \geq 1; \\ &\overline{s}_{0=2} + \bar{x}_{0=1} \geq 1; \quad \overline{s}_{0=3} + \bar{x}_{0=0} \geq 1; \quad \overline{s}_{0=4} + \bar{x}_{0=2} \geq 1; \end{aligned} \tag{5.15}$$

and these will all follow by RUP. Continuing, we recursively collect edges reachable from the initial node, consequently eliminating all outgoing edges for any node that is not reached. The result of this is shown in Figure 5.3.

At each layer, we ensure edge-removal justifications are logged as we explicitly remove outgoing edges. For example, we do not collect (q_3^2, q_3^3) at layer 2, since q_3^2 is not reached as part of the forward pass. So we would like to log the edge removal justification

$$\overline{s}_{2=3} + \bar{x}_{2=0} \geq 1; \tag{5.16}$$

but this does not follow by RUP. Despite us having logged constraints in the form of (5.4) for each eliminated incoming edge to q_2^3, none of these constraints contain either $s_{2=3}$ or $x_{2=0}$, as

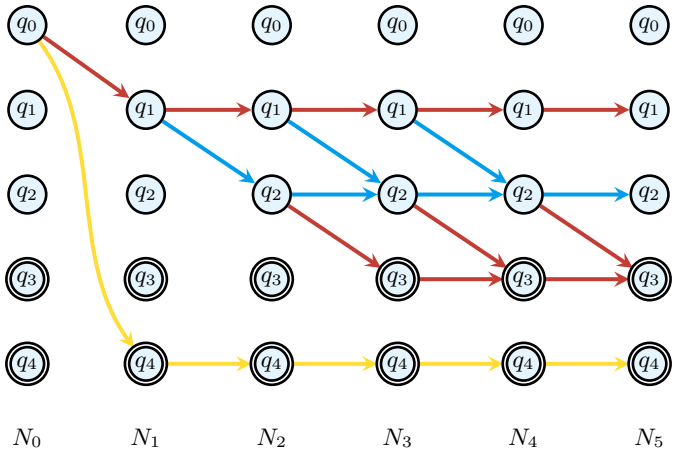


Figure 5.3: The same multigraph after the forward pass is complete.

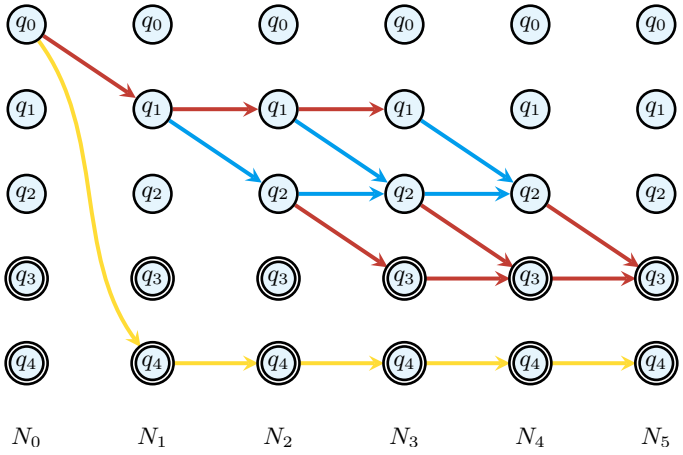


Figure 5.4: The same multigraph after the backwards pass is complete.

they concern the previous layer, and so no further propagation takes place from the negation of (5.16). Hence, the extra steps of Justification Subprocedure 5.4 are indeed necessary, and as shown we can first log

$$\overline{s}_{1=0} + \overline{s}_{2=3} \geq 1; \quad \overline{s}_{1=1} + \overline{s}_{2=3} \geq 1; \quad \overline{s}_{1=2} + \overline{s}_{2=3} \geq 1; \quad \overline{s}_{1=3} + \overline{s}_{2=3} \geq 1; \quad \overline{s}_{1=4} + \overline{s}_{2=3} \geq 1; \tag{5.17}$$

which all do follow by RUP, and then our constraint (5.16) can be logged.

After this process is complete, the backwards pass takes place, first eliminating incoming edges from any nodes in the last layer corresponding to non-final states, and then recursively eliminating all incoming edges from any node that lost all of its outgoing edges. The complete, correctly initialised state of the graph after this backwards pass is shown in Figure 5.4.

Following Justification Procedure 5.5 and then Justification Subprocedure 5.3 for eliminating incoming edges, we simply need to log one constraint as per line 2 for each removal, and these will follow by RUP.

At this point, some basic inferences about variable value pairs can already be made, such

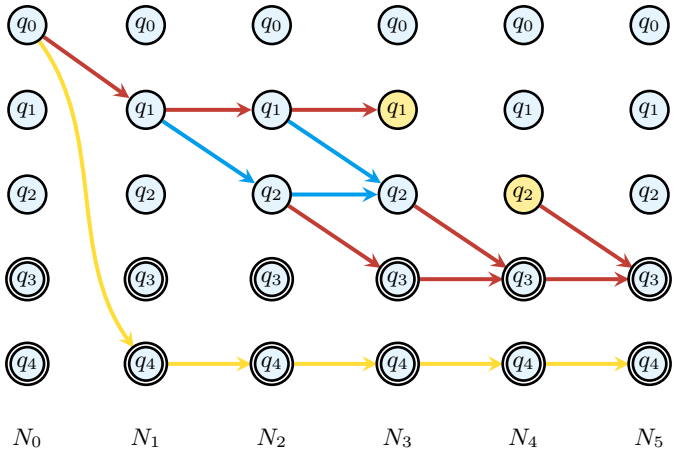


Figure 5.5: After inferring that $X_3 \neq 1$, the multigraph is in an invalid state.

as $X_0 \neq 1$ and $X_4 \neq 1$. Since we have justified all explicitly removed edges as required, a propagation justification for these inferences will follow by RUP, as per Justification Procedure 5.1. This concludes the graph initialisation for the Regular propagator, and from this point onwards the structure can be updated incrementally and restored upon backtrack, as described in detail by Pesant [163]. We will now demonstrate one such incremental update, corresponding to an execution of domain-consistent propagation for Regular. Suppose at point t through the course of solving, the value 1 is removed from the domain of X_3. This means the edges (q_1^3, q_2^4) and (q_2^3, q_2^4) are immediately implicitly removed from the graph, and the situation is as shown in Figure 5.5, with the two highlighted nodes having lost all of their outgoing and incoming edges respectively.

The recursive incremental update is then executed for each of these, further removing the edge (q_1^2, q_1^3) in a backwards pass, and so by Justification Subprocedure 5.3 we can log the RUP constraint

$$\mathsf{GenericR}_t(\mathcal{X}) \Rightarrow \overline{s}_{3=1} + \overline{x}_{2=0}. \tag{5.18}$$

The edge (q_2^4, q_3^5) is also removed, but because this is removed in a forward pass, we apply Justification Subprocedure 5.4 and log

$$\mathsf{GenericR}_t(\mathcal{X}) \Rightarrow \overline{s}_{3=0} + \overline{s}_{4=2} \geq 1; \tag{5.19}$$

$$\mathsf{GenericR}_t(\mathcal{X}) \Rightarrow \overline{s}_{3=1} + \overline{s}_{4=2} \geq 1; \tag{5.20}$$

$$\vdots \tag{5.21}$$

$$\mathsf{GenericR}_t(\mathcal{X}) \Rightarrow \overline{s}_{3=4} + \overline{s}_{4=2} \geq 1; \tag{5.22}$$

before logging the required

$$\mathsf{GenericR}_t(\mathcal{X}) \Rightarrow \overline{s}_{4=2} + \overline{x}_{2=0} \geq 1. \tag{5.23}$$

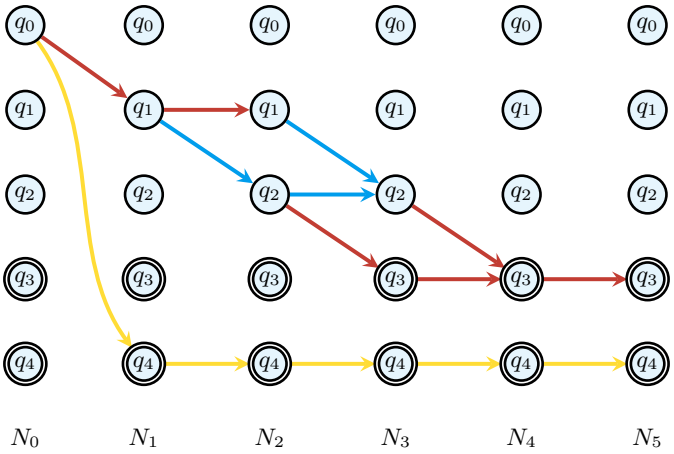


Figure 5.6: The same multigraph after its consistency properties have been re-established.

No further nodes lose their last incoming or outgoing edge as a result of this, and so the required properties (and hence domain consistency) have been re-established on the graph. The final consistent state is shown in Figure 5.6.

5.2 Decision Diagram-Based Constraints

We have seen how the use of this layered directed multigraph representing states and transitions allows for efficient propagation of the Regular constraint, and also efficient proof logging. Similar to smart table, Regular can be used to represent a number of recognised global constraints, including:

- Stretch; [163]
- Pattern; [163]
- GlobalContiguity; [20]
- ValuePrecedeChain; [20]
- Slide; [27]
- Some restricted cases of the Geost constraint for placement problems [27].

This means that we can already conclude that PB proof logging will be applicable to these too, albeit with the caveat that we need to trust the finite automaton representations of the constraints.

But it turns out that the justification procedures presented for Regular are also a useful stepping stone for dealing with other kinds of constraint reasoning based on states and transitions. We will explore this in the following subsections.

5.2.1 MDD Global Constraints

One easy generalisation of these methods allows us to deal with certain kinds of *multivalued decision diagram* (MDD) constraints [41, 74], which essentially use the graph representation to specify the constraint directly.

We can define a global MDD constraint using a labelled directed acyclic graph $G = (N, E)$ where, as with Regular, the nodes represent *states* and edges are labelled to represent *allowed transitions* associated with variable assignments. Concretely, for a set of variables $\mathcal{X} = \{X_1, \ldots, X_n\}$, every node $q_k \in N$ is labelled with a variable index $1 \leq i \leq n$, except for a distinguished terminating node (which we will assume is $q_{|N|}$); and every edge $(q_k, q_l) \in E$ is labelled with a value d in the domain of the variable associated with q. To keep notation relatively consistent with our Regular definitions we can write q_k^i for the node uniquely identified by $1 \leq k \leq |N| - 1$ and associated with the variable X_i; and q_k^T for the terminating node.

The $\mathsf{MDD}(\mathcal{X}; G)$ constraint then enforces that the sequence of values taken by the variables in X is recognised by the multivalued decision diagram specified by G. That is, an assignment is permitted if:

- there exists a path in G starting at a node q_k^0, and ending at $q_{|N|}^T$;
- every edge in the path is of the form (q_k^i, q_l^{i+1}); and
- each edge (q_k^i, q_l^{i+1}) has label v if and only if $X_i = v$.

We can assume that there is only one node q_0^0 in the first layer, since multiple layer 0 nodes can always be merged without changing the set of recognised sequences. Similarly, there is only one node in the special T layer (effectively layer n).

Domain-consistent propagation of a global MDD constraint of this form can be achieved with a similar recursive algorithm to the one used for Regular. This is because the layered, directed multigraph used in the Pesant propagator is effectively a multivalued-decision diagram, except with multiple start and end states, and no "long edges" that skip layers. The MDD definition above also does not permit long edges, since all variables' values must be accounted for in order in the recognised sequence. The procedure given by Cheng and Yap [41] recursively collects only the edges that are part of a path from the start node q_0^0 to the terminal nodes $q_{|N|}^T$, and then, as with Regular, identifies and removes variable-value pairs that are left with supporting edges.

It is easy to see that this propagation procedure ends up being equivalent from a proof logging perspective. Encoding Procedure 5.1 is easily adapted to represent an MDD constraint, simply by viewing the $s_{i=k}$ variables as corresponding precisely to the MDD nodes q_k^i, and requiring there is only one start and one end state. Since the invariant and support mechanism is exactly the same — edges should be excluded when they are no longer part of a valid path; and the explicit removals occur recursively in layer order — the same justification procedure relying on Justification Procedure 5.1 and Justification Subprocedure 5.3 can be applied. Note there are no particular assumptions about the data structures or incrementality features of a propagator in the soundness argument for these RUP-based derivations. Thus, a more incremental approach, such as the one proposed by Gange et al. [74], could also be extended with proof logging in this way.

5.2.2 Building Decision Diagrams during Propagation

Of course, having a global MDD constraint with a dedicated propagator is by no means the only way to incorporate decision diagram methods into a CP solver. The discipline of decision diagram-based constraint programming is an active and growing area of research [25], including techniques such as replacing the domain state itself with a decision diagram-based system [8], or propagating some subset of constraints using a decision diagram [46]. While the fact that we can formulate a proof logging procedure for Regular, and by extension MDD, gives a reasonable basis for expecting such methods to be amenable to pseudo-Boolean proof logging, a full treatment is outwith the scope of this thesis.

However, we can briefly demonstrate how the basic ideas for proof logging Regular and MDD propagators can be further adapted for individual constraints which are not necessarily *represented* using a decision diagram at the modelling level, but do employ decision-diagram techniques to maintain consistency. The representative example here is the Knapsack constraint, which encapsulates the classic knapsack packing problem as a CP global. There are many variations and views of this problem, but the basic idea is to choose a set of "items" from a fixed set, conceptually to put in a "knapsack", where restrictions on the sums of certain attributes of chosen items must be respected. Many CP toolkits represent items by two attributes, "weight" and "profit"; define variables W and P as the sums of these attributes for selected items; and enforce the constraint through 0-1 variables that indicate whether each item is selected. We will view the Knapsack global constraint primarily in this way, although there are no particular barriers either for propagation or proof logging to having more or fewer attributes.

For a sequence of 0-1 variables $\mathcal{X} := X_1, \ldots, X_n$ and finite domain variables W and P, consider a fixed sequence of *weight* constants $\mathbf{w} := w_1, \ldots, w_n$ and *profit* constants $\mathbf{p} := p_1, \ldots, p_n$. The constraint $\mathsf{Knapsack}(\mathcal{X}, W, P; \mathbf{w}, \mathbf{p})$ requires that

$$\sum_{i=1}^{n} w_i \cdot X_i = W \qquad \text{and} \qquad \sum_{i=1}^{n} p_i \cdot X_i = P. \tag{5.24}$$

Obviously this is already a set of linear constraints, and so regardless of how the constraint is propagated, from a proof logging point of view it would make little sense not to encode it in the natural way:

Encoding Procedure 5.3 *A* Knapsack *constraint as defined above can be encoded with the following four PB constraints.*

$$\sum_{i=1}^{n} w_i \cdot \mathsf{BinEnc}(X_i) - \mathsf{BinEnc}(W) \geq 0; \quad \sum_{i=1}^{n} -w_i \cdot \mathsf{BinEnc}(X_i) + \mathsf{BinEnc}(W) \geq 0; \tag{5.25}$$

$$\sum_{i=1}^{n} p_i \cdot \mathsf{BinEnc}(X_i) - \mathsf{BinEnc}(P) \ \geq 0; \quad \sum_{i=1}^{n} -p_i \cdot \mathsf{BinEnc}(X_i) + \mathsf{BinEnc}(P) \ \geq 0. \tag{5.26}$$

Note that here since for every i, $\mathrm{dom}(X_i) = \{0, 1\}$, $\mathsf{BinEnc}(X_i)$ is just a single PB variable x_{ib0}, which for simplicity we can write as x_i. This definition can be straightforwardly extended to an arbitrary number of attributes with fixed values for each item, and to non-0-1 variables, which corresponds to potentially taking multiple copies of each item.

Domain consistency for Knapsack defined as above can be achieved by a method first introduced by Trick [187]. This originally dealt with only 0-1 variables and a single "weight" attribute, but it can be naturally extended to the more general cases. The idea is to take a "dynamic programming approach" to constraint propagation, and define a partial feasibility evaluation function that should be set to 1 if there is a way to set the first i variables to obtain given partial weight, profit, or other-attribute sums equal to b, and 0 otherwise. This function has a recursive formulation. When all variables are 0-1 and the only attribute is weight, it is as follows.

$$f(0, 0) = 1; \tag{5.27}$$

$$f(i, b) = \max\{f(i-1, b), f(i-1, b-w_i)\}. \tag{5.28}$$

The two components of the max here correspond to $X_i = 0$ (not taking the i_{th} item) and $X_i = 1$ (taking the i_{th} item) respectively.

But although this recurrence naturally suggests a classic dynamic programming problem, Trick's exposition of the resulting CP propagator strongly resembles a decision diagram algorithm, with dynamic programming states corresponding to nodes in a layered directed acyclic graph. Indeed, as has been previously observed [120], decision diagram and dynamic programming methods are very closely related. In this case, the dynamic programming evaluation of the feasibility function coincides with the building of a decision diagram representation of the Knapsack constraint during propagation.

Essentially, the propagator considers each set of arguments to the feasibility function to be a diagram node. So for the above formulation each node would be identified by (i, b), but in general for k attributes we would use $(i, b_1, \ldots, b_k)$. The recurrence itself can then be seen as a rule for transitioning between nodes, and hence there should be an *edge* in the diagram if both

nodes are feasible and the evaluation of one is directly dependent on the other. The edges are labelled with their corresponding domain values depending on which recursive case the value enables. So in the basic case there would be an edge between (i, b) and $(i-1, b')$ if and only if $f(i, b) = f(i-1, b') = 1$, and this would be labelled with 0 if $b' = b$ and 1 if $b' = b - w_i$.

Once such a graph is built, it noticeably satisfies the same arrangement as the graph used for the global MDD constraint: the graph has layers corresponding to the i values, and there are only edges between nodes in consecutive layers. Furthermore, it can be shown [187] that there is a one-to-one correspondence between solutions to the Knapsack constraint and paths from the node in layer 0 to a node in layer n with attribute sums compatible with the domains of the corresponding CP variables $(W, P, \dots)$ for those attributes.

Domain consistency for Knapsack can thus be obtained in much the same way as for MDD and Regular. We first collect reachable nodes and edges from the starting state in a forward pass. This tells us exactly which values are feasible for the attribute sum variables, and we can immediately prune any unsupported values, as well as drop any states corresponding to values not in the variables' domains. We then perform a backwards pass collecting which nodes can reach the feasible final states, which finally allows us to prune any domain values from the X_i variables when there are no edges left in the graph labelled with that value.

Based on what we have already seen in this chapter, the procedure seems familiar enough that we should be very hopeful a similar proof logging method based on edge-removal justifications would be applicable. This turns out to be broadly the case, however, a distinct challenge is that we do not have any of the PB constraints encoding the decision diagram structure directly available in the model. Instead, we have only the basic inequalities from Encoding Procedure 5.3.

Fortunately, we can turn to the extension rule (redundance-based strengthening as per Theorem 2.4) to introduce state variables with the requisite definitions dynamically within the proof. We can then derive the PB apparatus we need for a proof structure similar to Justification Subprocedure 5.4 and Justification Subprocedure 5.3 directly from these definitions.

For the two-attribute weight/profit case and 0-1 X_i, a state variable (i, w, p) can be defined by first introducing partial state variables

$$\textbf{ext}\ \ s^{weight}_{i\geq w} \Leftrightarrow \sum_{j=1}^{i} w_j x_j \geq w; \qquad \textbf{ext}\ \ s^{weight}_{i\leq w} \Leftrightarrow \sum_{j=1}^{i} w_j x_j \leq w; \tag{5.29}$$

$$\textbf{ext}\ \ s^{profit}_{i\geq p} \Leftrightarrow \sum_{j=1}^{i} p_j x_j \geq p; \qquad \textbf{ext}\ \ s^{profit}_{i\leq p} \Leftrightarrow \sum_{j=1}^{i} p_j x_j \leq p; \tag{5.30}$$

and then defining

$$\textbf{ext}\ \ s_{i=w,p} \Leftrightarrow s^{weight}_{i\geq w} + s^{weight}_{i\leq w} + s^{profit}_{i\geq p} + s^{profit}_{i\leq p} \geq 4. \tag{5.31}$$

Recall that a fundamental part of the diagram-like encoding for Regular was the transitions

between state variables (see Encoding Procedure 5.1, Line 5). Once we have the above state definitions, we can derive similar PB constraints for Knapsack, using cutting planes and RUP. These are collected on a forwards pass and the following example demonstrates the general idea.

Example 5.2 (Constructing a transition relation constraint for Knapsack).
For a Knapsack$(\mathcal{X}, W, P; \mathbf{w}, \mathbf{p})$ *constraint encoded as in Encoding Procedure 5.3, consider an edge that corresponds to being in a state* $(i-1, w, p)$ *and not taking an item* i *(so labelled with* $X_i = 0$*).*
Suppose we have introduced partial state definitions as above for $s_{i-1,w,p}$ *and* $s_{i,w,p}$*.*
Then if we add together the constraints

$$\overline{s}^{weight}_{i\geq w} \Rightarrow -\sum_{j=1}^{i} w_j x_j \geq -w+1; \tag{5.32}$$

$$s^{weight}_{i-1\geq w} \Rightarrow \sum_{j=1}^{i-1} w_j x_j \geq w; \tag{5.33}$$

we obtain

$$s^{weight}_{i-1\geq w} \wedge \overline{s}^{weight}_{i\geq w} \Rightarrow \quad -w_i x_i \geq 1 \tag{5.34}$$

which allows us to derive by RUP (or syntactic implication)

$$s^{weight}_{i-1\geq q} \wedge \overline{x}_i \Rightarrow \quad s^{weight}_{i\geq w} \geq 1. \tag{5.35}$$

We can similarly obtain each of

$$s^{weight}_{i-1\leq w} \wedge \overline{x}_i \Rightarrow s^{weight}_{i\leq w} \geq 1; \tag{5.36}$$

$$s^{weight}_{i-1\geq w} \wedge \overline{x}_i \Rightarrow s^{profit}_{i\geq w} \geq 1; \tag{5.37}$$

$$s^{profit}_{i-1\leq w} \wedge \overline{x}_i \Rightarrow s^{profit}_{i\leq w} \geq 1; \tag{5.38}$$

with a small constant number of proof steps using the corresponding definition constraints.
Following this we can derive a transition/edge constraint

$$s_{i-1=w,p} \wedge \overline{x}_i \Rightarrow s_{i,w,p} \geq 1 \quad \textit{by RUP}. \tag{5.39}$$

For edges collected on the forward pass corresponding to *taking* an item ($X_i = 1$), we can similarly derive

$$s_{i-1=w,p} \wedge x_i \Rightarrow s_{i=w+w_i,p+p_i} \geq 1. \tag{5.40}$$

Or, if instead taking a particular item is infeasible (because we would already violate one of the

bounds of the variables in the overall Knapsack constraint), we can derive

$$\mathcal{R} \Rightarrow s_{i-1=w,p} \wedge x_i \Rightarrow 0 \geq 1. \tag{5.41}$$

This is analogous to the disallowed transition constraints in the Regular encoding (Encoding Procedure 5.1, line 8).

The next important step is to derive at-least-one constraints analogous to those on line 3 of Encoding Procedure 5.1 for Regular, which are required for RUP justifications to work. Note that initially, the $s_{0=0,0}$ is introduced by redundance as being trivially true, since the reverse reification is $\overline{s}_{0=0,0} \Rightarrow 0 \geq 1$, which is the same as $s_{0=0,0} \geq 1$, giving us a base case. When propagating Knapsack we can derive at-least-one constraints on subsequent layers by first deriving a constraint in the form

$$\mathcal{R} \wedge s_{i-1=w,p} \Rightarrow \sum_{(w',p')} s_{i=w',p'} \geq 1; \tag{5.42}$$

for all the possible successor states (i, w', p') depending on whether taking the i_{th} item is feasible for each state given the current domains of variables. Resolving (Theorem 2.2) all of these with the at-least-one constraint on the previous layer gives

$$\mathcal{R} \Rightarrow \sum_{(w',p')} s_{i=w',p'} \geq 1. \tag{5.43}$$

A suitable reason $\mathcal{R}$ would be $\mathsf{GenericR}_t(\mathcal{X} \cup \{W, P\})$ i.e. the generic reason over all the variables involved in the Knapsack constraint.

Once all of these transition and at-least-one constraints are derived as part of the forward pass, explicit edge removal justifications for the backwards pass (on elimination of states that cannot reach a feasible end state) will be RUP by the same argument as in Justification Subprocedure 5.3. This finally allows any unsupported propagations to be justified using a RUP propagation justification using the same $\mathcal{R}$, by the same argument as in Justification Procedure 5.1.

5.3 Implementation and Experiments

In the same vein as the previous chapter, we can demonstrate the practical workability of our pseudo-Boolean justification procedures by implementing and benchmarking new global constraints within the *Glasgow Constraint Solver*. Both Knapsack and Regular constraints have been implemented, but only the latter was the work of the author of this thesis so we will focus on it here.

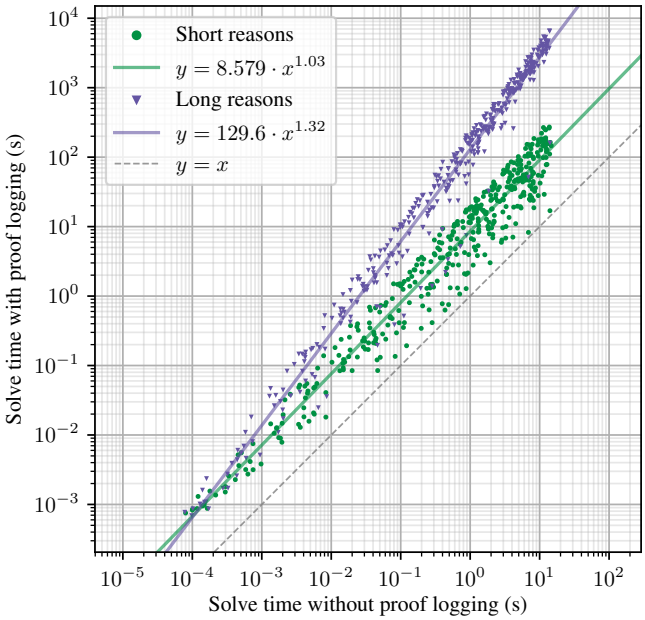
The propagation algorithm is a straightforward implementation of the one described by Pesant [163]. We used a stateful propagator that restores a persistent representation of the underlying

multigraph/decision-diagram on backtracking, and uses this to guide proof production. This makes it easy to ensure that the preconditions for the justification subprocedures are always satisfied. Another option would have been to reconstruct the full multigraph on each propagator execution, which avoids storing state at the cost of recomputation. As before, we don't attempt to address the question of which implementation methods are most beneficial, and deliberately ignore other options such as decomposing Regular to clauses, or implementing it as part of a more general MDD propagator.

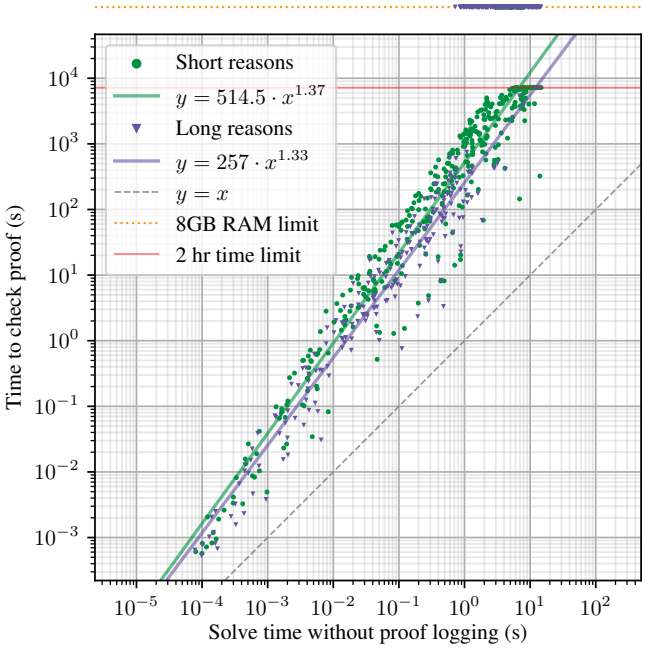
To validate the approach and roughly measure overhead in practice we generated randomised DFA transition tables on increasing sequences of variables of length 3 up to 150, choosing a random number of states and then populating a random transition table. Once again, single constraint instances such as these are not expected to tell us much about how useful our propagator is as part of a constraint solving toolkit. But we can be confident that all non-search-related logging statements come from our implementation of the new justification procedures and so this gives us a reasonable idea of the cost of logging and checking. We again include the "short reasons" optimisation from the previous chapter, and the following results make use of the same experimental setup and environment. All proofs that could be processed within the time limit and memory limits were verified using version 3.0 of the *VeriPB* proof checker. The results are shown in Figures 5.7 and 5.8.

We can make similar observations regarding overheads here. Proof logging for Regular adds a large but roughly constant factor overhead to solve time, and short reasons make a big difference to solving time at the expense of a modest checking time increase. Additionally, the size of the proofs using long reasons for some of the most difficult instances starts to push the boundary of "feasible" (in terms of logging and checking overheads), ranging of tens to hundreds of gigabytes, but remains more manageable using short reasons. This appears to be particularly relevant for checking, where even though short reasons generally take longer to check, the checker is able to use less memory in the process and thus finish verifying more proofs without running out of memory.

One implementation detail that may be contributing to the large checking overhead is the large number of auxiliary/extension variables that are permanently added to the checker's database. In the current apparatus of the *Glasgow Constraint Solver*, once an atomic literal such $x_{i=j}$ is introduced, its definition constraints are never deleted, to avoid having to redefine them later. With constraints such as Regular, where very large numbers of equality literals like these may be used at least once, this can lead to a build up of unnecessary constraints in the checking database, potentially slowing down RUP checks and eventually leading to run-time memory issues. A more aggressive deletion strategy is conceivable, but it would likely be a tradeoff between potentially having to overall log more constraints (when previously deleted atomic literal definitions are reintroduced) vs having a smaller checking database at any given point in the proof.

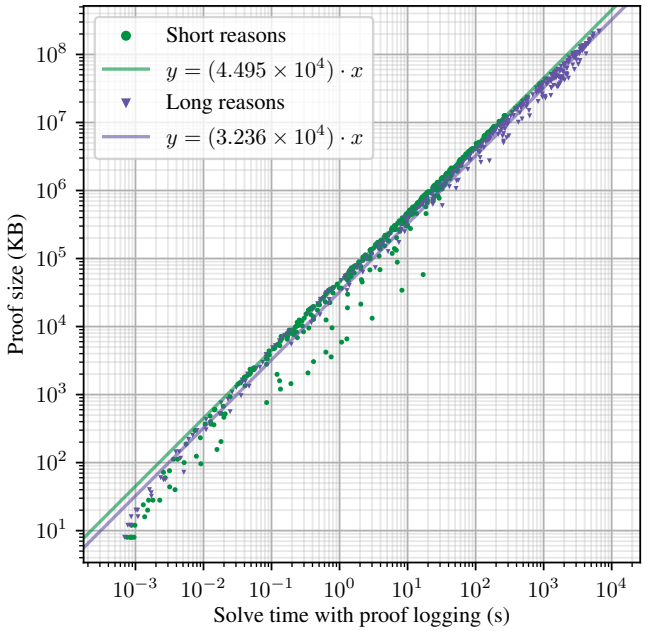


(a) Overhead of proof logging during solving.

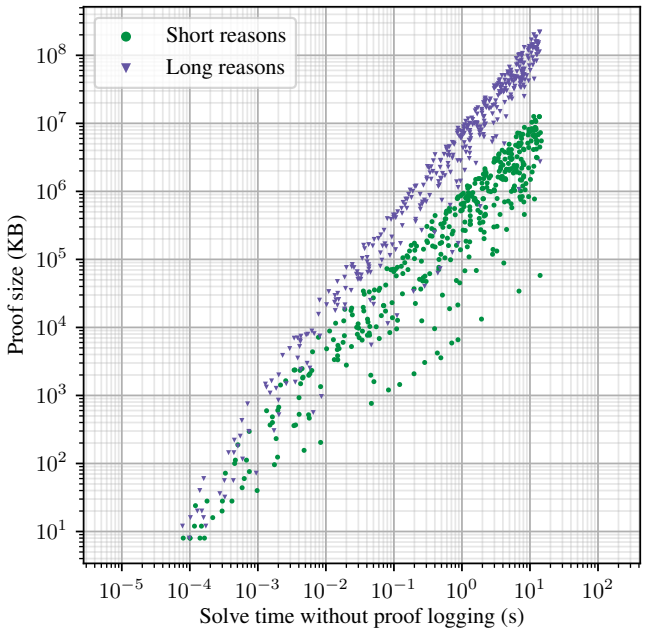


(b) Proof checking time relative to solve time.

Figure 5.7: Timing results for random Regular instances.



(a) Proof size relative to time to produce the proof.



(b) Proof size relative to solve time.

Figure 5.8: Proof size results for random Regular instances.

5.4 Conclusions

In this chapter we have devised further novel justification procedures for constraint propagation that have a level of general applicability. We have shown how consistency reasoning based on states and transitions and forwards-backwards passes can be justified by maintaining some straightforward invariants in the proof log. Again, we only needed RUP steps for this. Our methods were formulated specifically in the context of propagation of the regular-language membership constraint, which can already be used to implement a number of important global constraints. But we also saw that it can be adapted in principle for more general MDD-type reasoning, showing that pseudo-Boolean proof logging is applicable to constraints such as Knapsack, which employ decision-diagram methods for maintaining consistency.

The empirical logging and checking overheads are significant, but support the conclusion that the main remaining challenges are in the area of engineering rather than any fundamental intractability within the PB proof procedures.

Part III

Specialised Justification Procedures

Chapter 6

Hamiltonian Circuit Constraints

We have now seen some methods with varying degrees of generality for constructing justifications for CP constraints. If we know we can express a constraint compactly as a smart table or decision diagram, then we can make use of the methods in Chapters 4 and 5 to create PB proofs. But despite the existence of these techniques, it will still be desirable to have specialised PB justification procedures for certain propagators, either because generic methods do not apply, or in order to get improved proof logging performance.

In these final thesis chapters we will present specialised justification procedures for the Hamiltonian circuit global constraint (Circuit) and the ternary multiplication constraint $X \times Y = Z$. These are constraints that present unique challenges for PB proof logging, and are not amenable to any of the techniques discussed so far. They are also fundamental to many general-purpose CP solvers.

6.1 Propagating Circuit Constraints

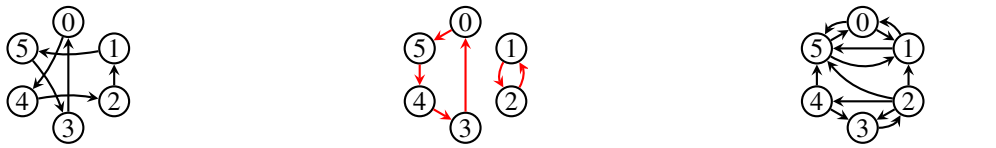
A common feature of the constraints that have been considered so far is that their usual propagation algorithms enforce some level of local consistency (either domain or bounds consistency) among the variables in scope. This has allowed us to argue for each constraint that because any domain or bounds consistent inference can be justified, we have a comprehensive procedure that can in principle be used to add proof-logging to any propagator implementation enforcing the same level of consistency. The same argument no longer works if a constraint is generally dealt with by propagators that do not enforce a clearly definable notion of consistency. And if we want to show that PB proof logging is applicable to CP solving in general, it is essential to demonstrate that the system is flexible enough to certify propagators that are based on ad-hoc propagation rules.

Propagation of the Circuit constraint provides a representative example of this situation. Enforcing domain consistency for Circuit is known to be NP-hard [129] and so it is generally propagated via ad-hoc propagation rules [39, 180, 72]. We will therefore work systematically through different inference types, from simple checking and basic lookahead, up to more advanced

propagation techniques based on depth-first search and identification of strongly connected components. In each case we will briefly outline the circumstances in which an inference can be made, and show that it can be justified either by a simple sequence of cutting planes steps, or via a conditional counting argument. This argument essentially consists of identifying a vertex which cannot reach every other vertex under some conditions, and deriving PB constraints over auxiliary variables that establish that the set of reachable vertices is too small. As with the other techniques in this thesis, the procedures have been tested by building a certifying Circuit propagator within the *Glasgow Constraint Solver*. It is comparable in propagation strength to well known open-source CP solver implementations [134, 44, 77], and it can produce verifiable proofs for a variety of instance sizes.

6.2 Definition and PB Encoding

The Circuit constraint uses a successor representation to treat a set of variables $\mathcal{X} := X_0 \dots X_{n-1}$, each with domain $\{0, \dots, n-1\}$ 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 6.1. This is useful for modelling problems such as vehicle routing [136, 137], activity scheduling [62] and other graph problems [39, 76]. 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 [39], but further lookahead and ad-hoc propagation rules are also possible [180, 72].



(a) Valid assignment. (b) Invalid assignment. (c) Partial assignment.

Figure 6.1: Interpretation of assignments for six variables constrained by Circuit.

To encode Circuit we can start by reusing the encoding for AllDifferent (Encoding Procedure 3.9). We then need to define additional PB constraints that eliminate the solutions representing subcycles. Looking at known SAT [105] and IP models [154] there are many possible options for this with different tradeoffs, but as usual we want to prioritise simplicity and obvious correspondence with the constraint's definition. We therefore define a straightforward PB encoding that is primarily adapted from a SAT encoding by Zhou [201, Sec. 4.2].

Encoding Procedure 6.1 (Circuit).
A linearisation of a constraint Circuit$(X_0, \ldots, X_{n-1})$ *can be achieved with an* AllDifferent *constraint encoding followed by subcycle elimination constraints involving auxiliary finite domain variables* $\{P_0, \ldots, P_{n-1}\}$*, as given by the following procedure.*

1: **def** $P_0 = 0$
2: **for** $i \in \{1, \ldots, n-1\}$
3: **def** $P_i \geq 1$
4: **def** $P_i \leq n-1$
5: **def** $x_{i=0} \Rightarrow P_0 - P_i = -n+1$
6: **for** $i \in \{0, \ldots, n-1\}$ and $j \in \{1, \ldots, n-1\}$
7: **def** $x_{i=j} \Rightarrow P_j - P_i = 1$

The idea is that the variables P_i are constrained to represent the *position* of vertex i in the cycle when counting from the vertex 0. We do this as shown in Encoding Procedure 6.1: define $P_0 = 0$, and then require $P_j = P_i + 1$ whenever $x_{i=j}$ is true, unless $j = 0$, in which case we 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".

Recall that in our PB encoding framework, because $P_0, \ldots, P_{n-1}$ are auxiliary (proof-only) variables that are non-0-1, they will be encoded using BinEnc replacement at step 4 of Encoding Procedure 3.1. So the actual PB constraints resulting from Encoding Procedure 6.1 will be in terms of BinEnc(P_i).

6.3 Justifying Simple Circuit Propagation

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 in scope represents a graph containing an invalid small cycle.

Based on the chosen encoding, no infeasibility justifications are required here. This is because unit propagation of a complete assignment will immediately establish a conflict; and hence Inv2 is already respected. It is straightforward to see this when examining Encoding Procedure 6.1. If all the bit values for the CP variables $X_0 \ldots X_n$ are fixed, then all the bits for the P_i variables must also be fixed by unit propagation (Theorem 2.8), unless there is a small cycle in which case a conflict will be reached. Specifically, $P_0 = 0$ together with $X_0 = v$ will fix $P_v = 1$; and then this together with $X_v = w$ will fix $P_w = 2$; and so on. When there is a small cycle, passing through X_0 (assuming AllDifferent has been correctly respected), at some point there will be an assignment $X_w = 0$, with BinEnc$(P_w) = m$ for $m < n - 1$ set by unit propagation, conflicting

with the requirement from the encoding that $\mathsf{BinEnc}(P_w) = n - 1$. Otherwise, propagation proceeds to set all P_i variables.

The same argument also tells us we can safely witness solutions to a CP optimisation problem involving a Circuit constraint using only atomic literals, as discussed in Section 3.3.3.

A better checking propagator can also return contradiction on *partial* assignments, when they encode a small cycle. This is what Francis and Stuckey [72] call check, and is a key component of the NoSubtour propagator of Pesant et al. [164] and the similar NoCycle propagator of Caseau and Laburthe [39]. 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 ignite the chain reaction of unit propagation and achieve the inconsistent setting of P_j variables. To create such a justification we can use cutting planes to sum up all the corresponding constraints for the position variables in the cycle, allowing us to create a conflict justification.

In fact, the above idea can be easily extended to produce propagation justifications for a basic lookahead version of the check propagator, called prevent by Francis and Stuckey [72], and described in various places in the literature [39, 164, 180]. This filters domains by disallowing any further assignments that would immediately complete a sub-cycle. Justification Procedure 6.1 below demonstrates how this works for both check and prevent.

Justification Procedure 6.1 (Check and prevent for Circuit).

Preconditions: $\mathsf{Circuit}(X_0, \ldots, X_{n-1})$ *encoded as per Encoding Procedure 6.1;* $m \in \{2, \ldots, n-1\}$*; and* $S := (v_0, \ldots, v_{m-1})$ *is a* (0-indexed) *sequence of* m *values where* $S[0] \in \mathrm{dom}_t(X_{S[m-1]})$ *and* $S[i] \in \mathrm{dom}_t(X_{S[i-1]})$ *for* $i \in \{1, \ldots, m-1\}$*; i.e. we have a small cycle of vertices.*

Procedure:

1: $\mathcal{R} \leftarrow \emptyset$
2: **for** $i \in \{0, \ldots, m-1\}$
3: $\quad v \leftarrow S[i]$
4: $\quad w \leftarrow S[i+1 \mod m]$
5: $\quad \mathcal{R} \leftarrow \mathcal{R} \cup \{x_{v=w}\}$
6: $\quad C_i \leftarrow$ **get** $x_{v=w} \Rightarrow \mathsf{BinEnc}(P_w) - \mathsf{BinEnc}(P_v) \geq 1$
7: **if** $0 \notin S$
8: $\quad D \leftarrow$ **cut** $\sum_{i=1}^{m} S_i$
9: $\quad$ **imp** $\mathcal{R} \Rightarrow 0 \geq 1$ **from** D
10: **else rup** $\mathcal{R} \Rightarrow 0 \geq 1$

To concisely express the correctness arguments for this procedure and others in this chapter, we will simply write P_j instead of $\mathsf{BinEnc}(P_j)$, as used in linearisations. We will also let

$K = \sum_{i=0}^{k-1} 2^i$, where k is the number bits used in each $\mathsf{BinEnc}(P_j)$, and then write K_t for $K + t$ for any $t \in \mathbb{Z}$. This allows us to compactly write out reified constraints without the $\Rightarrow$ syntactic sugar, which can be useful for computing cutting planes steps.

Correctness Proof for Justification Procedure 6.1. As discussed, if $\mathcal{R}$ includes equality assignments for a small cycle passing through the 0 vertex, then $\mathcal{R} \Rightarrow 0 \geq 1$. Otherwise, $0 \notin S$. Now recall that each constraint C_i from the encoding obtained is really of the form

$$K_1 \cdot \overline{x}_{S[i]=S[i+1]} + P_{S[i+1]} - P_{S[i]} \geq 1 \tag{6.1}$$

with K_1 as defined above, and hence each successive addition cancels the previous P_{v_i} value. This results in the constraint

$$K_1 \cdot \overline{x}_{S[0]=S[1]} + \cdots + K_1 \cdot \overline{x}_{S[m-1]=S[0]} - P_{S[0]} + P_{S[1]} - \ldots$$
$$\cdots - P_{S[m-1]} + P_{S[m]} - P_{S[m]} + P_{S[0]} \geq m, \tag{6.2}$$
$$\text{which telescopes to } K \cdot \overline{x}_{S[0]=S[1]} + \cdots + K \cdot \overline{x}_{S[m-1]=S[0]} \geq m. \tag{6.3}$$

This syntactically implies $\mathcal{R} \Rightarrow 0 \geq 1$, as required. □

6.4 Justifying SCC Circuit Propagation

There are several possibilities for stronger propagation of the Circuit constraint, although there is no consensus between solvers on which forms are worthwhile in practice. As usual, in this thesis we are not attempting to argue for one propagation strategy over any other; rather, the focus is to show that whatever propagator is chosen, it should be feasible to implement a proof logging version of it. We can 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 [72] and versions of it are implemented in the solvers *Gecode* [77], *Chuffed* [44], *JaCoP* [134], and *CP-SAT* [162] among others.

Let $G = (V, E)$ be a directed graph, and let r be the (directed) *reachability* relation on G — for $v, w \in V, (v, w) \in r$ if and only if there exists a path from v to w. We will denote by $\mathsf{Reach}(v)$ the set $\{w : (v, w) \in r\}$, 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 a 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 implied by the current domains of variables in scope we can backtrack early, as no satisfying Circuit assignment is possible.

Strongly connected components can be identified using the well-known algorithm by Tarjan [183], which exploits 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. But even if there is only a single SCC (and thus the Circuit constraint is not infeasible), the DFS tree itself provides information that can be used for domain pruning. This was first noted in the literature by Schulte and Tack [180] (credited to Mats Carlsson), and expanded on in more detail by Francis and Stuckey [72].

To properly motivate the way we add proof logging to such a propagator, and to attempt to make the explanation easier to follow, this section takes a slightly different approach to how we have presented justifications earlier in this thesis. We will first *assume* we have a certain justification subprocedure, which we call ReachTooSmall, asserting it can derive facts of a certain form given a domain state and potentially some ordering restrictions. We can then show that this subprocedure can be used to construct justifications for many of the filtering rules described by Francis and Stuckey [72]. After that we will present the full details of how ReachTooSmall can be implemented, before returning to present justification procedures for the remaining rules.

For now, we will say ReachTooSmall takes as input a domain state dom_t and an index r of a variable in the scope of a circuit Circuit constraint on variables $X_0, \ldots, X_{n-1}$ (so $v \in \{0, \ldots, n-1\}$). The domain state can be seen as encoding an intermediate graph with respect to the Circuit constraint with edges for values still in scope, i.e.

$$\mathsf{GraphRep}(\mathcal{X}, \mathrm{dom}_t) := (\{0, \ldots, n-1\}, \{(w, v) : v \in \mathrm{dom}_t(X_w)\}). \tag{6.4}$$

Our main assumption about ReachTooSmall is then that if from the given vertex there is at least one unreachable vertex in the represented graph, the procedure will derive an infeasibility justification. We can express this more formally as follows.

A1. If we have a Circuit constraint on the variables $\mathcal{X}$ encoded as per Encoding Procedure 6.1, and w is a vertex of $G := \mathsf{GraphRep}(\mathcal{X}, \mathrm{dom}_t)$ such that $|\mathsf{Reach}(w)| < |G|$, then ReachTooSmall$(w, \mathrm{dom}_t)$ will derive $\mathcal{R} \Rightarrow 0 \geq 1$, where $\mathcal{R}$ is a subset of $\mathsf{ImpLits}(\mathrm{dom}_t)$.

6.4.1 Justifying Filtering Rules using Reachability Proofs

In all the following justification procedures, we will implicitly include in the preconditions list that we have a Circuit constraint on variables $\mathcal{X} := X_0, \ldots, X_{n-1}$ encoded as per Encoding Procedure 6.1; a procedure ReachTooSmall satisfying A1; a domain state dom_t with $G := (\mathcal{X}, \mathrm{dom}_t)$; and a DFS on G has been executed as part of Tarjan's algorithm starting from the vertex v_0.

Let us start with the most basic inference related to the SCC algorithm. Once we have initiated a depth-first search (DFS) from a chosen arbitrary vertex, we can immediately return contradiction if any of its descendants are identified as the root of an SCC.

Justification Procedure 6.2 (More than one SCC for Circuit).
Preconditions: $w \neq v_0$ *is the root of an SCC detected by Tarjan's algorithm for* G *(and hence a* Circuit *propagator would detect infeasibility).*
Procedure: ReachTooSmall(w, dom_t)

Correctness Proof. We know 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 according to Tarjan's algorithm. So the conditions for A1 are met and ReachTooSmall derives a conflict justification. □

Next, 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 other vertex not reachable from v_0 and the propagator returns a contradiction. The correctness of the justification procedure for this is obvious.

Justification Procedure 6.3 (Disconnected graph for Circuit).
Preconditions: *The size of the set of vertices reached from* v_0 *in the DFS is less than* n.
Procedure: ReachTooSmall(v_0, dom_t)

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 examination of source code for open source solvers. As mentioned, several solvers such as *Gecode* and *Chuffed* also implement further ad-hoc propagation opportunities when multiple distinct subtrees are explored below v_0, and we can use ReachTooSmall to justify these too. They are easiest to understand with diagrams, with triangles representing subtrees visited bellow the root in the DFS. We add to the list of implicit preconditions here that the algorithm has not identified multiple SCCs or a disconnected graph, and that it has explored multiple distinct subtrees below the root.

We also need to make a further assumption about ReachTooSmall, namely, that it can take an optional third argument ℓ that is an atomic literal representing a *forced* or *forbidden* edge in the graph. Intuitively, if we assume $X_i = v$, and hence $X_j \neq v$ for all $j \neq i$ (by AllDifferent), if the reachable set for a given vertex is too small under these conditions, we should be able to derive a propagation justification for inferring $X_i \neq v$. Similarly, if we assume $X_i \neq v$, and the reachable set is too small, it will be possible to derive a propagation justification for setting $X_i = v$. More formally, we can state the following.

A2. Suppose we have a Circuit constraint on the variables $\mathcal{X}$ encoded as per Encoding Procedure 6.1; and an atomic literal ℓ for a variable $X_i \in \mathcal{X}$. Let dom_a be the modified domain

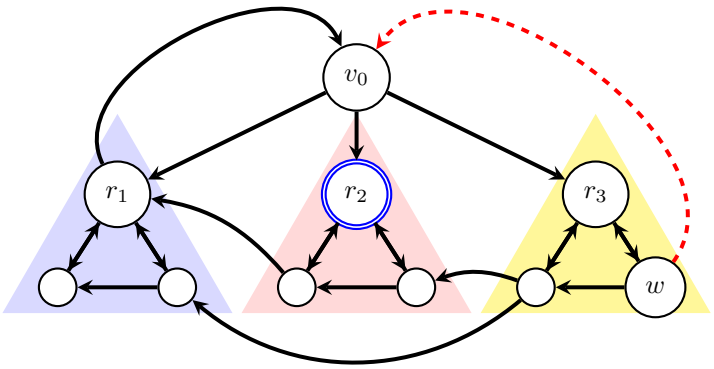


Figure 6.2: Justifying the "prune skip to root" inference. If the dotted edge (w, v_0) is used, (r_1, v_0) is eliminated and so there is no way to reach v_0 from r_2.

state given by

$$\mathrm{dom}_a(X_j) := \begin{cases} \mathrm{dom}_t(X_i) \cap \mathrm{dom}_{\{\ell\}}(X_i) & \text{if } j = i. \\ \mathrm{dom}_t(X_j) \setminus \{v\} & \text{if } j \neq i \text{ and } \mathrm{dom}_{\{\ell\}}(X_i) = \{v\} \\ \mathrm{dom}_t(X_j) & \text{otherwise,} \end{cases} \tag{6.5}$$

i.e., dom_a is the domain state given by making the modification described by ℓ, and additionally propagating the pairwise-disjoint requirement of an AllDifferent constraint on $\mathcal{X}$.

Then if w is a vertex of $G := \mathsf{GraphRep}(\mathcal{X}, \mathrm{dom}_a)$; such that $|\mathsf{Reach}(w)| < |G|$, then $\mathsf{ReachTooSmall}(w, \mathrm{dom}_t, \ell)$ will derive $\mathcal{R} \Rightarrow \ell$, where $\mathcal{R}$ is a subset of $\mathsf{ImpLits}(\mathrm{dom}_t)$.

With this new optional argument for an assumed edge, we can justify further filtering rules using ReachTooSmall.

Justification Procedure 6.4 (Pruning edges to the root for Circuit).
Preconditions: *w is not in the* earliest *visited subtree; (w, v_0) is an edge in G (that can therefore be pruned); and r is the root of a subtree visited earlier than the one containing w.*
Procedure:

$\mathsf{ReachTooSmall}(\mathrm{dom}_t, r, x_{w=v_0})$

Correctness Proof. Since we assume $X_w = v_0$ as a forced edge, we exclude in the intermediate graph any edges from descendants of r leading to v_0, as per A2. Vertices in this earlier subtree cannot have any edges leading to vertices in w's subtree or later, otherwise they would have

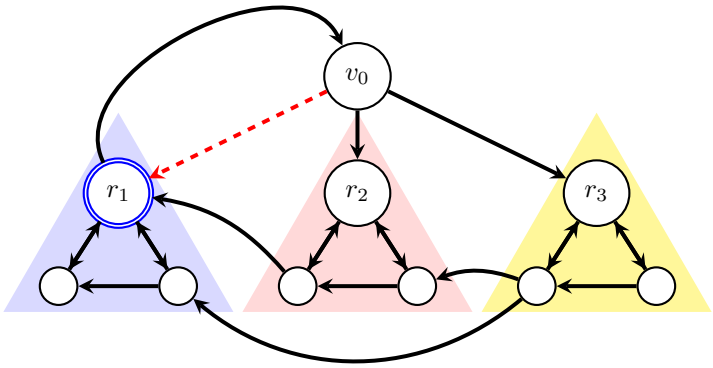


Figure 6.3: Justifying the "prune root" inference. If the dotted edge (v_0, r_1) is used, (v_0, r_2) and (v_0, r_3) are eliminated and so there is no way to reach e.g. r_2 from r_1.

been traversed as part of the same subtree by DFS. It follows that r cannot reach w. Hence, $\mathsf{ReachTooSmall}(\mathrm{dom}_t, r, (w, v_0))$ can be used to derive a propagation justification for the pruning. See Figure 6.2 for an intuition. □

Justification Procedure 6.5 (Prune edges from the root for Circuit).
Preconditions: *w is not in the* latest *visited subtree; (v_0, w) is an edge in G (that can therefore be pruned).*
Procedure:

$\mathsf{ReachTooSmall}(\mathrm{dom}_t, r, x_{v_0=w})$

Correctness Proof. Since w can only reach vertices in its own subtree or earlier, and v_0 no longer has edges to the later subtrees, it follows that not everything can be reached from w. See Figure 6.3. □

Justification Procedure 6.6 (Prune edges within subtrees for Circuit).
Preconditions: *w, v are vertices in G (different from v_0) 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 (hence $X_v = w$ can be pruned [72]).*
Procedure:

$\mathsf{ReachTooSmall}(\mathrm{dom}_t, w, x_{v=w})$

Correctness Proof. Fixing the successor of v to be w eliminates any possibility of reaching any nodes visited earlier than v from w. So in particular v_0 is not reachable from w. See Figure 6.4. □

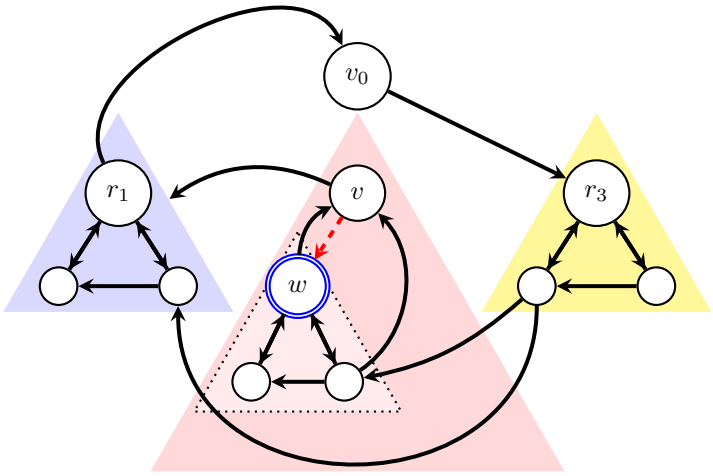


Figure 6.4: Justifying the "prune within" inference. If the dotted edge (v, w) is used, (v, r_1) is eliminated and so there is no way to reach e.g. v_0 from w.

There are some final filtering rules we would like to consider, based on the fact that edges cannot skip subtrees even when they do not begin or end with the root. To justify these, however, we will require one final, somewhat tricky, assumption type that is best expressed via extension variables internal to the ReachTooSmall procedure. So at this point we will go into the details of how ReachTooSmall can be implemented, which completes all the justification procedures outlined so far. After this we will return to consider our last filtering rules.

6.4.2 A Reachability Proof Worked Example

As stated, we have so far been assuming in our (short) justification procedures for Circuit filtering rules that we can construct a derivation of propagation or conflict justifications whenever we identify a domain state that represents a graph where, for some vertex v, $|\mathsf{Reach}(v)| \leq |G|$. It needs to be possible to construct this argument subject to an atomic literal assumption, modifying the domain state (including enforcing the pairwise-disjoint requirement of AllDifferent) in order to get a graph contradicting the reachability requirement.

To begin with, it will be easiest to understand how a ReachTooSmall proof can be structured by working through an example for the simplest case where the vertex of interest happens to be the 0 vertex and there are no literal assumptions. The basic idea is to collect the sets of possible position variables that can take certain values (as defined in Encoding Procedure 6.1) in a breadth-first search from the starting node. Eventually, we will have gathered a set of values with too few possible position variables to equal them all — a classic pigeonhole counting argument.

Note that even though the P_i variables are auxiliary variables and not part of the original CP model, they are still BinEnc encoded finite domain variables, so there is nothing to stop us introducing, effectively, atomic literals via the extension (redundance) rule for these as if they

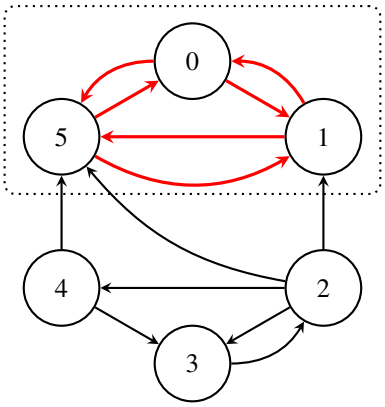


Figure 6.5: Domain state graph with reachable set from 0 marked.

were ordinary CP variables, i.e.,

$$\mathsf{Def}_{\Rightarrow}(p_{i\geq v}) := p_{i\geq v} \Rightarrow \mathsf{BinEnc}(P_i) \geq v; \tag{6.6}$$

$$\mathsf{Def}_{\Leftarrow}(p_{i\geq v}) := \overline{p}_{i\geq v} \Rightarrow -\mathsf{BinEnc}(P_i) \geq -v+1; \tag{6.7}$$

$$\mathsf{Def}_{\Rightarrow}(p_{i=v}) := p_{i=v} \Rightarrow p_{i\geq v} + p_{i\geq v+1} \geq 2; \tag{6.8}$$

$$\mathsf{Def}_{\Leftarrow}(p_{i=v}) := \overline{p}_{i=v} \Rightarrow \overline{p}_{i\geq v} + \overline{p}_{i\geq v+1} \geq 1. \tag{6.9}$$

We are therefore aiming to derive at-least-one and at-most-one constraints (expressed using these literals) over all the possible i values for each $k \in \{0, \ldots, |\mathsf{Reach}(0)|\}$.

For example, suppose the graph representation for a particular domain state dom_t is as shown in Figure 6.5. Clearly, $\mathsf{Reach}(0) = \{0, 1, 5\}$, which has fewer than 6 elements, so we can run $\mathsf{ReachTooSmall}(\mathrm{dom}_t, 0)$.

In this particular case the procedure should derive constraints that show for each $k \in \{0, 1, 2, 3\}$ that at least one of the vertices 0, 1, or 5 must have position value k. This begins with the at-least-one constraint:

$$\textbf{rup} \qquad p_{0=0} \geq 1; \tag{6.10}$$

and then looking at the possible values for X_0, deriving

$$\textbf{rup} \qquad p_{0=0} \wedge x_{0=1} \Rightarrow p_{1=1} \geq 1; \tag{6.11}$$

$$\textbf{rup} \qquad p_{0=0} \wedge x_{0=5} \Rightarrow p_{1=5} \geq 1; \tag{6.12}$$

$$\textbf{rup} \quad \mathsf{GenericR}_t(X_0) \Rightarrow x_{0=1} + x_{0=5} \geq 1; \tag{6.13}$$

which can be resolved (addition and saturation, recall Theorem 2.2), to yield the next at-least-one constraint

$$\mathsf{GenericR}_t(X_0) \Rightarrow p_{1=1} + p_{1=5} \geq 1. \tag{6.14}$$

Further at-least-one constraints on P_i literals can then be derived by resolving the previous at-least-one constraint with the possible transitions for each (P_i, X_j) combination. In a way, this resembles the forwards pass of the state-transition proof procedures from Chapter 5.

$$\textbf{rup} \quad p_{5=1} \wedge x_{5=0} \Rightarrow p_{0=2} \geq 1; \tag{6.15}$$

$$\textbf{rup} \quad p_{5=1} \wedge x_{5=1} \Rightarrow p_{1=2} \geq 1; \tag{6.16}$$

$$\textbf{rup} \quad \mathsf{GenericR}_t(X_5) \Rightarrow x_{5=0} + x_{5=1} \geq 1; \tag{6.17}$$

$$\text{(resolve)} \quad \mathsf{GenericR}_t(X_5) \wedge p_{5=1} \Rightarrow p_{0=2} + p_{1=2} \geq 1; \tag{6.18}$$

$$\vdots$$

$$\text{(similarly)} \quad \mathsf{GenericR}_t(X_1) \wedge p_{1=1} \Rightarrow p_{0=2} + p_{5=2} \geq 1; \tag{6.19}$$

$$\text{(resolve)} \quad \mathsf{GenericR}_t(X_0, X_1, X_5) \Rightarrow p_{0=2} + p_{1=2} + p_{5=2} \geq 1; \tag{6.20}$$

$$\vdots$$

$$\text{(similarly)} \quad \mathsf{GenericR}_t(X_0, X_1, X_5) \Rightarrow p_{0=3} + p_{1=3} + p_{5=3} \geq 1. \tag{6.21}$$

Next we can derive at-most-one constraints over the values for each P_i variables. There are several possible ways to do this: one method is simply to derive all pairs $\overline{p}_{i=k} + \overline{p}_{i=l} \geq 1$ for every $k \neq l$, and then recover a cardinality constraint as per Theorem 2.3. Another option uses proof by contradiction (i.e. redundance with an empty witness, or the PBC rule), see Justification Subprocedure 6.7. Regardless, in this example, we will be able to derive the constraints

$$-p_{0=0} - p_{0=2} - p_{0=3} \geq 1; \tag{6.22}$$

$$-p_{1=1} - p_{1=2} - p_{1=3} \geq 1; \tag{6.23}$$

$$-p_{5=1} - p_{5=2} - p_{5=3} \geq 1; \tag{6.24}$$

which precisely cancel out all the $p_{i=j}$ literals when summed with the at-least-one constraints (6.10), (6.14), (6.20) and (6.21), leaving us with

$$\mathsf{GenericR}_t(X_0, X_1, X_5) \Rightarrow 0 \geq 1, \qquad \text{as required.} \tag{6.25}$$

6.4.3 Reachability Proofs in Full

The above example establishes the general structure of the ReachTooSmall procedure: we collect at-least-one constraints over auxiliary position variables until we have more values than variables, and then add recovered at-most-one constraints to these to obtain contradiction. However, the specifics of this depended on us starting from the 0 vertex, as this is required in the encoding to have the fixed "position" 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

(a) Values of P_i for each i.

(b) Values of $\mathsf{dist}(2, i)$ for each i.

Figure 6.6: Comparison of position and shifted position labelling variables.

justifications from the previous section to work, we need to be able to run ReachTooSmall from an arbitrary vertex, and so we need a way to start the breadth-first search for possible positions without necessarily knowing what the position of the first node might be.

One potential way to achieve this could be to modify the PB encoding to omit the arbitrary choice of $P_0 = 0$, and then introduce a constraint $P_i = 0$ without loss of generality using *VeriPB*'s *dominance* rule in the proof as needed.

However, it is actually possible to make the analogous argument for another vertex using only (further) extension variables, and due to the difficulty of introducing and proving a new dominance *preorder* for every inference, especially in the context of *VeriPB*'s requirements for order change during the proof (namely that deletions of derived constraints after changing the preorder must be *checked* [32]), we exhibit here the extension variable version.

The idea is to conceptually introduce a new set of position labels $\{\mathsf{dist}(r, i) : 1 \leq i \leq n\}$ for each 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 $P_r = 0$. Or another way of thinking of these is that they represent the *distance* in terms of number of edges between two given vertices. Figure 6.6 demonstrates the required values for $\mathsf{dist}(2, i)$ labels compared with the P_i for an example assignment on six variables.

Specifically, we want to ensure that $\mathsf{dist}(r, i) = P_i - P_r \mod n$. This preserves the useful property that if $X_i = j$ then $\mathsf{dist}(r, j) = \mathsf{dist}(r, i) + 1 \mod n$, as is true for the p variables. By construction, we must have $\mathsf{dist}(r, r) = 0$, and so we should be able to collect sets of possible $\mathsf{dist}(r, i)$ variables for each subsequent value and use this to construct our ReachTooSmall argument similar to the worked example.

It is not necessary, however, to introduce a set of bit variables to specifically represent each $\mathsf{dist}(r, i)$. Since we just need to define 0-1 flags for the at-most-one and at-least-one constraints used by ReachTooSmall, we can reify atomic-literal-like flags directly on constraints over $\mathsf{BinEnc}(P_r)$ and $\mathsf{BinEnc}(P_i)$, ensuring they have the intended meaning by definition. This requires some further extensional variables which we will call *precedence flags*: $\mathsf{prec}(i, j)$ should be a 0-1 variable that is true if $P_i < P_j$ and false if $P_j < P_i$. So overall the definitions we require for a distance equality literal $\mathsf{dist}(r, i)_{=k}$ (where $r \neq i$) are as follows.

$$\begin{aligned}
\mathsf{Def}_{\Rightarrow}(\mathsf{prec}(r,i)) &:= \mathsf{prec}(r,i) \Rightarrow & \mathsf{BinEnc}(P_i) - \mathsf{BinEnc}(P_r) \geq 1;\\
\mathsf{Def}_{\Leftarrow}(\mathsf{prec}(r,i)) &:= \overline{\mathsf{prec}(r,i)} \Rightarrow & -\mathsf{BinEnc}(P_i) + \mathsf{BinEnc}(P_r) \geq 1;\\
\mathsf{Def}_{\Rightarrow}(\mathsf{dist}(r,i)_{\geq k}) &:= \mathsf{dist}(r,i)_{\geq k} \Rightarrow & \mathsf{BinEnc}(P_i) - \mathsf{BinEnc}(P_r) + n \cdot \mathsf{prec}(i,r) \geq k;\\
\mathsf{Def}_{\Leftarrow}(\mathsf{dist}(r,i)_{\geq k}) &:= \overline{\mathsf{dist}(r,i)}_{\geq k} \Rightarrow & -\mathsf{BinEnc}(P_i) + \mathsf{BinEnc}(P_r) - n \cdot \mathsf{prec}(i,r) \geq -k+1.\\
\mathsf{Def}_{\Rightarrow}(\mathsf{dist}(r,i)_{=k}) &:= \mathsf{dist}(r,i)_{=k} \Rightarrow & \mathsf{dist}(r,i)_{\geq k} + \overline{\mathsf{dist}(r,i)}_{\geq k+1} \geq 2;\\
\mathsf{Def}_{\Leftarrow}(\mathsf{dist}(r,i)_{=k}) &:= \overline{\mathsf{dist}(r,i)}_{=k} \Rightarrow & \overline{\mathsf{dist}(r,i)}_{\geq k} + \mathsf{dist}(r,i)_{\geq k+1} \geq 1.
\end{aligned}$$

In the case where $r = i$, a $\mathsf{prec}(r,i)$ flag is not needed, since we can just define $\mathsf{dist}(i,r)_{\geq k}$ to be trivially true or false depending on whether $k = 0$.

All of the above constraints can be introduced by redundance-based strengthening, however, for the precedence flags, the proof obligation is somewhat more tricky than the usual extension variable introduction. This because we are effectively relying on a NotEquals constraint on the P_i and P_r variables when we define $\mathsf{prec}(r,i)$ as above. So we will break down the method for introducing these flags into several subprocedures. Note that from this point onwards, once we have defined a justification procedure or subprocedure it will be an implicit precondition of later procedures that we have this procedure available, and its required conditions are met. We also omit the implicit preconditions that we have a Circuit constraint on variables $\mathcal{X} := X_0, \ldots, X_{n-1}$, encoded as in Encoding Procedure 6.1, and that any atomic literals over P_i variables are introduced as unique extension variables with corresponding definition constraints.

First, we need a subprocedure for recovering a constraint saying that each P_j variable can take at most one value out of some ordered set S.

Justification Subprocedure 6.7 (Recovering an at-most-1 over position variables).
Preconditions: $S := (v_1, \ldots, v_d)$; $v_i \in \{0, \ldots, n-1\}$ *and* $v_i < v_{i+1}$ *for each* $i \in \{1, \ldots, d\}$.
Procedure:

```
1: proc PosVarsRecoverAtMost1(S, j)
2:     AtMost1s[j] ← pbc  Σ_{i=1}^{d} −p_{j=v_i} ≥ −1
3:     subproof of Σ_{i=1}^{d} −p_{j=v_i} ≥ −1
4:         for i ∈ {1, ..., d}
5:             rup  p̄_{j=v_i} ≥ 1
6:     return AtMost1s[j]
```

Correctness Proof. For the **pbc** rule, we need to derive contradiction from $\neg\mathsf{AtMost1s}[j] = \sum_{i=1}^{d} p_{j=v_i} \geq 2$. From Theorem 2.8 each $p_{k=v_i}$ propagates $\overline{p}_{k=v_k}$ for every $j \neq k$ which

contradicts $\neg\mathsf{AtMost1s}[k]$. So all the constraints on line 5 are indeed RUP. Once they are in place, unit propagation reduces $\neg\mathsf{AtMost1s}[j]$ to $p_{j=v_d} \geq 2$, a contradiction. So this is sufficient to establish the proof obligation. □

Justification Subprocedure 6.8 (Recovering all-different over position variables).
Preconditions: *No further assumptions required.*
Procedure:

```
1:  proc PosVarsRecoverAllDiff( )
2:      AtLeast1s[0] ← rup p_{0=0} ≥ 1
3:      AtMost1s[0] ← get  − p_{0=0} ≥ −1
4:      for k ∈ {1, ..., n − 1}
5:          for i ∈ {0, ..., n − 1}
6:              for j ∈ {0, ..., n − 1}
7:                  C_ij ← rup p_{i=k−1} ∧ x_{i=j} ⇒ p_{j=k} ≥ 1
8:              C_i ← rup p_{i=k−1} ⇒ ∑_j p_{j=k} ≥ 1
9:              if k = 1 break
10:         AtLeast1s[k] := ∑_i p_{i=k} ≥ 1 ← cut sat(∑_i C_i + AtLeast1s[k − 1])
11:         AtMost1s[k] ← PosVarsRecoverAtMost1(k, (0, ..., n − 1))
12:     return AtLeast1s, AtMost1s
```

Correctness Proof. The constraint on Line 2 is RUP from the P_0 definition, and the constraint on line 3 is a literal axiom. Each RUP step on line 7 is immediate from the literal definitions and subcycle elimination encoding (line 7 in Encoding Procedure 6.1, and Theorem 2.8). Then each step on line 8 is RUP, since its negation will propagate $p_{i=k}$, and $\overline{p}_{j=k}$ for all j, which will cause the constraints derived on line 7 to propagate $\overline{x}_{i=j}$ for all j, leading to contradiction by Theorem 3.2. Line 10 is then performing a resolution step with the clauses derived on line 8 and the previously derived at-least-one constraint. So this will indeed produce the stated clause. The derivation of the AtMost1s follows directly from the correctness of Justification Subprocedure 6.7. □

This last subprocedure is somewhat expensive, requiring $O(n^3)$ steps. However, as we will see, it only needs to be executed once at the start of the proof, so that the derived AtLeast1s and AtMost1s are available for any other procedures that need them. The mechanism for defining $\mathsf{prec}(r, i)$ variables is then given by Justification Subprocedure 6.9 below.

Correctness Proof for Justification Subprocedure 6.9. C_1 can be introduced as per the usual extension rule application of redundance based strengthening Theorem 2.4. But C_2 is not in the form for the usual reverse implication introduction, and so subproofs may be needed. The only constraint touched by the witness ω is C_1, hence the only proof obligations (from the definition

Justification Subprocedure 6.9 (Defining precedence flags).

Preconditions: AtLeast1s *and* AtMost1s *constraints have already been derived, as per Justification Subprocedure 6.8;* Defined *is a map that keeps track of which extension variables are defined;* $r, i \in \{0, \dots, n-1\}$ *with* $r \neq i$.

Procedure:

1: **proc** DefinePrecFlag$(r, i, \mathsf{Defined})$
2: **if** $r > i$ **return** DefinePrecFlag$(i, r, \mathsf{Defined})$
3: **if** Defined$(\mathsf{prec}(r,i))$ **return** $\mathsf{prec}(r,i)$
4: $C_1 \leftarrow$ **red** $\mathsf{prec}(r,i) \Rightarrow \mathsf{BinEnc}(P_i) - \mathsf{BinEnc}(P_r) \geq 1$ **with** $\mathsf{prec}(r,i) \mapsto 0$
5: $C_2 \leftarrow$ **red** $\overline{\mathsf{prec}(r,i)} \Rightarrow -\mathsf{BinEnc}(P_i) + \mathsf{BinEnc}(P_r) \geq 1$ **with** $\omega := \mathsf{prec}(r,i) \mapsto 1$
6: **subproof of** $C_1\restriction_\omega = \mathsf{BinEnc}(P_i) - \mathsf{BinEnc}(P_r) \geq 1$
7: **for** $k \in \{0, \dots, n-1\}$
8: $D_k \leftarrow$ **rup** $p_{i=k} \Rightarrow p_{r=k} \geq 1$
9: **cut** $\sum_{k=0}^{n-1} \mathsf{sat}(\mathsf{AtLeast1s}[k] + D_k) + \sum_{\substack{k=0 \\ k\neq i}}^{n-1} \mathsf{AtMost1s}[k]$
10: Defined $\leftarrow$ Defined $\cup$ $(\mathsf{prec}(r,i), \mathit{true})$
11: **return** $\mathsf{prec}(r,i)$

of redundance-based strengthening Rule 8) are $C_2\restriction_\omega$, which is trivial and so does not need a subproof, and $C_1\restriction_\omega$, for which a subproof is provided.

Within the subproof context we have in addition to previously derived constraints, the temporary constraints

$$\neg C_2 = -K_1 \cdot \mathsf{prec}(r,i) + \mathsf{BinEnc}(P_i) - \mathsf{BinEnc}(P_r) \geq 0; \text{ and} \tag{6.26}$$

$$\neg C_1\restriction_\omega = -\mathsf{BinEnc}(P_i) + \mathsf{BinEnc}(P_r) \geq 0. \tag{6.27}$$

(recall our notation K_t introduced in Section 6.3 for reification coefficients) and from these a conflicting constraint must be derived. Note that under unit propagation of these we have $\mathsf{prec}(r,i) = 0$; so these are equivalent to the encoding of $P_i = P_r$. Hence, the conditions of Justification Procedure 3.12 are met and each constraint D_k indeed follows by RUP.

Then finally, for each k, the result of $\mathsf{sat}(\mathsf{AtLeast1s}[k] + D_k)$ is

$$\sum_{\substack{j=0 \\ j\neq i}}^{n-1} p_{j=k} \geq 1 \tag{6.28}$$

so the cutting planes operation on line 9 will derive $0 \geq 1$ as required. □

With this in place, we can define our subprocedure for introducing the distance/shifted-

position flags.

Justification Subprocedure 6.10 (Defining distance flags).
Preconditions: Defined *is a map that keeps track of which literals are defined.*
Procedure:

proc DefineDistEquals$(r, i, k,$ Defined$)$
  if Defined$(\mathsf{dist}(r,i)_{=k})$
    return $\mathsf{dist}(r,i)_{=k}$
  for $k' \in \{k, k+1\}$
    if not Defined$(\mathsf{dist}(r,i)_{\geq k'})$
      if $r = i \wedge k' = 0$
        ext $\overline{\mathsf{dist}(r,i)}_{\geq k} \Leftrightarrow 0 \geq 1$
      else if $r = i \wedge k' \neq 0$
        ext $\mathsf{dist}(r,i)_{\geq k} \Leftrightarrow 0 \geq 1$
      $\ell \leftarrow$ DefinePrecFlag$(i, r,$ Defined$)$
      ext $\mathsf{dist}(r,i)_{\geq k'} \Leftrightarrow \mathsf{BinEnc}(P_i) - \mathsf{BinEnc}(P_r) + n \cdot \ell \geq k'$
      Defined $\leftarrow$ Defined $\cup$ $(\mathsf{dist}(r,i)_{\geq k}, \mathit{true})$
  ext $\mathsf{dist}(r,i)_{=k} \Leftrightarrow \mathsf{dist}(r,i)_{\geq k} + \overline{\mathsf{dist}(r,i)}_{\geq k+1} \geq 2$
  Defined $\leftarrow$ Defined $\cup$ $(\mathsf{dist}(r,i)_{=k}, \mathit{true})$
  return $\mathsf{dist}(r,i)_{=k}$

Correctness Proof. Follows directly from Theorem 2.4, since we only make use of the extension rule. □

Since this procedure is effectively just defining literals as extension variables, we adopt the same convention as with ordinary atomic literals and assume in further procedures that if a flag for $\mathsf{dist}(r,i)_{=k}$ appears at any point in a procedure, that DefineDistEquals has been called at some point to retrieve or define it. The procedures are also intentionally written in such a way that only one of $\mathsf{prec}(r,i)$ and $\mathsf{prec}(i,r)$ will ever be defined, so we will also assume in what follows that names are canonicalised, and e.g. $\mathsf{prec}(i,r)$ can be substituted with $\overline{\mathsf{prec}(r,i)}$ wherever necessary.

Now that we have these distance and precedence flags, we can begin to define in full the subprocedures actually need to build ReachTooSmall. First, we have a somewhat involved process, detailed below in Justification Subprocedure 6.11 for deriving the constraints of the form

$$\mathsf{dist}(r,i)_{=k} \wedge x_{i=j} \Rightarrow \mathsf{dist}(r,r)_{=k+1} \geq 1. \tag{6.29}$$

These are intuitively true based on the definitions of the extension variables: if the vertex i is k steps away from the vertex r, and the successor of i is j, the vertex j must be $k+1$ steps from the vertex r. In the worked example with $r = 0$ analogous constraints were simply RUP, but

with arbitrary distance flags it involves more work, requiring somewhat tedious cutting planes additions to correctly cancel P_i variables in the definitions of the $\mathsf{dist}(r,i)_{\geq k}$ variables. When r is in fact 0 we can use the $p_{i=k}$ variables instead of $\mathsf{dist}(0,i)_{=k}$ variables, but for consistency we will use the latter as an alias for the former. We also need to deal separately with the degenerate case where $r = i$, since $\mathsf{prec}(r,r)$ is never defined, and we defined distance flags differently.

Justification Subprocedure 6.11 (Distance from r sucessor relation).
Preconditions: $k < n-1$; $i,j,k \in \{0,\ldots,n-1\}$ *with* $i \neq j$, *and if* $i = r$, $k = 0$.
Procedure: *See Justification Procedure 6.11 (cont.) below.*

Correctness Proof. Let us first deal with the RUP constraints in the easy cases.

If $r = 0$ and $j \neq 0$, then the negation of the constraint on line 3 will propagate $x_{i=j}$; and $\mathsf{dist}(0,i)_{=k}$; a.k.a. $p_{i=k}$. Either this is immediately contradictory (when $i = 0$ and $k \neq 0$) or it causes propagation such that $\mathsf{BinEnc}(P_i) = k$ and then $\mathsf{BinEnc}(P_j) = k+1$ from Encoding Procedure 6.1, contradicting $\overline{\mathsf{dist}(0,j)}_{=k+1}$.

On the other hand, if $j = 0$ and $r = 0$, then $x_{i=j}$ propagates from the negation of the constraint on line 5, causing propagation resulting in $\mathsf{BinEnc}(P_i) = n-1$ which conflicts with propagation setting $\mathsf{BinEnc}(P_i) = k$ resulting from $\mathsf{dist}(0,i)_{=k}$ a.k.a. $p_{i=k}$, since $k < n-1$.

Now we turn to the more awkward cases when $r \neq 0$. First, if $i = r$, $j \neq 0$ and $k = 0$, we have for C_3,

$$\begin{aligned}
&\mathsf{Def}_{\Rightarrow}(\mathsf{dist}(r,j)_{\geq 2}) = K_2 \cdot \overline{\mathsf{dist}(r,j)}_{\geq 2} + P_j - P_i + n \cdot \mathsf{prec}(j,r) && \geq 2;\\
+\quad & C_1 = K_{-1} \cdot \overline{x}_{i=j} \qquad - P_j + P_i && \geq -1;\\
\hline
&= K_2 \cdot \overline{\mathsf{dist}(r,j)}_{\geq 2} + K_{-1} \cdot \overline{x}_{i=j} + n \cdot \mathsf{prec}(j,r) \geq 1.
\end{aligned}$$

And then for C_4,

$$\begin{aligned}
&\mathsf{Def}_{\Leftarrow}(\mathsf{dist}(r,j)_{\geq 1}) = K_n \cdot \mathsf{dist}(r,j)_{\geq 1} - P_j + P_i - n \cdot \mathsf{prec}(j,r) && \geq 0;\\
+\quad & C_2 = K_1 \cdot \overline{x}_{i=j} \qquad + P_j - P_i && \geq 1;\\
\hline
&= \qquad K_2 \cdot \mathsf{dist}(r,j)_{\geq 1} + K_{-1} \cdot \overline{x}_{i=j} - n \cdot \mathsf{prec}(j,r) \geq 1.
\end{aligned}$$

Justification Subprocedure 6.11 (continued)

proc DeriveSucessorDistance(i, j, r, k)
  if $(r = 0 \wedge j \neq 0)$
    return rup $x_{i=j} \wedge \mathsf{dist}(r,i)_{=k} \Rightarrow \mathsf{dist}(r,j)_{=k+1}$
  else if $(r = 0 \wedge j = 0)$
    return rup $x_{i=j} \wedge \mathsf{dist}(r,i)_{=k} \Rightarrow 0 \geq 1$

  if $i = r$
    if $j \neq 0 \wedge k = 0$
      $C_1 \leftarrow x_{i=j} \Rightarrow -\mathsf{BinEnc}(P_j) + \mathsf{BinEnc}(P_i) \geq -1$
      $C_2 \leftarrow x_{i=j} \Rightarrow \mathsf{BinEnc}(P_j) - \mathsf{BinEnc}(P_i) \geq 1$
      $C_3 \leftarrow$ **cut** $\mathsf{Def}_{\Rightarrow}(\mathsf{dist}(r,j)_{\geq 2}) + C_1$
      $C_4 \leftarrow$ **cut** $\mathsf{Def}_{\Leftarrow}(\mathsf{dist}(r,j)_{\geq 1}) + C_2$
      $C_5 \leftarrow$ **cut** $C_2 + \mathsf{Def}_{\Rightarrow}(\mathsf{prec}(j,i))$
    return rup $x_{i=j} \wedge \mathsf{dist}(r,i)_{=k} \Rightarrow \mathsf{dist}(r,j)_{=k+1}$

  if $j \neq r$
    if $j \neq 0$
      $C_1 \leftarrow$ **get** $x_{i=j} \Rightarrow -\mathsf{BinEnc}(P_j) + \mathsf{BinEnc}(P_i) \geq -1$
      $C_2 \leftarrow$ **get** $x_{i=j} \Rightarrow \mathsf{BinEnc}(P_j) - \mathsf{BinEnc}(P_i) \geq 1$
      $C_3 \leftarrow$ **cut** $\mathsf{sat}(C_1 + \mathsf{Def}_{\Rightarrow}(\mathsf{prec}(i,r)) + \mathsf{Def}_{\Leftarrow}(\mathsf{prec}(j,r)))$
      $C_4 \leftarrow$ **cut** $\mathsf{sat}(C_2 + \mathsf{Def}_{\Leftarrow}(\mathsf{prec}(i,r)) + \mathsf{Def}_{\Rightarrow}(\mathsf{prec}(j,r)))/3$
    else
      $C_1 \leftarrow$ **get** $x_{i=j} \Rightarrow -\mathsf{BinEnc}(P_j) + \mathsf{BinEnc}(P_i) \geq n-1$
      $C_2 \leftarrow$ **get** $x_{i=j} \Rightarrow \mathsf{BinEnc}(P_j) - \mathsf{BinEnc}(P_i) \geq -n+1$
      $C_7 \leftarrow$ **rup** $x_{i=j} \Rightarrow \overline{\mathsf{prec}(i,r)} \geq 1$
      $C_8 \leftarrow$ **rup** $x_{i=j} \Rightarrow \mathsf{prec}(j,r) \geq 1$
      $C_3 \leftarrow$ **cut** $C_5 + C_6$
      $C_4 \leftarrow$ **rup** $x_{i=j} \Rightarrow \mathsf{prec}(i,r) + \overline{\mathsf{prec}(j,r)} \geq 0$
    $C_5 \leftarrow$ **cut** $n \cdot C_3 + C_2 + \mathsf{Def}_{\Rightarrow}(\mathsf{dist}(r,i)_{\geq k}) + \mathsf{Def}_{\Leftarrow}(\mathsf{dist}(r,j)_{\geq k+1}$
    $C_6 \leftarrow$ **cut** $n \cdot C_4 + C_1 + \mathsf{Def}_{\Leftarrow}(\mathsf{dist}(r,i)_{\geq k+1}) + \mathsf{Def}_{\Rightarrow}(\mathsf{dist}(r,j)_{\geq k+2}$
    return rup $\mathsf{dist}(r,i)_{=k} \wedge x_{i=j} \Rightarrow \mathsf{dist}(r,j)_{=k+1}$
  else
    $C_1 \leftarrow$ **get** $x_{i=j} \Rightarrow -\mathsf{BinEnc}(P_j) + \mathsf{BinEnc}(P_i) \geq -1$
    $C_2 \leftarrow$ **get** $x_{i=j} \Rightarrow \mathsf{BinEnc}(P_j) - \mathsf{BinEnc}(P_i) \geq 1$
    $C_3 \leftarrow$ **cut** $\mathsf{sat}(C_1 + \mathsf{Def}_{\Leftarrow}(\mathsf{dist}(r,i)_{\geq k+1}))$
    $C_4 \leftarrow$ **cut** $\mathsf{sat}(C_2 + \mathsf{Def}_{\Rightarrow}(\mathsf{dist}(r,i)_{\geq k}))$
    return rup $\mathsf{dist}(r,i)_{=k} \wedge x_{i=j} \Rightarrow 0 \geq 1.$

And for C_5,

$$\begin{aligned} C_2 = K_1 \cdot \overline{x}_{i=j} \quad + P_j - P_i \quad &\geq 1; \\ + \ \mathsf{Def}_{\Rightarrow}(\mathsf{prec}(j,i)) = K_1 \cdot \overline{\mathsf{prec}(j,i)} + P_i - P_j \quad &\geq 1; \\ \hline = \quad K_1 \cdot \overline{x}_{i=j} + K_1 \cdot \overline{\mathsf{prec}(j,i)}. \quad &\geq 2 \end{aligned}$$

These three constraints ensure that, from the negation of the RUP constraint on line 13, propagation of $x_{i=j}$ propagates $\overline{\mathsf{prec}(j,i)}$ (from C_5), which then, since $j = i$ causes $\overline{\mathsf{dist}(r,j)}_{\geq 2}$ and $\mathsf{dist}(r,j)_{\geq 1}$ to propagate from C_3 and C_4 respectively, contradicting $\overline{\mathsf{dist}(r,j)}_{=k+1} = \overline{\mathsf{dist}(r,j)}_{=1}$.

If $i = r$ and $k \neq 0$ then the constraint on line 13 is instead trivially RUP, since $\mathsf{dist}(r,j)_{\geq k}$ by definition propagates a contradiction (see Justification Subprocedure 6.10).

If $i = r$ and $j = 0$ the constraint on line 13 is also immediately RUP, since, after propagation of $x_{i=j}$, $P_i = P_r$ is fixed at $n-1$ and $\mathsf{dist}(i,j)_{=1}$ propagates from the definitions.

So from this point we can assume $i \neq r$ in addition to $r \neq 0$. Now suppose $j \neq r$ and $j \neq 0$. We have

$$\begin{aligned} C_1 = K_{-1} \cdot \overline{x}_{i=j} \quad &- P_j + P_i \geq -1; \\ + \ \mathsf{Def}_{\Rightarrow}(\mathsf{prec}(i,r)) = K_1 \cdot \overline{\mathsf{prec}(i,r)} \quad &+ P_r - P_i \geq 1; \\ + \ \mathsf{Def}_{\Leftarrow}(\mathsf{prec}(j,r)) = K_1 \cdot \mathsf{prec}(j,r) \quad &- P_r + P_j \geq 1; \\ \hline = K_{-1} \cdot \overline{x}_{i=j} + K_1 \cdot \overline{\mathsf{prec}(i,r)} \ + \ K_1 \cdot \mathsf{prec}(j,r) &\geq 1. \end{aligned}$$

So, after saturation

$$\begin{aligned} C_3 &= \overline{x}_{i=j} + \overline{\mathsf{prec}(i,r)} + \mathsf{prec}(j,r) \geq 1 \\ &= \overline{x}_{i=j} - \mathsf{prec}(i,r) + \mathsf{prec}(j,r) \geq 0. \end{aligned}$$

Similarly,

$$\begin{aligned} C_2 = K_1 \cdot \overline{x}_{i=j} \quad &P_j - P_i \geq 1; \\ + \ \mathsf{Def}_{\Leftarrow}(\mathsf{prec}(i,r)) = K_1 \cdot \mathsf{prec}(i,r) \quad &- P_r + P_i \geq 1; \\ + \ \mathsf{Def}_{\Rightarrow}(\mathsf{prec}(j,r)) = K_1 \cdot \overline{\mathsf{prec}(j,r)} \quad &+ P_r - P_j \geq 1; \\ \hline = \quad K_1 \cdot \overline{x}_{i=j} + K_1 \cdot \mathsf{prec}(i,r) + K_1 \cdot \overline{\mathsf{prec}(j,r)} &\geq 3. \end{aligned}$$

So, after saturation, and division

$$\begin{aligned} C_4 &= \overline{x}_{i=j} + \mathsf{prec}(i,r) + \overline{\mathsf{prec}(j,r)} \geq 1 \\ &= \overline{x}_{i=j} + \mathsf{prec}(i,r) - \mathsf{prec}(j,r) \geq 0. \end{aligned}$$

Then for line 27,

$$\begin{array}{rrlll} & n \cdot C_3 = n \cdot \overline{x}_{i=j} & & -n{\cdot}\mathsf{prec}(i,r){+}n \cdot \mathsf{prec}(j,r) & \geq 0; \\ + & C_2 = K_1 \cdot \overline{x}_{i=j} & + P_j - P_i & & \geq 1; \\ + & \mathsf{Def}_{\Rightarrow}(\mathsf{dist}(r,i)_{\geq k}) = K_k \cdot \overline{\mathsf{dist}(r,i)}_{\geq k} & + P_i - P_r{+}\ n{\cdot}\mathsf{prec}(i,r) & & \geq k; \\ + & \mathsf{Def}_{\Leftarrow}(\mathsf{dist}(r,j)_{\geq k+1}) = K_{n-k} \cdot \mathsf{dist}(r,i)_{\geq k+1} & - P_j + P_r & -n \cdot \mathsf{prec}(i,r) & \geq -k; \\ \hline & = & K_{1+n} \cdot \overline{x}_{i=j} + K_k \cdot \overline{\mathsf{dist}(r,i)}_{\geq k} + K_{n-k} \cdot \mathsf{dist}(r,i)_{\geq k+1} & & \geq 1. \end{array}$$

So after saturation,

$$C_5 = \overline{x}_{i=j} + \overline{\mathsf{dist}(r,i)}_{\geq k} + \mathsf{dist}(r,i)_{\geq k+1} \geq 1.$$

Similarly, for line 28

$$\begin{array}{rrlll} & n \cdot C_4 = n \cdot \overline{x}_{i=j} & & +n{\cdot}\mathsf{prec}(i,r){-}n \cdot \mathsf{prec}(j,r) & \geq 0; \\ + & C_1 = K_{-1} \cdot \overline{x}_{i=j} & - P_j + P_i & & \geq -1; \\ + & \mathsf{Def}_{\Leftarrow}(\mathsf{dist}(r,i)_{\geq k+1}) = K_{n-k} \cdot \mathsf{dist}(r,i)_{\geq k+1} & - P_i + P_r{-}\ n{\cdot}\mathsf{prec}(i,r) & & \geq -k; \\ + & \mathsf{Def}_{\Rightarrow}(\mathsf{dist}(r,j)_{\geq k+2}) = K_{k+2} \cdot \overline{\mathsf{dist}(r,j)}_{\geq k+2} & + P_j - P_r & +n \cdot \mathsf{prec}(j,r) & \geq k+2; \\ \hline & = & K_{-1+n} \cdot \overline{x}_{i=j} + K_{n-k} \cdot \mathsf{dist}(r,i)_{\geq k+1} + K_{k+2} \cdot \overline{\mathsf{dist}(r,i)}_{\geq k+2} & & \geq 1. \end{array}$$

So after saturation,

$$C_6 = \overline{x}_{i=j} + \mathsf{dist}(r,i)_{\geq k+1} + \overline{\mathsf{dist}(r,i)}_{\geq k+2} \geq 1.$$

With the constraints C_5 and C_6 in place, the final constraint derived on line 17 will be RUP.

Now consider the case where $j \neq r$ and $j = 0$. Due to the constraints from line 5 of Encoding Procedure 6.1, propagation of $x_{i=j}$ in this case will fix $P_i = n - 1$. This ensures C_7 and C_8 will be RUP by Theorem 2.7 as we would get contradictory bounds on P_r in each case. C_3 is therefore

$$2\overline{x}_{i=j} - \mathsf{prec}(i,r) + \mathsf{prec}(j,r) \geq 1, \tag{6.30}$$

while C_4 is trivially RUP and can be written as

$$\overline{x}_{i=j} + \mathsf{prec}(i,r) - \mathsf{prec}(j,r) \geq -1. \tag{6.31}$$

We can then run very similar calculations for lines 27 and 28.

$$\begin{aligned}
n \cdot C_3 &= 2n \cdot \overline{x}_{i=j} && -n{\cdot}\mathsf{prec}(i,r){+}n \cdot \mathsf{prec}(j,r) \geq n;\\
+ \quad C_2 &= K_{1-n} \cdot \overline{x}_{i=j} && + P_j - P_i && \geq 1-n;\\
+ \quad \mathsf{Def}_{\Rightarrow}(\mathsf{dist}(r,i)_{\geq k}) &= K_k \cdot \overline{\mathsf{dist}(r,i)}_{\geq k} && + P_i - P_r{+}\ n{\cdot}\mathsf{prec}(i,r) && \geq k;\\
+ \ \mathsf{Def}_{\Leftarrow}(\mathsf{dist}(r,j)_{\geq k+1}) &= K_{n-k} \cdot \mathsf{dist}(r,i)_{\geq k+1} && - P_j + P_r && -n \cdot \mathsf{prec}(i,r) \geq -k;\\
\hline
&= && K_{1+n} \cdot \overline{x}_{i=j} + K_k \cdot \overline{\mathsf{dist}(r,i)}_{\geq k} + K_{n-k} \cdot \mathsf{dist}(r,i)_{\geq k+1} \geq 1.
\end{aligned}$$

So after saturation,

$$C_5 = \overline{x}_{i=j} + \overline{\mathsf{dist}(r,i)}_{\geq k} + \mathsf{dist}(r,i)_{\geq k+1} \geq 1.$$

And:

$$\begin{aligned}
n \cdot C_4 &= n \cdot \overline{x}_{i=j} && +n{\cdot}\mathsf{prec}(i,r){-}n \cdot \mathsf{prec}(j,r) \geq -n;\\
+ \quad C_1 &= K_{n-1} \cdot \overline{x}_{i=j} && - P_j + P_i && \geq n-1;\\
+ \ \mathsf{Def}_{\Leftarrow}(\mathsf{dist}(r,i)_{\geq k+1}) &= K_{n-k} \cdot \mathsf{dist}(r,i)_{\geq k+1} && - P_i + P_r{-}\ n{\cdot}\mathsf{prec}(i,r) && \geq -k;\\
+ \ \mathsf{Def}_{\Rightarrow}(\mathsf{dist}(r,j)_{\geq k+2}) &= K_{k+2} \cdot \overline{\mathsf{dist}(r,i)}_{\geq k+2} && + P_j - P_r && +n \cdot \mathsf{prec}(i,r) \geq k+2;\\
\hline
&= && K_{2n-1} \cdot \overline{x}_{i=j} + K_{n-k} \cdot \mathsf{dist}(r,i)_{\geq k+1} + K_{k+2} \cdot \overline{\mathsf{dist}(r,i)}_{\geq k+2} \geq 1.
\end{aligned}$$

So again after saturation,

$$C_6 = \overline{x}_{i=j} + \mathsf{dist}(r,i)_{\geq k+1} + \overline{\mathsf{dist}(r,i)}_{\geq k+2} \geq 1.$$

And once again, C_5 and C_6 allow the final constraint to be RUP.

Finally, for the case where $j = r$ we have

$$\begin{aligned}
C_1 &= K_{-1} \cdot \overline{x}_{i=r} && - P_r + P_i && \geq -1;\\
+ \ \mathsf{Def}_{\Leftarrow}(\mathsf{dist}(r,i)_{\geq k+1}) &= K_{n-k} \cdot \mathsf{dist}(r,i)_{\geq k+1} && - P_i + P_r - n \cdot \mathsf{prec}(i,r) && \geq -k;\\
\hline
&= && K_1 \cdot \overline{x}_{i=r} + K_{n-k} \cdot \mathsf{dist}(r,i)_{\geq k} - n \cdot \mathsf{prec}(i,r) \geq -k-1;
\end{aligned}$$

and

$$\begin{aligned}
C_2 &= K_1 \cdot \overline{x}_{i=r} && + P_r - P_i && \geq 1;\\
+ \ \mathsf{Def}_{\Rightarrow}(\mathsf{dist}(r,i)_{\geq k}) &= K_k \cdot \overline{\mathsf{dist}(r,i)}_{\geq k} && + P_i - P_r + n \cdot \mathsf{prec}(i,r) && \geq k;\\
\hline
&= && K_1 \cdot \overline{x}_{i=r} + K_k \cdot \mathsf{dist}(r,i)_{\geq k} + n \cdot \mathsf{prec}(i,r) \geq k+1.
\end{aligned}$$

With these in place the negation of the constraint from line 35 propagates both $\mathsf{prec}(i,r)$ and $\overline{\mathsf{prec}(i,r)}$ (since $k < n$), and hence this will also be RUP. □

We also require procedures to derive at-most-one constraints over distance variables.

Justification Subprocedure 6.12 (Recovering at-most-one distance from r).
Preconditions: $S := (v_1, \ldots, v_d)$; $v_i \in \{0, \ldots, n-1)$ *and* $v_i < v_{i+1}$ *for each* $i \in \{1, \ldots, d\}$; $r \neq 0$.
Procedure:

```
1:  proc DistVarsRecoverAtMost1(r, S, j)
2:      if r = j
3:          AtMost1s[k] ← rup  Σ_{i=1}^{d} −dist(r,j)_{=v_i} ≥ −1
4:          return AtMost1s[k]
5:      AtMost1s[k] ← pbc  Σ_{i=1}^{d} −dist(r,j)_{=v_i} ≥ −1
6:      subproof of Σ_{i=1}^{d} −dist(r,j)_{=v_i} ≥ −1
7:          for i ∈ {1, ..., d−1}
8:              cut  Def_⇒(dist(r,j)_{≥v_i}) + Def_⇐(dist(r,j)_{≥v_{i+1}})
9:              cut  Def_⇒(dist(r,j)_{≥v_i+1}) + Def_⇐(dist(r,j)_{≥v_{i+1}+1})
10:         for i ∈ {1, ..., d}
11:             rup  ¬dist(r,j)_{=v_i} ≥ 1
12:     return AtMost1s[k]
```

Correctness Proof. First, if $r = k$, then $\mathsf{dist}(r,j)_{=v_i}$ is true by unit propagation for $v_i = 0$ and false for any other v_i. So the constraint from line 3 will be trivially RUP.

Otherwise, $r \neq k$. As in the proof for Justification Subprocedure 6.7, our only proof obligation for the "pbc" step is to derive contradiction from $\sum_{i=1}^{d} \mathsf{dist}(r,j)_{=v_i} \geq 2$.

For each $i \in \{1, \ldots, d\}$ the cutting planes steps compute

$$K_{v_i} \cdot \overline{\mathsf{dist}(r,j)}_{\geq v_i} + K_{n+v_{i+1}} \cdot \mathsf{dist}(r,j)_{\geq v_{i+1}} \geq (v_i - v_{i+1} + 1); \text{ and} \tag{6.32}$$

$$K_{v_i+1} \cdot \overline{\mathsf{dist}(r,j)}_{\geq v_i+1} + K_{n+v_{i+1}+1} \cdot \mathsf{dist}(r,j)_{\geq v_{i+1}+1} \geq (v_i - v_{i+1} + 1). \tag{6.33}$$

The presence of these two constraints for each value v_i ensures that for each constraint from line 11, propagation of $\mathsf{dist}(r,j)_{\geq v_i}$ causes every $\overline{\mathsf{dist}(r,j)}_{=v_k}$ to propagate for each $k \neq i$, contradicting the negation of the subproof constraint. This follows by the same argument as in the proof of Lemma 3.3. Once all these RUP constraints have been derived, we have a conflict by unit propagation, and the proof obligation is satisfied. □

Now we can finally detail the basic version of our ReachTooSmall procedure as Justification Subprocedure 6.13. We make use of the additional assumption that no variable has its own index in its domain (no self loops are possible), which follows immediately by unit propagation from the Circuit constraint definition.

Correctness Proof. First, since we meet the requirements of Justification Subprocedure 6.11,

Justification Subprocedure 6.13 (Reach is too small, with no assumptions).

Preconditions: SortAsTuple *converts a set to a sorted tuple;* dom_t *is a domain state such that for* $G := \mathsf{GraphRep}(\mathcal{X}, \mathrm{dom}_t)$, $|\mathsf{Reach}(r)| < |G|$*; and additionally* $i \notin \mathrm{dom}_t(X_i)$ *for any* i.

Procedure:

```
1:  proc ReachTooSmall(r, dom_t)
2:      AtLeast1s[0] ← rup dist(r, r)_{=0} ≥ 1
3:      allReached, lastReached ← {r}
4:      possibleDistances[r] ← {0}
5:      step ← 1
6:      while step ≤ |allReached|
7:          newlyReached ← ∅
8:          for i ∈ lastReached
9:              R_i ← GenericR_t(X_i)
10:             V ← dom_t(X_i)
11:             for j ∈ V
12:                 if j ≠ r
13:                     newlyReached ← newlyReached ∪ {j}
14:                     possibleDistances[j] ← possibleDistances[j] ∪ {step}
15:             C_ij ← DeriveSucessorDistance(i, j, r, step − 1)
16:         D ← rup R_i ⇒ Σ_{j∈V} x_{i=j} ≥ 1
17:         C_i ← cut D + Σ_j C_ij
18:     AtLeast1s[step] ← cut sat(AtLeast1s[step − 1] + Σ_i C_i)
19:     lastReached ← newlyReached
20:     allReached ← allReached ∪ newlyReached
21:     step ← step + 1

22:     for j ∈ allReached
23:         S ← SortAsTuple(possibleDistances[j])
24:         if r ≠ 0
25:             AtMost1s[j] ← DistVarsRecoverAtMost1(r, S, j)
26:         else
27:             AtMost1s[j] ← PosVarsRecoverAtMost1(S, j)

28:     cut Σ_{k=0}^{step−1} AtLeast1s[k] + Σ_{j∈allReached} AtMost1s[j]
```

each C_{ij} is by construction one of the forms

$$\overline{\mathsf{dist}(r,i)}_{=\mathsf{step}-1} + \overline{x}_{i=j} \geq 1 \quad \text{or} \quad \overline{\mathsf{dist}(r,i)}_{=\mathsf{step}-1} + \overline{x}_{i=r} + \mathsf{dist}(r,r)_{=\mathsf{step}} \geq 1. \tag{6.34}$$

Also, each constraint derived on line 16 will, as usual, be RUP by Theorem 3.2.

This means that (providing $\mathsf{step} \leq n-1$) the cutting planes operation on line 17 will cancel out the $x_{i=j}$ variables, and will result in C_i having the form

$$\mathcal{R}_i \wedge \mathsf{dist}(r,i)_{=\mathsf{step}-1} \Rightarrow \sum_{\substack{j \in \mathrm{dom}_t(X_i) \\ j \neq r}} \mathsf{dist}(r,j)_{=\mathsf{step}} \geq 1. \tag{6.35}$$

We can then argue inductively for $\mathsf{step} \leq n-1$ that at the start of each iteration $\mathsf{AtLeast1s}[\mathsf{step}-1]$ will always have the form

$$\mathcal{R}_{\mathsf{step}-1} \Rightarrow \sum_{i \in \mathsf{lastReached}} \mathsf{dist}(r,i)_{=\mathsf{step}-1} \geq 1. \tag{6.36}$$

This is initially true for the (trivially) RUP constraint derived on line 2, with $\mathcal{R}_0 = \emptyset$. Then, assuming for some $\mathsf{step} \leq n-1$ we have the required form, the cutting planes operation on line 18 will resolve each of the constraints of the form (6.35) with the last constraint of the form (6.36), resulting in

$$\mathcal{R}_{\mathsf{step}-1} \cup \{\mathcal{R}_i : i \in \mathsf{lastReached}\} \Rightarrow \sum_{j \in \mathsf{newlyReached}} \mathsf{dist}(r,j)_{=\mathsf{step}} \geq 1 \tag{6.37}$$

Letting $\mathcal{R}_{\mathsf{step}} := \mathcal{R}_{\mathsf{step}-1} \cup \{\mathcal{R}_i : i \in \mathsf{lastReached}\}$, once we have $\mathsf{lastReached} = \mathsf{newlyReached}$ and $\mathsf{step} = \mathsf{step}+1$ this is in the correct form for the next iteration.

The while loop terminates when $\mathsf{step} = |\mathsf{allReached}| + 1$. And by construction, we have $|\mathsf{allReached}| \leq |\mathsf{Reach}(r)|$ since we only collect nodes by following edges reachable from r. Since we are assuming $|\mathsf{Reach}(r)| < n$ we always have $\mathsf{step} \leq n-1$ during the loop body, and we must eventually terminate with $\mathsf{step} \leq n$.

The at-most-one constraints are correctly derived as per Justification Subprocedure 6.12 or Justification Subprocedure 6.7. And by construction they have, collectively, precisely the complementary literals to those appearing collectively in the at-least-one constraints.

This means that the final cutting planes summation is adding up step reified at-least-one constraints and $\mathsf{step}-1$ at-most-one constraints with complementary literals. Similar to Justification Procedure 3.16, this will result in $\mathcal{R} \Rightarrow 0 \geq 1$, where $\mathcal{R} := \bigcup_{i=0}^{\mathsf{step}-1} \mathcal{R}_i$. □

This procedure is sufficient for the conflict justifications constructed by Justification Procedure 6.2 and Justification Procedure 6.3, as by definition it fulfils the needs of property A1. However, to complete the justification procedures for the remainder of the filtering rules consid-

ered so far, we need to additionally respect A2, i.e. construct a propagation justification when a literal assumption implies the reach for a certain vertex in the represented graph is too small.

Justification Subprocedure 6.14 (Reach is too small, with an atomic literal assumption).

Preconditions: SortAsTuple *converts a set to a sorted tuple;* ℓ *is an atomic equality literal on a variable in* $\mathcal{X}$*;* dom_t *is a domain state such that for* $G := \mathsf{GraphRep}(\mathcal{X}, \mathrm{dom}_a)$*, with* dom_a *constructed using* ℓ *as in (6.5), we have* $|\mathsf{Reach}(r)| < |G|$.

Procedure: *Identical to Justification Subprocedure 6.13 except with an additional parameter* ℓ*, and between lines 11 and 12 we insert the following.*

12: **if** $\ell = \overline{x}_{i=j} \ \vee \ (\ell = x_{i=k} \wedge k \neq j) \ \vee \ (\ell = x_{k=j} \wedge k \neq i)$
13: $\quad \mathcal{R}_i \leftarrow \mathcal{R}_i \cup \{\ell\}$
14: $\quad V \leftarrow V \setminus \{j\}$
15: $\quad$ **continue**

Correctness Proof. Note that the added lines have the effect of excluding values not in dom_a, as defined by (6.5), from the breadth-first reachability search. Since by assumption, $|\mathsf{Reach}(r)| < |\mathsf{GraphRep}(\mathcal{X}, \mathrm{dom}_a)|$, all the of the arguments from the correctness of Justification Subprocedure 6.14 then apply, except possibly for the RUP constraint $\mathcal{R}_i \Rightarrow \sum_{j \in V} x_{i=j} \geq 1$ previously on line 16. Here we just need to observe for the case where $\ell = x_{k=j}$ and $k \neq i$ that reverse unit propagation of this constraint will exclude at least every value other than j from the domain of X_i. If conflict is then not already reached, further propagation would then fix $\mathsf{BinEnc}(X_i)$ at j as per Theorem 2.8 which then causes a conflict under propagation of ℓ due to the AllDifferent constraint encoded as part of Circuit. Hence, this constraint will still be RUP. □

6.4.4 Further Propagation Rules for Circuit

We have shown so far how a number of circuit pruning rules can be justified using a conditional counting argument formalised as the ReachTooSmall procedure. As mentioned, another pruning rule given by Francis and Stuckey [72] is the "prune skip" inference, which is illustrated in Figure 6.7. The subtrees explored by the DFS give us a required partial ordering over the vertices of G, and this is not respected by any edges that skip subtrees, allowing such edges to be pruned.

However, as illustrated in the diagram, even if we assume that an offending edge is taken, it can still be the case that every vertex is reachable from every other vertex, making ReachTooSmall not directly applicable.

In terms of once again reusing the same procedure, it might seem that hope is lost. But one encouraging observation here is that if we can somehow adapt ReachTooSmall to respect certain kinds of *ordering* assumptions then we can still in fact make use of a reachability argument.

Let $\mathsf{ord}(r, a, b)$ denote the property that, in a Hamiltonian cycle, the vertex a is visited *between* the vertices r and b. Or to think of it another way, the property saying that following any path from r, we must encounter a before we encounter b.

This can be straightforwardly expressed as a PB extension variable if we re-use the $\mathsf{prec}(i, j)$ flags we introduced as part of the distance variable definitions.

$$\mathsf{Def}_{\Rightarrow}(\mathsf{ord}(r, a, b)) := \mathsf{ord}(r, a, b) \Rightarrow \mathsf{prec}(r, a) + \mathsf{prec}(a, b) + \mathsf{prec}(b, r) \geq 2; \tag{6.38}$$

$$\mathsf{Def}_{\Leftarrow}(\mathsf{ord}(r, a, b)) := \overline{\mathsf{ord}(r, a, b)} \Rightarrow \overline{\mathsf{prec}(r, a)} + \overline{\mathsf{prec}(a, b)} + \overline{\mathsf{prec}(a, r)} \geq 1. \tag{6.39}$$

Note that, because there are only two possible cyclic orderings, $\overline{\mathsf{ord}(r, a, b)}$ is equivalent to $\mathsf{ord}(r, b, a)$. We can canonicalise literal names in the proof format so that equivalent sets of $\mathsf{prec}(i, r)$ and $\mathsf{ord}(r, i, j)$ literals are uniquely named in the actual proof log.

Now, we can define a new order-respecting reachable set for a directed graph G as the set

$$\mathsf{Reach}_{a \preceq b}(v) := \{w : \exists \text{ a path } P \text{ from } v \text{ to } w \text{ s.t. } a \preceq_P b\}, \tag{6.40}$$

where $a \preceq_P b$ denotes following the relation for a given path $P := (v_0, \ldots, v_p)$;

$$a \preceq_P b \iff \forall s (v_s = b \Rightarrow \exists r < s \text{ s.t. } v_r = a\}. \tag{6.41}$$

In other words $a \preceq_P b$ holds if j does not appear in the path P unless i appears earlier.

This allows us to express a new hope for our ReachTooSmall procedure, which should allow us to deal with the prune-skip rule.

A3. Suppose we have a Circuit constraint on the variables $\mathcal{X}$ encoded as per Encoding Procedure 6.1; and an atomic literal ℓ for a variable $X_i \in \mathcal{X}$. Let dom_a be the modified domain state as specified by (6.5). Then if $w \neq a \neq b$ are vertices of $G := \mathsf{GraphRep}(\mathcal{X}, \mathrm{dom}_a)$, such that $|\mathsf{Reach}_{a \preceq b}(w)| < |G|$, then $\mathsf{ReachTooSmall}(w, \mathrm{dom}_t, \ell, a, b)$ will derive $\mathcal{R} \wedge \mathsf{ord}(w, a, b) \Rightarrow \bar{\ell}$.

The key idea is that we run *two* ReachTooSmall arguments, assuming that they respect the above property. We then combine the results of these by adding up definition constraints to get a propagation justification. The intuition for why this works is that if an edge (v, w) skipping a subtree 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 skipped subtree, but also visit v_0 between visiting r and visiting v, and both of these cannot be simultaneously true.

Correctness Proof for Justification Procedure 6.15. First suppose $w \neq 0$. The cutting planes

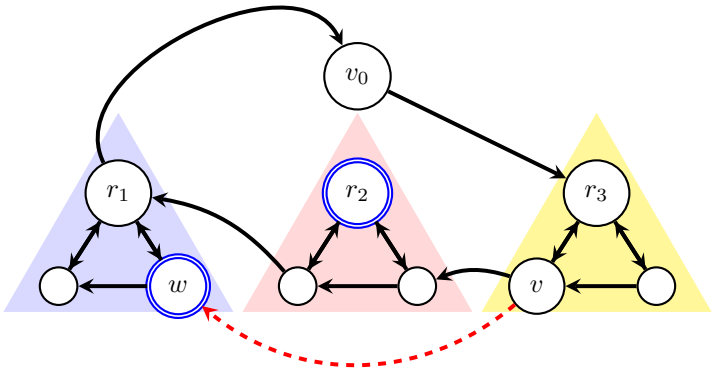


Figure 6.7: Justifying the "prune skip" inference. We can disprove the ordering assumption $\mathsf{ord}(w, r_2, v_0)$ with $\mathsf{ReachTooSmall}(w, \dots)$ and disprove the ordering assumption $\mathsf{ord}(r_2, v, v_0)$ with $\mathsf{ReachTooSmall}(r_2, \dots)$. These together imply that (v, w) cannot be used.

Justification Procedure 6.15 (Prune skip for Circuit).

Preconditions: *For some $i > 1$ there is an edge (v, w) in G beginning in the i_{th} subtree (explored by the DFS) and ending in the j_{th}, where $j > i + 1$, which can therefore be pruned; v_0 is the root the DFS; and r is the root of the $(i + 1)_{ith}$ visited subtree.*

Procedure:

1: $C_1 \leftarrow \mathcal{R}_1 \wedge \mathsf{ord}(w, r, v_0) \Rightarrow \overline{x}_{v=w} \geq 1 \leftarrow \mathsf{ReachTooSmall}(w, \mathrm{dom}_t, x_{v=w}, r, v_0)$
2: $C_2 \leftarrow \mathcal{R}_2 \wedge \mathsf{ord}(r, v, v_0) \Rightarrow \overline{x}_{v=w} \geq 1 \leftarrow \mathsf{ReachTooSmall}(r, \mathrm{dom}_t, x_{v=w}, r, v_0)$
3: **if** $w \neq 0$
4: $\quad C_3 \leftarrow$ **get** $x_{v=w} \Rightarrow \mathsf{BinEnc}P_w - \mathsf{BinEnc}P_v \geq 1$
5: $\quad C_4 \leftarrow$ **get** $x_{v=w} \Rightarrow \mathsf{BinEnc}{-P_w} + \mathsf{BinEnc}P_v \geq -1$
6: $\quad C_5 \leftarrow$ **cut** $\mathsf{sat}(C_3 + \mathsf{Def}_{\Rightarrow}(\mathsf{prec}(w, r)) + \mathsf{Def}_{\Rightarrow}(\mathsf{prec}(r, v)))/3$
7: $\quad C_6 \leftarrow$ **cut** $\mathsf{sat}(C_3 + \mathsf{Def}_{\Rightarrow}(\mathsf{prec}(v, v_0)) + \mathsf{Def}_{\Rightarrow}(\mathsf{prec}(v_0, w)))$
8: $\quad$ **cut** $C_5 + C_6 + \mathsf{Def}_{\Rightarrow}(\mathsf{ord}(w, r, v_0) + \mathsf{Def}_{\Rightarrow}(\mathsf{ord}(r, v, w))$
9: **rup** $\mathcal{R}_1 \cup \mathcal{R}_2 \Rightarrow x_{v=w} \geq 1$

steps can be computed as follows.

$$
\begin{aligned}
C_3 &= K_1 \cdot \overline{x}_{v=w} && +P_w - P_v \geq 1;\\
+\ \mathsf{Def}_{\Rightarrow}(\mathsf{prec}(w, r)) &= K_1 \cdot \overline{\mathsf{prec}(w, r)} && +P_r - P_w \geq 1;\\
+\ \mathsf{Def}_{\Rightarrow}(\mathsf{prec}(r, v)) &= K_1 \cdot \overline{\mathsf{prec}(r, v)} && +P_v - P_r \geq 1;\\
\hline
&= K_1 \cdot \overline{x}_{v=w} + K_1 \cdot \overline{\mathsf{prec}(w, r)} + K_1 \cdot \overline{\mathsf{prec}(r, v)} \geq 3.\\
\text{Hence,}\quad C_5 &= \overline{x}_{v=w} + \overline{\mathsf{prec}(w, r)} + \overline{\mathsf{prec}(r, v)} \geq 1.
\end{aligned}
$$

$$
\begin{aligned}
C_4 &= K_{-1} \cdot \overline{x}_{v=w} && -P_w + P_v \geq -1;\\
+\ \mathsf{Def}_{\Rightarrow}(\mathsf{prec}(v, v_0)) &= K_1 \cdot \overline{\mathsf{prec}(v, v_0)} && +P_{v_0} - P_v \geq 1;\\
+\ \mathsf{Def}_{\Rightarrow}(\mathsf{prec}(v_0, w)) &= K_1 \cdot \overline{\mathsf{prec}(v_0, w)} && +P_w - P_{v_0} \geq 1;\\
\hline
&= K_{-1} \cdot \overline{x}_{v=w} + K_1 \cdot \overline{\mathsf{prec}(v, v_0)} + K_1 \cdot \overline{\mathsf{prec}(v_0, w)} \geq 1.\\
\text{Hence,}\quad C_6 &= \overline{x}_{v=w} + \overline{\mathsf{prec}(v, v_0)} + \overline{\mathsf{prec}(v_0, w)} \geq 1
\end{aligned}
$$

So finally,

$$
\begin{aligned}
C_5 &= \overline{x}_{v=w} && - \mathsf{prec}(w, r) - \mathsf{prec}(r, v) && \geq -1;\\
+\quad C_6 &= \overline{x}_{v=w} && - \mathsf{prec}(v, v_0) - \mathsf{prec}(v_0, w) && \geq -1;\\
+\ \mathsf{Def}_{\Rightarrow}(\mathsf{ord}(w, r, v_0)) &= \overline{\mathsf{ord}(w, r, v_0)} && + \mathsf{prec}(w, r) + \mathsf{prec}(v_0, w) && \geq 2;\\
+\ \mathsf{Def}_{\Rightarrow}(\mathsf{ord}(r, v, v_0)) &= \overline{\mathsf{ord}(r, v, v_0)} && + \mathsf{prec}(v_0, r)\ + \mathsf{prec}(v, v_0) + \mathsf{prec}(r, v) && \geq 2;\\
\hline
&= && 2\overline{x}_{v=w} + \overline{\mathsf{ord}(w, r, v_0)} + \overline{\mathsf{ord}(r, v, v_0)} + \mathsf{prec}(r, v_0) + \mathsf{prec}(v_0, r) \geq 2.
\end{aligned}
$$

Recall that, since we are canonicalising names (c.f. Justification Subprocedure 6.9), we have $\mathsf{prec}(r, v_0) = \overline{\mathsf{prec}(v_0, r)}$. So this last constraint is in fact equivalent to

$$2\overline{x}_{v=w} + \overline{\mathsf{ord}(w, r, v_0)} + \overline{\mathsf{ord}(r, v, v_0)} \geq 2, \tag{6.42}$$

which, along with C_1 and C_2, allows the final constraint on line 9 to be RUP.

If, on the other hand $w = 0$, the situation is even simpler, since propagation of $x_{v=w}$ fixes $P_w = P_0 = 0$ and $P_v = n - 1$. So after propagation of $\mathsf{ord}(w, r, v_0)$ and $\mathsf{ord}(r, v, v_0)$, we have the following constraints active

$$\mathsf{prec}(0, r) + \mathsf{prec}(r, v_0) + \mathsf{prec}(v_0, 0) \geq 2; \tag{6.43}$$

$$\mathsf{prec}(r, v) + \mathsf{prec}(v, v_0) + \mathsf{prec}(v_0, r) \geq 2. \tag{6.44}$$

Here $\mathsf{prec}(0, r)$ and $\overline{\mathsf{prec}(v_0, 0)}$ must propagate, leading to $\mathsf{prec}(r, v_0)$ propagating. And since $\mathsf{prec}(r, v_0) = \overline{\mathsf{prec}(v_0, r)}$ we then propagate $\mathsf{prec}(v, v_0)$, ultimately resulting in the constraint $P_{v_0} \geq n$. Since we have in the encoding $P_{v_0} \leq n - 1$, this leads to contradiction via Theorem 2.7. □

In a moment we will demonstrate the final missing piece of this procedure, and show how ReachTooSmall can be adapted to additionally satisfy A3. Before that however, we can briefly note that, with the prune-skip rule in force, further propagation rules can be applied for Circuit, and yet again this can be justified using ReachTooSmall.

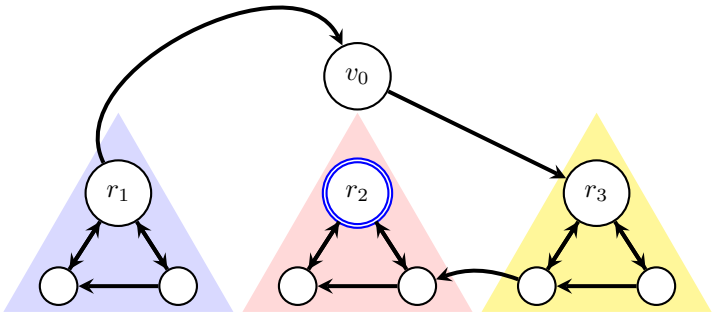


Figure 6.8: Justifying "no backedges" contradiction. There are no backedges from the subtree rooted at r_2, and since "prune skip" inferences have already been made, there is then no way to reach any nodes earlier than r_2 from r_2.

Justification Procedure 6.16 (Fix back-edges for Circuit).

Preconditions: *All edges which skip subtrees (as dealt with by Justification Procedure 6.15) have been pruned; all edges ending the root which did not begin in the earliest subtree (as dealt with by Justification Procedure 6.4) have been pruned; and for some $i > 1$, there is only one "backedge" (v, u) in G from the i_{th} subtree to a node in the $(i-1)_{th}$ subtree (which therefore must be used); and w is the root of the i_{th} subtree.*

Procedure:

ReachTooSmall$(\mathrm{dom}_t, w, \overline{x}_{v=u})$

Correctness Proof. Since any edges that skip subtrees have been removed by assumption, and the only edges left leading to the initial node v_0 come from the earliest subtree, the only way to escape the subtree rooted at w would be through the edge leading from the i_{th} subtree to the $(i-1)_{th}$ subtree.

So ReachTooSmall$(\mathrm{dom}_t, w, \overline{x}_{v=u})$ will derive $\mathcal{R} \Rightarrow x_{v=u}$ for some reason $\mathcal{R}$. □

Justification Procedure 6.17 (No back-edges for Circuit).

Preconditions: *All edges which skip subtrees (c.f. Justification Procedure 6.15) have been pruned; all edges ending the root which did not begin in the earliest subtree (as dealt with by Justification Procedure 6.4) have been pruned;for some $i > 1$, there are no "backedges" (v, u) in G from the i_{th} subtree to a node in the $(i-1)_{th}$ subtree (and hence* Circuit *is infeasible); and w is the root of the i_{th} subtree.*

Procedure:

ReachTooSmall(dom_t, w)

Correctness Proof. Analogous to Justification Procedure 6.16. See Figure 6.8. □

6.4.5 Reachability Proofs with Ordering Assumptions

We require some further subprocedures for the final modified version of ReachTooSmall.

The idea will be, when we have an ordering assumption $\mathsf{ord}(r, a, b)$ we should skip b in the breadth first search unless a is in the set of nodes we have reached so far. The first stepping stone to deriving this in terms of PB constraints will be to derive a constraint of the form $\mathsf{ord}(r, a, b) \wedge \mathsf{dist}(r, a)_{\geq k} \Rightarrow \mathsf{dist}(r, b)_{\geq k+1}$

Justification Subprocedure 6.18 (Ordering implies last is further from r than middle).

Preconditions: $r, a, b, k \in \{0, \ldots, n-1\}$ *with* $r \neq a \neq b$; $\mathsf{ord}(r, a, b)$ *is an ordering assumption literal defined as per (6.38) and (6.39).*

Procedure:

1: **proc** DeriveLastAfterMiddle(r, a, b, k)
2: $C_1 \leftarrow$ **pbc** $n \cdot \overline{\mathsf{prec}(a, b)} + P_b - P_a \geq 1$
3: **subproof of** C_1
4: $C_2 \leftarrow$ **rup** $-P_a \geq -n + 1$
5: **cut** $\neg C_1 + C_2$
6: **cut** $\neg C_1 + \mathsf{Def}_{\Rightarrow}(\mathsf{prec}(a, b))$
7: $C_3 \leftarrow$ **cut** $\mathsf{sat}((\mathsf{Def}_{\Rightarrow}(\mathsf{dist}(r, a)_{\geq k}) + \mathsf{Def}_{\Leftarrow}(\mathsf{dist}(r, b)_{\geq k+1}) + C_1)/n)$
8: $C_4 \leftarrow$ **cut** $\mathsf{sat}(C_3 + \mathsf{Def}_{\Rightarrow}(\mathsf{ord}(r, a, b))$
9: **return** $\mathsf{ord}(r, a, b) \wedge \mathsf{dist}(r, a)_{\geq k} \Rightarrow \mathsf{dist}(r, b)_{\geq k+1} \leftarrow C_4$

Correctness Proof. In the proof-by-contradiction subproof of C_1 we have

$$\begin{aligned} \neg C_1 + C_2 &= (-n \cdot \mathsf{prec}(a, b) - P_b + P_a \geq 0) + (-P_a \geq -n + 1) \\ &= -n \cdot \mathsf{prec}(a, b) - P_b \geq -n + 1; \\ \text{and } \neg C_1 + \mathsf{Def}_{\Rightarrow}(\mathsf{prec}(a, b)) &= (-n \cdot \mathsf{prec}(a, b) - P_b + P_a \geq 0) \\ &\quad + (K_1 \cdot \mathsf{prec}(a, b) + P_b - P_a \geq 1) \\ &= (K_1 - n) \cdot \mathsf{prec}(a, b) \geq 1. \end{aligned}$$

These propagate $\overline{\mathsf{prec}(a, b)}$ and $\mathsf{prec}(a, b)$ respectively, establishing the required contradiction for us to derive C_1 by redundance.

Then for line 7 we have,

$$\begin{array}{rrll}
\mathsf{Def}_{\Rightarrow}(\mathsf{dist}(r,a)_{\geq k}) = & K_k \cdot \overline{\mathsf{dist}(r,a)}_{\geq k} & +P_a - P_r + n \cdot \mathsf{prec}(a,r) & \geq k; \\
+\ \mathsf{Def}_{\Leftarrow}(\mathsf{dist}(r,b)_{\geq k+1}) = & K_{n-k} \cdot \mathsf{dist}(r,b)_{\geq k+1} & -P_b + P_r - n \cdot \mathsf{prec}(b,r) & \geq -k; \\
+ \qquad\qquad C_1 = & n \cdot \overline{\mathsf{prec}(a,b)} & +P_b - P_a & \geq 1; \\
\hline
= & K_k \cdot \overline{\mathsf{dist}(r,a)}_{\geq k} + K_{n-k} \cdot \mathsf{dist}(r,b)_{\geq k} + n \cdot \mathsf{prec}(a,r) + & & \\
& n \cdot \overline{\mathsf{prec}(a,b)} - n \cdot \mathsf{prec}(b,r) \geq 1; & &
\end{array}$$

$$\text{or, equivalently:} \quad K_k \cdot \overline{\mathsf{dist}(r,a)}_{\geq k} + K_{n-k} \cdot \mathsf{dist}(r,b)_{\geq k} + n \cdot \mathsf{prec}(a,r) + n \cdot \overline{\mathsf{prec}(a,b)} + n \cdot \overline{\mathsf{prec}(b,r)} \geq 1 + n.$$

Then, recalling the rounding property of the division rule (Rule 4), we know that $\lceil (1+n)/n \rceil = 2$, so we have

$$C_3 = 2 \cdot \overline{\mathsf{dist}(r,a)}_{\geq k} + 2 \cdot \mathsf{dist}(r,b)_{\geq k} + \mathsf{prec}(a,r) + \overline{\mathsf{prec}(a,b)} + \mathsf{prec}(b,r) \geq 2;$$

which since $\mathsf{prec}(a,r) = \overline{\mathsf{prec}(r,a)}$, is the same as

$$2 \cdot \overline{\mathsf{dist}(r,a)}_{\geq k} + 2 \cdot \mathsf{dist}(r,b)_{\geq k} + -\mathsf{prec}(r,a) - \mathsf{prec}(a,b) - \mathsf{prec}(b,r) \geq -1.$$

So adding this to $\mathsf{Def}_{\Rightarrow}(\mathsf{ord}(r,a,b))$ on line 8 cancels all the precedence variables, leading to $\mathsf{ord}(r,a,b) \wedge \mathsf{dist}(r,i)_{\geq k} \Rightarrow \mathsf{dist}(r,b)_{\geq k+1}$ after saturation, as claimed. □

Next, we want to be able to derive constraints of the form $\mathsf{dist}(r,j)_{=k} \Rightarrow \mathsf{dist}(r,a)_{=k} \geq 1$, i.e., that no two vertices can be the same distance from r. This is achieved by Justification Subprocedure 6.19

Justification Subprocedure 6.19 (Different nodes have different distances).
Preconditions: $r, j, a, k \in \{0, \ldots, n-1\}$ *with* $r \neq j \neq a$.
Procedure:

1: **proc** NotSameDistance(r, j, a, k)
2: **if** $r \neq 0$
3: $C_1 \leftarrow$ **cut** $\mathsf{sat}(\mathsf{Def}_{\Rightarrow}(\mathsf{prec}(a,j)) + \mathsf{Def}_{\Rightarrow}(\mathsf{prec}(j,r)) + \mathsf{Def}_{\Leftarrow}(\mathsf{prec}(a,r)))/3$
4: $C_2 \leftarrow$ **cut** $\mathsf{sat}(\mathsf{Def}_{\Leftarrow}(\mathsf{prec}(a,j)) + \mathsf{Def}_{\Leftarrow}(\mathsf{prec}(j,r)) + \mathsf{Def}_{\Rightarrow}(\mathsf{prec}(a,r)))/3$
5: **cut** $\mathsf{Def}_{\Rightarrow}(\mathsf{dist}(r,j)_{\geq k}) + \mathsf{Def}_{\Leftarrow}(\mathsf{dist}(r,a)_{\geq k+1}) + \mathsf{Def}_{\Rightarrow}(\mathsf{prec}(a,j)) + n \cdot C_1$
6: **cut** $\mathsf{Def}_{\Leftarrow}(\mathsf{dist}(r,j)_{\geq k+1}) + \mathsf{Def}_{\Rightarrow}(\mathsf{dist}(r,a)_{\geq k}) + \mathsf{Def}_{\Leftarrow}(\mathsf{prec}(a,j)) + n \cdot C_2$
7: **rup** $\overline{\mathsf{dist}(r,j)}_{=k} + \overline{\mathsf{dist}(r,a)}_{=k} \geq 1$

Correctness Proof for Justification Subprocedure 6.19. Once again we simply need to compute the cutting planes summations.

We have

$$\begin{aligned}
\mathsf{Def}_{\Rightarrow}(\mathsf{prec}(a,j)) &= K_1 \cdot \overline{\mathsf{prec}(a,j)} && +P_j - P_a \geq 1;\\
+\ \mathsf{Def}_{\Rightarrow}(\mathsf{prec}(j,r)) &= K_1 \cdot \overline{\mathsf{prec}(j,r)} && +P_r - P_j \geq 1;\\
+\ \mathsf{Def}_{\Leftarrow}(\mathsf{prec}(a,r)) &= K_1 \cdot \mathsf{prec}(a,r) && -P_r + P_a \geq 1;\\
\hline
&= K_1 \cdot \overline{\mathsf{prec}(a,j)} + K_1 \cdot \overline{\mathsf{prec}(j,r)} + K_1 \cdot \mathsf{prec}(a,r) && \geq 3.\\
\text{Hence, } C_1 = & \quad \overline{\mathsf{prec}(a,j)} + \overline{\mathsf{prec}(j,r)} + \mathsf{prec}(a,r) && \geq 1\\
\text{or, equivalently:} & \quad \overline{\mathsf{prec}(a,j)} - \mathsf{prec}(j,r) + \mathsf{prec}(a,r) && \geq 0.
\end{aligned}$$

$$\begin{aligned}
\mathsf{Def}_{\Leftarrow}(\mathsf{prec}(a,j)) &= K_1 \cdot \mathsf{prec}(a,j) && -P_j + P_a \geq 1;\\
+\ \mathsf{Def}_{\Leftarrow}(\mathsf{prec}(j,r)) &= K_1 \cdot \mathsf{prec}(j,r) && -P_r + P_j \geq 1;\\
+\ \mathsf{Def}_{\Rightarrow}(\mathsf{prec}(a,r)) &= K_1 \cdot \overline{\mathsf{prec}(a,r)} && +P_r - P_a \geq 1;\\
\hline
&= K_1 \cdot \overline{\mathsf{prec}(a,j)} + K_1 \cdot \overline{\mathsf{prec}(j,r)} + K_1 \cdot \mathsf{prec}(a,r) && \geq 3\\
\text{Hence } C_2 = & \quad \mathsf{prec}(a,j) + \mathsf{prec}(j,r) + \overline{\mathsf{prec}(a,r)} && \geq 1\\
\text{or, equivalently:} & \quad \mathsf{prec}(a,j) + \mathsf{prec}(j,r) - \mathsf{prec}(a,r) && \geq 0.
\end{aligned}$$

So, for line 5

$$\begin{aligned}
\mathsf{Def}_{\Rightarrow}(\mathsf{dist}(r,j)_{\geq k}) &= K_k \cdot \overline{\mathsf{dist}(r,j)}_{\geq k} && +P_j - P_r + n \cdot \mathsf{prec}(j,r) && \geq k;\\
+\ \mathsf{Def}_{\Leftarrow}(\mathsf{dist}(r,j)_{\geq k+1}) &= K_{n-k} \cdot \mathsf{dist}(r,j)_{\geq k+1} && -P_a + P_r - n \cdot \mathsf{prec}(j,r) && \geq -k;\\
+\ \mathsf{Def}_{\Rightarrow}(\mathsf{prec}(a,j)) &= K_1 \cdot \overline{\mathsf{prec}(a,j)} && +P_a - P_j && \geq 1;\\
+\ n \cdot C_1 &= n \cdot \overline{\mathsf{prec}(a,j)} && - n \cdot \mathsf{prec}(j,r) + n \cdot \mathsf{prec}(a,r) && \geq 0;\\
\hline
&= (K_1 + n) \cdot \overline{\mathsf{prec}(a,j)} + K_k \cdot \overline{\mathsf{dist}(r,j)}_{\geq k} + K_{n-k} \cdot \mathsf{dist}(r,a)_{\geq k+1} && \geq 1.
\end{aligned}$$

And then for line 6,

$$\begin{aligned}
\mathsf{Def}_{\Leftarrow}(\mathsf{dist}(r,j)_{\geq k+1}) &= K_{n-k} \cdot \mathsf{dist}(r,j)_{\geq k} && -P_j + P_r - n \cdot \mathsf{prec}(j,r) && \geq -k;\\
+\ \mathsf{Def}_{\Rightarrow}(\mathsf{dist}(r,j)_{\geq k}) &= K_k \cdot \mathsf{dist}(r,j)_{\geq k} && +P_a - P_r + n \cdot \mathsf{prec}(j,r) && \geq k;\\
+\ \mathsf{Def}_{\Leftarrow}(\mathsf{prec}(a,j)) &= K_1 \cdot \mathsf{prec}(a,j) && +P_a - P_j && \geq 1;\\
+\ n \cdot C_1 &= n \cdot \mathsf{prec}(a,j) && + n \cdot \mathsf{prec}(j,r) - n \cdot \mathsf{prec}(a,r) && \geq 0;\\
\hline
&= (K_1 + n) \cdot \mathsf{prec}(a,j) + K_{n-k} \cdot \mathsf{dist}(r,j)_{\geq k} + K_k \cdot \overline{\mathsf{dist}(r,a)}_{\geq k+1} && \geq 1.
\end{aligned}$$

These together guarantee that the final constraint will be RUP, since the negation will propagate

both $\mathsf{prec}(a, j)$ and $\overline{\mathsf{prec}(a, j)}$. In the case where $r = 0$, the final constraint is already RUP, since we are assuming an AllDifferent constraint over P_j variables has already been recovered (as per Justification Subprocedure 6.8) and the negation would fix both P_j and P_a to k, contradicting this. □

We can now describe the modified ReachTooSmall procedure that respects A3. Since there are a number of modifications to Justification Subprocedure 6.13 to make in different places, it will be clearest to rewrite most of the procedure again in full.

Justification Procedure 6.20 (Reach is too small, with both assumption types).
Preconditions: SortAsTuple *converts a set to a sorted tuple;* ℓ *is an atomic equality literal on a variable in* $\mathcal{X}$*;* dom_t *is a domain state;* $G := \mathsf{GraphRep}(\mathcal{X}, \mathrm{dom}_a)$*, with* dom_a *constructed using* ℓ *as in (6.5); and* $r \neq a \neq b$ *are vertices of* G *such that* $|\mathsf{Reach}_{a \prec b}(r)| < |G|$.
Procedure: *See Justification Procedure 6.20 (cont.)*

Correctness Proof. First note that the constraint on line 7 is RUP by Theorem 2.7 from the bounds on P_r in the encoding. Then the cutting planes step on line 8 derives

$$\begin{aligned}&(K_n \cdot \mathsf{dist}(r, a)_{\geq 1} - P_a + P_r - n \cdot \mathsf{prec}(a, r) \geq 0) + (-P_r \geq -n + 1)\\ &= K_n \cdot \mathsf{dist}(r, a)_{\geq 1} - P_a - n \cdot \mathsf{prec}(a, r) \geq -n + 1\end{aligned}$$

With this in place, the negation of the constraint on line 9 propagates both $\overline{\mathsf{prec}(a, r)}$ and $\mathsf{BinEnc}(P_a) = 0$, conflicting with $\mathsf{Def}_{\Leftarrow}(\mathsf{prec}(a, r))$, since this becomes $-\mathsf{BinEnc}(P_r) \geq 1$. So the constraint is indeed RUP.

Now assume at the start of each execution of the while loop that a has not yet been collected in allReached, and that we have derived

$$\mathcal{R}_{\mathsf{step}} \Rightarrow \mathsf{dist}(r, a)_{\geq \mathsf{step}} \geq 1. \tag{6.45}$$

From the above these things are true initially for $\mathsf{step} = 1$.

Then within the loop body, at-least-one constraints are initially derived in exactly the same way as in Justification Subprocedure 6.14, and so are correct by the same argument, except until the vertex a is seen, the vertex b is not added to the newlyReached set, due to the conditional block at line 24.

When this happens, the excludeByOrdering condition is set to true, causing the constraint

$$\mathsf{ord}(r, a, b) \wedge \mathsf{dist}(r, a)_{\geq \mathsf{step}} \Rightarrow \mathsf{dist}(r, b)_{\geq \mathsf{step}+1} \tag{6.46}$$

Justification Procedure 6.20 (continued)

proc ReachTooSmall$(r, \mathrm{dom}_t, \ell, a, b)$
  ⋮ ▷ *(The first 5 lines are identical to Justification Subprocedure 6.13).*
  $C \leftarrow$ **rup** $-\mathsf{BinEnc}(P_r) \geq -n+1$
  cut $\mathsf{Def}_{\Leftarrow}(\mathsf{dist}(r,a)_{\geq 1}) + C$
  rup $\mathsf{dist}(r,a)_{\geq 1} \geq 1$
  seenMid $\leftarrow$ *false*
  while step $\leq$ |allReached|
    newlyReached $\leftarrow \emptyset$, excludeByOrdering $\leftarrow$ *false*
    $\mathcal{R}_{\mathsf{step}} \leftarrow \emptyset$
    for $i \in$ lastReached
      $\mathcal{R}_i \leftarrow \mathsf{GenericR}_t(X_i)$
      $\mathcal{R}_{\mathsf{step}} \leftarrow \mathcal{R}_{\mathsf{step}} \cup \mathcal{R}_i$
      $V \leftarrow \mathrm{dom}_t(X_i)$
      for $j \in V$
        if $\ell = \overline{x}_{i=j} \ \vee \ (\ell = x_{i=k} \wedge k \neq j) \ \vee \ (\ell = x_{k=j} \wedge k \neq i)$
          $\mathcal{R}_i \leftarrow \mathcal{R}_i \cup \{\ell\}$
          $V \leftarrow V \setminus \{j\}$
          continue
        $C_{ij} \leftarrow \mathsf{DeriveSucessorDistance}(i, j, r, \mathsf{step}-1)$
        if $j = b \wedge \neg$seenMid
          excludeByOrdering $\leftarrow$ *true*
          continue
        if $j \neq r$
          seenMid $\leftarrow (j = a)$
          newlyReached $\leftarrow$ newlyReached $\cup \{j\}$
          possibleDistances$[j] \leftarrow$ possibleDistances$[j] \cup \{\mathsf{step}\}$
      $D \leftarrow$ **rup** $\mathcal{R}_i \Rightarrow \sum_{j \in V} x_{i=j} \geq 1$
      $C_i \leftarrow$ **cut** $D + \sum_j C_{ij}$
    AtLeast1s[step] $\leftarrow$ **cut** $\mathsf{sat}(\mathsf{AtLeast1s}[\mathsf{step}-1] + \sum_i C_i)$
    if excludeByOrdering
      $\mathsf{DeriveLastAfterMiddle}(r, a, b, \mathsf{step})$
      $E_1 \leftarrow$ **rup** $\mathcal{R}_{\mathsf{step}} \wedge \mathsf{ord}(r,a,b) \Rightarrow \overline{\mathsf{dist}(r,b)}_{=\mathsf{step}} \geq 1$
      AtLeast1s[step] $\leftarrow$ **cut** $\mathsf{sat}(\mathsf{AtLeast1s}[\mathsf{step}] + E_1)$
    if $\neg$seenMid
      for $j \in$ newlyReached
        $\mathsf{NotSameDistance}(r, j, a, \mathsf{step})$
      rup $\mathcal{R}_{\mathsf{step}} \Rightarrow \mathsf{dist}(r,a)_{\geq \mathsf{step}+1} \geq 1$
    ⋮ ▷ *(The remaining lines are identical to the last 10 lines of Justification Subprocedure 6.13).*

to be derived as per Justification Subprocedure 6.18. Since we have already derived a bound on the distance from r to a under $\mathcal{R}_{\mathsf{step}}$ (6.45) the negation of the constraint E_1 will immediately be RUP since $\mathsf{dist}(r, b)_{\geq \mathsf{step}+1}$ propagates $\overline{\mathsf{dist}(r, b)}_{=\mathsf{step}}$.

Then, the AtLeast1s[step] constraint will be updated by the cutting planes operation on line 37 with the $\mathsf{dist}(r, b)_{=\mathsf{step}}$ being resolved away, as follows

$$\begin{aligned}
&\mathsf{sat}((\mathcal{R}_{\mathsf{step}} \Rightarrow \mathsf{dist}(r, b)_{=\mathsf{step}} + \sum_{j \in \mathsf{newlyReached}} \mathsf{dist}(r, j)_{=\mathsf{step}} \geq 1) \\
&+ (\mathcal{R}_{\mathsf{step}} \wedge \mathsf{ord}(r, a, b) \Rightarrow \overline{\mathsf{dist}(r, b)}_{=\mathsf{step}} \geq 1)) \\
&= \mathcal{R}_{\mathsf{step}} \wedge \mathsf{ord}(r, a, b) \Rightarrow \sum_{j \in \mathsf{newlyReached}} \mathsf{dist}(r, j)_{=\mathsf{step}} \geq 1
\end{aligned}$$

If $a \notin \mathsf{newlyReached}$, seenMid will remain *false*, and on line 40 the constraint

$$\overline{\mathsf{dist}(r, j)}_{=\mathsf{step}} + \overline{\mathsf{dist}(r, a)}_{=\mathsf{step}} \geq 1 \tag{6.47}$$

for each $j \in \mathsf{newlyReached}$ will be derived as per Justification Subprocedure 6.19. This ensures the constraint on line 41 will be RUP, since together with (6.45), its negation will propagate $\mathsf{dist}(r, a)_{=\mathsf{step}}$, which then propagates $\overline{\mathsf{dist}(r, j)}_{=\mathsf{step}}$ for each $j \in \mathsf{newlyReached}$, contradicting the last at-least-one constraints. Hence, the invariant required for excluding b on the next step (if necessary) is maintained.

Overall, after each execution of the while loop, we derive an at-least-one constraint of the form $\mathcal{R}_{\mathsf{step}} \wedge \mathsf{ord}(r, a, b) \Rightarrow \sum_{j \in \mathsf{newlyReached}}$, and we can only collect vertices reachable from r via paths P where $a \prec_P b$. Since by assumption, $|\mathsf{Reach}_{a \prec b}(r)| < |G|$, an analogous argument to the proof of the correctness of Justification Subprocedure 6.13 is again applicable. We must derive step reified at-least-one constraints and $\mathsf{step} - 1$ at-most-one constraints which cancel to derive a constraint of the form $\mathcal{R} \wedge \mathsf{ord}(r, a, b) \Rightarrow \ell$.

□

This concludes the complete set of justification procedures and subprocedures necessary for justifying a set of inferences that could be made by a Circuit propagator.

6.5 Implementation and Experiments

We implemented a proof logging Circuit constraint within the *Glasgow Constraint Solver*, including an implementation of Tarjan's algorithm and detection of all the propagation opportunities discussed in the previous sections. The implementation keeps track of all used auxiliary variables such as $\mathsf{dist}(r, i)$ and $\mathsf{prec}(i, j)$ to avoid redefining them when possible. This is particularly important for the latter, since unlike other extension variable definitions appearing in this thesis,

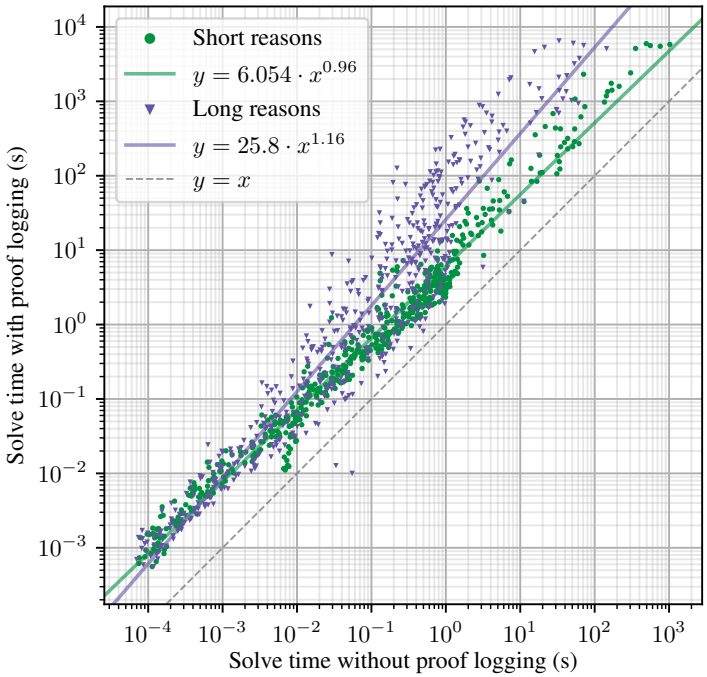
$\mathsf{prec}(i, j)$ require a non-trivial subproof with a non-constant number of proof steps to introduce, as demonstrated by Justification Subprocedure 6.9.

Following the experimental philosophy from previous chapters, we focussed on randomly generated instances with increasing sizes to measure proof logging overhead, proof size, and checking time. Since there are many specific filtering rules with different proof logging methods that we would ideally like to test, we formulated random instances of the "travelling salesperson problem" (TSP). This involves the Circuit constraint and also some side constraints and an objective to maximise, giving a slightly greater variety of solver behaviour than simply testing a single Circuit constraint. For sanity testing, we also formulated some specifically crafted small instances designed to trigger each of the inference rules, and we confirmed the corresponding justification procedure wrote to the proof log. As before, the key measure of success, which is that all the proofs were validated by version 3.0 of the *VeriPB* proof checker, was met.

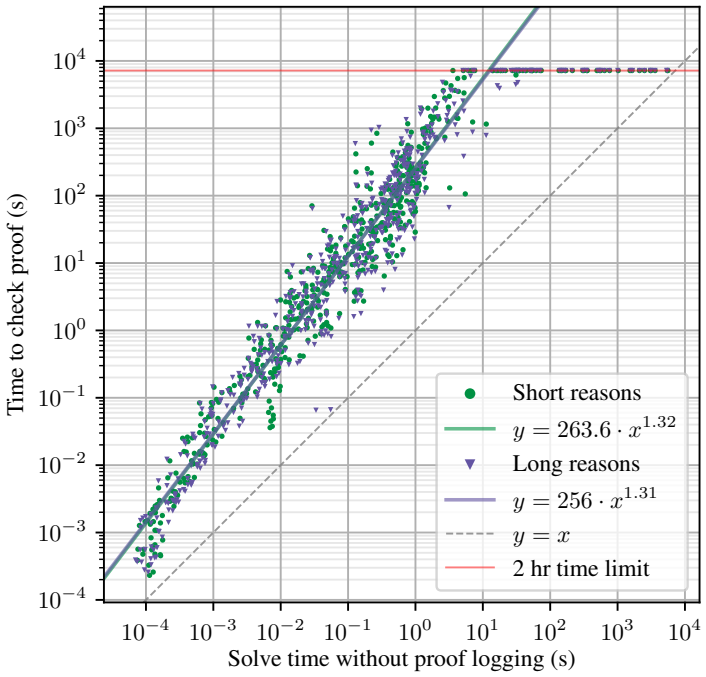
The results of the random TSP benchmarking are shown in Figures 6.9 and 6.10 This is in line with what we have seen previously: proof logging overhead is roughly within a constant factor of the amount of work performed by the propagator. Short reasons once again provide a benefit, although the difference is less marked than with other propagators, likely due to the fact that proportionately fewer of the intermediate justification steps need to be reified with a generic reason.

6.6 Conclusions

We have exhibited the first certifying Circuit propagator by making use of pseudo-Boolean proof logging. Although the justification procedures in this chapter are specialised to the particular propagation rules under consideration, they can be seen as a case study in how ad-hoc inference rules with complicated notions of consistency can be included in a proof logging 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 representations of these notions, or even of a graph. Even though the procedures are not immediately more generally applicable beyond Circuit, it is likely that the core concepts exemplified here — counting reachable vertices under implications; creating shifted auxiliary labels; and running proof procedures under ordering assumptions — will be useful for other constraints, particularly related graph constraints such as Path and SubCircuit.

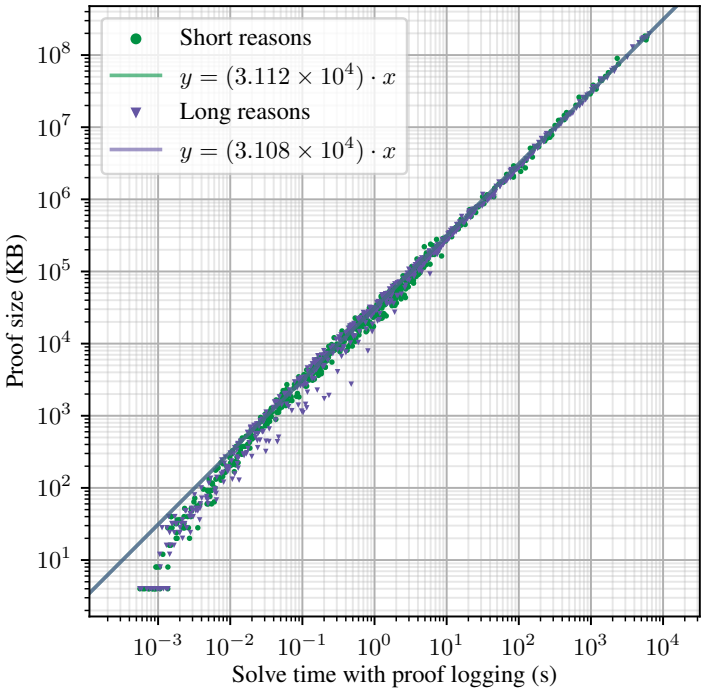


(a) Overhead of proof logging during solving.

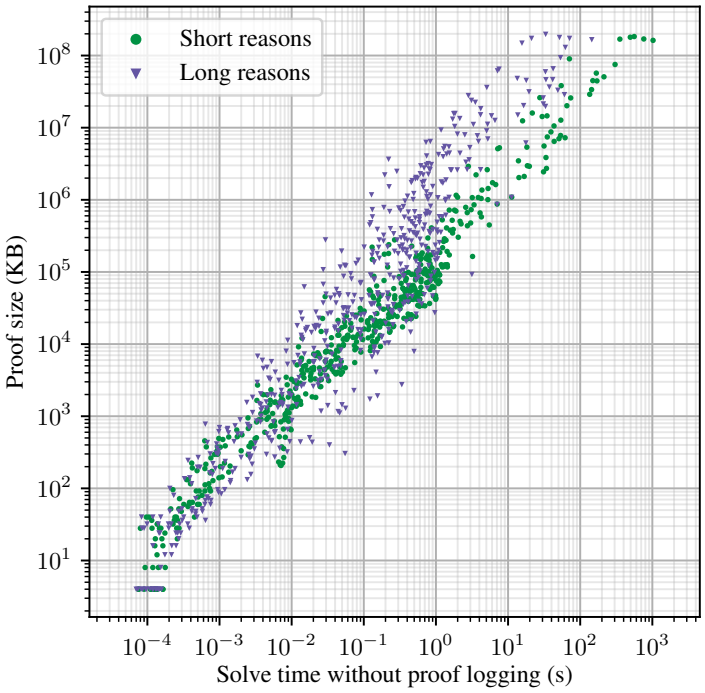


(b) Proof checking time relative to solve time.

Figure 6.9: Timing results for random TSP instances.



(a) Proof size relative to time to produce the proof.



(b) Proof size relative to solve time.

Figure 6.10: Proof size results for random TSP instances.

Chapter 7

Multiplication Constraints

We now move on to justifying propagation of multiplication constraints: $X \times Y = Z$ for CP variables X, Y, Z over integer domains. This fundamental arithmetic operation appears in a wide variety of problem instances, and can be used as an atomic constraint in the decomposition of arbitrary polynomial restrictions [9], as well as division and modulo constraints. In contrast to other simple arithmetic constraints, ternary multiplication propagators in practice generally enforce only *bounds-consistency*. In a bounds-consistent propagator, upper and lower bounds on domains of each variable are narrowed such that, for both bound values, there exist *real numbers* within the bounds of the other two variables where the multiplication relation holds. This corresponds to the *bounds($\mathcal{R}$)* consistency definition given by Choi et al. [42], which is the standard level of consistency enforced in state-of-the-art open source solver implementations such as *Gecode* [77] and *Chuffed* [44].

At first, it might appear that a fundamentally linear PB proof system based on cutting planes would be ill-equipped to handle such reasoning. But we are able to show that this is not the case. By using the binary representation of the CP variables and some explicit proofs by contradiction, we can derive justifications for any inference made by a bounds propagator for multiplication in a number of proof steps proportionate to the product of the numbers of bits required to encode the domains of X and Y. This includes handling negative domain values through the use of a case-based derivation over the signs of the variables.

In this chapter we will start with the easiest cases for this constraint, and build towards the more awkward ones. We begin by showing how to encode the constraint when X and Y have only non-negative domain values, and then demonstrate justification procedures for inferring bounds on Z based on bounds on X and Y. We then extend this encoding to negative values using a sign-bit-inspired representation, and demonstrate that the same justification procedure can be embedded in a case-based argument that considers all the possible sign combinations. Finally, using proof by contradiction, we show that this can be extended to any valid bounding inference on any of the variables X, Y, or Z. Once again, these procedures have been implemented as part of a new constraint in the *Glasgow Constraint Solver*, and we briefly discuss this at the end of

this chapter.

7.1 Non-negative Integer Multiplication

For a constraint $X \times Y = Z$, where X and Y have only *non-negative* domain values, encodings and bounds justifications are relatively straightforward.

7.1.1 PB Encoding for Non-negative Multiplication Constraints

If we have two binary numbers $x_{n-1} \dots x_0$ and $y_{m-1} \dots y_0$ the value of their product is given by

$$\left(\sum_{i=0}^{n-1} 2^i x^i\right) \cdot \left(\sum_{j=0}^{m-1} 2^j y^i\right) \tag{7.1}$$

$$= \sum_{i=0}^{n-1} \sum_{j=0}^{m-1} \left(2^{i+j} \cdot x_i \cdot y_j\right). \tag{7.2}$$

So a basic idea for a linearisation of $X \times Y = Z$ is to use binary representations, define auxiliary variables xy_{ij} constrained to represent each bit product, and then simply constrain Z to equal the sum corresponding to (7.2).

Encoding Procedure 7.1 (Ternary Multiplication with Positive Values).
Let $X \times Y = Z$ be a multiplication constraint with

$$\min(\mathrm{dom}_0(X)) \geq 0; \quad \min(\mathrm{dom}_0(Y)) \geq 0; \quad \textit{and} \quad \min(\mathrm{dom}_0(Z)) \geq 0.$$

A linearisation of the constraint is given by the following procedure.

1: $n \leftarrow |\mathsf{Vars}(\mathsf{BinEnc}(X))|$
2: $m \leftarrow |\mathsf{Vars}(\mathsf{BinEnc}(Y))|$
3: **for** $i \in \{0 \dots n-1\}$
4: ⎸ **for** $j \in \{0 \dots m-1\}$
5: ⎸ ⎸ **def** $xy_{ij} \Leftrightarrow x_i + y_j \geq 2$
6: **def** $X - \sum_{i=0}^{n-1} 2^i x_i = 0$
7: **def** $Y - \sum_{i=0}^{m-1} 2^i y_i = 0$
8: **def** $Z - \sum_{i=0}^{n-1} \sum_{j=0}^{m-1} 2^{i+j} xy_{ij} = 0$

It should not be controversial to claim that this is a direct translation into linear constraints of the multiplication of two binary-encoded variables. The constraints on lines 6 and 7 are only needed as a technicality in order to fit into the framework of Section 3.2 where an intermediate ILP is needed. In practice, we can assume that the x_i and y_i variables are unified with the same bit variables as used in $\mathsf{BinEnc}(X)$ and $\mathsf{BinEnc}(Y)$, in which case the equalities on X and Y are trivial.

In the following sections it will be useful to refer to the forward and reverse implication constraints from line 5 of this encoding by name. So let

$$\mathsf{Def}_{\Rightarrow}(xy_{ij}) := xy_{ij} \Rightarrow x_i + y_j \geq 2; \tag{7.3}$$

$$\mathsf{Def}_{\Leftarrow}(xy_{ij}) := \overline{xy}_{ij} \Rightarrow \overline{x}_i + \overline{y}_j \geq 1. \tag{7.4}$$

7.1.2 Deriving Lower Bounds on a Non-negative Product

Now with the linearisation in place, we can first show how to derive lower bounds on Z based on lower bounds on X and Y in a fairly natural way. The basic idea is to incrementally build up a product-of-lower-bounds constraint on a sum (7.2). For each fixed i we can compute a 2^j exponential sum of corresponding reverse implications for individual bit product variables, and then cancel out the y_j bits by saturating and adding the lower bound constraint on $\mathsf{BinEnc}Y$. We then compute a 2^i exponential sum over all the results of these, and cancel the y_j bits by adding the lower bounds constraint for $\mathsf{BinEnc}(X)$ multiplied by the lower bound for Y. This neatly works out to be exactly what we need, as the following examples illustrates.

Example 7.1 (Lower bounds on a non-negative product).
Suppose X has initial domain $\{3, \ldots, 5\}$ and Y has initial domain $\{4, \ldots, 6\}$. Then both domain ranges can be represented with just 3 bits.
A bounds propagator for the constraint should infer that $Z \geq 12$, so we would like to derive the justification

$$x_{\geq 3} \wedge y_{\geq 4} \Rightarrow z_{\geq 12} \tag{7.5}$$

Starting from bounds definitions, (and writing out BinEnc *in full) we have*

$$x_{\geq 3} \Rightarrow x_0 + 2x_1 + 4x_2 \geq 3; \tag{7.6}$$

$$\text{and } y_{\geq 4} \Rightarrow y_0 + 2y_1 + 4y_2 \geq 4. \tag{7.7}$$

Now, we can compute the following cutting planes steps for x_0:

$$\begin{aligned}
\overline{xy}_{00} &\Rightarrow & -x_0 - y_0 &\geq -1 && (7.8)\\
+2\cdot(\overline{xy}_{01} &\Rightarrow & -x_0 - y_1 &\geq -1) && (7.9)\\
+4\cdot(\overline{xy}_{02} &\Rightarrow & -x_0 - y_2 &\geq -1) && (7.10)\\
+(y_{\geq 4} &\Rightarrow & y_0 + 2y_1 + 4y_2 &\geq 4) && (7.11)\\
\hline
(\textit{synt. implies})\ y_{\geq 4} &\Rightarrow & xy_{00} + 2xy_{01} + 4xy_{02} - 7x_0 &\geq -3 && (7.12)\\
(\textit{synt. implies})\ y_{\geq 4} &\Rightarrow & xy_{00} + 2xy_{01} + 4xy_{02} - 4x_0 &\geq 0 && (7.13)
\end{aligned}$$

And similarly for x_1 and x_2 we can compute

$$\begin{aligned}
y_{\geq 4} &\Rightarrow\ xy_{10} + 2xy_{12} + 4xy_{12} - 4x_1 \geq 0 && (7.14)\\
y_{\geq 4} &\Rightarrow\ xy_{20} + 2xy_{23} + 4xy_{22} - 4x_2 \geq 0. && (7.15)
\end{aligned}$$

So we can in turn compute

$$\begin{aligned}
(y_{\geq 4} &\Rightarrow\ xy_{00} + 2xy_{01} + 4xy_{02} - 4x_0 \geq 0) && (7.16)\\
+2\cdot(y_{\geq 4} &\Rightarrow\ xy_{10} + 2xy_{11} + 4xy_{12} - 4x_1 \geq 0) && (7.17)\\
+4\cdot(y_{\geq 4} &\Rightarrow\ xy_{20} + 2xy_{21} + 4xy_{22} - 4x_2 \geq 0) && (7.18)\\
+4\cdot(x_{\geq 3} &\Rightarrow\ x_0 + 2x_1 + 4x_2 \geq 3) && (7.19)\\
+ &\quad (\textstyle\sum_{i=0}^{4} 2^i z_i - \sum_{i,j\in\{1,2,3\}} xy_{ij} \geq 0) && (7.20)\\
\hline
x_{\geq 3} \wedge y_{\geq 4} &\Rightarrow z_0 + 2z_1 + 4z_2 + 8z_3 + 16z_4 \geq 12 && (7.21)
\end{aligned}$$

which allows the desired constraint (7.5) to be RUP.

The process is articulated precisely in Justification Subprocedure 7.1 below.

Correctness Proof for Justification Subprocedure 7.1.

We can compute $\sum_j 2^j \cdot \mathsf{Def}_{\Leftarrow}(xy_{ij})$ as being

$$\sum_{j=0}^{m-1} 2^j xy_{ij} + (2^m - 1)\cdot \overline{x}_i + \sum_{j=0}^{m-1} 2^j \overline{y}_j \geq 2^m - 1,$$

or, equivalently:

$$\sum_{j=0}^{m-1} 2^j xy_{ij} + (2^m - 1)\cdot \overline{x}_i - \sum_{j=0}^{m-1} 2^j y_j \geq 0.$$

Justification Subprocedure 7.1 (Positive product lower bounds).

Preconditions: *A multiplication constraint $X \times Y = Z$ encoded as in Encoding Procedure 7.1; $l_X \geq 0$; $l_Y \geq 0$; and*

$$B_X := \mathcal{R} \Rightarrow \sum_{i=0}^{n-1} 2^i x_i \geq l_X, \qquad B_Y := \mathcal{R} \Rightarrow \sum_{i=0}^{m-1} 2^i y_i \geq l_Y,$$

for some set of literals $\mathcal{R}$.

Procedure:

1: **proc** DeriveProductLowerBound(B_X, B_Y)
2: **for** $i \in \{0 \dots n-1\}$
3: $D_i \leftarrow$ **cut** $\mathsf{sat}(B_Y + \sum_{j=0}^{m-1} 2^j \cdot \mathsf{Def}_{\Leftarrow}(xy_{ij}))$
4: $D'_i \leftarrow$ **imp** $\mathcal{R} \Rightarrow \sum_{j=0}^{m-1} 2^j xy_{ij} + l_Y \cdot \overline{x}_i \geq l_Y$ **from** D_i
5: $C_1 \leftarrow$ **get** $\mathsf{BinEnc}(Z) - \sum_{i=0}^{n-1}\sum_{j=0}^{m-1} 2^{i+j} xy_{ij} \geq 0$
6: $C_2 \leftarrow$ **cut** $C_1 + l_Y \cdot B_X + \sum_{i=0}^{n-1} 2^i \cdot D'_i$
7: **return imp** $\mathcal{R} \Rightarrow \sum_{i=0}^{k-1} 2^i z_k \geq l_X \cdot l_Y$ **from** C_2

Adding $B_Y\restriction_{\mathcal{R}} = \sum_j 2^j y_j \geq l_Y$ to this would yield

$$\sum_{j=0}^{m-1} 2^j xy_{ij} + (2^m - 1) \cdot \overline{x}_i \geq l_Y$$

which after saturation (relying on $l_Y \geq 0$) is

$$\sum_{j=0}^{m-1} \min(2^j, l_Y) \cdot xy_{ij} + l_Y \cdot \overline{x}_i \geq l_Y.$$

Since this syntactically implies $D'_i\restriction_{\mathcal{R}} = \sum_{j=0}^{m-1} 2^j xy_{ij} + l_Y \cdot \overline{x}_i \geq l_Y$, Corollary 2.1 tells us that D_i does indeed syntactically imply the constraint on line 4. Next, for the cutting planes step on

line 6, note that the summation $\sum_i 2^i \cdot (D'_i\restriction_{\mathcal{R}})$ is

$$\sum_{i=0}^{n-1}\sum_{j=0}^{m-1} 2^{i+j} xy_{ij} + l_Y \cdot \sum_{i=0}^{n-1} 2^i \overline{x}_i \geq (2^n - 1) \cdot l_Y$$

or, equivalently:

$$\sum_{i=0}^{n-1}\sum_{j=0}^{m-1} 2^{i+j} xy_{ij} - l_Y \cdot \sum_{i=0}^{n-1} 2^i x_i \geq 0$$

and so adding $l_Y \cdot B_X\restriction_{\mathcal{R}} = l_Y \cdot \sum_i 2^i x_i \geq l_Y \cdot l_X$ and then $C\restriction_{\mathcal{R}} = C$ to this would yield

$$\mathsf{BinEnc}(Z) \geq l_Y \cdot l_X.$$

This tells us that the final constraint is indeed syntactically implied by C_2. □

7.1.3 Deriving Upper Bounds on a Non-negative Product

The subprocedure for upper bounds is slightly more complicated, requiring a proof by contradiction to get intermediate steps in the required form. This can be observed by continuing the worked example below.

Example 7.2 (Upper bounds on a non-negative product).
Consider the same problem as in Example 7.1. It should also be possible to infer $Z \leq 30$. So we would like to derive a justification

$$\overline{x}_{\geq 6} \wedge \overline{y}_{\geq 7} \Rightarrow \overline{z}_{\geq 31} \geq 1, \tag{7.22}$$

To do this, first note that we can weaken the forwards implications for any bit product definition using syntactic implication to get either $-xy_{ij} + y_i \geq 0$ or $-xy_{ij} + x_i \geq 0$. Then for x_0, we can compute both

$$(xy_{00} \Rightarrow \quad y_0 \geq \quad 0) \tag{7.23}$$
$$+2\cdot(xy_{01} \Rightarrow \quad y_1 \geq \quad 0) \tag{7.24}$$
$$+4\cdot(xy_{02} \Rightarrow \quad y_2 \geq \quad 0) \tag{7.25}$$
$$+ \quad (\overline{y}_{\geq 7} \Rightarrow \quad -y_0 - 2y_1 - 4y_2 \geq -6) \tag{7.26}$$
$$\overline{y}_{\geq 7} \Rightarrow -xy_{00} - 2xy_{01} - 4xy_{02} \geq \quad -6 \tag{7.27}$$

and also

$$\begin{aligned}
(xy_{00} \Rightarrow & \quad x_0 \geq 0) && (7.28)\\
+2\cdot(xy_{01} \Rightarrow & \quad x_0 \geq 0) && (7.29)\\
+4\cdot(xy_{02} \Rightarrow & \quad x_0 \geq 0) && (7.30)\\
\hline
\overline{y}_{\geq 7} \Rightarrow & -xy_{00} - 2xy_{01} - 4xy_{02} + 7x_0 \geq 0. && (7.31)
\end{aligned}$$

And these two constraints can be seen to imply the conclusion

$$\overline{y}_{\geq 7} \Rightarrow -xy_{00} - 2xy_{01} - 4xy_{02} + 6x_0 \geq 0 \tag{7.32}$$

by considering each of the cases $x_0 \in \{0, 1\}$*: if* $x_0 = 0$ *it is equivalent to (7.31) and if* $x_0 = 1$ *it is equivalent to (7.27).*
So using a proof by contradiction, or fusion resolution Theorem 2.5, we can derive (7.32).
Next we can similarly derive

$$\begin{aligned}
\overline{y}_{\geq 7} &\Rightarrow -xy_{10} - 2xy_{11} - 4xy_{12} + 6x_1 \geq 0 && (7.33)\\
\overline{y}_{\geq 7} &\Rightarrow -xy_{20} - 2xy_{21} - 4xy_{22} + 6x_2 \geq 0 && (7.34)
\end{aligned}$$

So we can in turn compute

$$\begin{aligned}
(\overline{y}_{\geq 7} & \Rightarrow -xy_{00} - 2xy_{01} - 4xy_{02} + 6x_0 \geq 0) && (7.35)\\
2\cdot(\overline{y}_{\geq 7} & \Rightarrow -xy_{10} - 2xy_{11} - 4xy_{12} + 6x_1 \geq 0) && (7.36)\\
4\cdot(\overline{y}_{\geq 7} & \Rightarrow -xy_{20} - 2xy_{21} - 4xy_{22} + 6x_2 \geq 0) && (7.37)\\
6\cdot(\overline{x}_{\geq 6} & \Rightarrow -x_0 - 2x_1 - 4x_2 \geq -5) && (7.38)\\
- & \quad (-\textstyle\sum_{i=0}^{4} 2^i z_i + \sum_{i,j\in\{1,2,3\}} xy_{ij} \geq 0) && (7.39)\\
\hline
\overline{x}_{\geq 6} \wedge \overline{y}_{\geq 7} & \Rightarrow -z_0 - 2z_1 - 4z_2 - 8z_3 - 16z_4 \geq -30 && (7.40)
\end{aligned}$$

which allows the desired constraint (7.5) to be RUP.

We can articulate the process precisely in Justification Subprocedure 7.2 below.

Correctness Proof for Justification Subprocedure 7.2. First we have

$$W^x_{ij} = -xy_{ij} + y_i \geq 0 \quad \text{and} \quad W^y_{ij} = -xy_{ij} + x_i \geq 0. \tag{7.41}$$

Justification Subprocedure 7.2 (Positive product upper bounds).

Preconditions: *A multiplication constraint $X \times Y = Z$ encoded as in Encoding Procedure 7.1; $u_X \geq 0$; $u_Y \geq 0$; and $B_X := \mathcal{R} \Rightarrow -\sum_{i=0}^{n-1} 2^i x_i \geq u_X$, $B_Y := \mathcal{R} \Rightarrow -\sum_{i=0}^{m-1} 2^i y_i \geq u_Y$, for some set of literals $\mathcal{R}$.*

Procedure:

1: **proc** DeriveProductUpperBound(B_X, B_Y)
2: **for** $i \in \{0 \dots n-1\}$
3: **for** $j \in \{0 \dots m-1\}$
4: $W_{ij}^x \leftarrow$ **cut** $\mathsf{sat}(\mathsf{Def}_{\Rightarrow}(xy_{ij}) + (\overline{x}_i \geq 0))$
5: $W_{ij}^y \leftarrow$ **cut** $\mathsf{sat}(\mathsf{Def}_{\Rightarrow}(xy_{ij}) + (\overline{y}_j \geq 0))$
6: $W_i^x \leftarrow$ **cut** $\sum_{j=0}^{m-1} 2^j \cdot W_{ij}^x + B_Y$
7: $W_i^y \leftarrow$ **cut** $\sum_{j=0}^{m-1} 2^j \cdot W_{ij}^y$
8: $D_i \leftarrow$ **pbc** $\mathcal{R} \Rightarrow -\sum_{j=0}^{m-1} 2^j xy_{ij} + u_Y \cdot x_i \geq 0$
9: **subproof of** $\mathcal{R} \Rightarrow -\sum_{j=0}^{m-1} 2^j xy_{ij} + u_Y \cdot x_i \geq 0$
10: **cut** $\neg D_i + W_i^x$
11: **cut** $\neg D_i + W_i^y$
12: $C_1 \leftarrow$ **get** $-\mathsf{BinEnc}(Z) + \sum_{i=0}^{n-1}\sum_{j=0}^{m-1} 2^{i+j} xy_{ij} \geq 0$
13: $C_2 \leftarrow$ **cut** $C_1 + u_Y \cdot B_X + \sum_{i=0}^{n-1} 2^i \cdot D_i$
14: **return imp** $\mathcal{R} \Rightarrow -\sum_{i=0}^{k-1} 2^i z_i \geq -u_X \cdot u_Y$ **from** C_2.

Then we have

$$W_i^x\restriction_{\mathcal{R}} = -\sum_{j=0}^{m-1} 2^j xy_{ij} \geq -u_Y \qquad \text{and}$$

$$W_i^y\restriction_{\mathcal{R}} = -\sum_{j=0}^{m-1} 2^j xy_{ij} + (2^m - 1) \cdot x_i \geq 0.$$

Since $\neg D_i\restriction_{\mathcal{R}} = \sum_j 2^j xy_{ij} - u_Y \cdot x_i \geq 1$, we then have in the proof by contradiction subproof

$$\neg D_i\restriction_{\mathcal{R}} + W_i^x\restriction_{\mathcal{R}} = -u_Y \cdot x_i \geq -u_Y$$
$$\neg D_i\restriction_{\mathcal{R}} + W_i^y\restriction_{\mathcal{R}} = (2^m - 1 - u_Y) \cdot x_i \geq 1.$$

By assumption $u_Y \leq 2^m - 1$, so these together propagate both x_i and $\overline{x}_i$, a conflict. Then since $\neg D$ propagates $\mathcal{R}$, this tells us that a contradiction is reached by unit propagation after the constraints on lines 10 and 11 are derived. This fulfils the proof obligation for the PBC step.

Then $\sum_i 2^i D_i\restriction_{\mathcal{R}}$ is

$$-\sum_{i=0}^{n-1}\sum_{j=0}^{m-1} 2^{i+j} xy_{ij} + u_Y \cdot \sum_{i=0}^{n-1} 2^i \geq 0 \tag{7.42}$$

and so adding $u_X \cdot B_X\restriction_{\mathcal{R}} = -u_Y \cdot \sum_i^i x_i \geq -u_X \cdot u_Y$ and then adding $C_1\restriction_{\mathcal{R}} = C_1$ to this yields

$$-\mathsf{BinEnc}(Z) \geq -u_X \cdot u_Y. \tag{7.43}$$

Similar to the proof of Justification Subprocedure 7.2, this tells us that once the constraint on line 13 has been derived, the final returned constraint is syntactically implied. □

It is somewhat interesting that the lower bounds can be derived with pure cutting planes, while the upper bounds seem to require a proof by contradiction to get a compact derivation. This is related to the open question of whether there exists short cutting planes derivations for a *fusion-resolution* argument, which we only know how to implement as a proof by contradiction (recall Theorem 2.5).

If we look at the constraints $W_i^x\restriction_{\mathcal{R}}$ and $W_i^y\restriction_{\mathcal{R}}$ we can see that they have the form

$$-S \geq -b \quad \text{and} \quad -S + c \cdot x \geq 0 \tag{7.44}$$

where S is a PB summation, x is a PB variable, and c and b are integers such that $c \geq b$. As we saw in Example 7.2, we can reason informally that these imply $-S + b \cdot x \geq 0$ by case based-analysis on x. If $x = 1$ then the first constraint tells us that $-S + b \geq 0$ and hence the conclusion holds. And if $x = 0$ then the second constraint and the conclusion are the same.

Despite this being easy to see, it is not so clear how pure cutting planes, even with saturation, can derive this in a short number of steps. The reason this is related to fusion resolution is that we can view the two constraints as representing

$$x \Rightarrow -S + b \cdot x \geq 0 \qquad \text{and} \qquad \overline{x} \Rightarrow -S + b \cdot x \geq 0 \tag{7.45}$$

which is a special case of $x \Rightarrow C, \overline{x} \Rightarrow C$.

7.2 Multiplication with Negative Domain Values

Regardless of any minor annoyances, we have shown how to represent a positive multiplication constraint and derive bounds on the product with $O(n \cdot m)$-length proofs.

We would obviously like to be able to deal with negative values as well, but this presents an immediate difficulty. So far within the proof logging framework of this thesis we have used a two's complement representation for BinEnc when a variable has negative values in its initial domain. But if X and Y are two's complement encoded variables, with n and m bits respectively, the product (as computed by Baugh and Wooley [19]) is

$$2^{m+n-2} \cdot x_{n-1} \cdot y_{m-1} + \sum_{i=0}^{n-2} \sum_{j=0}^{m-2} 2^{i+j} \cdot x_i \cdot y_j - \sum_{i=0}^{m-2} 2^{n-1+i} \cdot x_{n-1} - \sum_{i=0}^{n-2} 2^{m-1+u} y_{m-1} \cdot x_i \cdot y_{m-1}. \tag{7.46}$$

While we could use the xy_{ij} we defined in Encoding Procedure 7.1 and define this directly, it is arguably stretching the requirement of "obvious correspondence" that we want from our PB representation of CP problems. Furthermore, it's not clear how we could adapt the fairly neat justification procedures from the previous section to be helpful with this case.

So to avoid starting from scratch, and have an (arguably) more understandable encoding, we propose a linearisation based on multiplying the magnitudes and signs of the represented integers separately.

7.2.1 PB Encoding for General Multiplication Constraints

Concretely, suppose a CP variable X is encoded with a sequence of PB bit variables $x_0, \ldots, x_{n-1}$ along with an additional two's complement bit x_n. To define a multiplication constraint in our PB model, we would introduce an additional set of $n + 1$ bit variables $|x|_0, \ldots, |x|_n$ to represent the absolute value of X and then define constraints for multiplication of the magnitudes; multiplication of the signs; and channelling between representations. To avoid confusion: each $|x|_i$ here represents a PB variable, separate from the x_i variables.

We will also make use of a function TCLength as part of our linearisation procedure. For a variable X with initial domain $\{l, \ldots, u\}$, we assume $\mathsf{TCLength}(X)$ returns the minimum number of bits h required to represent the range specifically with an $h - bit$ two's complement representation (regardless of whether l is negative).

As with the previous encoding, the constraints defined on lines 4 to 6 are not strictly needed if the CP variables will be replaced with a two's complement representation anyway when it comes to BinEnc replacement. However, making these constraints explicit allows a cleaner explanation of the justification procedures in full generality (including to cover the cases where some, but not all, of X, Y and Z can take negative values). The constraints from lines 7 and 12 ensure the *channelling* between the magnitude and two's complement bits. Essentially they say that if the

Encoding Procedure 7.2 (Ternary Multiplication).

A linearisation of a multiplication constraint $X \times Y = Z$ is as follows.

1: $n \leftarrow \mathsf{TCLength}(X) - 1$
2: $m \leftarrow \mathsf{TCLength}(Y) - 1$
3: $k \leftarrow \mathsf{TCLength}(Z) - 1$

4: **def** $X + 2^n x_n - \sum_{i=0}^{n-1} 2^i x_i = 0$

5: **def** $Y + 2^m y_m - \sum_{i=0}^{m-1} 2^i y_i = 0$

6: **def** $Z + 2^k z_k - \sum_{i=0}^{k-1} 2^i z_i = 0$

7: **def** $\overline{x}_n \Rightarrow \sum_{i=0}^{n-1} 2^i x_i - \sum_{i=0}^{n} 2_i |x|_i = 0$
8: **def** $x_n \Rightarrow \sum_{i=0}^{n-1} 2^i x_i + \sum_{i=0}^{n} 2^i |x|_i = 2^n$
9: **def** $\overline{y}_m \Rightarrow \sum_{i=0}^{m-1} 2^i y_i - \sum_{i=0}^{m} 2_i |y|_i = 0$
10: **def** $y_m \Rightarrow \sum_{i=0}^{m-1} 2^i y_i + \sum_{i=0}^{m} 2^i |y|_i = 2^m$
11: **def** $\overline{z}_k \Rightarrow \sum_{i=0}^{k-1} 2^i z_i - \sum_{i=0}^{k} 2_i |z|_i = 0$
12: **def** $z_k \Rightarrow \sum_{i=0}^{k-1} 2^i z_i + \sum_{i=0}^{k} 2^i |z|_i = 2^k$

13: **for** $i \in \{0 \dots n-1\}$
14: ⎸ **for** $j \in \{0 \dots m-1\}$
15: ⎸ ⎸ **def** $|xy|_{ij} \Leftrightarrow |x|_i + |y|_j \geq 2$

16: **def** $\sum_{i=0}^{k-1} 2^i |z|_i - \sum_{i=0}^{n-1} \sum_{j=0}^{m-1} 2^{i+j} |xy|_{ij} = 0$

17: $x_n \wedge y_m \Rightarrow \overline{s} \geq 1$
18: $\overline{x}_n \wedge \overline{y}_m \Rightarrow \overline{s} \geq 1$
19: $x_n \wedge \overline{y}_m \Rightarrow s \geq 1$
20: $\overline{x}_n \wedge y_m \Rightarrow s \geq 1$
21: $z_k \Leftrightarrow s + \overline{x}_{=0} + \overline{y}_{=0} \geq 3$

Auxiliary variables in include all the bit variables, magnitude bit variables and bit product variables.

sign bit is false (positive value), the variable and magnitude bits should be equal, and if the sign is true (negative value), then they should be complementary.

Each equality here is represented as two inequalities with opposite signs $\gamma \in \{1, -1\}$.

Borrowing from Buss and Nordström [36], for any PB literal w we can make use of the notation w^σ, $\sigma \in \{1, -1\}$, where $w^1 = w$ and $w^{-1} = \overline{w}$. We can then refer to any one of the four channelling constraints for any variable $W \in \{X, Y, Z\}$ unambiguously using $C_W(\sigma, \gamma)$, where σ specifies the sign of the reifying term and γ the direction of the inequality. For example $C_Y(-1, 1)$ is the first inequality defined due to line 9, and more generally

$$C_X(\sigma, \gamma) := x_n^\sigma \Rightarrow \gamma \sum_{i=0}^{n-1} 2^i x_i + \sigma\gamma \sum_{i=0}^{n} 2^i |x|_i \geq 2^{n-1} \cdot (\gamma\sigma + \gamma). \tag{7.47}$$

7.2.2 Channelling Subprocedures

With these channelling constraints, we can convert a bound on the original two's complement bits into a bound on the magnitude bits via a simple procedure. This is articulated in Justification Subprocedure 7.3. Every justification (sub)procedure that follows in the remainder of this chapter includes as an implicit precondition that we have a multiplication constraint $X \times Y = Z$ encoded as in Encoding Procedure 7.2, and that n, m, and k are the two's complement lengths of X, Y, and Z respectively.

Justification Subprocedure 7.3 (Channelling from two's compl. to magnitude bounds).
Preconditions: $b \in \mathbb{Z}$; $\gamma \in \{1, -1\}$; *$\mathcal{R}$ is a set of atomic literals for which we have already derived* $\mathcal{R} \Rightarrow -\gamma \cdot 2^n x_n + \gamma \cdot \sum_{i=0}^{n-1} 2^i x_i \geq b$
Procedure:

```
1:  proc ChannelTCToMag(b, γ)
2:      B ← get R ⇒ −γ · 2^n x_n + γ · Σ_{i=0}^{n−1} 2^i x_i ≥ b
3:      if γ · max(γb, 1) = b
4:          B′ ← cut B + C_X(−1, −γ)
5:          B_1 ← imp R ∧ x̄_n ∧ x̄_{=0} ⇒ γ · Σ_{i=0}^{n} 2^i |x|_i ≥ γ · max(γb, 1) from B′
6:          B_2 ← rup R ∧ x_n ∧ x̄_{=0} ⇒ −γ · Σ_{i=0}^{n} 2^i |x|_i ≥ γ · min(γb, −1)
7:          return {B_1, B_2}
8:      else
9:          B′ ← cut B + C_X(1, −γ)
10:         B_1 ← rup R ∧ x̄_n ∧ x̄_{=0} ⇒ γ · Σ_{i=0}^{n} 2^i |x|_i ≥ γ · max(γb, 1)
11:         B_2 ← imp R ∧ x_n ∧ x̄_{=0} ⇒ −γ · Σ_{i=0}^{n} 2^i |x|_i ≥ γ · min(γb, −1) from B′
12:         return {B_1, B_2}
```

Correctness Proof for Justification Subprocedure 7.3. First suppose $\gamma \cdot \max(\gamma b, 1) = b$. Then

$\gamma \cdot \min(\gamma b, -1) = -\gamma$, and $C_X(-1, -\gamma)$ is

$$\overline{x}_n \Rightarrow -\gamma \sum_{i=0}^{n-1} 2^i x_i + \gamma \sum_{i=0}^{n} 2^i |x|_i \geq 0$$

so clearly B_1 is syntactically implied by $B' = B + C_X(-1, -\gamma)$. Now if $\gamma = 1$, it must be the case that $b > 1$ and so B_2 is trivially RUP (since from B, $\overline{x}_n$ propagates to false). Otherwise, $\gamma = -1$ and B_2 is

$$\mathcal{R} \wedge x_n \wedge \overline{x}_{=0} \Rightarrow \sum_i 2^i |x|_i \geq 1 \tag{7.48}$$

and so the negation of this sets every $|x|_i$ to 0 by unit propagation. This in turn sets all the x_i to 0 (via the channelling constraints) and eventually $\mathsf{BinEnc}(X) = 0$, contradicting $x_{=0}$. So either way B_2 must be RUP.

Now consider the case where $\gamma \cdot \max(\gamma b, 1) \neq b$. It must be then that $\gamma \cdot \max(\gamma b, 1) = \gamma$, and so $\gamma \cdot \min(\gamma b, -1) = b$, and $C_1(-\gamma,)$ is

$$x_n \Rightarrow \gamma \sum_{i=0}^{n-1} 2^i x_i - \gamma \sum_{i=0}^{n} 2^i |x|_i \geq 0$$

so B_2 is in this case syntactically implied by B' and B_1 is RUP by precisely the same argument as B_2 in the previous case. □

Similarly, we can derive reified bounds on the original product variable Z from reified bounds on its magnitude via Justification Subprocedure 7.4 below.

Correctness Proof for Justification Subprocedure 7.4. C_2 is RUP, since $\mathcal{R}$ contains $\overline{x}_{=0}$ and $\overline{y}_{=0}$, which ensures via unit propagation $z_k = x_n \oplus y_m$ and hence $z_k^{-\sigma_X \cdot \sigma_Y}$ propagates after propagation of $x_n^{\sigma_X}$ and y_m.

Next, (using the fact that $\sigma_Z \cdot \sigma_Z = 1$), $C_Z(\sigma_Z, -\sigma_Z \gamma)$ is

$$z_n^{\sigma_Z} \Rightarrow -\sigma_Z \gamma \sum_{i=0}^{k-1} 2^i z_i - \gamma \sum_{i=0}^{k} 2^i |z|_i \geq -2^{k-1} \cdot \gamma \cdot (\sigma_Z + 1). \tag{7.49}$$

which can be written without reification syntactic sugar as

$$K \cdot z_n^{-\sigma_Z} - \sigma_Z \gamma \sum_{i=0}^{k-1} 2^i z_i - \gamma \sum_{i=0}^{k} 2^i |z|_i \geq -2^{k-1} \cdot \gamma \cdot (\sigma_Z + 1), \tag{7.50}$$

for some suitably large K. Since $2^{k+1} - 1 = \sum_{i=0}^{k} 2^i$, we may assume that $K \leq 2^{k+1} - 1$ since regardless of the values of σ_Z and γ this is certainly large enough to ensure the correctness of the reification (as per Proposition 2.2).

Justification Subprocedure 7.4 (Channelling magnitude bounds to two's complement).

Preconditions: $b \in \mathbb{Z}$; $\sigma_X, \sigma_Y, \gamma \in \{1, -1\}$; $\mathcal{R} \supseteq \{x_n^{\sigma_X}, y_{m-1}^{\sigma_Y}, \overline{x}_{=0}, \overline{y}_{=0}\}$ *is a set of atomic literals for which we have already derived* $\mathcal{R} \Rightarrow \gamma \cdot \sum_{i=0}^{k-1} 2^i |z|_i \geq b$.

Procedure:

1: **proc** ChannelMagToTC($b, \sigma_X, \sigma_Y, \mathcal{R}$)
2: $C_1 \leftarrow$ **get** $\mathcal{R} \Rightarrow \gamma \cdot \sum_{i=0}^{k-1} 2^i |z|_i \geq b$
3: $\sigma_Z \leftarrow -\sigma_X \cdot \sigma_Y$
4: $C_2 \leftarrow$ **rup** $\mathcal{R} \Rightarrow z_k^{\sigma_Z} \geq 1$
5: $C_3 \leftarrow$ **cut** $(2^{k+1} - 1) \cdot C_2 + C_Z(\sigma_Z, -\sigma_Z\gamma)$
6: $C_4 \leftarrow$ **imp** $\mathcal{R} \Rightarrow -\sigma_Z\gamma \sum_{i=0}^{k-1} 2^i z_i - \gamma \sum_{i=0}^{k} 2^i |z|_i \geq -2^{k-1} \cdot \gamma \cdot (\sigma_Z + 1)$ **from** C_3
7: **if** $\gamma = 1$
8: $C_5 \leftarrow C_2 \cdot 2^k$
9: **else**
10: $C_5 \leftarrow$ **imp** $\gamma\sigma_Z \cdot 2^k z_k \geq 2^k \cdot \gamma \cdot ({}^{(\sigma_Z+1)}/_2)$ **from** $0 \geq 0$
11: $C_6 \leftarrow$ **cut** $C_1 + C_4 + C_5$
12: **return imp** $\mathcal{R} \Rightarrow \sigma_Z\gamma \cdot 2^k z_k - \sigma_Z\gamma \cdot \sum_{i=0}^{k} 2^i z_i \geq b$ **from** C_6

For compactness, let us write $K' := 2^{k+1} - 1$. By considering each of the cases $\sigma_Z = \pm 1$ we can rewrite and compute the cutting planes operation on line 5

$$C_2{\restriction_{\mathcal{R}}} \cdot K' = \qquad K' \cdot \sigma_Z z_k \geq K' \cdot {}^{(\sigma_Z+1)}/_2;$$

$$C_Z(\sigma_Z, -\sigma_Z\gamma){\restriction_{\mathcal{R}}} = \qquad -\sigma_Z \cdot K \cdot z_n - \sigma_Z\gamma \sum_{i=0}^{k-1} 2^i z_i - \gamma \sum_{i=0}^{k} 2^i |z|_i \geq (-2^k \cdot \gamma - K) \cdot {}^{(\sigma_Z+1)}/_2;$$

$$\text{so} \quad C_3{\restriction_{\mathcal{R}}} = \sigma_Z \cdot (K' - K) \cdot z_n - \sigma_Z\gamma \sum_{i=0}^{k-1} 2^i z_i - \gamma \sum_{i=0}^{k} 2^i |z|_i \geq (K' - K - 2^k \cdot \gamma) \cdot {}^{(\sigma_Z+1)}/_2.$$

And we can similarly validate that

$$\sigma_z \cdot (K' - K) \cdot z_k \geq -(K' - K) \cdot {}^{(\sigma_Z+1)}/_2 \tag{7.51}$$

is always a multiple of literal axioms (since $K' - K \geq 0$).

This tells us that $C_4{\restriction_{\mathcal{R}}}$ is syntactically implied by $C_3{\restriction_{\mathcal{R}}}$ and hence, by Theorem 2.1, C_4 is syntactically implied by C_3.

Next, we can see that, if $\gamma = 1$, $C_5\restriction_{\mathcal{R}}$ is

$$\begin{aligned} & 2^k z_k^{\sigma_Z} \geq 2^k \\ = \quad & \sigma_Z \cdot 2^k z_k \geq 2^{k-1} \cdot (\sigma_Z + 1) \\ = \; & \gamma\sigma_Z \cdot 2^k z_k \geq \gamma \cdot 2^{k-1} \cdot (\sigma_Z + 1). \end{aligned}$$

On the other, hand, if $\gamma = -1$, then

$$\begin{aligned} & 2^k z_k^{-\sigma_Z} \geq 0 \\ = \; & -\sigma_Z \cdot 2^k z_k \geq -2^{k-1} \cdot (\sigma_Z + 1) \\ = \; & -\sigma_Z \cdot 2^k z_k \geq -2^{k-1} \cdot (\sigma_Z + 1) \\ = \quad & \gamma\sigma_Z \cdot 2^k z_k \geq \gamma \cdot 2^{k-1} \cdot (\sigma_Z + 1), \end{aligned}$$

and hence C_5 can indeed be built from literal axioms.

Either way, $C_5\restriction_{\mathcal{R}}$ ends up being the same thing. This finally, tells us that

$$C_1\restriction_{\mathcal{R}} + C_4\restriction_{\mathcal{R}} + C_5\restriction_{\mathcal{R}} \quad \text{yields} \quad \sigma_Z\gamma \cdot 2^k z_k - \sigma_Z\gamma \cdot \sum_{i=0}^{k} 2_i^z \geq b; \tag{7.52}$$

and so by Theorem 2.1 the returned constraint is indeed syntactically implied by $C_1 + C_4 + C_5$. □

7.3 Justifying Bounds-Consistent Multiplication

A CP bounds propagator for the multiplication constraint $X \times Y = Z$ will typically apply pruning rules to each variable in turn, narrowing the bounds on one variable based on the bounds on the other two.

Recall from Chapter 3 the notion of bounds consistency; which can be expressed more formally for integer domains (following Choi et al. [42]) as follows:

- A domain state dom_t is *bounds-$\mathbb{Z}$-consistent* for a constraint C, if for each variable $X_i \in \mathrm{scp}(\mathsf{C})$, and for each $b_i \in \{\min(\mathrm{dom}_t(X_i)), \max(\mathrm{dom}_t(X_i))\}$ there exists *integers* b_j, $j \in \{1, \ldots, n\} \setminus \{i\}$ with $\min(\mathrm{dom}_t(X_j)) \leq b_j \leq \max(\mathrm{dom}_t(X_j)$ such that $X_1 = b_1, \ldots, X_n = b_n$ satisfies C.

Our aim in this section is to show how to justify any inference made by a propagator as part of enforcing bounds consistency. This will make use of the subprocedures for deriving bounds and channelling that we introduced in the previous sections.

7.3.1 Justifying Infeasible Bounds

We can construct infeasibility justifications when the set of bounds on X, Y and Z do not allow for any solutions to $X \times Y = Z$. This works by writing out a case based argument in the proof log. We make assumptions about the signs of X and Y, and show that in each case, if the assumptions are possible, a contradictory bound on Z must be achieved.

Justification Procedure 7.5 (Justifying infeasible bounds for multiplication).

Preconditions: *$l_X \leq u_X; l_Y \leq u_Y; l_Z \leq u_Z$ are bounds for the variables X, Y, and Z respectively that are infeasible with respect to the multiplication constraint $X \times Y = Z$, i.e. there does not exist a triple of integers a, b, c with*

$$l_X \leq a \leq u_X; \quad l_Y \leq a \leq u_Y; \quad l_Z \leq a \leq u_Z; \quad \text{and } a \cdot b = c; \tag{7.53}$$

$\mathsf{TCBits}(V)$ gives the bit sum for the two's complement representation for each variable $V \in \{X, Y, Z\}$ used as part of Encoding Procedure 7.2; $\mathsf{IsLB}(C)$ returns true if a constraint is of the form $\mathcal{L} \Rightarrow B$ where B has all positive coefficients on the left-hand side, and false otherwise.

Procedure: *For definitions of DeriveConditionalBounds and ResolveSigns, see Justification Procedure 7.5 (cont.) below.*

1: DeriveConditionalBounds(l_X, u_X, l_Y, u_Y)
2: **pbc** $x_{\geq l_X} \wedge \overline{x}_{\geq u_X+1} \wedge y_{\geq l_Y} \wedge \overline{y}_{\geq u_Y+1} \wedge z_{\geq l_Z} \wedge \overline{z}_{\geq u_Z+1} \Rightarrow 0 \geq 1$
3: **subproof of** $x_{\geq l_X} \wedge \overline{x}_{\geq u_X+1} \wedge y_{\geq l_Y} \wedge \overline{y}_{\geq u_Y+1} \wedge z_{\geq l_Z} \wedge \overline{z}_{\geq u_Z+1} \Rightarrow 0 \geq 1$
4: ResolveSigns()

Correctness Proof. In what follows we will use the shorthands

$$\Sigma_X := \mathsf{TCBits}(X) = -2^n x_n + \textstyle\sum_{i=0}^{n-1} 2^i x_i, \quad \Sigma_{|X|} := \textstyle\sum_{i=0}^{n} 2^i |x|_i, \tag{7.54}$$

with Σ_Y, $\Sigma_{|Y|}$, Σ_Z, $\Sigma_{|Z|}$ defined similarly. For any integer b we will also use

$$b^+ := \max(b, 1), \quad \text{and} \quad b^- := \min(b, -1). \tag{7.55}$$

And finally we will let

$$\mathcal{R} := x_{\geq l_X} \wedge \overline{x}_{\geq u_X+1} \wedge y_{\geq l_Y} \wedge \overline{y}_{\geq u_Y+1}. \tag{7.56}$$

So with this notation, the first two lines in the initial loop of DeriveConditionalBounds will derive

$$\mathcal{R} \Rightarrow \Sigma_X \geq l_X; \qquad \mathcal{R} \Rightarrow -\Sigma_X \geq -u_X; \tag{7.57}$$

$$\mathcal{R} \Rightarrow \Sigma_Y \geq l_Y; \qquad \mathcal{R} \Rightarrow -\Sigma_Y \geq -u_Y. \tag{7.58}$$

Justification Procedure 7.5 (continued)

proc DeriveConditionalBounds(l_X, u_X, l_Y, u_Y)
  $\mathcal{R} \leftarrow \{x_{\geq l_X}, y_{\geq l_Y}, \overline{x}_{\geq u_X+1}, \overline{y}_{\geq u_Y+1}\}$;
  magBounds$[X] \leftarrow \emptyset$, magBounds$[Y] \leftarrow \emptyset$
  for $V \in \{X, Y\}$
    LB$[V] \leftarrow$ **rup** $\mathcal{R} \Rightarrow$ TCBits$(V) \geq l_V$
    UB$[V] \leftarrow$ **rup** $\mathcal{R} \Rightarrow -$TCBits$(V) \geq -u_V$
    magBounds$[V] \leftarrow$ magBounds $\cup$ ChannelTCToMag$(l_V, 1)$
    magBounds$[V] \leftarrow$ magBounds $\cup$ ChannelTCToMag$(-u_V, -1)$

  for $(C_X, C_Y) \in$ magBounds$[X] \times$ magBounds$[Y]$
    if IsLB$(C_X) \wedge$ IsLB(C_Y)
      $\mathcal{R} \wedge x_n^{\sigma_X} \wedge y_m^{\sigma_Y} \Rightarrow S \geq b \leftarrow$ DeriveProductLowerBound(C_X, C_Y)
      ChannelMagToTC$(b, \sigma_X, \sigma_Y, \mathcal{R})$
    else if $\neg$IsLB$(C_X) \wedge \neg$IsLB(C_Y)
      $\mathcal{R} \wedge x_n^{\sigma_X} \wedge y_m^{\sigma_Y} \Rightarrow S \geq b \leftarrow$ DeriveProductUpperBound(C_X, C_Y)
      ChannelMagToTC$(b, \sigma_X, \sigma_Y, \mathcal{R})$

  rup $x_{=0} \Rightarrow$ TCBits$[Z] \geq 0$
  rup $y_{=0} \Rightarrow$ TCBits$[Z] \geq 0$
  rup $x_{=0} \Rightarrow -$TCBits$[Z] \geq 0$
  rup $y_{=0} \Rightarrow -$TCBits$[Z] \geq 0$

proc ResolveSigns( )
  $C_1 \leftarrow$ **rup** $\overline{x}_n \wedge \overline{y}_n \wedge \overline{x}_{=0} \wedge \overline{y}_{=0} \Rightarrow 0 \geq 1$
  $C_2 \leftarrow$ **rup** $\overline{x}_n \wedge y_n \wedge \overline{x}_{=0} \wedge \overline{y}_{=0} \Rightarrow 0 \geq 1$
  $C_3 \leftarrow$ **rup** $x_n \wedge y_n \wedge \overline{x}_{=0} \wedge \overline{y}_{=0} \Rightarrow 0 \geq 1$
  $C_4 \leftarrow$ **rup** $x_n \wedge \overline{y}_n \wedge \overline{x}_{=0} \wedge \overline{y}_{=0} \Rightarrow 0 \geq 1$
  $C_5 \leftarrow$ **rup** $x_{=0} \Rightarrow 0 \geq 1$
  $C_6 \leftarrow$ **rup** $y_{=0} \Rightarrow 0 \geq 1$
  cut sat(sat(sat(sat$(C_1 + C_2)$ + sat$(C_3 + C_4)) + C_5) + C_6)$

These are RUP due to the definitions of the literals in $\mathcal{R}$, and the equality constraint with the TCBits defined in Encoding Procedure 7.2.

Furthermore, each of these are in the form required by the assumptions of Justification Subprocedure 7.3. So after both full iterations of the loop we will have derived the following

constraints, stored in magBounds$[X]$ and magBounds$[Y]$.

$$\mathcal{R} \wedge \overline{x}_n \wedge \overline{x}_{=0} \Rightarrow \Sigma_{|X|} \geq \quad l_X^+, \qquad \mathcal{R} \wedge x_n \wedge \overline{x}_{=0} \Rightarrow -\Sigma_{|X|} \geq \quad l_X^-, \tag{7.59}$$

$$\mathcal{R} \wedge x_n \wedge \overline{x}_{=0} \Rightarrow \Sigma_{|X|} \geq -u_X^-, \qquad \mathcal{R} \wedge \overline{x}_n \wedge \overline{x}_{=0} \Rightarrow -\Sigma_{|X|} \geq -u_X^+, \tag{7.60}$$

$$\mathcal{R} \wedge \overline{y}_n \wedge \overline{y}_{=0} \Rightarrow \Sigma_{|Y|} \geq \quad l_Y^+, \qquad \mathcal{R} \wedge y_n \wedge \overline{y}_{=0} \Rightarrow -\Sigma_{|Y|} \geq \quad l_Y^-, \tag{7.61}$$

$$\mathcal{R} \wedge y_n \wedge \overline{y}_{=0} \Rightarrow \Sigma_{|Y|} \geq -u_Y^-, \qquad \mathcal{R} \wedge \overline{y}_n \wedge \overline{y}_{=0} \Rightarrow -\Sigma_{|Y|} \geq -u_Y^+. \tag{7.62}$$

Since the constraints in the first column are (reified) positive lower bounds, and those in the second are reified positive upper bounds, the next loop will apply either Justification Subprocedure 7.1 or Justification Subprocedure 7.2 for each compatible set of bounds.

If we let ${}^{\sigma_X}\wedge^{\sigma_Y} := \mathcal{R} \wedge x_n^{-\sigma_X} \wedge y_n^{-\sigma_Y} \wedge \overline{x}_{=0} \wedge \overline{y}_{=0}$ where $\sigma_1, \sigma_2 \in \{+, -\}$ (meaning ± 1), the resulting eight constraints are

$${}^{+}\wedge^{+} \Rightarrow \Sigma_{|Z|} \geq \quad l_X^+ l_Y^+, \qquad {}^{+}\wedge^{-} \Rightarrow -\Sigma_{|Z|} \geq \quad u_X^+ l_Y^-, \tag{7.63}$$

$${}^{-}\wedge^{-} \Rightarrow \Sigma_{|Z|} \geq \quad u_X^- u_Y^-, \qquad {}^{-}\wedge^{+} \Rightarrow -\Sigma_{|Z|} \geq \quad l_X^- u_Y^+, \tag{7.64}$$

$${}^{-}\wedge^{+} \Rightarrow \Sigma_{|Z|} \geq -u_X^- l_Y^+, \qquad {}^{+}\wedge^{+} \Rightarrow -\Sigma_{|Z|} \geq -u_X^+ u_Y^+, \tag{7.65}$$

$${}^{+}\wedge^{-} \Rightarrow \Sigma_{|Z|} \geq -l_X^+ u_Y^-, \qquad {}^{-}\wedge^{-} \Rightarrow -\Sigma_{|Z|} \geq -l_X^- l_Y^-. \tag{7.66}$$

Each of these is in now in the required form to apply Justification Subprocedure 7.4. Note here that the product σ_X, σ_Y of the signs in the condition ${}^{\sigma_X}\wedge^{\sigma_Y}$ determines whether the inequality should be flipped. So after all the iterations of the second loop, we have

$${}^{+}\wedge^{+} \Rightarrow \quad \Sigma_Z \geq \quad l_X^+ l_Y^+, \qquad {}^{+}\wedge^{-} \Rightarrow \quad \Sigma_Z \geq \quad u_X^+ l_Y^-, \tag{7.67}$$

$${}^{-}\wedge^{-} \Rightarrow \quad \Sigma_Z \geq \quad u_X^- u_Y^-, \qquad {}^{-}\wedge^{+} \Rightarrow \quad \Sigma_Z \geq \quad l_X^- u_Y^+, \tag{7.68}$$

$${}^{-}\wedge^{+} \Rightarrow -\Sigma_Z \geq -u_X^- l_Y^+, \qquad {}^{+}\wedge^{+} \Rightarrow -\Sigma_Z \geq -u_X^+ u_Y^+, \tag{7.69}$$

$${}^{+}\wedge^{-} \Rightarrow -\Sigma_Z \geq -l_X^+ u_Y^-, \qquad {}^{-}\wedge^{-} \Rightarrow -\Sigma_Z \geq \quad -l_X^- l_Y^-. \tag{7.70}$$

Following this, the next four constraints to be derived (conditioned on $x_{=0}$ or $y_{=0}$), all follow by RUP because their negation will fix all the bits in TCBits(X) and TCBits(Y) respectively to 0, which then from Encoding Procedure 7.2 must fix TCBits(Z) to 0. In the current notation these are

$$x_{=0} \Rightarrow \quad \Sigma_Z \geq 0, \qquad y_{=0} \Rightarrow \quad \Sigma_Z \geq 0, \tag{7.71}$$

$$x_{=0} \Rightarrow -\Sigma_Z \geq 0, \qquad y_{=0} \Rightarrow -\Sigma_Z \geq 0. \tag{7.72}$$

Now finally, to see that the proof by contradiction (pbc) succeeds, we first note that the negation will propagate all the literals in $\mathcal{R}$. We then simply need to argue each of the constraints in the ResolveSigns procedure will be RUP, since the cutting planes operation on line 27 is

performing clausal resolution (Theorem 2.2) to derive $0 \geq 1$.

Starting with the first four constraints on lines 21 to 24, we can see that they have the form

$$x_n^{-\sigma_X} \wedge y_n^{-\sigma_Y} \wedge \overline{x}_{=0} \wedge \overline{y}_{=0} \Rightarrow 0 \geq 1 \tag{7.73}$$

for $\sigma_X, \sigma_Y \in \{1, -1\}$, and hence the negation of each propagates all the literals in ${}^{\sigma_1}\!\wedge^{\sigma_2}$ as defined above. Then looking at the constraints we derived in (7.67)–(7.70) there will be exactly two of these with matching reifying conditions on the right-hand side:

$${}^{\sigma_1}\!\wedge^{\sigma_2} \Rightarrow \Sigma_Z \geq a_X^{\sigma_1} \cdot a_Y^{\sigma_2} \qquad \text{and} \qquad {}^{\sigma_1}\!\wedge^{\sigma_2} \Rightarrow -\Sigma_Z \geq -b_X^{\sigma_1} \cdot b_Y^{\sigma_2} \tag{7.74}$$

where $a_X, b_X \in \{l_X, u_X\}$ and $a_Y, b_Y \in \{l_Y, u_Y\}$.

Now, we can exhaustively validate for these that propagation of ${}^{\sigma_X}\!\wedge^{\sigma_Y}$ must either reach an immediate contradiction due to incompatible bounds on X and Y, or else the right-hand side of the constraints are of the form $\pm a \cdot b$ where $l_X \leq a \leq u_X$ and $l_Y \leq b \leq u_Y$.

For example, for ${}^{-}\!\wedge^{+} \Rightarrow -\Sigma_Z \geq -u_X^- l_Y^+$ if x_n and $\overline{x}_{=0}$ are not immediately falsified by propagation of $x_{\geq l_X}$, then we must have $l_X \leq -1$ and so $l_X \leq u_X^- \leq u_X$; and $u_X \geq 1$ and so $l_X \leq l_X^+ \leq u_X$ (using the fact that $l_X \leq u_X$ and $l_Y \leq u_Y$). The other seven cases are very similar.

But then, assuming we don't get a propagation contradiction from ${}^{\sigma_X}\!\wedge^{\sigma_Y}$ alone, it must be the case that either $a_X^{\sigma_1} \cdot a_Y^{\sigma_2} > u_Z$ or $b_X^{\sigma_1} \cdot b_Y^{\sigma_2} < l_Z$, since otherwise these are bounds-compatible solutions to $X \times Y = Z$, which we are assuming as precondition do not exist. Hence, we would have at least one contradictory bound on Σ_Z for $\mathcal{R}$, and so we would reach a contradiction as per Theorem 2.7.

For four 0-bounds in (7.71) and (7.72) this is even easier to see: either $\mathcal{R}$ excludes 0, and so we get immediate contradiction, or else 0 is within the given bounds for X/Y and so 0 cannot be within the given bounds for Z, or we would have a feasible solution.

So all the resolvents are indeed RUP and thus the final proof by contradiction will succeed. □

7.3.2 Justifying Product Bounds

Now let us consider narrowing the bounds on Z based on the bounds of X and Y. Suppose the current domains of X and Y are known to be

$$\mathrm{dom}_t(X) \subseteq \{l_X \ldots u_X\}, \text{ and } \mathrm{dom}_t(Y) \subseteq \{l_Y \ldots u_Y\}. \tag{7.75}$$

Then the solver can infer $\mathrm{dom}_t(Z) \subseteq \{\inf E \ldots \sup E\}$ where $E = \{l_X l_Y, l_X u_Y, u_X l_Y, u_X u_Y\}$, and prune any values outside this range [179]. We can justify this in a manner very similar to the infeasibility justification procedure above.

Justification Procedure 7.6 (Justifying bounds on the product variable).

Preconditions: $l_X = \min(\mathrm{dom}_t(X))$; $u_X = \max(\mathrm{dom}_t(X))$; $l_Y = \min(\mathrm{dom}_t(Y))$; $u_Y = \max(\mathrm{dom}_t(Y))$; $l_Z = \min(\mathrm{dom}_t(z))$; $u_Z = \max(\mathrm{dom}_t(z))$; *(and hence a bounds consistent propagator can infer* $\min E \leq Z \leq \max E$ *for* E *defined as above);* $\mathsf{TCBits}(V)$ *gives the bit sum for the two's complement representation for each variable* $V \in \{X, Y, Z\}$ *used as part of Encoding Procedure 7.2;* $\mathsf{IsLB}(C)$ *returns true if a constraint is of the form* $\mathcal{L} \Rightarrow B$ *where* B *has all positive coefficients on the left-hand side, and false otherwise.*

Procedure: *For definitions of DeriveConditionalBounds and ResolveSigns, see again Justification Procedure 7.5 (cont.) above.*

1: DeriveConditionalBounds(l_X, u_X, l_Y, u_Y)
2: $l'_Z \leftarrow \min\{l_X l_Y, l_X u_Y, u_X l_Y, u_X u_Y\}$
3: $u'_Z \leftarrow \max\{l_X l_Y, l_X u_Y, u_X l_Y, u_X u_Y\}$
4: $\mathcal{R} \leftarrow \{x_{\geq l_X}, y_{\geq l_Y}, \overline{x}_{\geq u_X+1}, \overline{y}_{\geq u_Y+1}\}$;
5: **pbc** $\mathcal{R} \Rightarrow z_{\geq l'_Z} \geq 1$
6: **subproof of** $\mathcal{R} \Rightarrow z_{\geq l'_Z} \geq 1$
7: ResolveSigns()
8: **pbc** $\mathcal{R} \Rightarrow \overline{z}_{\geq u'_Z+1} \geq 1$
9: **subproof of** $\mathcal{R} \Rightarrow \overline{z}_{\geq u'_Z+1} \geq 1$
10: ResolveSigns()

Correctness Proof. The first four lines will derive the same constraints (7.67)–(7.72) by the same arguments as in Justification Procedure 7.5.

Then, the argument that the first pbc step succeeds can also be adapted.

This is because if $Z \geq l'_Z$ is a valid pruning inference, there can be no solutions with Z in the range $\{l_Z, \ldots, l'_Z - 1\}$, and X and Y in their respective ranges. So the negation of the constraint from line 5 will propagate an infeasible set of bound literals for $X \times Y = Z$ and hence the steps in ResolveSigns will succeed by exactly the same arguments.

The same is true for the second pbc step: if $Z \leq u'_Z$ is a valid pruning inference, there can be no bounds-compatible solutions with Z in the range $\{u'_Z + 1, \ldots, u_Z\}$. So this will also succeed. □

7.3.3 Justifying Multiplicand Bounds

The bounds propagator for $X \times Y = Z$ will also compute bounds on X based on bounds of Y and Z. If we show how to justify this, we have likewise shown by symmetry how to justify bounds on Y based on X and Z, and thus completed the description of our method for adding

proof logging to a bounds-consistent multiplication propagator. Unlike bounds on the product variable, these inferences cannot be compactly expressed as a simple min/max operation, since the solver must deal with different cases depending on whether 0 is within the bounds of Z.

Suppose that the current domains of Y and Z are known to be $D_Y \subseteq \{l_Y \dots u_Y\}$, and $D_z \subseteq \{l_Z \dots u_Z\}$. First, if $l_Y \leq 0 \leq u_Y$ and $l_Z \leq 0 \leq u_Z$ then we cannot infer anything about the bounds on X, since any value is possible. Next, if $l_Y = u_Y = 0$ and $0 \notin \{l_Z \dots u_Z\}$ then we can immediately infer contradiction, and the justification follows by RUP. In any other case, we can compute bounds on X which may reduce the domain, and a complete set of cases for this is given by Schulte and Stuckey [179] and also Apt and Zoeteweij [9].

Fortunately, we do not need to take each case in turn for constructing justifications; instead we can rely solely on the fact the propagator makes bounds-consistency-enforcing inferences. In particular, if a narrowing of the bounds is valid, there can be no solutions in the excluded range. Justification Procedure 7.7 and Justification Procedure 7.8 detail how this is achieved.

Justification Procedure 7.7 (Justifying lower bounds on the multiplicands).

Preconditions: $l_X = \min(\text{dom}_t(X))$; $u_X = \max(\text{dom}_t(X))$; $l_Z = \min(\text{dom}_t(Z))$; $u_Z = \max(\text{dom}_t(Z))$; $l_Y = \min(\text{dom}_t(Y))$; $u_Y = \max(\text{dom}_t(Y))$; $l'_X > l_X$ *is a new lower bound on* X *computed by a bounds-consistency propagator based on* dom_t; $\mathsf{TCBits}(V)$ *gives the bit sum for the two's complement representation for each variable* $V \in \{X, Y, Z\}$ *used as part of Encoding Procedure 7.2;* $\mathsf{IsLB}(C)$ *returns true if a constraint is of the form* $\mathcal{L} \Rightarrow B$ *where* B *has all positive coefficients on the left-hand side, and false otherwise;* DeriveConditionalBounds *and* ResolveSigns *are as defined in Justification Procedure 7.6.*

Procedure:

1: $\mathsf{DeriveConditionalBounds}(l_X, l'_X - 1, l_Y, u_Y)$
2: $\mathcal{R} \leftarrow \{x_{\geq l_X}, y_{\geq l_Y}, \overline{y}_{\geq u_Y+1}, z_{\geq l_Z} \wedge \overline{z}_{\geq u_Z+1}\}$
3: **pbc** $\mathcal{R} \Rightarrow x_{\geq l'_X} \geq 1$
4: **subproof of** $\mathcal{R} \Rightarrow x_{\geq l'_X} \geq 1$
5: $\quad \mathsf{ResolveSigns}()$

Correctness Proof for Justification Procedure 7.7.

Justification Procedure 7.8 (Justifying upper bounds on the multiplicands).

Preconditions: *Identical to Justification Procedure 7.7, except instead of new lower bounds $u'_X < u_X$ is a new* upper *bound on X computed by a bounds-consistency propagator based on* dom_t.

Procedure:

1: $\mathcal{R} \leftarrow \{\overline{x}_{\geq u_X+1}, y_{\geq l_Y}, \overline{y}_{\geq u_Y+1}, z_{\geq l_Z} \wedge \overline{z}_{\geq u_Z+1}\}$
2: **pbc** $\mathcal{R} \Rightarrow \overline{x}_{\geq u'_X+1} \geq 1$
3: **subproof of** $\mathcal{R} \Rightarrow \overline{x}_{\geq u'_X+1} \geq 1$
4: ResolveSigns()

Letting $u_X = l'_X - 1$, the procedure DeriveConditionalBounds will derive the constraints

$$^{+}\wedge^{+} \Rightarrow \Sigma_Z \geq l^{+}_X l^{+}_Y; \qquad ^{-}\wedge^{+} \Rightarrow \Sigma_Z \geq l^{-}_X u^{+}_Y; \tag{7.76}$$

$$^{+}\wedge^{-} \Rightarrow -\Sigma_Z \geq -l^{+}_X u^{-}_Y; \qquad ^{-}\wedge^{-} \Rightarrow -\Sigma_Z \geq -l^{-}_X l^{-}_Y; \tag{7.77}$$

$$x_{=0} \Rightarrow \Sigma_Z \geq 0; \qquad x_{=0} \Rightarrow -\Sigma_Z \geq 0; \tag{7.78}$$

$$y_{=0} \Rightarrow \Sigma_Z \geq 0; \qquad y_{=0} \Rightarrow -\Sigma_Z \geq 0. \tag{7.79}$$

This follows again by same argument as in the proof for Justification Procedure 7.6.

Now, assuming increasing the lower bound of X to l'_X is a valid inference, there can be no solutions to $X \times Y = Z$ compatible with the bounds propagated by the negation of the PBC constraint. So exactly the same argument as in Justification Procedure 7.5 applies, to argue that the proof by contradiction statement will succeed. □

Correctness Proof for Justification Procedure 7.8.

Analogous to the proof of Justification Procedure 7.7. □

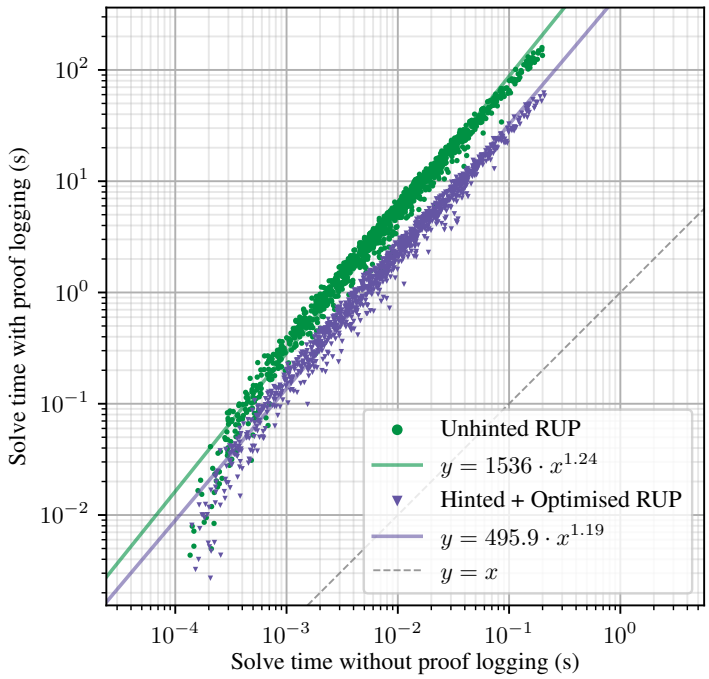
7.4 Implementation and Experiments

To validate the approach, we implemented a new proof logging bounds-consistent multiplication propagator within the *Glasgow Constraint Solver*. The solver previously had support for domain-consistent multiplication and division via tabulation: writing out all possible solutions in advance. Once again we ran experiments to test correctness and roughly measure overhead. This made use of the same experimental set up as in previous chapters, with a set of synthetic instances consisting of a single ternary multiplication constraint and randomly generated domain bounds of increasing size.

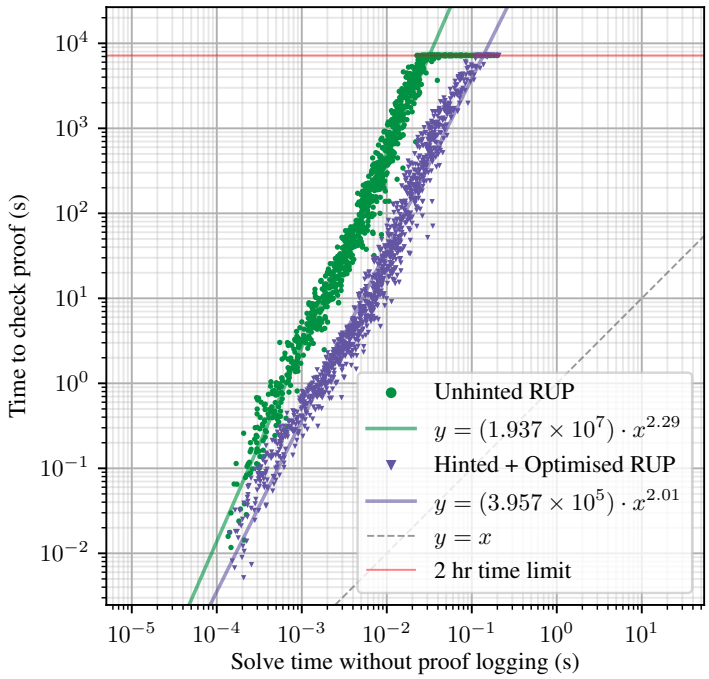
Instances of this kind often lead to the solver discovering a solution and terminating almost immediately. When this happens, it tells us nothing about the practical performance of the implemented justification procedures. Hence, we only present results here where the randomly generated bounds lead the solver to explore at least one failing search path, and the bounds-consistent multiplication propagator to be executed at least once.

Unlike previous chapters, the short reasons optimisation is not applicable for the justification procedures being tested here. This is because the number of variables is fixed at three, and the reifying reasons are always just the atomic bounds literals for the current domain state. We did take the opportunity to instead test another simple implementation optimisation: that of adding *hints* to RUP steps. This involves annotating any RUP derivations with a set of constraint IDs that are sufficient to allow for the unit propagation condition. These IDs can be extracted from the correctness proofs for the justification procedure: we simply look to where we have proved on paper that a particular (small) set of constraints allow a further constraint to be RUP (for example line 6 from Justification Subprocedure 7.3). In principle, this should allow for faster verification, since the checker can avoid propagating over the whole current database, at the cost of slightly more writing to the proof log. In the case where the RUP hint would consist of a single constraint ID we can also simplify the proof procedure on the fly by not re-deriving a constraint that already exists. This can apply for example when recovering bounds constraints (e.g. line 5 in Justification Procedure 7.6) in the case when TCBits are the same as the bits on the original variable. Since this can result in writing fewer constraints to the proof overall, keeping track of hints in this way can also provide a saving for logging overhead. The results of the experiments, with and without RUP hints, are shown in Figures 7.1 and 7.2.

First, it is clear here that the overheads of proof logging for these are significant, more so than for other propagators implemented in this thesis. Writing proof logs for the more difficult multiplication instances incurs a slowdown of several orders of magnitude. This is not particularly surprising, given that the bounds-consistent computations are a constant number of integer multiplication operations, which are extremely cheap on modern hardware for fixed-width integers. In contrast, our PB proof logging procedures require writing a polynomial number of steps to the proof for each execution of the propagator ($O(n \cdot m)$ where n and m are the number of bits required to represent the range of values in the initial domains of X and Y respectively). Furthermore, checking time grows steeply as the difficulty of the problem increases. This is partially due to the structure of the instances: since trivial cases where a solution is found immediately are ignored, the remaining harder instances appear to fall into a bad case for the propagator implementation, where every value in a large variable domain is branched on. Shallow, high-breadth search trees are in turn a poor case for checking RUP-based backtracking justifications, since all the backtracking justifications for failures have to be kept in the checker's database at the current decision level, meaning each RUP step can get successively more expensive with more branches. Improvements to the *Glasgow Constraint Solver*'s branching strategy or to

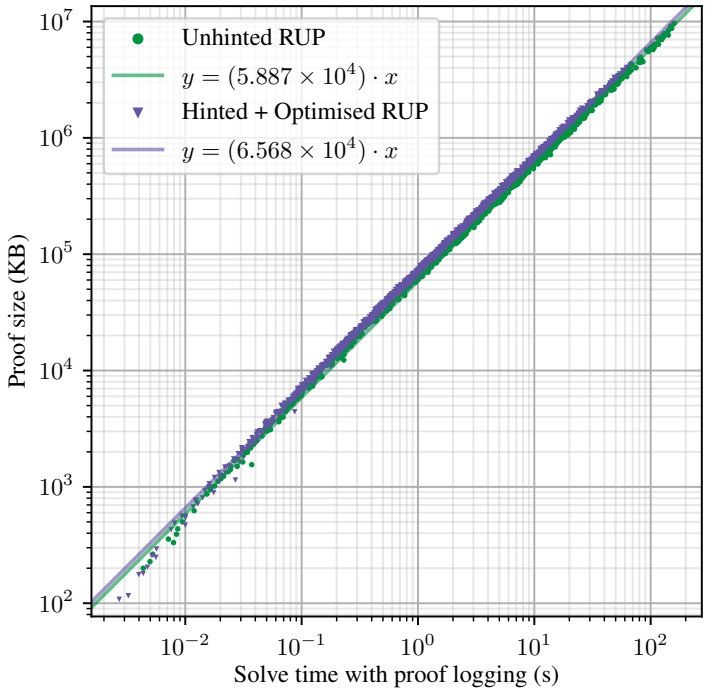


(a) Overhead of proof logging during solving.

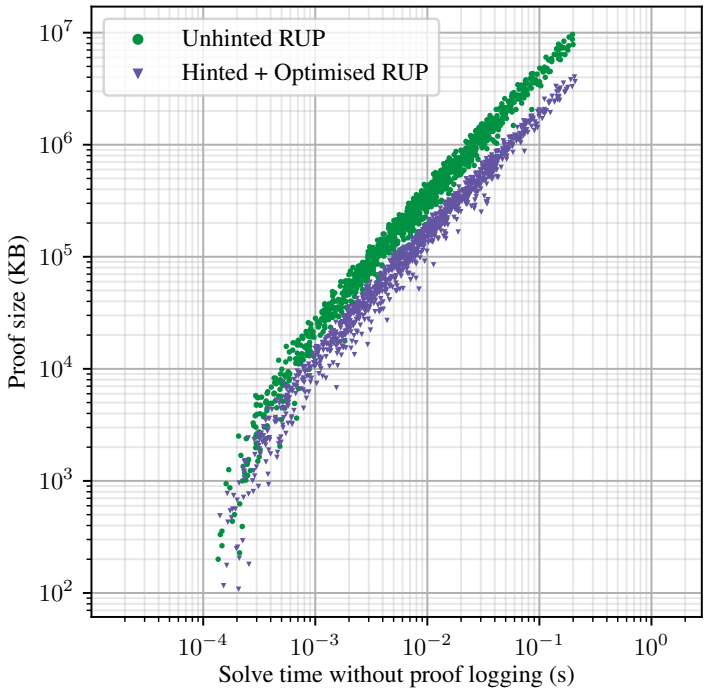


(b) Proof checking time relative to solve time.

Figure 7.1: Timing results for random multiplication instances.



(a) Proof size relative to time to produce the proof.



(b) Proof size relative to solve time.

Figure 7.2: Proof size results for random multiplication instances.

the checker, could be considered to avoid such pathological behaviour, although this falls outwith the scope of justifying constraint propagation itself.

The optimisations of tracking hints and outputting fewer RUP steps do make a difference, but they improve logging time only by a moderate constant factor and checking time by about one order of magnitude. Even in this implementation a majority of RUP steps remain unhinted, since adding hints to backtracking justifications requires more significant bookkeeping within the core search mechanism of the solver.

7.5 Conclusions

While many constraint propagators have been almost trivial to add pseudo-Boolean proof logging, bounds-consistent multiplication requires significantly more effort and much more intricate justification procedures. Unlike Circuit, where it was perhaps not too surprising that a relatively complex propagator would require a complex justification procedure, the underlying propagator here is quite simple, and can be extremely efficient for narrowing domain bounds. So on the one hand we have shown that pseudo-Boolean proof logging is still applicable for non-linear constraints that are less easily expressible in terms of linear inequalities. On the other hand, this is the closest we have to a case where we can say that PB reasoning has been found to be ill-suited for justifying a CP technique.

Nevertheless, this is, to the author's knowledge, the first certifying propagator for a bounds-consistent multiplication constraint. As with Circuit, the specialised justifications here are not directly applicable to other constraints, but the techniques exemplified here, such as using proof by contradiction to effectively handle case-based arguments relating to signed values, are likely to help inform future design of justification procedures. Additionally, the multiplication constraint itself can obviously be used as a building block to implement propagators for division, modulo and polynomial constraint types, expanding once again the set of constraints that can in principle support proof logging.

Part IV

Conclusions

Chapter 8

Conclusions and Future Work

In this thesis we have advanced pseudo-Boolean certification methods for propagation algorithms in constraint programming. In the process we have gained a better understanding of how pseudo-Boolean derivations relate to CP solving techniques, along with general principles for building justification procedures to support a proof logging solver. We have already given a short summary of contributions and conclusions in isolation at the end of each substantive chapter, but here we will review them in a wider context. We will first return to the original thesis statement and research topics from Chapter 1, and summarise what was achieved in relation to these. Next we will finish with a discussion of possible future work and questions that remain open, including outlining the next steps in the mission to certify CP tools and techniques.

8.1 Summary and Contributions

Our first research topic was as follows.

R1. Expanding and more rigorously defining the methodology of Gocht et al., making explicit the properties required from encodings and proof logging propagation algorithms to guarantee correct proof production.

This was achieved within Chapters 2 and 3, in conjunction with reviewing the necessary background material. We formulated specifically for the first time what it means to represent or "encode" a CP problem as a PB problem for the purpose of a proof logging framework. Namely, we required the encodings to be the result of a six-step transformation procedure. This creates a faithful *linearisation* of constraints with respect to bounds constraints, original CP variables, and auxiliary variables, and then maps it to PB using binary exponential sums. We showed that, due to limited unit propagation properties of PB constraints involving such sums (proved in Chapter 2) we can rely on certain obvious facts described by "atomic" literals always propagating based on an encoded CP problem. This fact was shown to be essential to explaining why a PB-RUP-based methodology for justifying backtracking search in a CP solver always works. It also motivated

the need for creating individual justification procedures using stronger PB reasoning for different propagation inferences made during search.

With this in mind, we specified precisely what a "justification" constraint for a propagation inference must look like and defined "reasons" for the context of PB proof logging. We then standardised a notation for specifying and proving correctness for pseudo-Boolean justification procedures. We used this to explicitly write down previous described or implemented justification procedures and prove that they will always work.

Having motivated further work on pseudo-Boolean justification procedures, and provided a solid basis for their design, we dedicated parts II and III to tackling the remaining research topics.

R2. Developing proof logging methods for efficiently justifying important propagation algorithms for specific constraints types not covered by previous approaches.

We devised new justification procedures for propagation of the following constraints.

1. The SmartTable constraint.
2. The Regular language membership constraint.
3. The Circuit constraint.
4. The ternary multiplication constraint.

In each of these cases we presented the first certifying propagator for the constraint type, to the author's knowledge — Item 4 is a possible exception, since it is unclear whether multiplication constraints were included in the unpublished work of Gange et al. [75]. We designed new specific pseudo-Boolean justification procedures to support derivations for a useful set of inferences.

Importantly, we did not have to make compromises in the way the underlying algorithms work in order to support proof logging. The justification procedures are formulated such that they can be simply added to implementations of established propagation algorithms. This is in keeping with the general philosophy of proof logging; being more akin to adding extra print statements to an otherwise standard solver architecture, rather than re-engineering a solver to itself become a proof production or proof search program.

The further significance of these particular constraints is that they might, for various reasons, have been expected to pose particular barriers to a proof logging approach. SmartTable and Regular have fairly involved, stateful propagators with underlying graph representations; while Circuit has the significant collection of non-monotonic, ad-hoc filtering rules; and multiplication obviously requires non-linear reasoning. With this work, we have refuted the not-unreasonable assumption that there might have been insurmountable challenges for pseudo-Boolean proof logging posed by one or more of these complications.

Furthermore, in part 2 of the thesis we made progress in the third research direction.

R3. Extracting more general proof logging methods from ideas developed for R2 to show, in principle, that larger families of constraint types can be efficiently justified.

In Chapter 4 we argued that since we know how to justify any domain-consistency inferences for a basic SmartTable, this means we also in principle know that pseudo-Boolean proof logging is applicable for propagation of any constraint that is expressible as a smart table. This includes LexGreater, AtMostOne, NotAllEqual, and ValuePrecede. We also extended the justification procedure to support general disjunctions of conjunctions of constraints for which we already have a proof logging procedure (with some caveats), expanding the potential applicability even further. Then in Chapter 5, we achieved a similar general applicability for justifying propagation using states and transitions, both for constraints expressible using Regular, and for more general methods based on decision diagram reasoning.

Lastly, we evidenced new contributions under the fourth research topic.

R4. Implementing proof logging methods within a proof-of-concept CP solver, and demonstrating empirical evidence that they work on a feasible timescale.

Each of the four constraints mentioned above have come with a corresponding new propagator implementation for the *Glasgow Constraint Solver*. Even though we did not go into software engineering details here, the code exists and is publically available as a reference implementation, as well as a means to create strong empirical evidence that the justification procedures work [150]. We have verified thousands of proofs as part of the experimental evaluations in this thesis, and have recorded no checking failures resulting from the final versions of the implemented propagators. Of course, many proof failures were observed during propagator development, and this occasionally allowed the author to identify subtle bugs in the code that may have been difficult to spot using conventional testing.

The overheads observed in practice from adding proof logging remain large, but as noted in the relevant sections, this is in line with similar work in the area. Logging performance on more difficult instances supports the conclusion that large constant factors from disk writing are the biggest barrier to low-overhead certification, even when the proof procedures are theoretically "efficient" and free from any exponential blow-up. We discuss future approaches to tackle this in the next section.

So, to return to the thesis statement:

> *"Pseudo-Boolean proof logging provides a complete and practical means to certify and audit a wide range of modern constraint propagation techniques";*

there is a strong argument that this has been convincingly demonstrated. We have considered disparate propagators that work in several different ways, and showed that despite concepts such as graphs, states, transitions, and components seeming superficially dissimilar from linear inequalities, pseudo-Boolean reasoning remains applicable, both in theory and in practice.

Of course, it is not possible for us to conclude that, therefore, PB proof logging *must* be usable for *all* aspects of CP. There are many propagation methods not covered by any of our existing justification approaches, as we will note in the next section. And even if that were not

the case, CP defined broadly encompasses a huge and ever-expanding range of problem solving techniques, beyond even finite domain integer variables and deterministic filtering algorithms. Settling forever the question of certification in a fully comprehensive way for everything is unlikely to be achievable.

We should also acknowledge a counter-interpretation of some the findings presented in this thesis: that pseudo-Boolean proof logging is not entirely practical for certain kinds of constraint reasoning. A sceptical reader might point to the large overheads observed for logging and checking, particularly for the primitive multiplication constraint, and argue that although we have shown that PB reasoning *can* be applied, the results indicate that it is at least somewhat ill-suited. While we have shown that there is certainly scope for optimising proof writing and verification on top of the conceptual justification algorithm design, there is undeniably a fundamental gap between the concise CP representation of $X \times Y = Z$ and the quadratic justifications based on bitwise-multiplier encoding that we presented in Chapter 7.

Ultimately, we can only rebut this by returning to the fundamental tradeoffs of proof logging as discussed in Chapter 1. We can either have a small set of efficiently checkable and definitely trustworthy rules in our system, or we can natively express more kinds of high-level reasoning directly, potentially at the cost of trustworthy verification. It is not clear whether the current form of PB reasoning as used in this thesis draws the line in precisely the right place. Perhaps adding primitive multiplication constraints and corresponding bounds reasoning would be an achievable extension for *VeriPB* in a future version. But it is undeniable that PB reasoning has at least achieved the best balance to date of any approach for certifying general constraint reasoning, allowing both reasonably short proofs and reasonably efficient, formally verified checking.

8.2 Future Work and Open Questions

There are many strands of work that could build on the research presented in this thesis.

8.2.1 Further Propagation Justification Procedures

To restate the obvious, we did not cover all existing constraint propagation techniques. So for an immediate strand of future work, we would like to examine the applicability of pseudo-Boolean reasoning to major global constraints not yet covered. The most prominent of these are probably the Disjunctive and Cumulative scheduling constraints [14]. These encompass various restrictions on *tasks* with start times and end times, requiring that they do not overlap or do not exceed some resource limits. PB justifications for the specific case of "timetable" reasoning were presented by Flippo et al. [70], and MDD propagation methods have been explored [45], which would likely allow for PB justifications via techniques from Chapter 5 of this work. But how to justify a Cumulative or Disjunctive propagator with a more comprehensive suite of state-of-the art filtering techniques is currently an active research area.

Another line of future work would be to consider "*cost*" variants of various global constraints. This associates numeric penalties with some aspect of the constraint semantics, for example, edges have costs in CostMDD [161] and CostCircuit [22]; and variable-value assignments have costs in CostAllDifferent [71]. PB justification methods would need to be adapted or perhaps revised entirely to support such constraints.

8.2.2 Further General Justification Procedures

It would also be interesting to explore the merits of further general methods that can produce justifications for a variety of constraint propagators, beyond decomposition to acyclic disjunctions of conjunctions, or to decision diagrams. For example, almost all global constraints can be decomposed to simpler constraints by at least *some* method, but there may be a hindrance to the level of domain filtering that can be achieved. Nevertheless, we could imagine justifying inferences from a propagator where we *don't* have a justification procedure, using a decomposition into constraints where we *do*. This could work by re-running a proof logging solver on a small sub-problem consisting of just the (decomposed) definition of the constraint, along with the negation of the justification clause. Assuming the inference really is implied by the constraint, the resulting proof should end in contradiction and hence be easily integrated as a *VeriPB* PBC step with a subproof. We would expect this to be significantly slower for cases when the decomposition is known to induce exponentially more work than bespoke propagators, but we could nevertheless perform some empirical evaluation to gauge how usable general decomposition-based justification can be in practice.

An alternative general method could be to try to leverage the *intermediate ILP* representation of the constraint as used as part of the encoding process. This requires somewhat more detail to explain properly. We will take the opportunity here to briefly outline an idea which represents currently ongoing research by the author of this thesis.

Justifications via Linear Programming

Standard results from the theory of integer/linear programming [178], in particular consequences of *Farkas' Lemma*, tell us that any linear consequence of a feasible linear programme (LP) can be written as a linear combination of its constraints. Moreover, these Farkas' coefficients can in principle be found efficiently via exact LP solving methods, for which highly performant implementations exist [121, 86].

This is not immediately a *VeriPB* justification mechanism, since the coefficients are only guaranteed to be rational, and the linear combinations (Rule 3) supported by the current proof checker are required to be integral. However, we can note that for certain constraint types, the coefficient matrix of the intermediate ILP satisfies the *totally unimodular* (TU) property. This means that the determinant of every sub-matrix is either 0 or ± 1, and importantly, is well known

to be a necessary condition for an associated LP to have integral extreme points and hence integral Farkas' coefficients for linear consequences. A TU coefficient matrix also ensures that any linear inequality implied by the ILP is also implied by the LP (relaxation).

So putting these facts together, for any constraint with a TU-linear-integer-programming (TULIP) representation, it *must* be possible to construct a justification for any inference expressible as a linear constraint implied by said TULIP, by simply taking a single linear combination of some subset of the PB encoding constraints. Note that since BinEnc is a linear operator here, the final step of the PB translation process from Section 3.2 does not change the results of linear combinations, so it will be possible to justify any bounding inferences on variables appearing directly in the intermediate ILP, and domain-consistent inferences for variables appearing as a collection of equality variables.

Although it appears to be a very strong restriction, such constraints are not obscure nor particularly uncommon. A good example is the GlobalCardinality constraint [166], for which the most obvious linearisation can be shown to be a TULIP for an arbitrary number of variables. More broadly, it should apply to constraints based on network flow algorithms [65], since it has long been known that network-flow or "circulation" problems can always be viewed as TULIPs [117].

A known TULIP representation might not actually be the one we would like to use for a PB model, since another linearisation might be more natural or more compact. But providing we can show that we can efficiently *derive* the TULIP constraints from the chosen linearisation, then we can still claim that justification construction is known to be efficient. In fact, this explains exactly why the justification procedures for AllDifferent, first described by Elffers et al. [67] (and documented in Section 3.4.4 of this thesis) are successful in the way that they are. Effectively, the AllDifferent constraint has a TULIP representation that can be derived from the clique of NotEquals constraints, and then the connection between maximum matchings and network flow allows us to recover the Farkas' coefficients.

Although this is an interesting perspective, the question that remains is how best to turn the observation into a usable general justification process. The most direct instantiation would be to execute an exact LP solver as part of the justification procedure to recover Farkas' coefficients for cases where we know this will work. But this requires far more heavyweight instrumentation for proof logging compared with any justification process examined so far. And previous research in using LP solving for propagation itself has reported a slowdown factor of 20 to 100 compared with bespoke propagators for achieving the same filtering [84]. This would be on top of the usual overheads expected for proof logging.

Perhaps the connection could be better exploited for TULIP constraints if the LP justifying logic was removed from the propagator itself and executed as part of post-processing or checking. We discuss approaches related to this in the next subsection. It might also be useful to analyse flow-based propagator implementations in more detail, to generalise the methods for AllDifferent

and consider how Farkas' coefficients could in general be discovered directly at the same time as an inference.

8.2.3 Better Justification Mechanisms

Besides a direct continuation of the work in this thesis — certifying ever more kinds of constraint propagation using PB reasoning — there are also natural extensions to the justification mechanism itself to consider. If the eventual goal is to have more solvers implement a proof logging mode as standard, there will undoubtedly have to be serious efforts towards making existing certification methods more straightforward to add to an existing code-base and less detrimental to overall runtime. In this thesis, justification procedures were implemented directly inside a solver, and executed immediately as soon as the corresponding inference was made. This was useful for debugging and for ease of implementing a first prototype. But a future implementation could extract the justification framework as an external tool, that is either called as a library via a suitable API, or as part of a proof post-processing step from a "scaffold" proof or similar intermediate format [169, 70]. The justification procedures presented in this thesis only depend on the domain information (encoded via reasons) and sometimes some additional data such as the state information from the Regular propagator, or reachability starting points for Circuit. It is therefore conceivable that they could be reimplemented as function calls to an external tool, or by reading in "hints" from a skeleton proof format.

As well as making proof logging easier to implement in different solvers, and reducing logging overhead, this would likely speed up checking, by taking advantage of "trimming" techniques on a proof scaffold/skeleton, and only justifying inferences that are actually needed for the final conclusion [169, 70]. It would also be an opportunity to implement more generic mechanisms to tackle inference types without a known justification method. For example, a proof post-processor could have fallbacks to try justifying based on a decomposition, LP relaxation, or even outsourcing to a proof logging pseudo-Boolean solver [132] or SAT solver [30] to construct justifications when all else fails.

Another open question is how formally-verified bespoke checkers for specific kinds of constraint reasoning could be integrated into an otherwise unified proof system. We have in this work been investigating PB reasoning and the *VeriPB* checker as conceptually fixed, developing justification procedures that work based only on the available rules. This has been viewed as an alternative to attempting to develop a more complex system that has bespoke rules for every constraint and inference type. But it is perhaps fair to ask whether a middle ground is possible. Could we have a system that uses primarily a small set of very general rules, but facilitates "plug-in" external inference checkers for those cases where full PB justification is deemed to be too cumbersome?

8.2.4 Verified Problem Representations

Finally, on the topic of formal verification, there is the question of how to create more trustworthy problem representations. Everything we have done here has required the basic assumption that the PB problem is effectively *the same* problem as the CSP/COP of interest. As discussed in Chapter 3, this has hinged on a very careful translation of problems using limited and straightforward encodings. But admittedly, the guarantees of correctness for the encoding process are not formal in the same way as the checker's verification of the proof log. Unlike formally verified solving, having a formally verified *translation*, at least from a flat list of variables and constraints into PB constraints seems achievable in the near-future. End-to-end verification, from a high-level problem description to a final answer, has already been demonstrated for constraint-based subgraph finding algorithms [95]. Naturally, the difficulties increase the closer to the native, highest-level problem description one wants the certification guarantees to start. Formally certifying the various unrolling and decomposition techniques performed, for example, by the *MiniZinc* compiler would be an important, but formidable undertaking.

If higher-level representations could be better integrated into formal proof logging frameworks, this would raise the intriguing possibility of exploiting them on the justification side for shorter, faster, proofs even without (technically) extending the rules beyond the current *VeriPB* rule set. This might work by viewing the high-level model translator as an oracle that can *yield* PB constraints during the proof checking process, allowing for large, potentially exponentially large, PB representations to be drawn upon without having to write them out explicitly.

Appendix A

Example Issue Reports from Solver Repositories

What follows is a short sample of issue reports which appear to indicate cases in the past where bugs or other problems in open source solvers led to them giving a wrong answer. This is meant to be illustrative only, and the author makes no claim to have analysed these problems or their underlying causes in any great detail.

Choco

- https://github.com/chocoteam/choco-solver/issues/162
- https://github.com/chocoteam/choco-solver/issues/601
- https://github.com/chocoteam/choco-solver/issues/841
- https://github.com/chocoteam/choco-solver/issues/1035
- https://github.com/chocoteam/choco-solver/issues/1037
- https://github.com/chocoteam/choco-solver/issues/1041
- https://github.com/chocoteam/choco-solver/issues/1042
- https://github.com/chocoteam/choco-solver/issues/1043
- https://github.com/chocoteam/choco-solver/issues/1044
- https://github.com/chocoteam/choco-solver/issues/1045

OR-Tools

- https://github.com/google/or-tools/issues/139
- https://github.com/google/or-tools/issues/298
- https://github.com/google/or-tools/issues/2419
- https://github.com/google/or-tools/issues/2618
- https://github.com/google/or-tools/issues/2649
- https://github.com/google/or-tools/issues/2784
- https://github.com/google/or-tools/issues/2896
- https://github.com/google/or-tools/issues/3246
- https://github.com/google/or-tools/issues/3407

- https://github.com/google/or-tools/issues/3483
- https://github.com/google/or-tools/issues/3591
- https://github.com/google/or-tools/issues/3643
- https://github.com/google/or-tools/issues/3681
- https://github.com/google/or-tools/issues/4077
- https://github.com/google/or-tools/issues/4102
- https://github.com/google/or-tools/issues/4199
- https://github.com/google/or-tools/issues/4324

Gecode

- https://github.com/Gecode/gecode/issues/101
- https://github.com/Gecode/gecode/issues/151
- https://github.com/Gecode/gecode/issues/166
- https://github.com/Gecode/gecode/issues/167
- https://github.com/Gecode/gecode/issues/168
- https://github.com/Gecode/gecode/issues/170

Chuffed

- https://github.com/chuffed/chuffed/issues/10
- https://github.com/chuffed/chuffed/issues/32
- https://github.com/chuffed/chuffed/issues/124
- https://github.com/chuffed/chuffed/issues/128
- https://github.com/chuffed/chuffed/issues/137
- https://github.com/chuffed/chuffed/issues/146
- https://github.com/chuffed/chuffed/issues/147
- https://github.com/chuffed/chuffed/issues/152

JaCoP

- https://github.com/radsz/jacop/issues/7
- https://github.com/radsz/jacop/issues/21
- https://github.com/radsz/jacop/issues/60
- https://github.com/radsz/jacop/issues/61
- https://github.com/radsz/jacop/issues/63

Bibliography

[1] VeriPB. GitLab repository, 2025. URL https://gitlab.com/MIAOresearch/software/VeriPB/-/releases/3.0.0.

[2] Berhan Oumer Adame, Bart Bogaerts, Benjamin Bogø, Simon Dold, Arthur Gontier, Wietze Koops, Ciaran McCreesh, Matthew McIlree, Jakob Nordström, Andy Oertel, Adrian Rebola-Pardo, and Mark Turnbull. VeriPB Proof Trimming. Unpublished manuscript, 2026.

[3] Magnus Ågren, Tamás Szeredi, Nicolas Beldiceanu, and Mats Carlsson. Tracing and Explaining Execution of CLP(FD) Programs. In Alexandre Tessier, editor, *Proceedings of the 12th International Workshop on Logic Programming Environments, WLPE 2002, Copenhagen, Denmark, July 31, 2002*, pages 1–16, 2002.

[4] Özgür Akgün, Ian P. Gent, Christopher Jefferson, Ian Miguel, and Peter Nightingale. Metamorphic Testing of Constraint Solvers. In John N. Hooker, editor, *Principles and Practice of Constraint Programming*, Lecture Notes in Computer Science, pages 727–736, Cham, 2018. Springer International Publishing. doi:10.1007/978-3-319-98334-9_46.

[5] Roberto Amadini, Maurizio Gabbrielli, and Jacopo Mauro. SUNNY-CP: A sequential CP portfolio solver. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, SAC '15, pages 1861–1867, New York, NY, USA, April 2015. Association for Computing Machinery. ISBN 978-1-4503-3196-8. doi:10.1145/2695664.2695741.

[6] Roberto Amadini, Maurizio Gabbrielli, and Jacopo Mauro. SUNNY-CP and the MiniZinc challenge. *Theory and Practice of Logic Programming*, 18(1):81–96, January 2018. ISSN 1471-0684, 1475-3081. doi:10.1017/S1471068417000205.

[7] Markus Anders, Bart Bogaerts, Benjamin Bogø, Arthur Gontier, Wietze Koops, Ciaran McCreesh, Magnus O Myreen, Jakob Nordstrom, Adrian Rebola-Pardo, Andy Oertel, and Yong Kiam Tan. Documentation of VeriPB and CakePB for the SAT Competition 2023, 2025. URL https://satcompetition.github.io/2025/downloads/checkers/veripb.pdf.

[8] Henrik R. Andersen, Tarik Hadžić, John N. Hooker, and Peter Tiedemann. A Constraint Store Based on Multivalued Decision Diagrams. In Christian Bessière, editor, *Principles and Practice of Constraint Programming – CP 2007*, pages 118–132, Berlin, Heidelberg, 2007. Springer. ISBN 978-3-540-74970-7. doi:10.1007/978-3-540-74970-7_11.

[9] Krzysztof R. Apt and Peter Zoeteweij. A Comparative Study of Arithmetic Constraints on Integer Intervals. In *Recent Advances in Constraints*, volume 3010, pages 1–24. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004. doi:10.1007/978-3-540-24662-6_1.

[10] Fahiem Bacchus. GAC Via Unit Propagation. In Christian Bessière, editor, *Principles and Practice of Constraint Programming – CP 2007*, Lecture Notes in Computer Science, pages 133–147, Berlin, Heidelberg, 2007. Springer. ISBN 978-3-540-74970-7. doi:10.1007/978-3-540-74970-7_12.

[11] Fahiem Bacchus and Toby Walsh. Propagating logical combinations of constraints. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence*, IJCAI'05, pages 35–40, San Francisco, CA, USA, July 2005. Morgan Kaufmann Publishers Inc.

[12] Fahiem Bacchus, Matti Järvisalo, and Ruben Martins. Maximum Satisfiabiliy. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability - Second Edition*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, pages 929–991. IOS Press, 2021. doi:10.3233/FAIA201008.

[13] Seulkee Baek, Mario Carneiro, and Marijn J. H. Heule. A Flexible Proof Format for SAT Solver-Elaborator Communication. *Logical Methods in Computer Science*, Volume 18, Issue 2, April 2022. ISSN 1860-5974. doi:10.46298/lmcs-18(2:3)2022.

[14] Philippe Baptiste and Claude Le Pape. Constraint Propagation and Decomposition Techniques for Highly Disjunctive and Highly Cumulative Project Scheduling Problems. *Constraints*, 5(1):119–139, January 2000. ISSN 1572-9354. doi:10.1023/A:1009822502231.

[15] Haniel Barbosa. Challenges in SMT Proof Production and Checking for Arithmetic Reasoning (Invited Paper). In Erika Ábrahám and Thomas Sturm, editors, *Proceedings of the 8th SC-Square Workshop Co-Located with the 48th International Symposium on Symbolic and Algebraic Computation, SC-Square@ISSAC 2023, Tromsø, Norway, July 28, 2023*, volume 3455 of *CEUR Workshop Proceedings*, pages 1–9. CEUR-WS.org, 2023.

[16] Haniel Barbosa, Andrew Reynolds, Gereon Kremer, Hanna Lachnitt, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Arjun Viswanathan, Scott Viteri, Yoni Zohar, Cesare Tinelli, and Clark W. Barrett. Flexible Proof Production in an Industrial-Strength SMT Solver. In Jasmin Blanchette, Laura Kovács, and Dirk Pattinson, editors, *Automated Reasoning - 11th International Joint Conference, IJCAR 2022, Haifa, Israel,*

August 8-10, 2022, Proceedings, volume 13385 of *Lecture Notes in Computer Science*, pages 15–35. Springer, 2022. doi:10.1007/978-3-031-10769-6_3.

[17] Clark Barrett, Leonardo De Moura, and Pascal Fontaine. Proofs in satisfiability modulo theories. *All about proofs, Proofs for all*, 55(1):23–44, 2015.

[18] Clark Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Chapter 33. Satisfiability Modulo Theories. In *Handbook of Satisfiability*, pages 1267–1329. IOS Press, 2021. doi:10.3233/FAIA201017.

[19] Charles R. Baugh and Bruce A. Wooley. A Two's Complement Parallel Array Multiplication Algorithm. *IEEE Transactions on Computers*, C-22(12):1045–1047, December 1973. ISSN 1557-9956. doi:10.1109/T-C.1973.223648.

[20] Nicolas Beldiceanu, Mats Carlsson, and Jean-Xavier Rampon. Global Constraint Catalog, 2025. https://sofdem.github.io/gccat/gccat/titlepage.html [Accessed 19/02/2025].

[21] Anton Belov, Daniel Diepold, Marijin Heule, and Matti Järvisalo. SAT Competition 2014, certified unsat, 2014. URL http://www.satcompetition.org/2014/certunsat.shtml.

[22] Pascal Benchimol, Willem-Jan van Hoeve, Jean-Charles Régin, Louis-Martin Rousseau, and Michel Rueher. Improved filtering for weighted circuit constraints. *Constraints*, 17(3): 205–233, July 2012. ISSN 1572-9354. doi:10.1007/s10601-012-9119-x.

[23] Jeremias Berg, Bart Bogaerts, Jakob Nordström, Andy Oertel, and Dieter Vandesande. Certified Core-Guided MaxSAT Solving. *Proceedings of the 29th International Conference on Automated Deduction*, May 2023.

[24] Jeremias Berg, Bart Bogaerts, Jakob Nordström, Andy Oertel, Tobias Paxian, and Dieter Vandesande. Certifying Without Loss of Generality Reasoning in Solution-Improving Maximum Satisfiability. In Paul Shaw, editor, *30th International Conference on Principles and Practice of Constraint Programming (CP 2024)*, volume 307 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 4:1–4:28, Dagstuhl, Germany, 2024. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-95977-336-2. doi:10.4230/LIPIcs.CP.2024.4.

[25] 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. ISBN 978-3-319-42847-5. doi:10.1007/978-3-319-42849-9.

[26] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development*. Texts in Theoretical Computer Science An EATCS Series. Springer, Berlin, Heidelberg, 2004. ISBN 978-3-642-05880-6 978-3-662-07964-5. doi:10.1007/978-3-662-07964-5.

[27] Christian Bessiere, Emmanuel Hebrard, Brahim Hnich, Zeynep Kiziltan, Claude-Guy Quimper, and Toby Walsh. Reformulating Global Constraints: The Slide and Regular Constraints. In Ian Miguel and Wheeler Ruml, editors, *Abstraction, Reformulation, and Approximation*, pages 80–92, Berlin, Heidelberg, 2007. Springer. ISBN 978-3-540-73580-9. doi:10.1007/978-3-540-73580-9_9.

[28] Armin Biere. TraceCheck, 2006. URL https://fmv.jku.at/tracecheck/.

[29] Armin Biere. PicoSAT Essentials. *Journal on Satisfiability, Boolean Modeling and Computation*, 4(2-4):75–97, January 2008. doi:10.3233/SAT190039.

[30] Armin Biere, Tobias Faller, Katalin Fazekas, Mathias Fleury, Nils Froleyks, and Florian Pollitt. CaDiCaL 2.0. In Arie Gurfinkel and Vijay Ganesh, editors, *Computer Aided Verification*, pages 133–152, Cham, 2024. Springer Nature Switzerland. ISBN 978-3-031-65627-9. doi:10.1007/978-3-031-65627-9_7.

[31] Bart Bogaerts, Jakob Nordström, Andy Oertel, and Çağrı Uluç Yıldırımoğlu. BreakID-kissat and BreakID-kissat-WithUNSATCertificates in SAT Competition 2022 (System Description). In *Proceedings of SAT Competition 2022 Solver and Benchmark Descriptions*, Department of Computer Science Series of Publications B, page 12. Department of Computer Science, University of Helsinki, 2022.

[32] Bart Bogaerts, Stephan Gocht, Ciaran McCreesh, and Jakob Nordström. Certified Dominance and Symmetry Breaking for Combinatorial Optimisation. *Journal of Artificial Intelligence Research*, 77:1539–1589, August 2023. ISSN 1076-9757. doi:10.1613/jair.1.14296.

[33] Endre Boros and Peter L. Hammer. Pseudo-Boolean optimization. *Discrete Applied Mathematics*, 123(1):155–225, November 2002. ISSN 0166-218X. doi:10.1016/S0166-218X(01)00341-9.

[34] Frédéric Boussemart, Christophe Lecoutre, and Cédric Piette. XCSP3: an integrated format for benchmarking combinatorial constrained problems. *CoRR*, abs/1611.03398, 2016. URL http://arxiv.org/abs/1611.03398.

[35] Frederic Boussemart, Christophe Lecoutre, Gilles Audemard, and Cédric Piette. XCSP3: An Integrated Format for Benchmarking Combinatorial Constrained Problems, August 2024.

[36] Sam Buss and Jakob Nordström. Proof Complexity and SAT Solving. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability - Second Edition*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, pages 233–350. IOS Press, 2021. doi:10.3233/FAIA200990.

[37] Sam Buss and Neil Thapen. DRAT Proofs, Propagation Redundancy, and Extended Resolution. In Mikoláš Janota and Inês Lynce, editors, *Theory and Applications of Satisfiability Testing – SAT 2019*, pages 71–89, Cham, 2019. Springer International Publishing. ISBN 978-3-030-24258-9. doi:10.1007/978-3-030-24258-9_5.

[38] Matthieu Carlier, Catherine Dubois, and Arnaud Gotlieb. A Certified Constraint Solver over Finite Domains. In Dimitra Giannakopoulou and Dominique Méry, editors, *FM 2012: Formal Methods*, Lecture Notes in Computer Science, pages 116–131, Berlin, Heidelberg, 2012. Springer. ISBN 978-3-642-32759-9. doi:10.1007/978-3-642-32759-9_12.

[39] Yves Caseau and François Laburthe. Solving small TSPs with constraints. In Lee Naish, editor, *Logic Programming, Proceedings of the Fourteenth International Conference on Logic Programming, Leuven, Belgium, July 8-11, 1997*, pages 316–330. MIT Press, 1997.

[40] B.M.W. Cheng, Jimmy H.M. Lee, and J.C.K. Wu. A Nurse Rostering System Using Constraint Programming and Redundant Modeling. *IEEE Transactions on Information Technology in Biomedicine*, 1(1):44–54, March 1997. ISSN 1558-0032. doi:10.1109/4233.594027.

[41] Kenil C. K. Cheng and Roland H. C. Yap. Maintaining Generalized Arc Consistency on Ad Hoc r-Ary Constraints. In Peter J. Stuckey, editor, *Principles and Practice of Constraint Programming*, pages 509–523, Berlin, Heidelberg, 2008. Springer. ISBN 978-3-540-85958-1. doi:10.1007/978-3-540-85958-1_34.

[42] Chiu W. Choi, Warwick Harvey, Jimmy H. M. Lee, and Peter J. Stuckey. Finite Domain Bounds Consistency Revisited. In Abdul Sattar and Byeong-ho Kang, editors, *AI 2006: Advances in Artificial Intelligence*, Lecture Notes in Computer Science, pages 49–58, Berlin, Heidelberg, 2006. Springer. ISBN 978-3-540-49788-2. doi:10.1007/11941439_9.

[43] Chiu Wo Choi, Jimmy Ho Man Lee, and Peter J. Stuckey. Propagation Redundancy in Redundant Modelling. In Francesca Rossi, editor, *Principles and Practice of Constraint Programming – CP 2003*, pages 229–243, Berlin, Heidelberg, 2003. Springer. ISBN 978-3-540-45193-8. doi:10.1007/978-3-540-45193-8_16.

[44] Geoffrey Chu, Peter J. Stuckey, Andreas Schutt, Thorsten Ehlers, Graeme Gange, and Kathryn Francis. Chuffed, a lazy clause generation solver, 2023. URL `https://github.com/chuffed/chuffed`.

[45] André A. Ciré and Willem Jan van Hoeve. MDD Propagation for Disjunctive Scheduling. In Lee McCluskey, Brian Charles Williams, José Reinaldo Silva, and Blai Bonet, editors, *Proceedings of the Twenty-Second International Conference on Automated Planning and Scheduling, ICAPS 2012, Atibaia, São Paulo, Brazil, June 25-19, 2012*. AAAI, 2012.

[46] Andre A. Cire and Willem-Jan Van Hoeve. Multivalued Decision Diagrams for Sequencing Problems. *Operations Research*, 61(6):1411–1428, December 2013. ISSN 0030-364X, 1526-5463. doi:10.1287/opre.2013.1221.

[47] Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors. *Handbook of Model Checking*. Springer International Publishing, Cham, 2018. ISBN 978-3-319-10574-1 978-3-319-10575-8. doi:10.1007/978-3-319-10575-8.

[48] Hélène Collavizza, Michel Rueher, and Pascal Van Hentenryck. CPBPV: A constraint-programming framework for bounded program verification. *Constraints*, 15(2):238–264, April 2010. ISSN 1572-9354. doi:10.1007/s10601-009-9089-9.

[49] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, pages 151–158, New York, NY, USA, May 1971. Association for Computing Machinery. ISBN 978-1-4503-7464-4. doi:10.1145/800157.805047.

[50] Stephen A. Cook and Robert A. Reckhow. The relative efficiency of propositional proof systems. *Journal of Symbolic Logic*, 44(1):36–50, March 1979. ISSN 0022-4812, 1943-5886. doi:10.2307/2273702.

[51] William Cook, Colette R. Coullard, and György. Turán. On the complexity of cutting-plane proofs. *Discrete Applied Mathematics*, 18(1):25–38, September 1987. ISSN 0166-218X. doi:10.1016/0166-218X(87)90039-4.

[52] Luís Cruz-Filipe, Marijn J. H. Heule, Warren A. Hunt, Matt Kaufmann, and Peter Schneider-Kamp. Efficient Certified RAT Verification. In Leonardo de Moura, editor, *Automated Deduction – CADE 26*, Lecture Notes in Computer Science, pages 220–236, Cham, 2017. Springer International Publishing. ISBN 978-3-319-63046-5. doi:10.1007/978-3-319-63046-5_14.

[53] Luís Cruz-Filipe, João Marques-Silva, and Peter Schneider-Kamp. Efficient Certified Resolution Proof Checking. In Axel Legay and Tiziana Margaria, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, pages 118–135, Berlin, Heidelberg, 2017. Springer. ISBN 978-3-662-54577-5. doi:10.1007/978-3-662-54577-5_7.

[54] Martin Davis and Hilary Putnam. A Computing Procedure for Quantification Theory. *Journal of the ACM*, 7(3):201–215, July 1960. ISSN 0004-5411. doi:10.1145/321033.321034.

[55] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, July 1962. ISSN 0001-0782. doi:10.1145/368273.368557.

[56] Joel D. Day, Thorsten Ehlers, Mitja Kulczynski, Florin Manea, Dirk Nowotka, and Danny Bøgsted Poulsen. On Solving Word Equations Using SAT. In Emmanuel Filiot, Raphaël Jungers, and Igor Potapov, editors, *Reachability Problems*, pages 93–106, Cham, 2019. Springer International Publishing. ISBN 978-3-030-30806-3. doi:10.1007/978-3-030-30806-3_8.

[57] Maxence Delorme, Sergio García, Jacek Gondzio, Jörg Kalcsics, David Manlove, and William Pettersson. New Algorithms for Hierarchical Optimization in Kidney Exchange Programs. *Operations Research*, January 2023. ISSN 0030-364X. doi:10.1287/opre.2022.2374.

[58] Jordan Demeulenaere, Renaud Hartert, Christophe Lecoutre, Guillaume Perez, Laurent Perron, Jean-Charles Régin, and Pierre Schaus. Compact-Table: Efficiently Filtering Table Constraints with Reversible Sparse Bit-Sets. In Michel Rueher, editor, *Principles and Practice of Constraint Programming*, pages 207–223, Cham, 2016. Springer International Publishing. ISBN 978-3-319-44953-1. doi:10.1007/978-3-319-44953-1_14.

[59] Emir Demirović, Ciaran McCreesh, Matthew J. McIlree, Jakob Nordström, Andy Oertel, and Konstantin Sidorov. Pseudo-Boolean Reasoning About States and Transitions to Certify Dynamic Programming and Decision Diagram Algorithms. In Paul Shaw, editor, *30th International Conference on Principles and Practice of Constraint Programming, CP 2024, September 2-6, 2024, Girona, Spain*, volume 307 of *LIPIcs*, pages 9:1–9:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2024. doi:10.4230/LIPICS.CP.2024.9.

[60] Jo Devriendt. Watched Propagation of 0-1-Integer Linear Constraints. In Helmut Simonis, editor, *Principles and Practice of Constraint Programming*, pages 160–176, Cham, 2020. Springer International Publishing. ISBN 978-3-030-58475-7. doi:10.1007/978-3-030-58475-7_10.

[61] Jo Devriendt, Bart Bogaerts, and Maurice Bruynooghe. Symmetric Explanation Learning: Effective Dynamic Symmetry Handling for SAT. In Serge Gaspers and Toby Walsh, editors, *Theory and Applications of Satisfiability Testing - SAT 2017 - 20th International Conference, Melbourne, VIC, Australia, August 28 - September 1, 2017, Proceedings*, volume 10491 of *Lecture Notes in Computer Science*, pages 83–100. Springer, 2017. doi:10.1007/978-3-319-66263-3_6.

[62] Luca Di Gaspero and Tommaso Urli. A CP/LNS Approach for Multi-day Homecare Scheduling Problems. In Maria J. Blesa, Christian Blum, and Stefan Voß, editors, *Hybrid Metaheuristics*, Lecture Notes in Computer Science, pages 1–15, Cham, 2014. Springer International Publishing. ISBN 978-3-319-07644-7. doi:10.1007/978-3-319-07644-7_1.

[63] Edsger W Dijkstra et al. On the cruelty of really teaching computing science. *Communications of the ACM*, 32(12):1398–1404, 1989.

[64] Simon Dold, Malte Helmert, Jakob Nordström, Gabriele Röger, and Tanja Schindler. Pseudo-Boolean proof logging for optimal classical planning. In *International Conference on Automated Planning and Scheduling*, volume 35, pages 54–63, 2025.

[65] Nicholas Downing, Thibaut Feydy, and Peter J. Stuckey. Explaining flow-based propagation. In Nicolas Beldiceanu, Narendra Jussien, and Eric Pinson, editors, *Integration of AI and OR Techniques in Contraint Programming for Combinatorial Optimzation Problems - 9th International Conference, CPAIOR 2012, Nantes, France, May 28 - June1, 2012. Proceedings*, volume 7298 of *Lecture Notes in Computer Science*, pages 146–162. Springer, 2012. doi:10.1007/978-3-642-29828-8_10. URL https://doi.org/10.1007/978-3-642-29828-8_10.

[66] Catherine Dubois. Formally verified constraints solvers: a guided tour. Talk at Conference on Intelligent Computer Mathematics 16, Emmanuel College, Cambridge, UK, April 2020. URL https://cicm-conference.org/2020/.

[67] Jan Elffers, Stephan Gocht, Ciaran McCreesh, and Jakob Nordström. Justifying All Differences Using Pseudo-Boolean Reasoning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 1486–1494, April 2020. doi:10.1609/aaai.v34i02.5507.

[68] Torsten Fahle, Stefan Schamberger, and Meinolf Sellmann. Symmetry Breaking. In Toby Walsh, editor, *Principles and Practice of Constraint Programming — CP 2001*, pages 93–107, Berlin, Heidelberg, 2001. Springer. ISBN 978-3-540-45578-3. doi:10.1007/3-540-45578-7_7.

[69] Andreas Falkner, Gerhard Friedrich, Alois Haselbock, Gottfried Schenner, and Herwig Schreiner. Twenty-five years of successful application of constraint technologies at Siemens. *AI Magazine*, 37(4):67–81, December 2016. ISSN 07384602.

[70] Maarten Flippo, Konstantin Sidorov, Imko Marijnissen, Jeff Smits, and Emir Demirović. A Multi-Stage Proof Logging Framework to Certify the Correctness of CP Solvers, 2024. Forthcoming.

[71] Filippo Focacci, Andrea Lodi, and Michela Milano. Cost-Based Domain Filtering. In Joxan Jaffar, editor, *Principles and Practice of Constraint Programming – CP'99*, pages

189–203, Berlin, Heidelberg, 1999. Springer. ISBN 978-3-540-48085-3. doi:10.1007/978-3-540-48085-3_14.

[72] Kathryn Glenn Francis and Peter J. Stuckey. Explaining circuit propagation. *Constraints*, 19(1):1–29, January 2014. ISSN 1572-9354. doi:10.1007/s10601-013-9148-0.

[73] Alan M. Frisch, Warwick Harvey, Chris Jefferson, Bernadette Martínez-Hernández, and Ian Miguel. Essence: A constraint language for specifying combinatorial problems. *Constraints*, 13(3):268–306, September 2008. ISSN 1572-9354. doi:10.1007/s10601-008-9047-y.

[74] Graeme Gange, Peter J. Stuckey, and Radoslaw Szymanek. MDD propagators with explanation. *Constraints*, 16(4):407–429, October 2011. ISSN 1572-9354. doi:10.1007/s10601-011-9111-x.

[75] Graeme Gange, Geoffrey Chu, and Peter J. Stuckey. Certifying optimality in constraint programming. Unpublished manuscript, 2017. URL https://people.eng.unimelb.edu.au/pstuckey/papers/certified-cp.pdf.

[76] Luca Di Gaspero, Andrea Rendl, and Tommaso Urli. Balancing bike sharing systems with constraint programming. *Constraints*, 21(2):318–348, April 2016. ISSN 1572-9354. doi:10.1007/s10601-015-9182-1.

[77] Gecode Team. Gecode: Generic constraint development environment, 2023. URL http://www.gecode.org.

[78] Pieter Andreas Geelen. Dual viewpoint heuristics for binary constraint satisfaction problems. In *Proceedings of the 10th European Conference on Artificial Intelligence*, ECAI '92, pages 31–35, USA, August 1992. John Wiley & Sons, Inc. ISBN 978-0-471-93608-4.

[79] Allen Van Gelder. Extracting (Easily) Checkable Proofs from a Satisfiability Solver that Employs both Preorder and Postorder Resolution. In *International Symposium on Artificial Intelligence and Mathematics, AI&M 2002, Fort Lauderdale, Florida, USA, January 2-4, 2002*, 2002.

[80] 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.

[81] Allen van Gelder. Producing and verifying extremely large propositional refutations. *Annals of Mathematics and Artificial Intelligence*, 65(4):329–372, August 2012. ISSN 1573-7470. doi:10.1007/s10472-012-9322-x.

[82] Ian P. Gent. Arc Consistency in SAT. In Frank van Harmelen, editor, *Proceedings of the 15th European Conference on Artificial Intelligence, ECAI'2002, Lyon, France, July 2002*, pages 121–125. IOS Press, 2002.

[83] Ian P. Gent, Ian Miguel, and Peter Nightingale. Generalised arc consistency for the AllDifferent constraint: An empirical survey. *Artificial Intelligence*, 172(18):1973–2000, December 2008. ISSN 0004-3702. doi:10.1016/j.artint.2008.10.006.

[84] Grigori German, Olivier Briant, Hadrien Cambazard, and Vincent Jost. Arc Consistency via Linear Programming. In J. Christopher Beck, editor, *Principles and Practice of Constraint Programming*, pages 114–128, Cham, 2017. Springer International Publishing. ISBN 978-3-319-66158-2. doi:10.1007/978-3-319-66158-2_8.

[85] Xavier Gillard, Pierre Schaus, and Yves Deville. SolverCheck: Declarative Testing of Constraints. In Thomas Schiex and Simon de Givry, editors, *Principles and Practice of Constraint Programming*, Lecture Notes in Computer Science, pages 565–582, Cham, 2019. Springer International Publishing. doi:10.1007/978-3-030-30048-7_33.

[86] Ambros M. Gleixner, Daniel E. Steffy, and Kati Wolter. Iterative Refinement for Linear Programming. *INFORMS Journal on Computing*, 28(3):449–464, July 2016. ISSN 1091-9856. doi:10.1287/ijoc.2016.0692.

[87] Stephan Gocht. Personal communication, 2022. Email to Ciaran McCreesh, January 2022.

[88] Stephan Gocht. *Certifying Correctness for Combinatorial Algorithms: By Using Pseudo-Boolean Reasoning*. PhD thesis, Lund University, Sweden, 2022.

[89] 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.

[90] Stephan Gocht, Jakob Nordström, and Amir Yehudayoff. On Division Versus Saturation in Pseudo-Boolean Solving. In *Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19*, pages 1711–1718, Macao, China, August 2019. International Joint Conferences on Artificial Intelligence Organization. ISBN 978-0-9992411-4-1. doi:10.24963/ijcai.2019/237.

[91] 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, September 2020. doi:10.1007/978-3-030-58475-7_20.

[92] Stephan Gocht, Ciaran McCreesh, and Jakob Nordström. Subgraph Isomorphism Meets Cutting Planes: Solving With Certified Solutions. In *Electronic Proceedings of IJCAI 2020*, volume 2, pages 1134–1140, July 2020. doi:10.24963/ijcai.2020/158.

[93] Stephan Gocht, Ruben Martins, Jakob Nordström, and Andy Oertel. Certified CNF Translations for Pseudo-Boolean Solving. In Kuldeep S. Meel and Ofer Strichman, editors, *25th International Conference on Theory and Applications of Satisfiability Testing (SAT 2022)*, volume 236 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 16:1–16:25, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-95977-242-6. doi:10.4230/LIPIcs.SAT.2022.16.

[94] Stephan Gocht, Ciaran McCreesh, and Jakob Nordström. An Auditable Constraint Programming Solver. In Christine Solnon, editor, *Proceeding of the 28th International Conference on Principles and Practice of Constraint Programming*, volume 235 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 25:1–25:18, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-95977-240-2. doi:10.4230/LIPIcs.CP.2022.25.

[95] 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. URL https://doi.org/10.1609/aaai.v38i8.28642.

[96] Stephan Gocht, Ciaran McCreesh, Magnus O. Myreen, Jakob Nordström, Andy Oertel, and Yong Kiam Tan. Proof Logging for Projected Enumeration (and Counting?) Problems in VeriPB. Unpublished, 2026.

[97] Evguenii Goldberg and Yakov Novikov. Verification of proofs of unsatisfiability for CNF formulas. In *Automation and Test in Europe Conference and Exhibition 2003 Design*, pages 886–891, March 2003. doi:10.1109/DATE.2003.1253718.

[98] Tseitin Grigori. On the complexity of derivation in propositional calculus. *Studies in Constructive Mathematics and Mathematical Logic*, pages 115–125, 1968.

[99] Tias Guns. Increasing modeling language convenience with a universal n-dimensional array, CPpy as python-embedded example. In *Proceedings of the 18th workshop on Constraint Modelling and Reformulation at CP (Modref 2019)*, volume 19, 2019.

[100] Şeyda Gür, Tamer Eren, and Hacı Mehmet Alakaş. Surgical Operation Scheduling with Goal Programming and Constraint Programming: A Case Study. *Mathematics*, 7(3):251, March 2019. ISSN 2227-7390. doi:10.3390/math7030251.

[101] Armin Haken. The intractability of resolution. *Theoretical Computer Science*, 39:297–308, January 1985. ISSN 0304-3975. doi:10.1016/0304-3975(85)90144-6.

[102] Philip Hall. On Representatives of Subsets. *Journal of the London Mathematical Society*, s1-10(1):26–30, 1935. ISSN 1469-7750. doi:10.1112/jlms/s1-10.37.26.

[103] Marijin Heule. Trustworthy automated reasoning. Talk at Satisfiability: Theory, Practice, and Beyond, Simons Institute for the Theory of Computing, University of California Berkeley, April 2023. URL https://www.youtube.com/watch?v=dFTPC8cQ3Z4.

[104] Marijn J. H. Heule. Schur number five. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence and Thirtieth Innovative Applications of Artificial Intelligence Conference and Eighth AAAI Symposium on Educational Advances in Artificial Intelligence*, AAAI'18/IAAI'18/EAAI'18, pages 6598–6606, New Orleans, Louisiana, USA, February 2018. AAAI Press. ISBN 978-1-57735-800-8.

[105] Marijn J. H. Heule. Chinese Remainder Encoding for Hamiltonian Cycles. In Chu-Min Li and Felip Manyà, editors, *Theory and Applications of Satisfiability Testing - SAT 2021 - 24th International Conference, Barcelona, Spain, July 5-9, 2021, Proceedings*, volume 12831 of *Lecture Notes in Computer Science*, pages 216–224. Springer, 2021. doi:10.1007/978-3-030-80223-3_15.

[106] Marijn J. H. Heule, Warren A. Hunt, and Nathan Wetzler. Verifying Refutations with Extended Resolution. In Maria Paola Bonacina, editor, *Automated Deduction – CADE-24*, Lecture Notes in Computer Science, pages 345–359, Berlin, Heidelberg, 2013. Springer. ISBN 978-3-642-38574-2. doi:10.1007/978-3-642-38574-2_24.

[107] Marijn J. H. Heule, Warren A. Hunt Jr., and Nathan Wetzler. Bridging the gap between easy generation and efficient verification of unsatisfiability proofs. *Software Testing, Verification and Reliability*, 24(8):593–607, 2014. ISSN 1099-1689. doi:10.1002/stvr.1549.

[108] Marijn J. H. Heule, Warren A. Hunt, and Nathan Wetzler. Expressing Symmetry Breaking in DRAT Proofs. In Amy P. Felty and Aart Middeldorp, editors, *Automated Deduction - CADE-25*, Lecture Notes in Computer Science, pages 591–606, Cham, 2015. Springer International Publishing. ISBN 978-3-319-21401-6. doi:10.1007/978-3-319-21401-6_40.

[109] Marijn J. H. Heule, Oliver Kullmann, and Victor W. Marek. Solving and Verifying the Boolean Pythagorean Triples Problem via Cube-and-Conquer. In Nadia Creignou and Daniel Le Berre, editors, *Theory and Applications of Satisfiability Testing – SAT 2016*, Lecture Notes in Computer Science, pages 228–245, Cham, 2016. Springer International Publishing. ISBN 978-3-319-40970-2. doi:10.1007/978-3-319-40970-2_15.

[110] Marijn J. H. Heule, Benjamin Kiesl, and Armin Biere. Short Proofs Without New Variables. In Leonardo de Moura, editor, *Automated Deduction – CADE 26*, Lecture Notes in Computer Science, pages 130–147, Cham, 2017. Springer International Publishing. ISBN 978-3-319-63046-5. doi:10.1007/978-3-319-63046-5_9.

[111] Marijn J. H. Heule, Benjamin Kiesl, and Armin Biere. Strong Extension-Free Proof Systems. *Journal of Automated Reasoning*, 64(3):533–554, March 2020. ISSN 1573-0670. doi:10.1007/s10817-019-09516-0.

[112] Marijn J.H. Heule, Warren A. Hunt, and Nathan Wetzler. Trimming while checking clausal proofs. In *2013 Formal Methods in Computer-Aided Design*, pages 181–188, October 2013. doi:10.1109/FMCAD.2013.6679408.

[113] S Hitarth, Cayden Codel, Hanna Lachnitt, and Bruno Dutertre. Extending DRAT to SMT. In *2024 Formal Methods in Computer-Aided Design (FMCAD)*, pages 01–11, October 2024. doi:10.34727/2024/isbn.978-3-85448-065-5_8.

[114] Pierre-Emmanuel Hladik, Hadrien Cambazard, Anne-Marie Déplanche, and Narendra Jussien. Solving a real-time allocation problem with constraint programming. *Journal of Systems and Software*, 81(1):132–149, January 2008. ISSN 0164-1212. doi:10.1016/j.jss.2007.02.032.

[115] Alexander Hoen, Andy Oertel, Ambros M. Gleixner, and Jakob Nordström. Certifying MIP-Based Presolve Reductions for 0-1 Integer Linear Programs. In Bistra Dilkina, editor, *Integration of Constraint Programming, Artificial Intelligence, and Operations Research - 21st International Conference, CPAIOR 2024, Uppsala, Sweden, May 28-31, 2024, Proceedings, Part I*, volume 14742 of *Lecture Notes in Computer Science*, pages 310–328. Springer, 2024. doi:10.1007/978-3-031-60597-0_20. URL https://doi.org/10.1007/978-3-031-60597-0_20.

[116] Jochen Hoenicke and Tanja Schindler. A Simple Proof Format for SMT. In David Déharbe and Antti E. J. Hyvärinen, editors, *Proceedings of the 20th Internal Workshop on Satisfiability Modulo Theories Co-Located with the 11th International Joint Conference on Automated Reasoning (IJCAR 2022) Part of the 8th Federated Logic Conference (FLoC 2022), Haifa, Israel, August 11-12, 2022*, volume 3185 of *CEUR Workshop Proceedings*, pages 54–70. CEUR-WS.org, 2022.

[117] Alan J. Hoffman. Total unimodularity and combinatorial theorems. *Linear Algebra and its Applications*, 13(1):103–108, 1976. ISSN 0024-3795. doi:10.1016/0024-3795(76)90047-1.

[118] John N. Hooker. Generalized resolution and cutting planes. *Annals of Operations Research*, 12(1):217–239, December 1988. ISSN 1572-9338. doi:10.1007/BF02186368.

[119] John N. Hooker. Generalized resolution for 0–1 linear inequalities. *Annals of Mathematics and Artificial Intelligence*, 6(1):271–286, March 1992. ISSN 1573-7470. doi:10.1007/BF01531033.

[120] John N. Hooker. Decision Diagrams and Dynamic Programming. In Carla Gomes and Meinolf Sellmann, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pages 94–110, Berlin, Heidelberg, 2013. Springer. ISBN 978-3-642-38171-3. doi:10.1007/978-3-642-38171-3_7.

[121] Qi Huangfu and Julian. A. J. Hall. Parallelizing the dual revised simplex method. *Mathematical Programming Computation*, 10(1):119–142, March 2018. ISSN 1867-2957. doi:10.1007/s12532-017-0130-5.

[122] Sophie Huczynska, Christopher Jefferson, and Silvia Nepšinská. Strong external difference families in abelian and non-abelian groups. *Cryptography and Communications*, 13(2): 331–341, March 2021. ISSN 1936-2455. doi:10.1007/s12095-021-00473-3.

[123] Hannes Ihalainen, Andy Oertel, Yong Kiam Tan, Jeremias Berg, Matti Järvisalo, Magnus O. Myreen, and Jakob Nordström. Certified MaxSAT Preprocessing. In Christoph Benzmüller, Marijn J.H. Heule, and Renate A. Schmidt, editors, *Automated Reasoning*, pages 396–418, Cham, 2024. Springer Nature Switzerland. ISBN 978-3-031-63498-7. doi:10.1007/978-3-031-63498-7_24.

[124] Matti Järvisalo, Daniel Le Berre, and Olivier Roussel. SAT 2011 Competition: Main competition, 2011. URL https://web.archive.org/web/20220819175637/www.cril.univ-artois.fr/SAT11/results/results.php?idev=45.

[125] Matti Järvisalo, Daniel Le Berre, Olivier Roussel, and Laurent Simon. The International SAT Solver Competitions. *AI Magazine*, 33(1):89–92, March 2012. ISSN 2371-9621. doi:10.1609/aimag.v33i1.2395.

[126] Matti Järvisalo, Marijn J. H. Heule, and Armin Biere. Inprocessing Rules. In Bernhard Gramlich, Dale Miller, and Uli Sattler, editors, *Automated Reasoning*, Lecture Notes in Computer Science, pages 355–370, Berlin, Heidelberg, 2012. Springer. ISBN 978-3-642-31365-3. doi:10.1007/978-3-642-31365-3_28.

[127] Christopher Jefferson. *Representations in Constraint Programming*. PhD thesis, University of York, UK, 2007.

[128] 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*, Lecture Notes in Computer Science, pages 54–60, Berlin, Heidelberg, 2006. Springer. ISBN 978-3-540-37207-3. doi:10.1007/11814948_8.

[129] Richard M. Karp. Reducibility among Combinatorial Problems. In Raymond E. Miller, James W. Thatcher, and Jean D. Bohlinger, editors, *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*, The IBM Research Symposia Series, pages 85–103. Springer US, Boston, MA, 1972. ISBN 978-1-4684-2001-2. doi:10.1007/978-1-4684-2001-2_9.

[130] Philip Kilby and Paul Shaw. Vehicle Routing. In Francesca Rossi, Peter van Beek, and Toby Walsh, editors, *Foundations of Artificial Intelligence*, volume 2 of *Handbook of Constraint Programming*, pages 801–836. Elsevier, January 2006. doi:10.1016/S1574-6526(06)80027-1.

[131] Wietze Koops. Contradicting Inequalities by Unit Propagation, 2024. Personal Communication.

[132] Wietze Koops, Daniel Le Berre, Magnus O. Myreen, Jakob Nordström, Andy Oertel, Yong Kiam Tan, and Marc Vinyals. Practically Feasible Proof Logging for Pseudo-Boolean Optimization. In Maria Garcia de la Banda, editor, *31st International Conference on Principles and Practice of Constraint Programming (CP 2025)*, volume 340 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 21:1–21:27, Dagstuhl, Germany, 2025. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-95977-380-5. doi:10.4230/LIPIcs.CP.2025.21.

[133] Jan Krajíček. *Proof Complexity*. Encyclopedia of Mathematics and Its Applications. Cambridge University Press, Cambridge, 2019. ISBN 978-1-108-41684-9. doi:10.1017/9781108242066.

[134] Krzysztof Kuchcinski and Radoslaw Szymanek. 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.

[135] Ramana Kumar, Eric Mullen, Zachary Tatlock, and Magnus O. Myreen. Software Verification with ITPs Should Use Binary Code Extraction to Reduce the TCB. In Jeremy

Avigad and Assia Mahboubi, editors, *Interactive Theorem Proving*, pages 362–369, Cham, 2018. Springer International Publishing. ISBN 978-3-319-94821-8. doi:10.1007/978-3-319-94821-8_21.

[136] Edward Lam and Pascal Van Hentenryck. A branch-and-price-and-check model for the vehicle routing problem with location congestion. *Constraints*, 21(3):394–412, July 2016. ISSN 1572-9354. doi:10.1007/s10601-016-9241-2.

[137] Edward Lam, Pascal Van Hentenryck, and Philip Kilby. 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*, pages 654–670. Springer, 2015. doi:10.1007/978-3-319-23219-5_45.

[138] Evelyn Lamb. Two-hundred-terabyte maths proof is largest ever. *Nature*, 534(7605): 17–18, June 2016. ISSN 1476-4687. doi:10.1038/nature.2016.19990.

[139] Massimo Lauria. Lecture 6 – Cutting Planes Proofs. Introduction to Proof Complexity, Tokyo Institute of Technologies, 2015. `https://www.massimolauria.net/courses/2015.ProofComplexity/lecture6.pdf` [Accessed 15/01/2025].

[140] Christophe Lecoutre. STR2: Optimized simple tabular reduction for table constraints. *Constraints*, 16(4):341–371, October 2011. ISSN 1572-9354. doi:10.1007/s10601-011-9107-6.

[141] Christophe Lecoutre. *Constraint Networks: Targeting Simplicity for Techniques and Algorithms*. John Wiley & Sons, 2013.

[142] Kevin Lotz, Amit Goel, Bruno Dutertre, Benjamin Kiesl-Reiter, Soonho Kong, Rupak Majumdar, and Dirk Nowotka. Solving String Constraints Using SAT. In Constantin Enea and Akash Lal, editors, *Computer Aided Verification*, pages 187–208, Cham, 2023. Springer Nature Switzerland. ISBN 978-3-031-37703-7. doi:10.1007/978-3-031-37703-7_9.

[143] Alan K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1): 99–118, February 1977. ISSN 0004-3702. doi:10.1016/0004-3702(77)90007-8.

[144] Jean-Baptiste Mairy, Yves Deville, and Christophe Lecoutre. The Smart Table Constraint. In Laurent Michel, editor, *12th International Conference on Integration of AI and OR Techniques in Constraint Programming (CPAIOR 2015)*, Lecture Notes in Computer Science, pages 271–287, Cham, 2015. Springer International Publishing. ISBN 978-3-319-18008-3. doi:10.1007/978-3-319-18008-3_19.

[145] João Marques-Silva. Practical applications of boolean satisfiability. In *2008 9th International Workshop on Discrete Event Systems*, pages 74–80. IEEE, 2008.

[146] João. Marques Silva and Karem A. Sakallah. GRASP-A new search algorithm for satisfiability. In *Proceedings of International Conference on Computer Aided Design*, pages 220–227, November 1996. doi:10.1109/ICCAD.1996.569607.

[147] Raul Mazo, Paul Grünbacher, Wolfgang Heider, Rick Rabiser, Camille Salinesi, and Daniel Diaz. Using constraint programming to verify DOPLER variability models. In *Proceedings of the 5th International Workshop on Variability Modeling of Software-Intensive Systems*, VaMoS '11, pages 97–103, New York, NY, USA, January 2011. Association for Computing Machinery. ISBN 978-1-4503-0570-9. doi:10.1145/1944892.1944904.

[148] Ross McBride. A Maximum Clique Algorithm with Verifiable Output. Master's thesis, University of Glasgow, 2020.

[149] Ross M. McConnell, Kurt Mehlhorn, Stefan Näher, and Pascal Schweitzer. Certifying algorithms. *Computer Science Review*, 5(2):119–161, May 2011. ISSN 1574-0137. doi:10.1016/j.cosrev.2010.09.009.

[150] Ciaran McCreesh and Matthew J. McIlree. The Glasgow Constraint Solver, 2025. https://github.com/ciaranm/glasgow-constraint-solver [Accessed: 12/02/2025].

[151] 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)*, volume 280 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 26:1–26:17, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-95977-300-3. doi:10.4230/LIPIcs.CP.2023.26.

[152] Matthew J. McIlree and Ciaran McCreesh. Certifying Bounds Propagation for Integer Multiplication Constraints. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pages 11309–11317, 2025.

[153] Matthew J. McIlree, Ciaran McCreesh, and Jakob Nordström. Proof Logging for the Circuit Constraint. In Bistra Dilkina, editor, *Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 38–55, Cham, 2024. Springer Nature Switzerland. ISBN 978-3-031-60599-4. doi:10.1007/978-3-031-60599-4_3.

[154] C. E. Miller, Albert W. Tucker, and R A. Zemlin. Integer Programming Formulation of Traveling Salesman Problems. *J. ACM*, 7(4):326–329, October 1960. ISSN 0004-5411. doi:10.1145/321043.321046.

[155] Mia Müßig and Jan Johannsen. Improving Watched Pseudo-Boolean Propagation with Significant Literals. In Jochen Hoenicke, Mikoláš Janota, Aina Niemetz, and Sophie Tourret, editors, *Joint Proceedings of the 23rd International Workshop on Satisfiability*

Modulo Theories and the 16th Pragmatics of SAT International Workshop, volume 4008 of *CEUR Workshop Proceedings*, pages 177–189, Glasgow, UK, August 2025. CEUR.

[156] Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. MiniZinc: Towards a Standard CP Modelling Language. In Christian Bessière, editor, *Principles and Practice of Constraint Programming – CP 2007*, pages 529–543, Berlin, Heidelberg, 2007. Springer. ISBN 978-3-540-74970-7. doi:10.1007/978-3-540-74970-7_38.

[157] Robert Nieuwenhuis, Albert Oliveras, Enric Rodríguez-Carbonell, and Rui Zhao. Speeding up Pseudo-Boolean Propagation. In Supratik Chakraborty and Jie-Hong Roland Jiang, editors, *27th International Conference on Theory and Applications of Satisfiability Testing (SAT 2024)*, volume 305 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 22:1–22:18, Dagstuhl, Germany, 2024. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-95977-334-8. doi:10.4230/LIPIcs.SAT.2024.22.

[158] Peter Nightingale, Özgür Akgün, Ian P. Gent, Christopher Jefferson, Ian Miguel, and Patrick Spracklen. Automatically improving constraint models in Savile Row. *Artificial Intelligence*, 251:35–61, October 2017. ISSN 0004-3702. doi:10.1016/j.artint.2017.07.001.

[159] Tobias Nipkow, Markus Wenzel, Lawrence C. Paulson, Gerhard Goos, Juris Hartmanis, and Jan Van Leeuwen, editors. *Isabelle/HOL*, volume 2283 of *Lecture Notes in Computer Science*. Springer, Berlin, Heidelberg, 2002. ISBN 978-3-540-43376-7 978-3-540-45949-1. doi:10.1007/3-540-45949-9.

[160] Olga Ohrimenko, Peter J. Stuckey, and Michael Codish. Propagation via lazy clause generation. *Constraints*, 14(3):357–391, September 2009. ISSN 1572-9354. doi:10.1007/s10601-008-9064-x.

[161] Guillaume Perez and Jean-Charles Régin. Soft and cost MDD propagators. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*, AAAI'17, pages 3922–3928, San Francisco, California, USA, February 2017. AAAI Press.

[162] Laurent Perron and Frédéric Didier. Cp-sat, 2025. URL `https://developers.google.com/optimization/cp/cp_solver/`.

[163] Gilles Pesant. A Regular Language Membership Constraint for Finite Sequences of Variables. In Mark Wallace, editor, *10th International Conference on Principles and Practice of Constraint Programming (CP 2004)*, Lecture Notes in Computer Science, pages 482–495, Berlin, Heidelberg, 2004. Springer. ISBN 978-3-540-30201-8. doi:10.1007/978-3-540-30201-8_36.

[164] Gilles Pesant, Michel Gendreau, Jean-Yves Potvin, and Jean-Marc Rousseau. An Exact Constraint Logic Programming Algorithm for the Traveling Salesman Problem with Time Windows. *Transportation Science*, 32(1):12–29, February 1998. ISSN 0041-1655, 1526-5447. doi:10.1287/trsc.32.1.12.

[165] Pavel Pudlák. Lower Bounds for Resolution and Cutting Plane Proofs and Monotone Computations. *The Journal of Symbolic Logic*, 62(3):981–998, 1997. ISSN 0022-4812. doi:10.2307/2275583.

[166] Claude-Guy Quimper, Alejandro López-Ortiz, Peter van Beek, and Alexander Golynski. Improved Algorithms for the Global Cardinality Constraint. In Mark Wallace, editor, *Principles and Practice of Constraint Programming – CP 2004*, pages 542–556, Berlin, Heidelberg, 2004. Springer. ISBN 978-3-540-30201-8. doi:10.1007/978-3-540-30201-8_40.

[167] Layali Rashid, Karthik Pattabiraman, and Sathish Gopalakrishnan. Characterizing the Impact of Intermittent Hardware Faults on Programs. *IEEE Transactions on Reliability*, 64(1):297–310, March 2015. ISSN 1558-1721. doi:10.1109/TR.2014.2363152.

[168] Adrián Rebola-Pardo. Unsatisfiability proofs in SAT solving with parity reasoning. Master's thesis, TU Dresden, 2015.

[169] Joseph E. Reeves, Benjamin Kiesl-Reiter, and Marijn J. H. Heule. Propositional Proof Skeletons. In Sriram Sankaranarayanan and Natasha Sharygina, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 329–347, Cham, 2023. Springer Nature Switzerland. ISBN 978-3-031-30823-9. doi:10.1007/978-3-031-30823-9_17.

[170] Jean-Charles Régin. A filtering algorithm for constraints of difference in CSPs. In *AAAI*, volume 94, pages 362–367, 1994.

[171] Jean-Charles Régin. A Filtering Algorithm for Constraints of Difference in CSPs. In Barbara Hayes-Roth and Richard E. Korf, editors, *Proceedings of the 12th National Conference on Artificial Intelligence, Seattle, WA, USA, July 31 - August 4, 1994, Volume 1*, pages 362–367. AAAI Press / The MIT Press, 1994.

[172] Alexandre Riazanov and Andrei Voronkov. Vampire. In Harald Ganzinger, editor, *Automated Deduction — CADE-16*, pages 292–296, Berlin, Heidelberg, 1999. Springer. ISBN 978-3-540-48660-2. doi:10.1007/3-540-48660-7_26.

[173] John Alan Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *J. ACM*, 12(1):23–41, 1965. doi:10.1145/321250.321253.

[174] Francesca Rossi, Charles J. Petrie, and Vasant Dhar. On the equivalence of constraint satisfaction problems. In *Proceedings of the 9th European Conference on Artificial Intelligence*, ECAI'90, pages 550–556, USA, 1990. Pitman Publishing, Inc.

[175] Francesca Rossi, Peter van Beek, and Toby Walsh, editors. *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*. Elsevier, 2006. ISBN 978-0-444-52726-4.

[176] Olivier Roussel. General OPB Format, 2024. https://www.cril.univ-artois.fr/PB24/OPBgeneral.pdf [Accessed 5/02/2025].

[177] Alexander Schrijver. On Cutting Planes. In Peter L. Hammer, editor, *Annals of Discrete Mathematics*, volume 9 of *Combinatorics 79*, pages 291–296. Elsevier, January 1980. doi:10.1016/S0167-5060(08)70085-2.

[178] Alexander Schrijver. *Theory of Linear and Integer programming*. Wiley-Interscience, 1986.

[179] Christian Schulte and Peter J. Stuckey. When do bounds and domain propagation lead to the same search space. In *Proceedings of the 3rd ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, PPDP '01, pages 115–126, New York, NY, USA, September 2001. Association for Computing Machinery. ISBN 978-1-58113-388-2. doi:10.1145/773184.773197.

[180] Christian Schulte and Guido Tack. Weakly Monotonic Propagators. In Ian P. Gent, editor, *Principles and Practice of Constraint Programming - CP 2009*, Lecture Notes in Computer Science, pages 723–730, Berlin, Heidelberg, 2009. Springer. ISBN 978-3-642-04244-7. doi:10.1007/978-3-642-04244-7_56.

[181] Mate Soos, Karsten Nohl, and Claude Castelluccia. Extending SAT Solvers to Cryptographic Problems. In Oliver Kullmann, editor, *Theory and Applications of Satisfiability Testing - SAT 2009*, pages 244–257, Berlin, Heidelberg, 2009. Springer. ISBN 978-3-642-02777-2. doi:10.1007/978-3-642-02777-2_24.

[182] Gregory F. Sullivan and Gerald M. Masson. Using certification trails to achieve software fault tolerance. In *[1990] Digest of Papers. Fault-Tolerant Computing: 20th International Symposium*, pages 423–431, June 1990. doi:10.1109/FTCS.1990.89397.

[183] Robert Tarjan. Depth-First Search and Linear Graph Algorithms. *SIAM Journal on Computing*, 1(2):146–160, June 1972. ISSN 0097-5397. doi:10.1137/0201010.

[184] Tip ten Brink. Proof Step Checking in a Constraint Programming Unsatisfiability Proof Checker. Master's thesis, TU Delft, 2025.

[185] Maurice H. ter Beek, Rod Chapman, Rance Cleaveland, Hubert Garavel, Rong Gu, Ivo ter Horst, Jeroen J. A. Keiren, Thierry Lecomte, Michael Leuschel, Kristin Yvonne Rozier, Augusto Sampaio, Cristina Seceleanu, Martyn Thomas, Tim A. C. Willemse, and Lijun Zhang. Formal Methods in Industry. *Form. Asp. Comput.*, 37(1):7:1–7:38, December 2024. ISSN 0934-5043. doi:10.1145/3689374.

[186] Christian Timpe. Solving planning and scheduling problems with combined integer and constraint programming. *OR Spectrum*, 24(4):431–448, November 2002. ISSN 1436-6304. doi:10.1007/s00291-002-0107-1.

[187] Michael A. Trick. A Dynamic Programming Approach for Consistency and Propagation for Knapsack Constraints. *Annals of Operations Research*, 118(1):73–84, February 2003. ISSN 1572-9338. doi:10.1023/A:1021801522545.

[188] Grigori S. Tseitin. On the Complexity of Derivation in Propositional Calculus. In Jörg H. Siekmann and Graham Wrightson, editors, *Automation of Reasoning*, pages 466–483. Springer Berlin Heidelberg, Berlin, Heidelberg, 1983. ISBN 978-3-642-81957-5 978-3-642-81955-1. doi:10.1007/978-3-642-81955-1_28.

[189] Julian R. Ullmann. Partition search for non-binary constraint satisfaction. *Information Sciences*, 177(18):3639–3678, 2007. doi:10.1016/j.ins.2007.03.030.

[190] Alasdair Urquhart. The symmetry rule in propositional logic. *Discrete Applied Mathematics*, 96–97:177–193, October 1999. ISSN 0166-218X. doi:10.1016/S0166-218X(99)00039-6.

[191] Dieter Vandesande, Wolf De Wulf, and Bart Bogaerts. QMaxSATpb: A Certified MaxSAT Solver. In Georg Gottlob, Daniela Inclezan, and Marco Maratea, editors, *Logic Programming and Nonmonotonic Reasoning*, Lecture Notes in Computer Science, pages 429–442, Cham, 2022. Springer International Publishing. ISBN 978-3-031-15707-3. doi:10.1007/978-3-031-15707-3_33.

[192] Wout Vanroose, Ignace Bleukx, Jo Devriendt, Dimos Tsouros, Hélène Verhaeghe, and Tias Guns. Mutational Fuzz Testing for Constraint Modeling Systems. In Paul Shaw, editor, *30th International Conference on Principles and Practice of Constraint Programming (CP 2024)*, volume 307 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 29:1–29:25, Dagstuhl, Germany, 2024. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-95977-336-2. doi:10.4230/LIPIcs.CP.2024.29.

[193] Michael Veksler and Ofer Strichman. A Proof-Producing CSP Solver. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 24, pages 204–209, July 2010. doi:10.1609/aaai.v24i1.7543.

[194] Hélène Verhaeghe. *The extensional constraint.* PhD thesis, Catholic University of Louvain, Louvain-la-Neuve, Belgium, 2021.

[195] Hélène Verhaeghe, Christophe Lecoutre, Yves Deville, and Pierre Schaus. Extending Compact-Table to Basic Smart Tables. In J. Christopher Beck, editor, *Principles and Practice of Constraint Programming*, pages 297–307, Cham, 2017. Springer International Publishing. ISBN 978-3-319-66158-2. doi:10.1007/978-3-319-66158-2_19.

[196] Nathan Wetzler, Marijn J. H. Heule, and Warren A. Hunt. 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*, Lecture Notes in Computer Science, pages 422–429, Cham, 2014. Springer International Publishing. ISBN 978-3-319-09284-3. doi:10.1007/978-3-319-09284-3_31.

[197] Alexander Wold, Andreas Agne, and Jim Torresen. Module Placement Using Constraint Programming in Run-Time Reconfigurable Systems. In *2014 IEEE International Parallel & Distributed Processing Symposium Workshops*, pages 287–292, May 2014. doi:10.1109/IPDPSW.2014.39.

[198] Zhang Yuanlin and Roland H. C. Yap. Arc Consistency on n-ary Monotonic and Linear Constraints. In Rina Dechter, editor, *Principles and Practice of Constraint Programming – CP 2000*, pages 470–483, Berlin, Heidelberg, 2000. Springer. ISBN 978-3-540-45349-9. doi:10.1007/3-540-45349-0_34.

[199] Tallys Yunes, Ionuţ D. Aron, and John N. Hooker. An Integrated Solver for Optimization Problems. *Oper. Res.*, 58(2):342–356, March 2010. ISSN 0030-364X. doi:10.1287/opre.1090.0733.

[200] Lintao Zhang and Sharad Malik. Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications. In *Automation and Test in Europe Conference and Exhibition 2003 Design*, pages 880–885, March 2003. doi:10.1109/DATE.2003.1253717.

[201] Neng-Fa Zhou. In Pursuit of an Efficient SAT Encoding for the Hamiltonian Cycle Problem. In Helmut Simonis, editor, *Principles and Practice of Constraint Programming*, volume 12333, pages 585–602, Cham, 2020. Springer International Publishing. ISBN 978-3-030-58474-0 978-3-030-58475-7. doi:10.1007/978-3-030-58475-7_34.

[202] James F. Ziegler and William A. Lanford. Effect of Cosmic Rays on Computer Memories. *Science*, 206(4420):776–788, November 1979. doi:10.1126/science.206.4420.776.