Constraint Integer Programming

vorgelegt von
Dipl.-Math. Dipl.-Inf. Tobias Achterberg
Berlin

der Fakultät II – Mathematik und Naturwissenschaften
der Technischen Universität Berlin
zur Erlangung des akademischen Grades

Doktor der Naturwissenschaften
– Dr. rer. nat. –

genehmigte Dissertation

Berichter: Prof. Dr. Dr. h.c. Martin Grötschel
Technische Universität Berlin

Prof. Dr. Robert E. Bixby
Rice University, Houston, USA

Tag der wissenschaftlichen Aussprache: 12. Juli 2007

Berlin 2007
D 83

Für Julia

Zusammenfassung

Diese Arbeit stellt einen integrierten Ansatz aus *Constraint Programming* (CP) und Gemischt-Ganzzahliger Programmierung (*Mixed Integer Programming*, MIP) vor, den wir *Constraint Integer Programming* (CIP) nennen. Sowohl Modellierungs- als auch Lösungstechniken beider Felder fließen in den neuen integrierten Ansatz ein, um die unterschiedlichen Stärken der beiden Gebiete zu kombinieren. Als weiteren Beitrag stellen wir der wissenschaftlichen Gemeinschaft die Software SCIP zur Verfügung, die ein Framework für Constraint Integer Programming darstellt und zusätzlich Techniken des SAT-Lösens beinhaltet. SCIP ist im Source Code für akademische und nicht-kommerzielle Zwecke frei erhältlich.

Unser Ansatz des Constraint Integer Programming ist eine Verallgemeinerung von MIP, die zusätzlich die Verwendung beliebiger Constraints erlaubt, solange sich diese durch lineare Bedingungen ausdrücken lassen falls alle ganzzahligen Variablen auf feste Werte eingestellt sind. Die Constraints werden von einer beliebigen Kombination aus CP- und MIP-Techniken behandelt. Dies beinhaltet insbesondere die *Domain Propagation*, die Relaxierung der Constraints durch lineare Ungleichungen, sowie die Verstärkung der Relaxierung durch dynamisch generierte Schnittebenen.

Die derzeitige Version von SCIP enthält alle Komponenten, die für das effiziente Lösen von Gemischt-Ganzzahligen Programmen benötigt werden. Die vorliegende Arbeit liefert eine ausführliche Beschreibung dieser Komponenten und bewertet verschiedene Varianten in Hinblick auf ihren Einfluß auf das Gesamt-Lösungsverhalten anhand von aufwendigen praktischen Experimenten. Dabei wird besonders auf die algorithmischen Aspekte eingegangen.

Der zweite Hauptteil der Arbeit befasst sich mit der Chip-Design-Verifikation, die ein wichtiges Thema innerhalb des Fachgebiets der *Electronic Design Automation* darstellt. Chip-Hersteller müssen sicherstellen, dass der logische Entwurf einer Schaltung der gegebenen Spezifikation entspricht. Andernfalls würde der Chip fehlerhaftes Verhalten aufweisen, dass zu Fehlfunktionen innerhalb des Gerätes führen kann, in dem der Chip verwendet wird. Ein wichtiges Teilproblem in diesem Feld ist das Eigenschafts-Verifikations-Problem, bei dem geprüft wird, ob der gegebene Schaltkreisentwurf eine gewünschte Eigenschaft aufweist. Wir zeigen, wie dieses Problem als Constraint Integer Program modelliert werden kann und geben eine Reihe von problemspezifischen Algorithmen an, die die Struktur der einzelnen Constraints und der Gesamtschaltung ausnutzen. Testrechnungen auf Industrie-Beispielen vergleichen unseren Ansatz mit den bisher verwendeten SAT-Techniken und belegen den Erfolg unserer Methode.

Abstract

This thesis introduces the novel paradigm of *constraint integer programming* (CIP), which integrates *constraint programming* (CP) and *mixed integer programming* (MIP) modeling and solving techniques. It is supplemented by the software SCIP, which is a solver and framework for constraint integer programming that also features SAT solving techniques. SCIP is freely available in source code for academic and non-commercial purposes.

Our constraint integer programming approach is a generalization of MIP that allows for the inclusion of arbitrary constraints, as long as they turn into linear constraints on the continuous variables after all integer variables have been fixed. The constraints, may they be linear or more complex, are treated by any combination of CP and MIP techniques: the propagation of the domains by constraint specific algorithms, the generation of a linear relaxation and its solving by LP methods, and the strengthening of the LP by cutting plane separation.

The current version of SCIP comes with all of the necessary components to solve mixed integer programs. In the thesis, we cover most of these ingredients and present extensive computational results to compare different variants for the individual building blocks of a MIP solver. We focus on the algorithms and their impact on the overall performance of the solver.

In addition to mixed integer programming, the thesis deals with *chip design verification*, which is an important topic of electronic design automation. Chip manufacturers have to make sure that the logic design of a circuit conforms to the specification of the chip. Otherwise, the chip would show an erroneous behavior that may cause failures in the device where it is employed. An important subproblem of chip design verification is the *property checking problem*, which is to verify whether a circuit satisfies a specified property. We show how this problem can be modeled as constraint integer program and provide a number of problem-specific algorithms that exploit the structure of the individual constraints and the circuit as a whole. Another set of extensive computational benchmarks compares our CIP approach to the current state-of-the-art SAT methodology and documents the success of our method.

Acknowledgements

Working at the Zuse Institute Berlin was a great experience for me, and thanks to the very flexible people in the administrations and management levels of ZIB and my new employer ILOG, this experience continues. It is a pleasure to be surrounded by lots of nice colleagues, even though some of them have the nasty habit to come into my office (without being formally invited!) with the only purpose of wasting my time by asking strange questions about SCIP and Cplex. And eating a piece of sponsored cake from time to time while discussing much more interesting topics than optimization or mathematical programming is always a valuable distraction. Thank you, Marc!

Work on this thesis actually began already in 2000 with the work on my computer science master's thesis. This was about applying neural networks to learn good branching strategies for MIP solvers. Research on such a topic is only possible if you have the source code of a state-of-the-art MIP solver. Fortunately, the former ZIB member Alexander Martin made his solver SIP available to me, such that I was relieved from inventing the wheel a second time. With the help of Thorsten Koch, I learned a lot by analyzing his algorithms.

Since I studied both, mathematics and computer science, I always looked for a topic that combines the two fields. In 2002, it came to my mind that the integration of integer programming and constraint programming would be a perfect candidate in this regard. Unfortunately, such an integration was way beyond the scope of SIP, such that I had to start from scratch at the end of 2002 with a quickly growing code that I called SCIP in order to emphasize its relation to SIP. At this point, I have to thank my advisor Martin Grötschel for his patience and for the freedom he offered me to do whatever I liked. For almost two years, I did not publish a single paper! Instead, I was sitting in my office, hacking the whole day on my code in order to get a basis on which I can conduct the actual research. And then, the miracle occurred, again thanks to Martin Grötschel: his connections to the group of Wolfram Büttner at Infineon (which later became the spin-off company OneSpin Solutions) resulted in the perfect project at the perfect moment: solving the chip design verification problem, an ideal candidate to tackle with constraint integer programming methods.

During the two-year period of the project, which was called Valse-XT, I learned a lot about the logic design of chips and how its correctness can be verified. I thank Raik Brinkmann for all his support. Having him as the direct contact person of the industry partner made "my" project considerably different from most of the other projects we had at ZIB: I obtained lots of test and benchmark data, and I received the data early enough to be useful within the duration of the project.

Another interesting person I learned to know within the project is Yakov Novikov. He is a brilliant SAT researcher from Minsk who is now working for OneSpin Solutions in Munich. Because the Berlin administration is a little more "flexible" than the one of Bavaria, he came to Berlin in order to deal with all the bureaucratic affairs regarding his entry to Germany. We first met when I picked him up at a subway station after he arrived with the train at Berlin-Ostbahnhof. At the Ostbahnhof, he was mugged by a gang of Russians. They thought that he is a "typical" Eastern

European who carries a lot of cash in order to buy a car in Germany. Fortunately, by showing some of his papers he could convince them that he is only a poor researcher who does not have any money. After telling me this story, he explained to me the key concepts in SAT solving during the remaining 10 minutes in the subway train: *conflict analysis* and the *two watched literals scheme* for fast Boolean constraint propagation. I was very excited and started with the integration of the *two watched literals scheme* into the set covering constraint handler a few days later. The generalization of conflict analysis to mixed integer programming followed after three weeks. As conflict analysis turned out to be a key ingredient for solving the chip verification problem with constraint integer programming, I am very thankful to Yakov for pointing me into this direction.

I thought after having implemented more than 250 000 lines of C code for SCIP and the chip verification solver, the writing of the thesis would be a piece of cake. What a mistake! The time passed by, and suddenly I exceeded my self-imposed deadline (to be finished before my 30th birthday) without having even started to write my thesis. Of course, SCIP improved considerably during that time, which is also the contribution of my great students Kati Wolter and Timo Berthold. Additionally, the public visibility of SCIP increased dramatically, thanks to Hans Mittelmann's benchmarking website. Nevertheless, it needed Thorsten Koch to convince me that starting to write things down is more important than to improve the performance of SCIP by another 10 %. I am very thankful for this advice!

Furthermore, I am very grateful to all the proof-readers of my thesis, which are Timo Berthold, Stefan Heinz, Thorsten Koch, and Marc Pfetsch from ZIB, Markus Wedler from TU Kaiserslautern, and Lloyd Clarke and Zonghao Gu from ILOG. All of you helped me to improve the thesis quite a bit. Most of all, I would like to thank Marc Pfetsch, who actually read (and commented on) around 75 % of the thesis. I also thank Kathleen Callaway (ILOG) for her willingness to review the grammar and punctuation, but unfortunately she became sick and could not make it in time. Therefore, the reader has to live with my language deficiencies.

Finally, I thank my ZIB colleagues Andreas Eisenblätter, Stefan Heinz, Marika Neumann, Sebastian Orlowski, Christian Raack, Thomas Schlechte, and Steffen Weider, who are the ones working on machines that are identical to mine, and who allowed me to spend their spare CPU cycles on my benchmark runs.

Am allermeisten aber danke ich Dir, Julia. Du hast mir die ganze Zeit über den Rücken freigehalten und es insbesondere in den letzten vier Monaten ertragen, dass ich fast jeden Tag erst nach Mitternacht nach Hause gekommen bin und zum Teil auch die Wochenenden am ZIB verbracht habe. Ich bedaure es sehr, dass ich Dich und die Kinder in dieser Zeit kaum zu Gesicht bekommen habe. Insbesondere sind die ersten sechs Lebensmonate von Mieke ziemlich schnell an mir vorübergezogen. Ab jetzt werde ich aber zu einem geregelten Familienleben zurückfinden. Ich liebe Dich!

Tobias Achterberg
Berlin, May 2007

Contents

Introduction

This thesis introduces *constraint integer programming* (CIP), which is a novel way to combine constraint programming (CP) and mixed integer programming (MIP) methodologies. CIP is a generalization of MIP that supports the notion of general constraints as in CP. This approach is supported by the CIP framework SCIP, which also integrates techniques from SAT solving.

We demonstrate the usefulness of SCIP on two tasks. First, we apply the constraint integer programming approach to pure mixed integer programs. Computational experiments show that SCIP is almost competitive to current state-of-the-art commercial MIP solvers, even though it incurs the overhead to support the more general constraint integer programming model. We describe the fundamental building blocks of MIP solvers and specify how they are implemented in SCIP. For all involved components, namely branching, node selection, domain propagation, cutting plane separation, primal heuristics, and presolving, we review existing ideas and introduce new variants that improve the runtime performance. Additionally, we generalize *conflict analysis*—a technique originating from the SAT community—to constraint and mixed integer programming. This novel concept in MIP solving yields noticeable performance improvements.

As a second application, we employ the SCIP framework to solve chip design verification problems as they arise in the logic design of integrated circuits. Although this problem class features a substantial kernel of linear constraints that can be efficiently handled by MIP techniques, it involves a few highly non-linear constraint types that are very hard to handle by pure mixed integer programming solvers. In this setting, the CIP approach is very effective: it can apply the full sophisticated MIP machinery to the linear part of the problem, while it is still able to deal with the non-linear constraints outside the MIP kernel by employing constraint programming techniques.

The idea of combining modeling and solving techniques from CP and MIP is not new. In the recent years, several authors showed that an integrated approach can help to solve optimization problems that were intractable with either of the two methods alone. For example, Timpe [205] applied a hybrid procedure to solve chemical industry planning problems that include lot-sizing, assignment, and sequencing as subproblems. Other examples of successful integration include the assembly line balancing problem (Bockmayr and Pisaruk [50]) and the parallel machine scheduling problem (Jain and Grossmann [122]).

Different approaches to integrate general constraint and mixed integer programming into a single framework have been proposed in the literature. For example, Bockmayr and Kasper [49] developed the framework Coupe, that unifies CP and MIP by observing that both techniques rely on branching and inference. In this setting, cutting planes and domain propagation are just specific types of inference. Althaus et al. [10] presented the system Scil, which introduces symbolic constraints on top of mixed integer programming solvers. Aron et al. [21] developed Simpl, a system for integrated modeling and solving. They view both, CP and MIP, as a special case of an infer-relax-restrict cycle in which CP and MIP techniques closely interact at any stage.

Our approach differs from the existing work in the level of integration. SCIP combines the CP, SAT, and MIP techniques on a very low level. In particular, all involved algorithms operate on a single search tree which yields a very close interaction. For example, MIP components can base their heuristic decisions on statistics that have been gathered by CP algorithms or vice versa, and both can use the dual information provided by the LP relaxation of the current subproblem. Furthermore, the SAT-like conflict analysis evaluates both the deductions discovered by CP techniques and the information obtained through the LP relaxation.

Content of the Thesis

This thesis consists of three parts. We now describe their content in more detail.

The first part illustrates the basic concepts of constraint programming, SAT solving, and mixed integer programming. Chapter 1 defines the three model types and gives a rough overview of how they can be solved in practice. The chapter concludes with the definition of the constraint integer program that forms the basis of our approach to integrate the solving and modeling techniques of the three areas. Chapter 2 presents the fundamental algorithms that are applied to solve CPs, MIPs, and SAT problems, namely branch-and-bound, cutting plane separation, and domain propagation. Finally, Chapter 3 explains the design principles of the CIP solving framework SCIP to set the stage for the description of the domain specific algorithms in the subsequent parts. In particular, we present sophisticated memory management methods, which yield an overall runtime performance improvement of 8 %.[1]

The second part of the thesis deals with the solution of mixed integer programs. After a general introduction to mixed integer programming in Chapter 4, we present the ideas and algorithms for the key components of branch-and-bound based MIP solvers as they are implemented in SCIP. Many of the techniques are gathered from the literature, but some components as well as a lot of algorithmic subtleties and small improvements are new developments. Except the introduction, every chapter of the second part concludes with computational experiments to evaluate the impact of the discussed algorithms on the MIP solving performance. Overall, this constitutes one of the most extensive computational studies on this topic that can be found in the literature. In total, we spent more than one CPU year on the preliminary and final benchmarks, solving 244 instances with 115 different parameter settings each, which totals to 28060 runs.

Chapter 5 addresses branching rules. We review the most popular strategies and introduce a new rule called *reliability branching*, which generalizes many of the previously known strategies. We show the relations of the various other rules to different parameter settings of *reliability branching*. Additionally, we propose a second novel branching approach, which we call *inference branching*. This rule is influenced by ideas of the SAT and CP communities and is particularly tailored for pure feasibility problems. Using *reliability branching* and *inference branching* in a hybrid fashion outperforms the previous state-of-the-art *pseudocost branching*

[1] We measure the performance in the geometric mean relative to the default settings of SCIP. For example, a performance improvement of 100 % for a default feature means that the solving process takes twice as long in the geometric mean if the feature is disabled.

with strong branching initialization rule by 8 %. On feasibility problems, we obtain a performance improvement of more than 50 %. Besides improving the branching strategies, we demonstrate the deficiencies of the still widely used *most infeasible branching*. Our computational experiments show that this rule, although seemingly a natural choice, is almost as poor as selecting the branching variable randomly.

Branching rules usually generate a "score" or "utility" value for the two child nodes associated to each branching candidate. The pseudocost estimates for the LP objective changes in the two branching directions are an example for such values. An important aspect of the branching variable selection is the combination of these two values into a single score value that is used to compare the branching candidates. Commonly, one uses a convex combination

$$\text{score}(q^-, q^+) = (1 - \mu) \cdot \min\{q^-, q^+\} + \mu \cdot \max\{q^-, q^+\}$$

of the two child node score values q^- and q^+ with parameter $\mu \in [0, 1]$. We propose a novel approach which employs a product based function

$$\text{score}(q^-, q^+) = \max\{q^-, \epsilon\} \cdot \max\{q^+, \epsilon\}$$

with $\epsilon = 10^{-6}$. Our computational results show that even for the best of five different μ values, the product function outperforms the linear approach by 14 %.

Chapter 6 deals with the node selection, which together with the branching rule forms the search component of the solver. Again, we review existing ideas and present several mixed strategies that aim to combine the advantages of the individual methods. Here, the impact on the solving performance is not as strong as for the branching rules. Compared to the basic *depth first* and *best first search* rules, however, the hybrid node selection strategy that we employ achieves an overall speedup of about 30 %.

Domain propagation and cutting plane separation constitute the inference engine of the solver. Chapter 7 deals with the former and commences with a detailed discussion of the propagation of general linear constraints, including numerical issues that have to be considered. A key concept in the theory of constraint programming to evaluate domain propagation algorithms is the notion of *local consistency* for which several variants are distinguished. Two of them are *bound consistency* and the stronger *interval consistency*. We show that bound consistency can be achieved easily for general linear constraints, but deciding interval consistency for linear equations is $\mathcal{NP}$-complete. However, if the constraint is a simple inequality $a^T x \leq \overline{\beta}$, our algorithm attains interval consistency. This means, the propagation is optimal in the sense that no further deductions can be derived by only looking at one constraint at a time together with the bounds and integrality restrictions of the involved variables.

In addition to general linear inequalities and equations, Chapter 7 deals with special cases of linear constraints like, for example, binary knapsack and set covering constraints. If restricted to propagating the constraints one at a time, we cannot get better than interval consistency. The data structures and algorithms, however, can be improved to obtain smaller memory consumption and runtime costs. In particular, the so-called *two watched literals* scheme of SAT solvers can be applied to set covering constraints.

Chapter 8 deals with the separation of cutting planes. As most of the details of cutting plane separation in SCIP can be found in the diploma thesis of Kati Wolter [218] and a comprehensive survey of the theory was recently given by Klar [132], we cover the topic only very briefly. We describe the different classes of cuts that are generated by SCIP and give a few comments on the theoretical

background and the implementation of the separation algorithms. As in the previous chapters, we conclude with a computational study to evaluate the effectiveness of the various cut separators. It turns out that cutting planes yield a performance improvement of more than 100 % with the complemented mixed integer rounding cuts having the largest impact. Besides the separation of the different classes of cutting planes, it is also important to have good selection criteria in order to decide which of the generated cuts should actually be added to the LP relaxation. Our experiments show that very simple strategies like adding all the cuts that have been found or adding only one cut per round increase the total runtime by 70 % and 80 %, respectively, compared to a sophisticated rule that carefully selects a subset of the available cutting planes. More interestingly, choosing cuts which are pairwise almost orthogonal yields a 20 % performance improvement over the common strategy of only considering the cut violations.

Chapter 9 gives an overview of the primal heuristics included in SCIP. Similar to the cutting planes we do not go into the details, since they can be found in the diploma thesis of Timo Berthold [41]. We describe only the general ideas of the various heuristics and conclude with a computational study. Our results indicate that the contribution of primal heuristics to decrease the time to solve MIP instances to optimality is rather small. Disabling all primal heuristics increases the time to find the optimal solution and to prove that no better solution exists by only 14 %. However, proving optimality is not always the primary goal of a user. For practical applications, it is usually enough to find feasible solutions of reasonable quality quickly. For this purpose, primal heuristics are a useful tool.

Chapter 10 presents the presolving techniques that are incorporated in SCIP. Besides calling the regular domain propagation algorithms for the global bounds of the variables as a subroutine, they comprise more sophisticated methods to alter the problem structure with the goal of decreasing the size of the instance and strengthening its LP relaxation. As for domain propagation, we first discuss the presolving of general linear constraints and continue with the special cases like binary knapsack or set covering constraints. In addition, we present four methods that can be applied to any constraint integer program, independent from the involved constraint types.

While all of these presolving techniques are well known in the MIP community, Chapter 10 includes the additional method of *restarts*. This method has not been used by MIP solvers in the past, although it is a key ingredient in modern SAT solvers. It means to interrupt the branch-and-bound solving process, reapply presolving, and perform another pass of branch-and-bound search. The information about the problem instance that was discovered in the previous solving pass can lead to additional presolving reductions and to improved decisions in the subsequent run, for example in the branching variable selection. Although SAT solvers employ periodic restarts throughout the whole solving process, we concluded that in the case of MIP it is better to restart only directly after the root node has been solved. We restart the solving process if a certain amount of additional variable fixings have been generated, for example by cutting planes or strong branching. The computational results at the end of the chapter show that the regular presolving techniques yield a 90 % performance improvement, while restarts achieve an additional reduction of almost 10 %.

Finally, Chapter 11 contributes another successful integration of a SAT technique into the domain of mixed integer programming, namely the idea of *conflict analysis*. Using this method, one can extract structural knowledge about the problem instance at hand from the infeasible subproblems that are processed during the branch-and-bound search. We show how conflict analysis as employed for SAT can

be generalized to the much richer modeling constructs available in mixed integer programming, namely general linear constraints and integer and continuous variables. A particularly interesting aspect is the analysis of infeasible or bound exceeding LPs for which we use dual information in order to obtain an initial starting point for the subsequent analysis of the branchings and propagations that lead to the conflict. The computational experiments identify a performance improvement of more than 10 %, which can be achieved by a reasonable effort spent on the analysis of infeasible subproblems.

In the third part of the thesis, we discuss the application of our constraint integer programming approach to the chip design verification problem. The task is to verify whether a given logic design of a chip satisfies certain desired properties. All the transition operators that can be used in the logic of a chip, for example addition, multiplication, shifting, or comparison of registers, are expressed as constraints of a CIP model. Verifying a property means to decide whether the CIP model is feasible or not.

Chapter 12 gives an introduction to the application and an overview of current state-of-the-art solving techniques. The property checking problem is formally defined in Chapter 13, and we present our CIP model together with a list of all constraint types that can appear in the problem instances. In total, 22 different operators have to be considered.

In Chapter 14 we go into the details of the implementation. For each constraint type it is explained how an LP relaxation can be constructed and how the domain propagation and presolving algorithms exploit the special structure of the constraint class to efficiently derive deductions. Since the semantics of some of the operators can be represented by constraints of a different operator type, we end up with 10 non-trivial constraint handlers. In addition, we need a supplementary constraint class that provides the link between the bit and word level representations of the problem instance.

The most complex algorithms deal with the multiplication of two registers. These constraints feature a highly involved LP relaxation using a number of auxiliary variables. In addition, we implemented three domain propagation algorithms that operate on different representations of the constraint: the LP representation, the bit level representation, and a symbolic representation. For the latter, we employ term algebra techniques and define a term rewriting system. We state a term normalization algorithm and prove its termination by providing a well-founded partial ordering on the operations of the underlying algebraic signature.

In regular mixed integer programming, every constraint has to be modeled with linear inequalities and equations. In contrast, in our constraint integer programming approach we can treat each constraint class by CP or MIP techniques alone, or we can employ both of them simultaneously. The benefit of this flexibility is most apparent for the *shifting* and *slicing* operators. We show, for example, that a reasonable LP relaxation of a single *shift left* constraint on 64-bit registers includes 2 145 auxiliary variables and 6 306 linear constraints with a total of 16 834 non-zero coefficients. Therefore, a pure MIP solver would have to deal with very large problem instances. In contrast, the CIP approach can handle these constraints outside the LP relaxation by employing CP techniques alone, which yields much smaller node processing times.

Chapter 15 introduces two application specific presolving techniques. The first is the use of a term rewriting system to generate problem reductions on a symbolic level. As for the symbolic propagation of multiplication constraints, we present a term normalization algorithm and prove that it terminates for all inputs. The

normalized terms can then be compared in order to identify fixings and equivalences of variables. The second presolving technique analyzes the *function graph* of the problem instance in order to identify parts of the circuit that are irrelevant for the property that should be verified. These irrelevant parts are removed from the problem instance, which yields a significant reduction in the problem size on some instances.

Chapter 16 gives a short overview of the search strategies, i.e., branching and node selection rules, that are employed for solving the property checking problem. Finally, computational results in Chapter 17 demonstrate the effectiveness of our integrated approach by comparing its performance to the state-of-the-art in the field, which is to apply SAT techniques for modeling and solving the problem. While SAT solvers are usually much faster in finding counter-examples that prove the invalidity of a property, our CIP based procedure can be—depending on the circuit and property—several orders of magnitude faster than the traditional approach.

SOFTWARE

As a supplement to this thesis we provide the constraint integer programming framework SCIP, which is freely available in source code for academic and non-commercial use and can be downloaded from `http://scip.zib.de`. It has LP solver interfaces to CLP [87], CPLEX [118], MOSEK [167], SOPLEX [219], and XPRESS [76]. The current version 0.90i consists of 223 178 lines of C code and C++ wrapper classes, which breaks down to 145 676 lines for the CIP framework and 77 502 lines for the various plugins. For the special plugins dealing with the chip design verification problem, an additional 58 363 lines of C code have been implemented.

The development of SCIP started in October 2002. Most ideas and algorithms of the then state-of-the-art MIP solver SIP of Alexander Martin [159] were transfered into the initial version of SCIP. Since then, many new features have been developed that further have improved the performance and the usability of the framework. As a stand-alone tool, SCIP in combination with SOPLEX as LP solver is the fastest non-commercial MIP solver that is currently available, see Mittelmann [166]. Using CPLEX 10 as LP solver, the performance of SCIP is even comparable to the today's best commercial codes CPLEX and XPRESS: the computational results in Appendix C show that SCIP 0.90i is on average only 63 % slower than CPLEX 10.

As a library, SCIP can be used to develop branch-cut-and-price algorithms, and it can be extended to support additional classes of non-linear constraints by providing so-called constraint handler plugins. The solver for the chip design verification problem is one example of this usage. It is the hope of the author that the performance and the flexibility of the software combined with the availability of the source code fosters research in the area of constraint and mixed integer programming. Apart from the chip design verification problem covered in this thesis, SCIP has already been used in various other projects, see, for example, Pfetsch [187], Anders [12], Armbruster et al. [19, 20], Bley et al. [48], Joswig and Pfetsch [126], Koch [135], Nunkesser [176], Armbruster [18], Bilgen [45], Ceselli et al. [58], Dix [81], Kaibel et al. [127], Kutschka [138], or Orlowski et al. [178]. Additionally, it is used for teaching graduate students, see Achterberg, Grötschel, and Koch [3].

Part I

Concepts

Chapter 1

Basic Definitions

In this chapter, we present three model types of search problems—constraint programs, satisfiability problems, and mixed integer programs. We specify the basic solution strategies of the three fields and highlight the key ideas that make the approaches efficient in practice. Finally, we derive a problem class which we call *constraint integer program*. This problem class forms the basis of our approach to integrate the modeling and solving techniques from the three domains into a single framework.

1.1 Constraint Programs

The basic concept of general logical *constraints* was used in 1963 by Sutherland [202, 203] in his interactive drawing system Sketchpad. In the 1970's, the concept of *logic programming* emerged in the artificial intelligence community in the context of automated theorem proving and language processing, most notably with the logic programming language Prolog developed by Colmerauer et al. [64, 66] and Kowalski [136]. In the 1980's, constraint solving was integrated into logic programming, resulting in the so-called *constraint logic programming* paradigm, see, e.g., Jaffar and Lassez [121], Dincbas et al. [80], or Colmerauer [65].

In its most general form, the basic model type that is addressed by the above approaches is the *constraint satisfaction problem* (CSP), which is defined as follows:

Definition 1.1 (constraint satisfaction problem). A *constraint satisfaction problem* is a pair $\text{CSP} = (\mathfrak{C}, \mathfrak{D})$ with $\mathfrak{D} = \mathcal{D}_1 \times \ldots \times \mathcal{D}_n$ representing the domains of finitely many variables $x_j \in \mathcal{D}_j$, $j = 1, \ldots, n$, and $\mathfrak{C} = \{\mathcal{C}_1, \ldots, \mathcal{C}_m\}$ being a finite set of constraints $\mathcal{C}_i : \mathfrak{D} \to \{0, 1\}$, $i = 1, \ldots, m$. The task is to decide whether the set

$$X_{\text{CSP}} = \{x \mid x \in \mathfrak{D},\ \mathfrak{C}(x)\}, \text{ with } \mathfrak{C}(x) :\Leftrightarrow \forall i = 1, \ldots, m : \mathcal{C}_i(x) = 1$$

is non-empty, i.e., to either find a solution $x \in \mathfrak{D}$ satisfying $\mathfrak{C}(x)$ or to prove that no such solution exists. A CSP where all domains $\mathcal{D} \in \mathfrak{D}$ are finite is called a *finite domain constraint satisfaction problem* (CSP(FD)).

Note that there are no further restrictions imposed on the constraint predicates $\mathcal{C}_i \in \mathfrak{C}$. The optimization version of a constraint satisfaction problem is called *constraint optimization program* or, for short, *constraint program* (CP):

Definition 1.2 (constraint program). A *constraint program* is a triple $\text{CP} = (\mathfrak{C}, \mathfrak{D}, f)$ and consists of solving

$$\text{(CP)} \quad f^\star = \min\{f(x) \mid x \in \mathfrak{D},\ \mathfrak{C}(x)\}$$

with the set of domains $\mathfrak{D} = \mathcal{D}_1 \times \ldots \times \mathcal{D}_n$, the constraint set $\mathfrak{C} = \{\mathcal{C}_1, \ldots, \mathcal{C}_m\}$, and an objective function $f : \mathfrak{D} \to \mathbb{R}$. We denote the set of feasible solutions by

$X_{\text{CP}} = \{x \mid x \in \mathfrak{D},\ \mathfrak{C}(x)\}$. A CP where all domains $\mathcal{D} \in \mathfrak{D}$ are finite is called a *finite domain constraint program* (CP(FD)).

Like the constraint predicates $\mathcal{C}_i \in \mathfrak{C}$ the objective function f may be an arbitrary mapping.

Existing constraint programming solvers like CAL [7], CHIP [80], CLP($\mathcal{R}$) [121], PROLOG III [65], or ILOG SOLVER [188] are usually restricted to finite domain constraint programming.

To solve a CP(FD), the problem is recursively split into smaller subproblems (usually by splitting a single variable's domain), thereby creating a branching tree and implicitly enumerating all potential solutions (see Section 2.1). At each subproblem (i.e., node in the tree) domain propagation is performed to exclude further values from the variables' domains (see Section 2.3). These domain reductions are inferred by the single constraints (primal reductions) or by the objective function and a feasible solution $\hat{x} \in X_{\text{CP}}$ (dual reductions). If every variable's domain is thereby reduced to a single value, a new primal solution is found. If any of the variables' domains becomes empty, the subproblem is discarded and a different leaf of the current branching tree is selected to continue the search.

The key element for solving constraint programs in practice is the efficient implementation of domain propagation algorithms, which exploit the structure of the involved constraints. A CP solver usually includes a library of constraint types with specifically tailored propagators. Furthermore, it provides infrastructure for managing local domains and representing the subproblems in the tree, such that the user can integrate algorithms into the CP framework in order to control the search or to deal with additional constraint classes.

1.2 Satisfiability Problems

The satisfiability problem (SAT) is defined as follows. The Boolean truth values *false* and *true* are identified with the values 0 and 1, respectively, and Boolean formulas are evaluated correspondingly.

Definition 1.3 (satisfiability problem). Let $\mathfrak{C} = \mathcal{C}_1 \wedge \ldots \wedge \mathcal{C}_m$ be a logic formula in conjunctive normal form (CNF) on Boolean variables $x_1, \ldots, x_n$. Each clause $\mathcal{C}_i = \ell_1^i \vee \ldots \vee \ell_{k_i}^i$ is a disjunction of literals. A literal $\ell \in L = \{x_1, \ldots, x_n, \bar{x}_1, \ldots, \bar{x}_n\}$ is either a variable x_j or the negation of a variable $\bar{x}_j$. The task of the *satisfiability problem* (SAT) is to either find an assignment $x^\star \in \{0,1\}^n$, such that the formula $\mathfrak{C}$ is satisfied, i.e., each clause $\mathcal{C}_i$ evaluates to 1, or to conclude that $\mathfrak{C}$ is unsatisfiable, i.e., for all $x \in \{0,1\}^n$ at least one $\mathcal{C}_i$ evaluates to 0.

SAT was the first problem shown to be $\mathcal{NP}$-complete by Cook [68]. Since SAT is a special case of a constraint satisfaction problem, CSP is $\mathcal{NP}$-complete as well. Besides its theoretical relevance, SAT has many practical applications, e.g., in the design and verification of integrated circuits or in the design of logic based intelligent systems. We refer to Biere and Kunz [44] for an overview of SAT techniques in chip verification and to Truemper [206] for details on logic based intelligent systems.

Modern SAT solvers like BERKMIN [100], CHAFF [168], or MINISAT [82] rely on the following techniques:

- ▷ using a branching scheme (the *DPLL-algorithm* of Davis, Putnam, Logemann, and Loveland [77, 78]) to split the problem into smaller subproblems (see Section 2.1),
- ▷ applying *Boolean constraint propagation* (BCP) [220] on the subproblems, which is a special form of domain propagation (see Section 2.3),
- ▷ analyzing infeasible subproblems to produce conflict clauses [157], which help to prune the search tree later on (see Chapter 11), and
- ▷ restarting the search in a periodic fashion in order to revise the branching decisions after having gained new knowledge about the structure of the problem instance, which is captured by the conflict clauses, see Gomes et al. [101].

The DPLL-algorithm creates two subproblems at each node of the search tree by fixing a single variable to zero and one, respectively. The nodes are processed in a depth first fashion.

1.3 Mixed Integer Programs

A mixed integer program (MIP) is defined as follows.

Definition 1.4 (mixed integer program). Given a matrix $A \in \mathbb{R}^{m \times n}$, vectors $b \in \mathbb{R}^m$, and $c \in \mathbb{R}^n$, and a subset $I \subseteq N = \{1, \ldots, n\}$, the *mixed integer program* $\text{MIP} = (A, b, c, I)$ is to solve

$$\text{(MIP)} \quad c^\star = \min \{c^T x \mid Ax \leq b,\ x \in \mathbb{R}^n,\ x_j \in \mathbb{Z} \text{ for all } j \in I\}.$$

The vectors in the set $X_{\text{MIP}} = \{x \in \mathbb{R}^n \mid Ax \leq b,\ x_j \in \mathbb{Z} \text{ for all } j \in I\}$ are called *feasible solutions* of MIP. A feasible solution $x^\star \in X_{\text{MIP}}$ of MIP is called *optimal* if its objective value satisfies $c^T x^\star = c^\star$.

MIP solvers usually treat simple bound constraints $l_j \leq x_j \leq u_j$ with $l_j, u_j \in \mathbb{R} \cup \{\pm\infty\}$ separately from the remaining constraints. In particular, integer variables with bounds $0 \leq x_j \leq 1$ play a special role in the solving algorithms and are a very important tool to model "yes/no" decisions. We refer to the set of these *binary variables* with $B := \{j \in I \mid l_j = 0 \text{ and } u_j = 1\} \subseteq I \subseteq N$. In addition, we denote the continuous variables by $C := N \setminus I$.

Common special cases of MIP are linear programs (LPs) with $I = \emptyset$, integer programs (IPs) with $I = N$, mixed binary programs (MBPs) with $B = I$, and binary programs (BPs) with $B = I = N$. The satisfiability problem is a special case of a BP without objective function. Since SAT is $\mathcal{NP}$-complete, BP, IP, and MIP are $\mathcal{NP}$-hard. Nevertheless, linear programs are solvable in polynomial time, which was first shown by Khatchiyan [130, 131] using the so-called *ellipsoid method*.

Note that in contrast to constraint programming, in mixed integer programming we are restricted to

- ▷ linear constraints,
- ▷ a linear objective function, and
- ▷ integer or real-valued domains.

Despite these restrictions in modeling, practical applications prove that MIP, IP, and BP can be very successfully applied to many real-word problems. However, it often requires expert knowledge to generate models that can be solved with current general purpose MIP solvers. In many cases, it is even necessary to adapt the solving process itself to the specific problem structure at hand. This can be done with the help of an MIP framework.

Like CP and SAT solvers, most modern MIP solvers recursively split the problem into smaller subproblems, thereby generating a branching tree (see Section 2.1). However, the processing of the nodes is different. For each node of the tree, the LP relaxation is solved, which can be constructed from the MIP by removing the integrality conditions:

Definition 1.5 (LP relaxation of an MIP). Given a mixed integer program $\text{MIP} = (A, b, c, I)$, its *LP relaxation* is defined as

$$\text{(LP)} \quad \check{c} = \min \{c^T x \mid Ax \leq b,\ x \in \mathbb{R}^n\}.$$

$X_{\text{LP}} = \{x \in \mathbb{R}^n \mid Ax \leq b\}$ is the set of *feasible solutions* of the LP relaxation. An LP-feasible solution $\check{x} \in X_{\text{LP}}$ is called *LP-optimal* if $c^T \check{x} = \check{c}$.

The LP relaxation can be strengthened by cutting planes which use the LP information and the integrality restrictions to derive valid inequalities that cut off the solution of the current LP relaxation without removing integral solutions (see Section 2.2). The objective value $\check{c}$ of the LP relaxation provides a lower bound for the whole subtree, and if this bound is not smaller than the value $\hat{c} = c^T \hat{x}$ of the current best primal solution $\hat{x}$, the node and its subtree can be discarded. The LP relaxation usually gives a much stronger bound than the one that is provided by simple dual propagation of CP solvers. The solution of the LP relaxation usually requires much more time, however.

The most important ingredients of an MIP solver implementation are a fast and numerically stable LP solver, cutting plane separators, primal heuristics, and presolving algorithms (see Bixby et al. [46]). Additionally, the applied branching rule is of major importance (see Achterberg, Koch, and Martin [5]). Necessary infrastructure includes the management of subproblem modifications, LP warm start information, and a cut pool.

Modern MIP solvers like CBC [86], CPLEX [118], GLPK [99], LINDO [147], MINTO [171, 172], MOSEK [167], SIP [159], SYMPHONY [190], or XPRESS [76] offer a variety of different general purpose separators which can be activated for solving the problem instance at hand (see Atamtürk and Savelsbergh [26] for a feature overview for a number of MIP solvers). It is also possible to add problem specific cuts through callback mechanisms, thus providing some of the flexibility a full MIP framework offers. These mechanisms are in many cases sufficient to solve a given problem instance. With the help of modeling tools like AIMMS [182], AMPL [89], GAMS [56], LINGO [148], MOSEL [75], MPL [162], OPL [119], or ZIMPL [133] it is often even possible to formulate the model in a mathematical fashion, to automatically transform the model and data into solver input, and to solve the instance within reasonable time. In this setting, the user does not need to know the internals of the MIP solver, which is used as a black-box tool.

Unfortunately, this rapid mathematical prototyping chain (see Koch [134]) does not yield results in acceptable solving time for every problem class, sometimes not even for small instances. For these problem classes, the user has to develop special

purpose codes with problem specific algorithms. To provide the necessary infrastructure like the branching tree and LP management, or to give support for standard general purpose algorithms like LP based cutting plane separators or primal heuristics, a branch-and-cut framework like Abacus [204], the tools provided by the Coin project [63], or the callback mechanisms provided, for example, by Cplex or Xpress can be used. As we will see in the following, SCIP can also be used in this fashion.

1.4 Constraint Integer Programs

As described in the previous sections, most solvers for constraint programs, satisfiability problems, and mixed integer programs share the idea of dividing the problem into smaller subproblems and implicitly enumerating all potential solutions. They differ, however, in the way of processing the subproblems.

Because MIP is a very specific case of CP, MIP solvers can apply sophisticated problem specific algorithms that operate on the subproblem as a whole. In particular, they use the simplex algorithm invented by Dantzig [73] to solve the LP relaxations, and cutting plane separators like the Gomory cut separator [104].

In contrast, due to the unrestricted definition of CPs, CP solvers cannot take such a global perspective. They have to rely on the constraint propagators, each of them exploiting the structure of a single constraint class. Usually, the only communication between the individual constraints takes place via the variables' domains. An advantage of CP is, however, the possibility to model the problem more directly, using very expressive constraints which contain a lot of structure. Transforming those constraints into linear inequalities can conceal their structure from an MIP solver, and therefore lessen the solver's ability to draw valuable conclusions about the instance or to make the right decisions during the search.

SAT is also a very specific case of CP with only one type of constraints, namely Boolean clauses. A clause $\mathcal{C}_i = \ell_1^i \vee \ldots \vee \ell_{k_i}^i$ can easily be linearized with the set covering constraint $\ell_1^i + \ldots + \ell_{k_i}^i \geq 1$. However, this LP relaxation of SAT is rather useless, since it cannot detect the infeasibility of subproblems earlier than domain propagation: by setting all unfixed variables to $\check{x}_j = \frac{1}{2}$, the linear relaxations of all clauses with at least two unfixed literals are satisfied. Therefore, SAT solvers mainly exploit the special problem structure to speed up the domain propagation algorithm and to improve the underlying data structures.

The hope of combining CP, SAT, and MIP techniques is to combine their advantages and to compensate for their individual weaknesses. We propose the following slight restriction of a CP to specify our integrated approach:

Definition 1.6 (constraint integer program). A *constraint integer program* $\text{CIP} = (\mathfrak{C}, I, c)$ consists of solving

$$\text{(CIP)} \quad c^\star = \min\{c^T x \mid \mathfrak{C}(x),\ x \in \mathbb{R}^n,\ x_j \in \mathbb{Z} \text{ for all } j \in I\}$$

with a finite set $\mathfrak{C} = \{\mathcal{C}_1, \ldots, \mathcal{C}_m\}$ of constraints $\mathcal{C}_i : \mathbb{R}^n \to \{0,1\}$, $i = 1, \ldots, m$, a subset $I \subseteq N = \{1, \ldots, n\}$ of the variable index set, and an objective function vector $c \in \mathbb{R}^n$. A CIP has to fulfill the following condition:

$$\forall \hat{x}_I \in \mathbb{Z}^I\ \exists (A', b') : \{x_C \in \mathbb{R}^C \mid \mathfrak{C}(\hat{x}_I, x_C)\} = \{x_C \in \mathbb{R}^C \mid A' x_C \leq b'\} \tag{1.1}$$

with $C := N \setminus I$, $A' \in \mathbb{R}^{k \times C}$, and $b' \in \mathbb{R}^k$ for some $k \in \mathbb{Z}_{\geq 0}$.

Restriction (1.1) ensures that the remaining subproblem after fixing the integer variables is always a linear program. This means that in the case of finite domain integer variables, the problem can be—in principle—completely solved by enumerating all values of the integer variables and solving the corresponding LPs. Note that this does not forbid quadratic or even more involved expressions. Only the remaining part after fixing (and thus eliminating) the integer variables must be linear in the continuous variables.

The linearity restriction of the objective function can easily be compensated by introducing an auxiliary objective variable z that is linked to the actual non-linear objective function with a non-linear constraint $z = f(x)$. We just demand a linear objective function in order to simplify the derivation of the LP relaxation. The same holds for omitting the general variable domains $\mathfrak{D}$ that exist in Definition 1.2 of the constraint program. They can also be represented as additional constraints. Therefore, every CP that meets Condition (1.1) can be represented as constraint integer program. In particular, we can observe the following:

Proposition 1.7. The notion of constraint integer programming generalizes finite domain constraint programming and mixed integer programming:

(a) Every CP(FD) and CSP(FD) can be modeled as a CIP.

(b) Every MIP can be modeled as CIP.

Proof. The notion of a constraint is the same in CP as in CIP. The linear system $Ax \leq b$ of an MIP is a conjunction of linear constraints, each of which is a special case of the general constraint notion in CP and CIP. Therefore, we only have to verify Condition (1.1).

For a CSP(FD), all variables have finite domain and can therefore be equivalently represented as integers, leaving Condition (1.1) empty. In the case of a CP(FD), the only non-integer variable is the auxiliary objective variable z, i.e., $x_C = (z)$. Therefore, Condition (1.1) can be satisfied for a given $\hat{x}_I$ by setting

$$A' := \begin{pmatrix} 1 \\ -1 \end{pmatrix} \quad \text{and} \quad b' := \begin{pmatrix} f(\hat{x}_I) \\ -f(\hat{x}_I) \end{pmatrix}.$$

For an MIP, partition the constraint matrix $A = (A_I, A_C)$ into the columns of the integer variables I and the continuous variables C. For a given $\hat{x}_I \in \mathbb{Z}^I$ set $A' := A_C$ and $b' := b - A_I\hat{x}_I$ to meet Condition (1.1). □

Like for a mixed integer program, we can define the notion of the LP relaxation for a constraint integer program:

Definition 1.8 (LP relaxation of a CIP). Given a constraint integer program CIP $= (\mathfrak{C}, I, c)$, a linear program

$$\text{(LP)} \quad \check{c} = \min \{c^T x \mid Ax \leq b,\ x \in \mathbb{R}^n\}$$

is called *LP relaxation of CIP* if

$$\{x \in \mathbb{R}^n \mid Ax \leq b\} \supseteq \{x \in \mathbb{R}^n \mid \mathfrak{C}(x), x_j \in \mathbb{Z} \text{ for all } j \in I\}.$$

Chapter 2

Algorithms

This chapter presents algorithms that can be used to solve constraint programs, satisfiability problems, and mixed integer programs. All of the three problem classes are commonly solved by *branch-and-bound*, which is explained in Section 2.1.

State-of-the-art MIP solvers heavily rely on the linear programming (LP) relaxation to calculate lower bounds for the subproblems of the search tree and to guide the branching decision. The LP relaxation can be tightened to improve the lower bounds by *cutting planes*, see Section 2.2.

In contrast to MIP, constraint programming is not restricted to linear constraints to define the feasible set. This means, there usually is no canonical linear relaxation at hand that can be used to derive lower bounds for the subproblems. Therefore, one has to stick to other algorithms to prune the search tree as much as possible in order to avoid the immense running time of complete enumeration. A method that is employed in practice is *domain propagation*, which is a restricted version of the so-called *constraint propagation*. Section 2.3 gives an overview of this approach. Note that MIP solvers are also applying domain propagation on the subproblems in the search tree. However, the MIP community usually calls this technique "*node preprocessing*".

Although the clauses that appear in a SAT problem can easily be represented as linear constraints, the LP relaxation of a satisfiability problem is almost useless since SAT has no objective function and the LP can always be satisfied by setting $x_j = \frac{1}{2}$ for all variables (as long as each clause contains at least two literals). Therefore, SAT solvers operate similar to CP solvers and rely on branching and domain propagation.

Overall, the three algorithms presented in this chapter (branch-and-bound, LP relaxation strengthened by cutting planes, and domain propagation) form the basic building blocks of our integrated constraint integer programming solver SCIP.

2.1 Branch and Bound

The branch-and-bound procedure is a very general and widely used method to solve optimization problems. It is also known as *implicit enumeration*, *divide-and-conquer*, *backtracking*, or *decomposition*. The idea is to successively divide the given problem instance into smaller subproblems until the individual subproblems are easy to solve. The best of the subproblems' solutions is the global optimum. Algorithm 2.1 summarizes this procedure.

The splitting of a subproblem into two or more smaller subproblems in Step 7 is called *branching*. During the course of the algorithm, a *branching tree* is created with each node representing one of the subproblems (see Figure 2.1). The root of the tree corresponds to the initial problem R, while the leaves are either "easy" subproblems that have already been solved or subproblems in $\mathcal{L}$ that still have to be processed.

The intention of the *bounding* in Step 5 is to avoid a complete enumeration of all potential solutions of R, which are usually exponentially many. In order for bounding

Algorithm 2.1 Branch-and-bound

Input: Minimization problem instance R.

Output: Optimal solution $x^\star$ with value $c^\star$, or conclusion that R has no solution, indicated by $c^\star = \infty$.

1. Initialize $\mathcal{L} := \{R\}$, $\hat{c} := \infty$. [*init*]
2. If $\mathcal{L} = \emptyset$, stop and return $x^\star = \hat{x}$ and $c^\star = \hat{c}$. [*abort*]
3. Choose $Q \in \mathcal{L}$, and set $\mathcal{L} := \mathcal{L} \setminus \{Q\}$. [*select*]
4. Solve a relaxation Q_{relax} of Q. If Q_{relax} is empty, set $\check{c} := \infty$. Otherwise, let $\check{x}$ be an optimal solution of Q_{relax} and $\check{c}$ its objective value. [*solve*]
5. If $\check{c} \geq \hat{c}$, goto Step 2. [*bound*]
6. If $\check{x}$ is feasible for R, set $\hat{x} := \check{x}$, $\hat{c} := \check{c}$, and goto Step 2. [*check*]
7. Split Q into subproblems $Q = Q_1 \cup \ldots \cup Q_k$, set $\mathcal{L} := \mathcal{L} \cup \{Q_1, \ldots, Q_k\}$, and goto Step 2. [*branch*]

to be effective, good lower (dual) bounds $\check{c}$ and upper (primal) bounds $\hat{c}$ must be available. Lower bounds are calculated with the help of a relaxation Q_{relax} which should be easy to solve. Upper bounds can be found during the branch-and-bound algorithm in Step 6, but they can also be generated by primal heuristics.

The node selection in Step 3 and the branching scheme in Step 7 determine important decisions of a branch-and-bound algorithm that should be tailored to the given problem class. Both of them have a major impact on how early good primal solutions can be found in Step 6 and how fast the lower bounds of the open subproblems in $\mathcal{L}$ increase. They influence the bounding in Step 5, which should cut off subproblems as early as possible and thereby prune large parts of the search

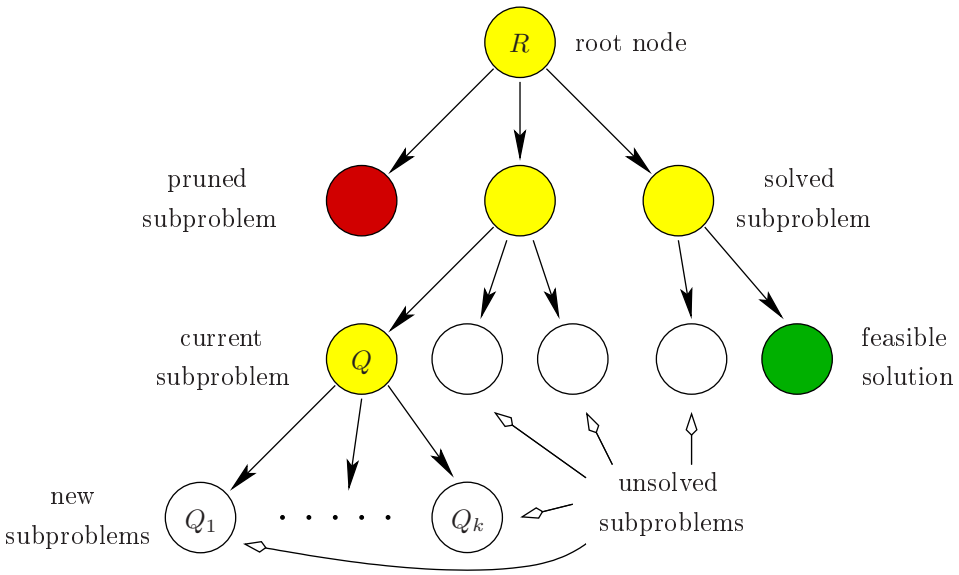

Figure 2.1. Branch-and-bound search tree.

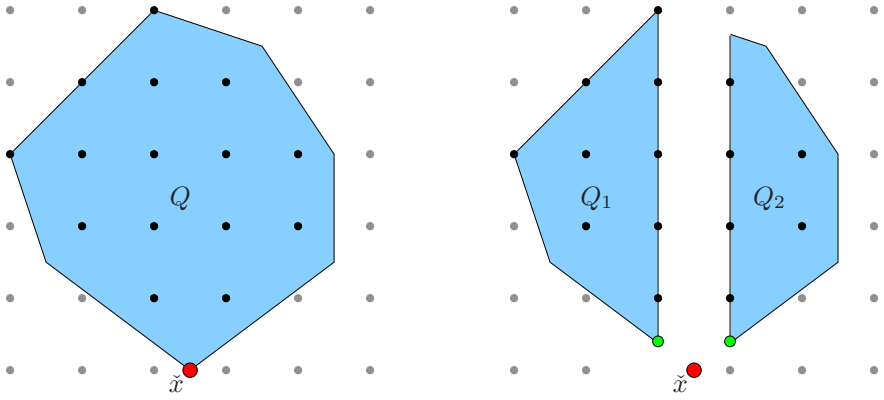

Figure 2.2. LP based branching on a single fractional variable.

tree. Even more important for a branch-and-bound algorithm to be effective is the type of relaxation that is solved in Step 4. A reasonable relaxation must fulfill two usually opposing requirements: it should be easy to solve, and it should yield strong dual bounds.

In mixed integer programming, the most widely used relaxation is the LP relaxation (see Definition 1.5), which proved to be very successful in practice. Currently, almost all efficient commercial and academic MIP solvers are LP relaxation based branch-and-bound algorithms. This includes the solvers mentioned in Section 1.3.

Besides supplying a dual bound that can be exploited for the bounding in Step 5, the LP relaxation can also be used to guide the branching decisions of Step 7. The most popular branching strategy in MIP solving is to split the domain of an integer variable x_j, $j \in I$, with fractional LP value $\check{x}_j \notin \mathbb{Z}$ into two parts, thus creating the two subproblems $Q_1 = Q \cap \{x_j \leq \lfloor \check{x}_j \rfloor\}$ and $Q_2 = Q \cap \{x_j \geq \lceil \check{x}_j \rceil\}$ (see Figure 2.2). Methods to select a fractional variable as branching variable are discussed in Chapter 5.

In constraint programming, the branching step is usually carried out by selecting an integer variable x_j and fix it to a certain value $x_j = v \in \mathcal{D}_j$ in one child node and rule out the value in the other child node by enforcing $x_j \in \mathcal{D}_j \setminus \{v\}$. In contrast to MIP, constraint programs do not have a strong canonical relaxation like the LP relaxation. Although there might be good relaxations for special types of constraint programs, there is no useful relaxation available for the general model. Therefore, CP solvers implement the bounding Step 5 of Algorithm 2.1 only by propagating the objective function constraint $f(x) < \hat{c}$ with $\hat{c}$ being the value of the current incumbent solution. Thus, the strength of the bounding step heavily depends on the propagation potential of the objective function constraint. In fact, CP solvers are usually inferior to MIP solvers on problems where achieving feasibility is easy, but finding the optimal solution is hard.

The branching applied in SAT solvers is very similar to the one of constraint programming solvers. Since all variables are binary, however, it reduces to selecting a variable x_j and fixing it to $x_j = 0$ in one child node and to $x_j = 1$ in the other child node. Actually, current SAT solvers do not even need to represent the branching decisions in a tree. Because they apply depth first search, they only need to store the nodes on the path from the root node to the current node. This simplification

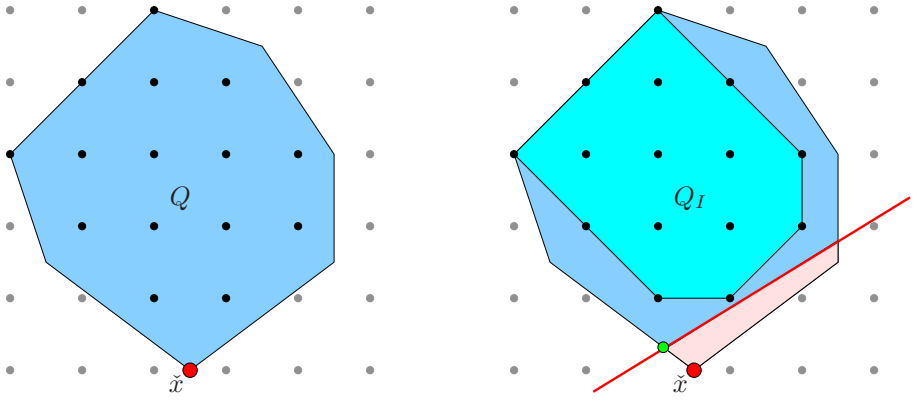

Figure 2.3. A cutting plane that separates the fractional LP solution $\check{x}$ from the convex hull Q_I of integer points of Q.

in data structures is possible since the node selection of Step 3 is performed in a depth-first fashion and conflict clauses (see Chapter 11) are generated for infeasible subproblems that implicitly lead the search to the opposite fixing of the branching variable after backtracking has been performed.

As SAT has no objective function, there is no need for the bounding Step 5 of Algorithm 2.1. A SAT solver can immediately abort after having found the first feasible solution.

2.2 Cutting Planes

Besides splitting the current subproblem Q into two or more easier subproblems by branching, one can also try to tighten the subproblem's relaxation in order to rule out the current solution $\check{x}$ and to obtain a different one. Since MIP is the only of the three investigated problem classes that features a generally applicable useful relaxation, this technique is in this form unique to MIP.

The LP relaxation can be tightened by introducing additional linear constraints $a^T x \leq b$ that are violated by the current LP solution $\check{x}$ but do not cut off feasible solutions from Q (see Figure 2.3). Thus, the current solution $\check{x}$ is *separated* from the convex hull of integer solutions Q_I by the *cutting plane* $a^T x \leq b$, i.e.,

$$\check{x} \notin \{x \in \mathbb{R} \mid a^T x \leq b\} \supseteq Q_I.$$

Gomory presented a general algorithm [102, 103] to find such cutting planes for integer programs. He also proved [104] that his algorithm is finite for integer programs with rational data, i.e., an optimal IP solution is found after adding a finite number of cutting planes. His algorithm, however, is not practicable since it usually adds an exponential number of cutting planes, which dramatically decreases the performance and the numerical stability of the LP solver.

To benefit from the stronger relaxations obtained by cutting planes without hampering the solvability of the LP relaxations, today's most successful MIP solvers combine branching and cutting plane separation in one of the following fashions:

Cut-and-branch. The LP relaxation R_{LP} of the initial (root) problem R is strengthened by cutting planes as long as it seems to be reasonable and does not reduce numerical stability too much. Afterwards, the problem is solved with branch-and-bound.

Branch-and-cut. The problem is solved with branch-and-bound, but the LP relaxations Q_{LP} of *all* subproblems Q (including the initial problem R) might be strengthened by cutting planes. Here one has to distinguish between globally valid cuts and cuts that are only valid in a local part of the branch-and-bound search tree, i.e., cuts that were deduced by taking the branching decisions into account. Globally valid cuts can be used for all subproblems during the course of the algorithm, but local cuts have to be removed from the LP relaxation after the search leaves the subtree for which they are valid.

Marchand et al. [154] and Fügenschuh and Martin [90] give an overview of computationally useful cutting plane techniques. A recent survey of cutting plane literature can be found in Klar [132]. For further details, we refer to Chapter 8 and the references therein.

2.3 Domain Propagation

Constraint propagation is an integral part of every constraint programming solver. The task is to analyze the set of constraints of the current subproblem and the current domains of the variables in order to infer additional valid constraints and domain reductions, thereby restricting the search space. The special case where only the domains of the variables are affected by the propagation process is called *domain propagation*. If the propagation only tightens the lower and upper bounds of the domains without introducing holes it is called *bound propagation* or *bound strengthening*.

In mixed integer programming, the concept of bound propagation is well-known under the term *node preprocessing*. One usually applies a restricted version of the preprocessing algorithm that is used before starting the branch-and-bound process to simplify the problem instance (see, e.g., Savelsbergh [199] or Fügenschuh and Martin [90]). Besides the integrality restrictions, only linear constraints appear in mixed integer programming problems. Therefore, MIP solvers only employ a very limited number of propagation algorithms, the most prominent being the bound strengthening on individual linear constraints (see Section 7.1).

In contrast, a constraint programming model can include a large variety of constraint classes with different semantics and structure. Thus, a CP solver provides specialized constraint propagation algorithms for every single constraint class. Figure 2.4 shows a particular propagation of the ALLDIFF constraint, which demands that the involved variables have to take pairwise different values. Fast domain propagation algorithms for ALLDIFF constraints include the computation of maximal matchings in bipartite graphs (see Régin [192]). Bound propagation algorithms identify so-called *Hall intervals* (Puget [189], López-Ortis et al. [151]).

The following example illustrates constraint propagation and domain propagation for clauses of the satisfiability problem (see Section 1.2).

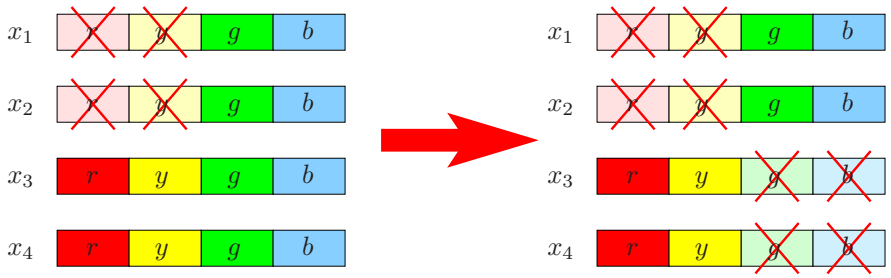

Figure 2.4. Domain propagation on an ALLDIFF constraint. In the current subproblem on the left hand side, the values *red* and *yellow* are not available for variables x_1 and x_2 (for example, due to branching). The propagation algorithm detects that the values *green* and *blue* can be ruled out for the variables x_3 and x_4.

Example 2.1 (SAT constraint propagation). Consider the clauses $\mathcal{C}_1 = x \vee y \vee v$ and $\mathcal{C}_2 = \bar{x} \vee y \vee w$ with binary variables $x, y, v, w \in \{0, 1\}$. The following *resolution* can be performed:

$$\begin{array}{lccccccc} \mathcal{C}_1: & x & \vee & y & \vee & v & & \\ \mathcal{C}_2: & \bar{x} & \vee & y & & & \vee & w \\ \hline \mathcal{C}_3: & & & y & \vee & v & \vee & w \end{array}$$

The resolution process yields a valid clause, namely $\mathcal{C}_3 = y \vee v \vee w$, which can be added to the problem. Thus, resolution is a form of *constraint propagation*.

As a second example, suppose that we branched on $v = 0$ and $w = 0$ to obtain the current subproblem. Looking at $\mathcal{C}_3$, we can deduce $y = 1$ because the constraint would become unsatisfiable with $y = 0$. This latter constraint propagation is a *domain propagation*, because the deduced constraint $y = 1$ directly restricts the domain of a variable. In fact, since the lower bound of y is raised to 1, it is actually a *bound propagation*. In the nomenclature of SAT solving, clause $\mathcal{C}_3$ with $v = w = 0$ is called a *unit clause* and the bound propagation process that fixes the remaining literal y is called *Boolean constraint propagation* (BCP).

As Example 2.1 shows, one propagation can trigger additional propagations. In the example, the generation of the inferred constraint $\mathcal{C}_3$ lead to the subsequent fixing of $y = 1$. Such chains of iteratively applied propagations happen very frequently in constraint propagation algorithms. Therefore, a constraint propagation framework has to provide infrastructure that allows a fast detection of problem parts that have to be inspected again for propagation. For example, the current state-of-the-art to implement BCP for SAT is to apply the so-called *two watched literals* scheme (Moskewicz et al. [168]) where only two of the unfixed literals in a clause need to be watched for changes of their current domains. SCIP uses an event system to reactivate constraints for propagation (see Section 3.1.10). The constraint handler of SCIP that treats SAT clauses implements the *two watched literals* scheme by tracking the bound change events on two unfixed literals per clause, see Section 7.4.

The ultimate goal of a constraint propagation scheme is to decide the *global consistency* of the problem instance at hand.

Definition 2.2 (global consistency). A constraint satisfaction problem CSP = $(\mathfrak{C}, \mathfrak{D})$ with constraint set $\mathfrak{C}$ and domains of variables $\mathfrak{D}$ (see Definition 1.1) is called

globally consistent if there exists a solution $x^\star \in \mathfrak{D}$ with $\mathfrak{C}(x^\star)$.

Since CSP is $\mathcal{NP}$-complete, it is unlikely that efficient propagation schemes exist that decide global consistency. Therefore, the iterative application of constraint propagation usually aims to achieve some form of *local consistency*, which is a weaker form of global consistency: a locally consistent CSP does not need to be globally consistent, but global consistency implies local consistency. In the following we present only some basic notions of local consistency that will be used in this thesis. A more thorough overview can be found in Apt [17].

Definition 2.3 (node consistency). Consider a constraint satisfaction problem $\mathrm{CSP} = (\mathfrak{C}, \mathfrak{D})$ with constraint set $\mathfrak{C}$ and domains of variables $\mathfrak{D}$ (see Definition 1.1). A unary constraint $\mathcal{C} \in \mathfrak{C}$ on a variable x_j

$$\mathcal{C} : \mathcal{D}_j \to \{0, 1\}$$

is called *node consistent* if $\mathcal{C}(x_j) = 1$ for all values $x_j \in \mathcal{D}_j$. A CSP is called *node consistent* if all of its unary constraints are node consistent.

Definition 2.4 (arc consistency). A binary constraint $\mathcal{C} \in \mathfrak{C}$ on variables x_i and x_j, $i \neq j$,

$$\mathcal{C} : \mathcal{D}_i \times \mathcal{D}_j \to \{0, 1\},$$

of a constraint satisfaction problem $\mathrm{CSP} = (\mathfrak{C}, \mathfrak{D})$ is called *arc consistent* if

$$\begin{aligned} &\forall x_i \in \mathcal{D}_i \; \exists x_j \in \mathcal{D}_j : \mathcal{C}(x_i, x_j) = 1, \text{ and} \\ &\forall x_j \in \mathcal{D}_j \; \exists x_i \in \mathcal{D}_i : \mathcal{C}(x_i, x_j) = 1. \end{aligned}$$

A CSP is called *arc consistent* if all of its binary constraints are arc consistent.

Definition 2.5 (hyper-arc consistency). An arbitrary constraint $\mathcal{C} \in \mathfrak{C}$ on variables $x_{j_1}, \ldots, x_{j_k}$,

$$\mathcal{C} : \mathcal{D}_{j_1} \times \ldots \times \mathcal{D}_{j_k} \to \{0, 1\},$$

of a constraint satisfaction problem $\mathrm{CSP} = (\mathfrak{C}, \mathfrak{D})$ is called *hyper-arc consistent* if

$$\forall i \in \{1, \ldots, k\} \, \forall x_{j_i} \in \mathcal{D}_{j_i} \, \exists x^\star \in \mathcal{D}_{j_1} \times \ldots \times \mathcal{D}_{j_k} : x^\star_{j_i} = x_{j_i} \wedge \mathcal{C}(x^\star) = 1.$$

A CSP is called *hyper-arc consistent* if all of its constraints are hyper-arc consistent.

Hyper-arc consistency is the strongest possible local consistency notion with respect to a single constraint. It demands that for each constraint $\mathcal{C}$ each value in the involved domains participates in a solution which satisfies $\mathcal{C}$. In other words, no further values can be excluded from the domains of the variables by considering the constraints individually.

An algorithm which aims to achieve hyper-arc consistency for a given constraint has to remove all values from the domains of the involved variables that do not take part in a solution for the constraint. For domains $\mathcal{D} \subseteq \mathbb{R}$, one usually has to introduce holes in order to achieve hyper-arc consistency. Such algorithms can be very time-consuming. Additionally, the LP relaxation of a constraint integer program can only deal with continuous intervals without holes. Therefore, we will usually regard one of the following relaxed versions of hyper-arc consistency, which only deal with the bounds of the interval domains:

Definition 2.6 (interval consistency). An arbitrary constraint $\mathcal{C} \in \mathfrak{C}$ on variables $x_{j_1}, \ldots, x_{j_k}$,

$$\mathcal{C} : \mathcal{D}_{j_1} \times \ldots \times \mathcal{D}_{j_k} \to \{0, 1\},$$

of a constraint satisfaction problem CSP $= (\mathfrak{C}, \mathfrak{D})$ with interval domains $\mathcal{D}_{j_i} = [l_{j_i}, u_{j_i}]$, $l_{j_i}, u_{j_i} \in \mathbb{R}$, or $\mathcal{D}_{j_i} = \{l_{j_i}, \ldots, u_{j_i}\}$, $l_{j_i}, u_{j_i} \in \mathbb{Z}$, $i = 1, \ldots, k$, is called *interval consistent* if

$$\forall i \in \{1, \ldots, k\} \, \forall x_{j_i} \in \{l_{j_i}, u_{j_i}\} \, \exists x^\star \in \mathcal{D}_{j_1} \times \ldots \times \mathcal{D}_{j_k} : x^\star_{j_i} = x_{j_i} \wedge \mathcal{C}(x^\star) = 1.$$

A CSP with interval domains is called *interval consistent* if all of its constraints are interval consistent.

Definition 2.7 (bound consistency). Let $\mathcal{C} \in \mathfrak{C}$,

$$\mathcal{C} : \mathcal{D}^r_{j_1} \times \ldots \times \mathcal{D}^r_{j_k} \to \{0, 1\},$$

be a constraint defined on real-valued variables $x_{j_1}, \ldots, x_{j_k}$, $\mathcal{D}^r_{j_i} = [l_{j_i}, u_{j_i}]$, $l_{j_i}, u_{j_i} \in \mathbb{R}$, $i = 1, \ldots, k$, which is part of a constraint satisfaction problem CSP $= (\mathfrak{C}, \mathfrak{D})$ with interval domains $\mathcal{D}_{j_i} = [l_{j_i}, u_{j_i}]$, $l_{j_i}, u_{j_i} \in \mathbb{R}$, or $\mathcal{D}_{j_i} = \{l_{j_i}, \ldots, u_{j_i}\}$, $l_{j_i}, u_{j_i} \in \mathbb{Z}$, $i = 1, \ldots, k$. Then, $\mathcal{C}$ is called *bound consistent* if

$$\forall i \in \{1, \ldots, k\} \, \forall x_{j_i} \in \{l_{j_i}, u_{j_i}\} \, \exists x^\star \in \mathcal{D}^r_{j_1} \times \ldots \times \mathcal{D}^r_{j_k} : x^\star_{j_i} = x_{j_i} \wedge \mathcal{C}(x^\star) = 1.$$

A CSP with constraints defined on real-valued variables and variables with interval domains is called *bound consistent* if all of its constraints are bound consistent.

Note that bound consistency is weaker than interval consistency: every interval consistent CSP in which the constraints are defined on real-valued variables is bound consistent. On the other hand, there are bound consistent CSPs that are not interval consistent, as the following example illustrates:

Example 2.8. Let $\mathcal{C} : [0, 1] \times [0, 1] \times [0, 1] \to \{0, 1\}$ be the linear constraint

$$\mathcal{C}(x) = 1 \quad \Leftrightarrow \quad 2x_1 + 2x_2 + 2x_3 = 3.$$

Now consider the constraint satisfaction problem CSP $= (\mathfrak{C}, \mathfrak{D})$ with $\mathfrak{C} = \{\mathcal{C}\}$, $\mathfrak{D} = \mathcal{D}_1 \times \mathcal{D}_2 \times \mathcal{D}_3$, and integer domains $\mathcal{D}_1 = \mathcal{D}_2 = \mathcal{D}_3 = \{0, 1\}$. This CSP is bound consistent: the vector $x^{l_1} = (0, 1, 0.5)$ supports the lower bound of x_1 while $x^{u_1} = (1, 0, 0.5)$ supports the upper bound of x_1, and similar support vectors can be constructed for the bounds of x_2 and x_3. These vectors are feasible solutions to the real-valued constraint $\mathcal{C}(x)$, although they are not feasible solutions to the CSP due to the fractionality of one of their components. On the other hand, the CSP is not interval consistent, since there are no vectors $x^\star \in \mathfrak{D}$ that support the bounds of the variables.

Chapter 3

SCIP as a CIP Framework

This chapter describes the constraint integer programming framework SCIP (an acronym for "Solving Constraint Integer Programs"). SCIP is being developed at the Konrad-Zuse-Zentrum für Informationstechnik Berlin (ZIB) since 2001. It is the successor of the mixed integer programming solver SIP of Alexander Martin [159] and adopts several ideas and algorithms of its predecessor. Nevertheless, it was implemented from scratch in order to obtain a much more flexible design, which is capable of supporting constraint integer programming techniques and a wide variety of user plugins that can be included via a callback mechanism.

Section 3.1 describes the various types of user plugins that can enrich the basic CIP framework and explains their role in the solving process. The algorithmic design and the main sequence of the solving steps are illustrated in Section 3.2. Finally, Section 3.3 covers the infrastructure which is supplied by SCIP in order to provide data structures and efficient methods to represent and access the problem data and to allow interaction and sharing of information between the plugins.

3.1 Basic Concepts of SCIP

SCIP is a constraint integer programming framework that provides the infrastructure to implement very flexible branch-and-bound based search algorithms. In addition, it includes a large library of default algorithms to control the search. These main algorithms of SCIP are part of external *plugins*, which are user defined callback objects that interact with the framework through a very detailed interface. The current distribution of SCIP contains the necessary plugins to solve MIPs (see Part II). In the following, we describe the different plugin types and their role in solving a CIP.

3.1.1 Constraint Handlers

Since a CIP consists of constraints, the central objects of SCIP are the *constraint handlers*. Each constraint handler represents the semantics of a single class of constraints and provides algorithms to handle constraints of the corresponding type.

The primary task of a constraint handler is to check a given solution for feasibility with respect to all constraints of its type existing in the problem instance. This feasibility test suffices to turn SCIP into an algorithm which correctly solves CIPs with constraints of the supported type, at least if no continuous variables are involved. However, the resulting procedure would be a complete enumeration of all potential solutions, because no additional information about the problem structure would be available.

To improve the performance of the solving process constraint handlers may provide additional algorithms and information about their constraints to the framework, namely

- ▷ presolving methods to simplify the problem's representation,
- ▷ propagation methods to tighten the variables' domains,
- ▷ a linear relaxation, which can be generated in advance or on the fly, that strengthens the LP relaxation of the problem, and
- ▷ branching decisions to split the problem into smaller subproblems, using structural knowledge of the constraints in order to generate a well-balanced branching tree.

Example 3.1 (knapsack constraint handler). A binary knapsack constraint is a specialization of a linear constraint

$$a^T x \leq \bar{\beta} \tag{3.1}$$

with non-negative integral right hand side $\bar{\beta} \in \mathbb{Z}_{\geq 0}$, non-negative integral coefficients $a_j \in \mathbb{Z}_{\geq 0}$, and binary variables $x_j \in \{0,1\}$, $j \in N$.

The feasibility test of the knapsack constraint handler is very simple: it only adds up the coefficients a_j of variables x_j set to 1 in the given solution and compares the result with the right hand side $\bar{\beta}$. Presolving algorithms for knapsack constraints include modifying the coefficients and right hand side in order to tighten the LP relaxation, and fixing variables with $a_j > \bar{\beta}$ to 0, see Savelsbergh [199] and Section 10.2.

The propagation method fixes additional variables to 0, that would not fit into the knapsack together with the variables that are already fixed to 1 in the current subproblem.

The linear relaxation of the knapsack constraint initially consists of the knapsack inequality (3.1) itself. Additional cutting planes like lifted cover cuts (see, for example, Balas [28], Balas and Zemel [35] or Martin and Weismantel [160]) or GUB cover cuts (see Wolsey [217]) are dynamically generated to enrich the knapsack's relaxation and to cut off the current LP solution; see also Section 8.1.

Example 3.2 (NOSUBTOUR constraint handler). The symmetric traveling salesman problem (TSP) on a graph $G = (V, E)$ with edge lengths $c_{uv} \in \mathbb{R}_{\geq 0}$, $uv \in E$, can be stated as a constraint integer program in the following way:

$$\begin{aligned}
\min \quad & \sum_{uv \in E} c_{uv}\, x_{uv} & & \\
\text{s.t.} \quad & \sum_{u \in \delta(v)} x_{uv} = 2 & \text{for all } v \in V & \quad (3.2)\\
& \text{NOSUBTOUR}(G, x) & & \quad (3.3)\\
& x_{uv} \in \{0,1\} & \text{for all } uv \in E & \quad (3.4)
\end{aligned}$$

Formally, this model consists of $|V|$ degree constraints (3.2), one NOSUBTOUR constraint (3.3), and $|E|$ integrality constraints (3.4). The NOSUBTOUR constraint is a non-linear constraint which is defined as

$$\text{NOSUBTOUR}(G, x) \;\Leftrightarrow\; \nexists C \subseteq \{uv \in E \mid x_{uv} = 1\} : C \text{ is a cycle of length } |C| < |V|.$$

This constraint must be supported by a constraint handler, which for a given integral solution $x \in \{0,1\}^E$ checks whether the corresponding set of edges contains a subtour

C. The linear relaxation of the NOSUBTOUR constraint consists of exponentially many *subtour elimination inequalities*

$$\sum_{uv \in E(S)} x_{uv} \leq |S| - 1 \text{ for all } S \subset V \text{ with } 2 \leq |S| \leq |V| - 2,$$

which can be separated and added on demand to the LP relaxation. Additionally, the constraint handler could separate various other classes of valid inequalities for the traveling salesman problem that can be found in the literature, see, for example, Grötschel and Padberg [108, 109], Grötschel and Holland [107], Clochard and Naddef [62], Applegate et al. [14, 15, 16], or Naddef [169].

3.1.2 Presolvers

In addition to the constraint based (primal) presolving mechanisms provided by the individual constraint handlers, additional presolving algorithms can be applied with the help of *presolvers*, which interact with the whole set of constraints. They may, for example, perform dual presolving reductions which take the objective function into account.

For instance, if the value of a variable x_j can always be decreased without rendering any constraint infeasible (an information, the constraint handlers have to provide by setting variable locks, see Section 3.3.3), and if the objective value c_j of the variable is non-negative, the *dual fixing presolver* fixes the variable to its lower bound, see Section 10.8. In the setting of an MIP with inequality system $Ax \leq b$, this condition is satisfied if and only if $A_{\cdot j} \geq 0$ and $c_j \geq 0$.[1]

3.1.3 Cut Separators

In SCIP, we distinguish between two different types of cutting planes. The first type are the constraint based cutting planes, that are valid inequalities or even facets of the polyhedron described by a single constraint or a subset of the constraints of a single constraint class. These cutting planes may also be strengthened by lifting procedures that take information about the full problem into account, for example the implication graph, see Section 3.3.5. They are generated by the constraint handlers of the corresponding constraint types. Prominent examples are the different types of knapsack cuts that are generated in the knapsack constraint handler, see Example 3.1, or the cuts for the traveling salesman problem like subtour elimination and comb inequalities which can be separated by the NOSUBTOUR constraint handler, see Example 3.2.

The second type of cutting planes are general purpose cuts, which are using the current LP relaxation and the integrality conditions to generate valid inequalities. Generating those cuts is the task of *cut separators*. Examples are Gomory fractional and Gomory mixed integer cuts (Gomory [104]), complemented mixed integer rounding cuts (Marchand and Wolsey [155]), and strong Chvátal-Gomory cuts (Letchford and Lodi [142]).

3.1.4 Domain Propagators

Like for cutting planes, there are two different types of domain propagations. Constraint based (primal) domain propagation algorithms are part of the corresponding

[1] Here, $A_{\cdot j}$ is the j'th column of the coefficient matrix A.

constraint handlers. For example, the ALLDIFF[2] constraint handler excludes certain values of the variables' domains with the help of a bipartite matching algorithm, see Regin [192], or applies bound propagation by so-called Hall intervals, see Leconte [140], Puget [189], or López-Ortiz et al. [151].

In contrast, *domain propagators* provide dual propagations, i.e., propagations that can be applied due to the objective function and the currently best known primal solution. An example is the simple objective function propagator of Section 7.6 that tightens the variables' domains with respect to the objective bound

$$c^T x < \hat{c}$$

with $\hat{c}$ being the objective value of the currently best primal solution.

3.1.5 Variable Pricers

Several optimization problems are modeled with a huge number of variables, e.g., with each path in a graph or each subset of a given set corresponding to a single variable. In this case, the full set of variables cannot be generated in advance. Instead, the variables are added dynamically to the problem whenever they may improve the current solution. In mixed integer programming, this technique is called *column generation*.

SCIP supports dynamic variable creation by *variable pricers*. They are called during the subproblem processing and have to generate additional variables that reduce the lower bound of the subproblem. If they operate on the LP relaxation, they would usually calculate the reduced costs of the not yet existing variables with a problem specific algorithm and add some or all of the variables with negative reduced costs. Note that since variable pricers are part of the model, they are always problem class specific. Therefore, SCIP does not contain any "default" variable pricers.

3.1.6 Branching Rules

If the LP solution of the current subproblem is fractional, the integrality constraint handler calls the *branching rules* to split the problems into subproblems. Additionally, branching rules are called as a last resort on integral solutions that violate one or more constraints for which the associated constraint handlers were not able to resolve the infeasibility in a more sophisticated way, see Section 3.2.8.

Usually, a branching rule creates two subproblems by splitting a single variable's domain. If applied on a fractional LP solution, commonly an integer variable x_j with fractional value $\check{x}_j$ is selected, and the two branches $x_j \leq \lfloor \check{x}_j \rfloor$ and $x_j \geq \lceil \check{x}_j \rceil$ are created. The well-known *most infeasible*, *pseudocost*, *reliability*, and *strong branching* rules are examples of this type (see Achterberg, Martin, and Koch [5] and Chapter 5). It is also possible to implement much more general branching schemes, for example by creating more than two subproblems, or by adding additional constraints to the subproblems instead of tightening the domains of the variables.

3.1.7 Node Selectors

Node selectors decide which of the leaves in the current branching tree is selected as next subproblem to be processed. This choice can have a large impact on the

[2] ALLDIFF$(x_1, \ldots, x_k)$ requires the integer variables $x_1, \ldots, x_k$ to take pairwise different values.

solver's performance, because it influences the finding of feasible solutions and the development of the global dual bound.

Constraint programming was originally developed for *constraint satisfaction problems* (CSPs). In this setting, the solver only has to check whether there is a feasible solution or not. Therefore, many of the available CP solvers employ *depth first search*. The same holds for the satisfiability problem. SAT solvers are even more tailored to depth first search, since one of their key components—conflict analysis (see Chapter 11)—is best suited for the use inside a depth first search algorithm. A more extensive discussion of this topic can be found in Section 16.2.

With the addition of an objective function, depth first search is usually an inferior strategy. It tends to evaluate many nodes in the tree that could have been discarded if a good or optimal solution were known earlier. In mixed integer programming, several node selection strategies are known that try to discover good feasible solutions early during the search process. Examples of those strategies are *best first* and *best estimate* search. See Chapter 6 for a comparison of node selection strategies for MIP.

3.1.8 Primal Heuristics

Feasible solutions can be found in two different ways during the traversal of the branching tree. On the one hand, the solution of a node's relaxation may be feasible w.r.t. the constraints. On the other hand, feasible solutions can be discovered by *primal heuristics*. They are called periodically during the search.

SCIP provides specific infrastructure for diving and probing heuristics. *Diving heuristics* iteratively resolve the LP after making a few changes to the current subproblem, usually aiming at driving the fractional values of integer variables to integrality. *Probing heuristics* are even more sophisticated. Besides solving LP relaxations, they may call the domain propagation algorithms of the constraint handlers after applying changes to the variables' domains, and they can undo these changes using backtracking.

Other heuristics without special support in SCIP include local search heuristics like *tabu search* [97], *rounding heuristics* which try to round the current fractional LP solution to a feasible integral solution, and *improvement heuristics* like *local branching* [85] or RINS [72], which try to generate improved solutions by inspecting one or more of the feasible solutions that have already been found. Chapter 9 provides an overview of the heuristics included in SCIP to solve mixed integer programs.

3.1.9 Relaxation Handlers

SCIP provides specific support for LP relaxations: constraint handlers implement callback methods for generating the LP, additional cut separators may be included to further tighten the LP relaxation, and there are a lot of interface methods available to access the LP information at the current subproblem.

In addition, it is also possible to include other relaxations, e.g., Lagrange relaxations or semidefinite relaxations. This is possible through *relaxation handler* plugins. The relaxation handler manages the necessary data structures and calls the relaxation solver to generate dual bounds and primal solution candidates. However, the data to define a single relaxation must either be extracted by the relaxation handler itself (e.g., from the user defined problem data, the LP information, or the

integrality conditions), or be provided by the constraint handlers. In the latter case, the constraint handlers have to be extended to support this specific relaxation.

Like with LP relaxations, support for managing warm start information is available to speed up the resolves at the subproblems. At each subproblem, the user may solve any number of relaxations, including the LP relaxation. In particular, it is possible to refrain from solving any relaxation, in which case SCIP behaves like a CP solver.

3.1.10 Event Handlers

SCIP contains a sophisticated event system, which can be used by external plugins to be informed about certain events. These events are processed by *event handler* plugins. Usually, the event handlers pass the information to other objects, e.g., to a constraint handler. It is very common in SCIP that a constraint handler closely interacts with an event handler in order to improve its own runtime performance.

For example, a constraint handler may want to be informed about the domain changes of the variables involved in its constraints. This can be used to avoid unnecessary work in preprocessing and propagation: a constraint has only to be processed again, if at least one domain of the involved variables was changed since the last preprocessing or propagation call. Events can also be used to update certain internal values (e.g., the total weight of variables currently fixed to 1 in a knapsack constraint) in order to avoid frequent recalculations.

Other potential applications for the event system include a dynamic graphical display of the currently best solution and the online visualization of the branching tree. These are supported by events triggered whenever a new primal solution has been found or a node has been processed.

3.1.11 Conflict Handlers

Current state-of-the-art SAT solvers employ analysis of infeasible subproblems to generate so-called *conflict clauses* (see Marques-Silva and Sakallah [157]). These are implied constraints that may help to prune the branching tree. In the CP community, a generalization of those clauses is known as *no-goods*.

SCIP adopts this mechanism and extends it to the analysis of infeasible LPs, see Chapter 11. Whenever a conflict was found by the internal analysis algorithms, the included *conflict handlers* are called to create a conflict constraint out of the set of conflicting variables. Conflict handlers usually cooperate with constraint handlers by calling the constraint creation method of the constraint handler and adding the constraint to the model.

3.1.12 File Readers

File readers are called to parse an input file and generate a CIP model. They create constraints and variables and activate variable pricers if necessary. Each file reader is hooked to a single file name extension. It is automatically called if the user wants to read in a problem file of corresponding name. Examples of file formats are the *MPS format* [117] and the Cplex *LP format* [118] for linear and mixed integer programs, the *CNF format* for SAT instances in conjunctive normal form, and the *TSP format* [193] for instances of the traveling salesman problem.

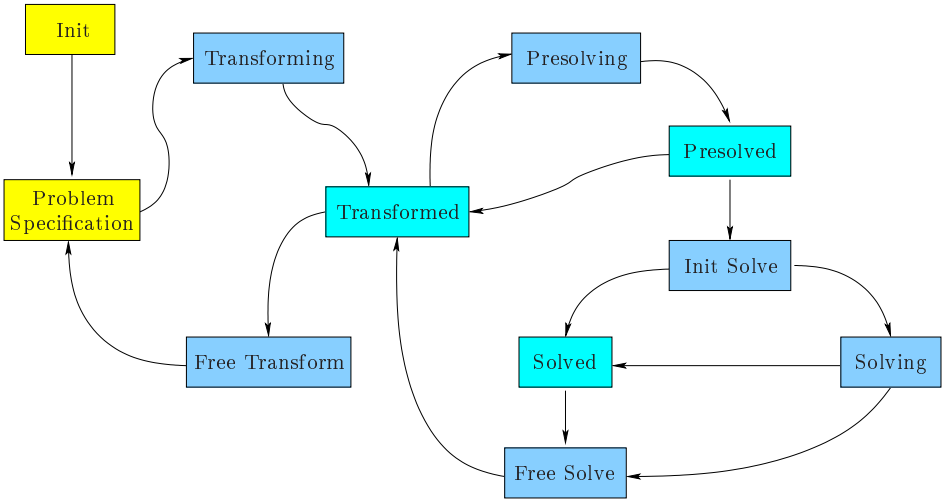

Figure 3.1. Operational stages of SCIP. The arrows represent possible transitions between stages.

3.1.13 Display Columns

While solving a constraint integer program, SCIP displays status information in a column-like fashion. The current number of processed branching tree nodes, the solving time, and the relative gap between primal and dual bound are examples of such *display columns*. There already exists a wide variety of display columns which can be activated or deactivated on demand. Additionally, the user can implement his own display columns in order to track problem or algorithm specific values.

3.1.14 Dialog Handlers

SCIP comes with a command line shell which allows the user to read in problem instances, modify the solver's parameters, initiate the optimization, and display certain statistics and solution information. This shell can be extended by *dialog handlers*. They are linked to the shell's calling tree and executed whenever the user enters the respective command. The default shell itself is also generated by dialog handlers and is therefore completely adjustable to the needs of the software developer.

3.1.15 Message Handlers

All screen output of SCIP is passed through a message handler. By overwriting the appropriate callback methods, the user can easily redirect or suppress the output.

3.2 Algorithmic Design

Figure 3.1 shows a flow chart of the main *operational stages* that are traversed during the execution of SCIP. In this section we specify which callback methods of the different plugins are executed and which operations the user may perform during the different stages. It is explained how the problem is represented in SCIP's data

structures and which transformations are being applied during the course of the algorithm. Compare also the data flow illustrated in Figure 3.5 on page 37.

3.2.1 Init Stage

In the *init stage*, the basic data structures are allocated and initialized. The user has to include the required plugins with calls to the `SCIPinclude...()` methods. Each included plugin may allocate its own private data. With a call to `SCIPcreateProb()` or `SCIPreadProb()`, the solver leaves the *init stage* and enters the *problem specification stage*, the latter one executing a file reader to create the problem instance.

3.2.2 Problem Specification Stage

During the *problem specification stage*, the user can define and modify the original problem instance that he wants to solve. He can create constraints and variables and activate included variable pricers. A file reader that is called during the *init stage* switches to the *problem specification stage* with a call to `SCIPcreateProb()` and subsequently creates the necessary problem data.

3.2.3 Transforming Stage

Before the actual solving process begins, SCIP creates a working copy of the original problem instance. The working copy is called the *transformed problem* and protects the original problem instance from modifications applied during presolving or solving. The original problem can only be modified in the *problem specification stage*.

In the *transforming stage*, the data of variables and constraints are copied into a separate memory area. Because SCIP does not know how the constraints are represented, it has to call the constraint handlers to create copies of their constraints.

3.2.4 Transformed Stage

After the copying process of the *transforming stage* is completed, the *transformed stage* is reached. This state is only an intermediate state, from which the user may initiate the *presolving stage* or free the solving process data by switching into the *free transform stage*.

3.2.5 Presolving Stage

In the *presolving stage*, permanent problem modifications on the transformed problem are applied by the presolvers and the presolving methods of the constraint handlers. These plugins are called iteratively until no more reductions can be found or until a specified limit is reached.

One of the main tasks of presolving is to detect fixings and aggregations of variables, which are stored in the variable aggregation graph, see Section 3.3.4. Fixed and aggregated variables are deleted from the transformed problem and replaced by their fixed value or their representing active variables, respectively.

Constraint handlers may also upgrade their constraints to a more specific constraint type. For example, as explained in Section 10.1, the linear constraint handler

provides an upgrading mechanism for its constraints

$$\underline{\beta} \leq a^T x \leq \overline{\beta}.$$

Other constraint handlers can be hooked into this mechanism to be called for converting linear constraints into constraints of their own type. For example, the knapsack constraint handler (see Example 3.1) checks whether the linear constraint consists of only binary variables, integral weights, and only one finite side $\underline{\beta}$ or $\overline{\beta}$. If the check succeeds, the linear constraint is converted into a knapsack constraint, possibly by negating some of the binary variables or inverting the inequality. Such an upgrading of a constraint into a more specific type has the advantage that the specialized constraint handler can store the constraint data in a more compact form and can employ specialized, more efficient algorithms.

3.2.6 Presolved Stage

Like the *transformed stage*, the *presolved stage* is an intermediate stage, which is reached after the presolving is completed. Thereafter, the actual solving process may be launched. If the presolving already solved the problem instance by detecting infeasibility or unboundness or by fixing all variables, SCIP automatically switches via the *init solve stage* to the *solved stage*.

3.2.7 Init Solve Stage

In the *init solve stage* all necessary data structures for the solving process are set up. For example, the root node of the branching tree is created and the LP solver is initialized. Additionally, the plugins are informed about the beginning of the solving process in order to enable them to create and initialize their private data.

3.2.8 Solving Stage

If the problem was not already solved in the *presolving stage*, the branch-and-bound process is performed in the *solving stage* to implicitly enumerate the potential solutions. This stage contains the main solving loop of SCIP which consists of five different steps that are called successively until the problem is solved or the solving process is interrupted (see Figure 3.2).

Node Selection

The first step of each iteration in the main solving loop is the selection of the next subproblem. The node selector of highest priority (the *active node selector*) is called to select one of the leaves in the branching tree to be processed. It can decide between the current node's children and siblings, and the "best" of the remaining leaves stored in the tree. The ordering relation of the tree's leaves is also defined by the active node selector.

Successively choosing a child or sibling of the current node is called *plunging* or *diving*. Selecting the best leaf of the tree ends the current plunging sequence and starts the next one. During plunging, the setup of the subproblems to be processed is computationally less expensive, since the children and siblings are most likely to be closely related to the current node. Switching to the best leaf of the tree is more

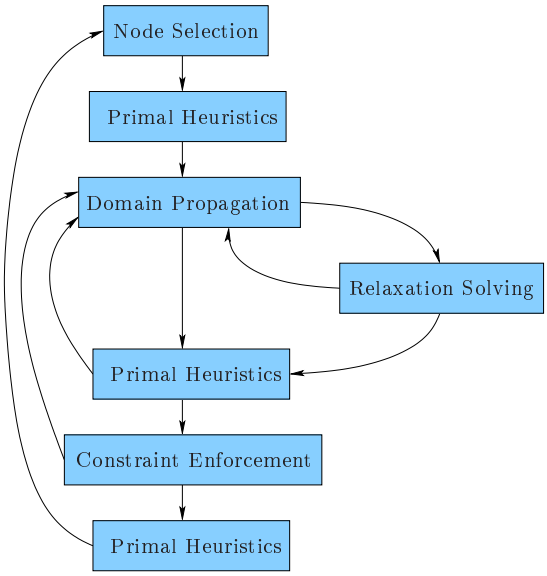

Figure 3.2. Main solving loop of the *solving stage*.

expensive, but has the advantage that the search can be brought to regions in the search space that are more promising to contain feasible solutions of small objective value—at least if the ordering of the leaves corresponds to the subproblems' dual (i.e., lower) bounds. Additionally, it helps to improve the global dual bound more quickly. Efficient node selectors for MIP employ a mixture of plunging and best first search. SAT and CSP solvers usually perform depth first search since these two problems are pure feasibility problems, which do not contain an objective function.

SCIP has two different operation modes: the *standard mode* and the *memory saving mode*. If the memory limit—given as a parameter by the user—is nearly reached, SCIP switches to the *memory saving mode* in which different priorities for the node selectors are applied. Usually, the *depth first search* node selector has highest priority in memory saving mode, since it does not produce as many unprocessed nodes as strategies like *best first search* and tends to reduce the number of open leaves, thereby releasing allocated memory. If the memory consumption decreased sufficiently, SCIP switches back to *standard mode*.

Primal Heuristics

Primal heuristics have different entry points during the solving process. If applicable, a primal heuristic can be called directly after the next subproblem to be processed is selected. This is particularly useful for heuristics that do not need to access the LP solution of the current node. If such a heuristic finds a feasible solution, the leaves of the branching tree exceeding the new primal bound are pruned. It may happen that even the current node can be cut off without solving the LP relaxation. Very fast heuristics that require an LP solution can also be called during the "Relaxation Solving" loop, see below. Most heuristics, however, are called either after the LP relaxation was solved or after the node has been completely processed, which means that the node was either cut off or a branching was applied.

Like most plugins in SCIP, primal heuristics do not need to be executed at every

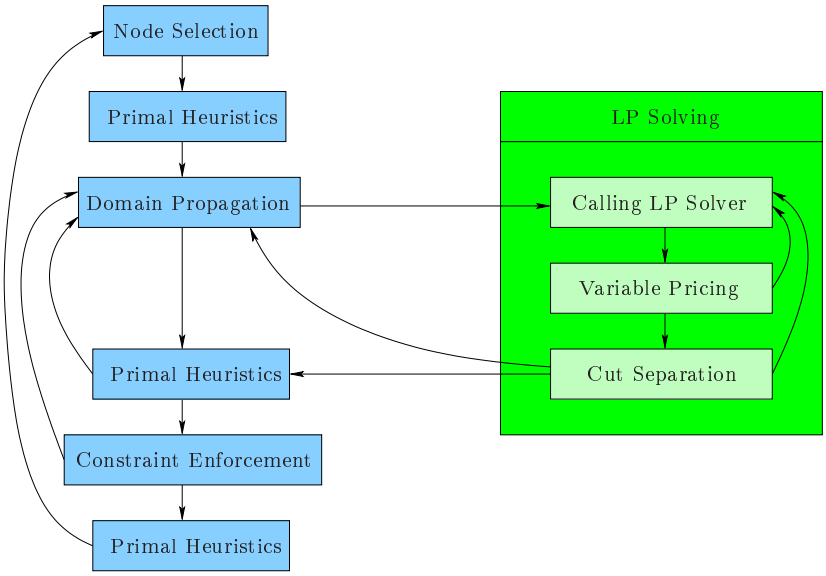

Figure 3.3. Main solving loop of the *solving stage* with detailed LP solving loop.

single node. They are usually called with a certain frequency, i.e., at specific depth levels in the branching tree, with the more expensive heuristics being called less often.

Domain Propagation

After a node is selected to be processed, the corresponding subproblem is set up, and the applicable primal heuristics have been called, the domain propagators and the domain propagation methods of the constraint handlers are called to tighten the variables' local domains. This propagation is applied iteratively until no more reductions are found or a propagation limit set by the user is reached. Domain propagation does not have to be applied at every node. Every constraint handler and domain propagator can decide whether it wants to invest the effort of trying to tighten the variables' domains.

Relaxation Solving

The next step of the solving loop is to solve the subproblem's relaxations, in particular the LP relaxation. Like domain propagation, the solving of relaxations can be skipped or applied as needed. If there are active variable pricers, however, the LP relaxation has to be solved in order to generate new variables and to obtain a feasible dual bound.

The LP solving consists of an inner loop as can be seen in Figure 3.3. It is executed as long as changes to the LP have been applied in the variable pricing or cut separation steps.

Calling LP Solver. The first step is to call the LP solver to solve the initial LP relaxation of the subproblem. In the root node, this is defined by the relaxations of constraints that are marked to be *initial*: the constraint handlers are asked to

enrich the LP with rows that correspond to their *initial* constraints before the first LP is solved. The initial LP relaxation of a subsequent node equals its parent's relaxation modified by the additional bound changes of the node. Note that branching on constraints does affect the LP relaxation of the child nodes directly only if the branching constraints are marked to be initial. Otherwise, the branching only modifies the CIP subproblem, and the corresponding constraint handlers may then tighten the LP in their cut separation or constraint enforcement methods.

After an LP basis is loaded from the warm start information stored in the branching tree into the LP solver, the LP is solved with the primal or dual simplex algorithm, depending on the feasibility status of the current basis. It is also possible to use an interior point method like the barrier algorithm to solve the LP relaxations, if such an algorithm is available. Note, however, that current interior point methods are not able to exploit warm start information. Therefore, they are usually inferior to simplex solvers for processing the relaxations of the subproblems. Nevertheless, for some problem classes it can be beneficial to use the barrier algorithm even for the subproblems. For example, Koch [134] reported performance gains in using the barrier algorithm for some instances of the Steiner tree packing problem.

After the LP solver has solved the relaxation, the resulting LP solution is checked for stability. In a numerically unstable situation, different LP solver parameter settings are tried in order to achieve a stable solution. If this fails, the LP relaxation of the current subproblem is discarded, and the solving process continues as if the LP was not solved at the current node. This is a valuable feature for solving numerically difficult problems. Since SCIP does not need to solve the LP at every node, it can easily leap over numerical troubles in the LP solver without having to abandon the whole solving process.

Variable Pricing. After the initial LP is solved, the variable pricers are called to create new variables and add additional columns to the LP. Variable pricers can be *complete* or *incomplete*. A complete pricer generates at least one new variable if the current LP solution is not optimal in the relaxation of the full variable space. If an incomplete pricer is used, the objective value of the optimal LP solution is not necessarily a dual bound of the subproblem and cannot be used to apply bounding, since there may exist other variables which would further reduce the LP value.

The pricing is performed in rounds. In each round, several new variables are created with their associated LP columns stored in a pricing storage, see Section 3.3.9. After each pricing round, some or all of the columns in the pricing store are added to the LP, and the LP solver is called again to resolve the relaxation. Note that the primal simplex algorithm can be used to quickly resolve the LP after new columns have been added, since new columns do not affect the primal feasibility of the current basis.

Cut Separation. After the pricing is performed and the LP is resolved, the cut separators and the separation methods of the constraint handlers are called to tighten the LP relaxation with additional cutting planes. In each iteration of the LP solving loop, cutting planes are collected in a separation storage, and only some of them are added to the LP afterwards, see Section 3.3.8. Note that the well-known *reduced cost strengthening* (see Nemhauser and Wolsey [174] and Section 8.8) is implemented as a general purpose cutting plane separator, and does therefore not appear as an explicit step in the algorithm.

Some cutting planes found by the cut separators or constraint handlers might be

simple bound changes, i.e., cuts with only one non-zero coefficient. In particular, the reduced cost strengthening cut separator produces inequalities of this type. Needless to say, such trivial cuts are not added as rows to the LP, but modify the local bounds of the variables directly. If bound changes have been applied, the domain propagation is called again with the hope to tighten even more bounds. If no bound changes, but other cutting planes have been found, the LP is resolved. The dual simplex algorithm can be applied efficiently, since added rows and modified bounds do not affect the dual feasibility of the current basis. If no cutting planes have been generated, the LP solving loop is finished, and the applicable primal heuristics are called.

Constraint Enforcement

After the domain propagation has been applied and the relaxations are solved, the constraint handlers are asked to process one of the relaxations' primal solutions. In MIP, they usually use the solution of the LP relaxation.

In contrast to the constraint handlers' *feasibility tests*, which only check a given primal solution (generated by a primal heuristic) for feasibility, the *enforcement* methods should also try to resolve an infeasibility. The constraint handler has different options of dealing with an infeasibility (see Figure 3.4):

- ▷ reducing a variable's domain to exclude the infeasible solution from the local set of domains,
- ▷ adding an additional valid constraint that can deal appropriately with the infeasible solution,
- ▷ adding a cutting plane to the LP relaxation that cuts off the infeasible solution,
- ▷ creating a branching with the infeasible solution no longer being feasible in the relaxations of the child nodes,
- ▷ concluding that the current subproblem is infeasible as a whole and can be pruned from the branching tree,
- ▷ stating that the solution is infeasible without resolving the infeasibility.

Constraint handlers can also answer that the current solution is feasible for all of its constraints.

The constraint handlers' enforcement methods are called in an order specified by the constraint handlers' enforcement priorities. Depending on the result of each constraint enforcement method, SCIP proceeds differently. If the constraint handler tightened a variable's domain or added a constraint, the enforcement cycle is aborted and the algorithm jumps back to domain propagation. Adding a cutting plane invokes the LP solving again. Branching and pruning the current node finishes the processing of the node after which the primal heuristics are called. If the constraint handler detects the solution to be infeasible without resolving it, or if the solution is feasible for the constraints of the constraint handler, the next constraint handler is asked to process the current solution.

The constraint enforcement cycle can have three different outcomes:

1. A constraint handler has resolved an infeasibility, after which the node processing is continued appropriately.

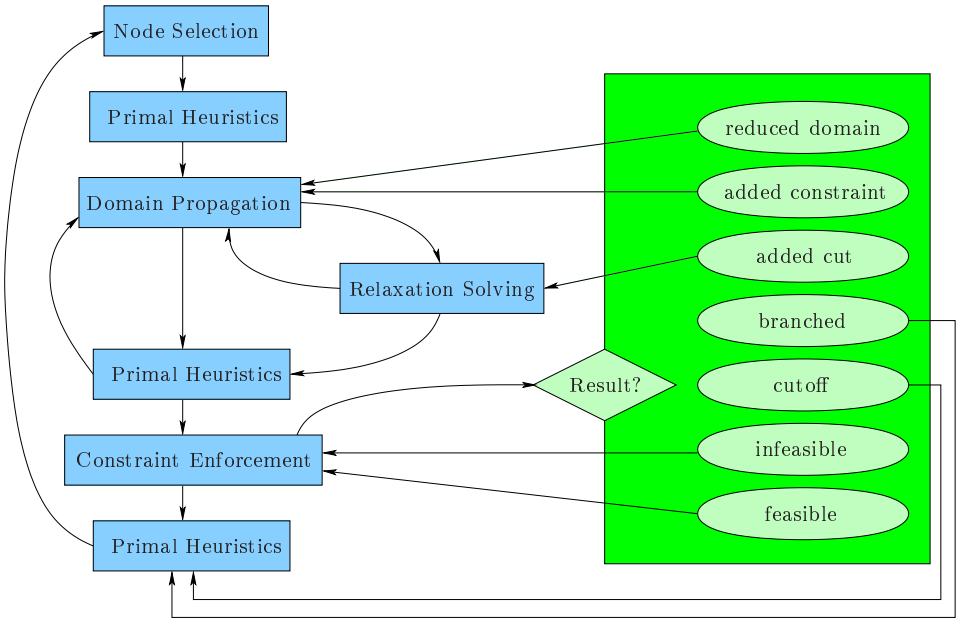

Figure 3.4. Constraint enforcement results. Depending on the enforcement result of a constraint handler, the solving process continues differently.

2. All constraint handlers have declared the solution to be feasible, which means that a new feasible solution has been found.

3. At least one constraint handler has detected an infeasibility, but none of them has resolved it. This is a very undesirable case, since the solution is not feasible but no additional information on the problem structure is available to guide the search. As a last resort, the branching rules are called to create a branching by splitting an integer variable's domain. Ultimately, this leads to a subproblem in which all integer variables are fixed. Due to Definition 1.6 of the constraint integer program, such a subproblem can be solved to optimality by solving the LP relaxation.

Note that the *integrality constraint handler* enforces its constraint by calling the branching rules, if at least one of the integer variables has a fractional value. The *integrality constraint handler* has an enforcement priority of 0, so that constraint handlers may decide whether they want to be called only on integral solutions (in which case they should have a negative priority) or to be also called on fractional solutions (with a positive priority). To be called only on integral solutions can be useful if an efficient feasibility test of the constraint handler can only be applied on integral solutions, e.g., if the solution selects edges in a graph and the feasibility test is some graph algorithm. To be called on fractional solutions can be useful if one wants to apply a constraint specific branching rule. For example, the constraint handler for set partitioning constraints

$$x_1 + \ldots + x_q = 1 \quad \text{with } x_j \in \{0,1\}, \;\; j = 1, \ldots, q$$

may want to apply the so-called *special ordered set branching* (see Beale and Tomlin [38]). This means to branch on a subset of the variable set using the disjunction

$$x_1 = \ldots = x_k = 0 \;\; \vee \;\; x_{k+1} = \ldots = x_q = 0,$$

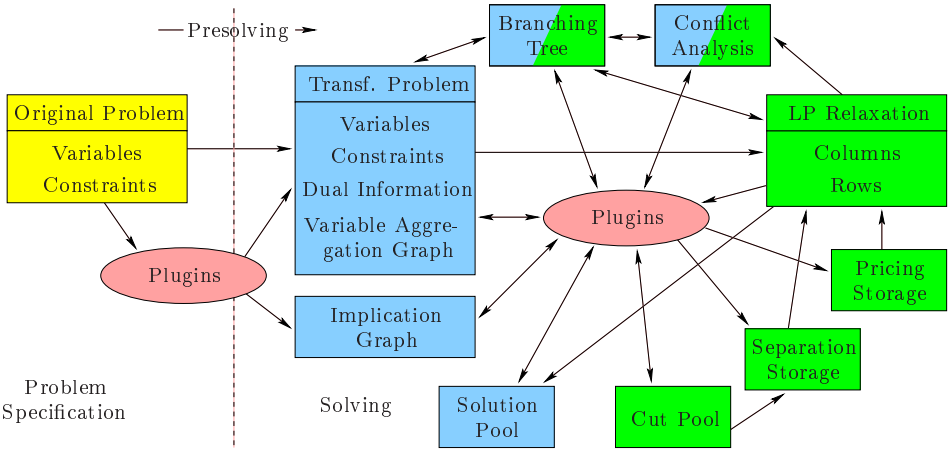

Figure 3.5. Infrastructure provided by SCIP. The arrows denote the data flow between the components.

with $k \in \{1, \ldots, q-1\}$.

3.3 Infrastructure

SCIP provides all necessary infrastructure to implement branch-and-bound based algorithms for solving CIPs. It manages the branching tree along with all subproblem data, automatically updates the LP relaxations, and handles all necessary transformations due to the preprocessing problem modifications. Additionally, a cut pool, pricing and separation storage management, and a SAT-like conflict analysis mechanism (see Chapter 11) are available. SCIP provides an efficient memory allocation shell, which also includes a simple leak detection if compiled in debug mode. Finally, statistical output can be generated to support the diagnosis of the user's algorithms. In particular, the branching tree can be visualized with the help of Sebastian Leipert's VBC Tool [141].

Figure 3.5 gives a rough sketch of the different components of SCIP and how they interact with each other and with the external plugins. The problem information is represented in three different parts of the diagram. Initially, the user states the CIP instance as *original problem*. The constraint handler and presolver plugins generate the *transformed problem*, which is an equivalent but usually more compact and smaller formulation of the problem instance. Both objects are CIP representations of the model consisting of *variables* and general *constraints*. Feasible solutions for the instance—i.e., value assignments for the variables such that all constraints are satisfied—are stored in the *solution pool*. Optionally, an *implication graph* and a *clique table* can be associated to the transformed problem.

The third representation of the problem instance is only a partial representation, namely the LP relaxation. It consists of *columns*, each with lower and upper bound and objective coefficient, and *rows* which are linear inequalities or equations over the columns. The LP relaxation is populated via intermediate storage components, the *pricing storage* and the *separation storage*. Additionally, the *cut pool* can store valid inequalities that can be added on demand to the LP through the separation storage. The *branching tree* and *conflict analysis* components operate on both rep-

resentations, the CIP model of the transformed problem and the LP relaxation. The user *plugins* can access all of the components, although the LP relaxation can only be modified through the pricing and separation storages.

The following sections take a closer look at the different components and describe their role in the solving process and their interaction with the user plugins.

3.3.1 Original Problem

The original problem stores the data of the problem instance as it was entered by the user. It consists of variables and constraints. Each variable x_j has an objective function coefficient $c_j \in \mathbb{R}$, a lower bound $l_j \in \mathbb{R}$ and an upper bound $u_j \in \mathbb{R}$. Additionally, it is known whether the variable is of integer or continuous type. In contrast to this explicit information about the variables, the constraints are just abstract data objects. To define their semantics, the user has to provide external plugins (*constraint handlers*) for each class of constraints. SCIP calls these constraint handlers through a callback interface at different points in the solving process, see Section 3.1.1.

3.3.2 Transformed Problem

The transformed problem is created as a direct copy of the original problem. While the original problem instance is retained in a separate data area, the transformed problem is modified in the presolving and solving steps of SCIP. For example, variables and constraints of the transformed problem can be eliminated or replaced, the domains of variables can be tightened, or the constraint data can be altered. Nevertheless, the transformed problem remains equivalent to the original problem in the sense that the transformed problem is feasible if and only if the original problem is feasible, and that every feasible (optimal) solution of the transformed problem can be converted into a feasible (optimal) solution of the original problem.

3.3.3 Dual Information

One main drawback of the abstract constraint approach of SCIP is the inaccessibility of dual information about the variables. For example, a component like a presolving plugin cannot answer the question in which constraints a variable appears, since the data of the constraints are private to the corresponding constraint handler.

To remedy this situation, SCIP requires the constraint handlers to provide at least a minimum of dual information which is stored in the data structures of the variables. This information consists of the number of *down-locks* and *up-locks* for each variable.

Definition 3.3 (variable locks). Let $\mathcal{C}_i : \mathbb{R}^n \to \{0,1\}$ be a constraint of a constraint integer program CIP $= (\mathfrak{C}, I, c)$. We say that $\mathcal{C}_i$ *down-locks* (*up-locks*) x_j if there exist two vectors $\hat{x}, \dot{x} \in \mathbb{R}^n$ with $\mathcal{C}_i(\hat{x}) = 1$, $\mathcal{C}_i(\dot{x}) = 0$, $\dot{x}_k = \hat{x}_k$ for all $k \neq j$, $\hat{x}_j, \dot{x}_j \in \mathbb{Z}$ if $j \in I$, and $\dot{x}_j < \hat{x}_j$ ($\dot{x}_j > \hat{x}_j$). The number of constraints which down-lock and up-lock variable x_j is denoted by ζ_j^- and ζ_j^+, respectively.

The variable locks ζ_j^- and ζ_j^+ can be interpreted as the number of constraints that "block" the shifting of x_j in direction to its lower or upper bound.

Example 3.4 (variable locks for inequality systems). For a mixed integer program with constraint system $Ax \leq b$, the variable lock numbers are given by the number of negative and positive coefficients per column: $\zeta_j^- = |\{i \mid A_{ij} < 0\}|$ and $\zeta_j^+ = |\{i \mid A_{ij} > 0\}|$.

Example 3.5 (variable locks for general constraints). Consider the constraint integer program

$$\begin{aligned}
\mathcal{C}_1 &: \quad 3x_1 + 5x_2 - 2x_3 + x_4 - 2x_5 \leq 8 \\
\mathcal{C}_2 &: \quad 4x_3 + 3x_5 = 5 \\
\mathcal{C}_3 &: \text{ALLDIFF}(\, x_1\,, \quad x_2\,, \quad x_3\,)
\end{aligned}$$

with variables $x_1, x_2, x_3 \in \mathbb{Z}_{\geq 0}$ and $x_4, x_5 \in \mathbb{R}_{\geq 0}$. The linear inequality $\mathcal{C}_1$ down-locks x_3 and x_5, and up-locks x_1, x_2, and x_4. The equation $\mathcal{C}_2$ down-locks and up-locks both involved variables, x_3 and x_5. The ALLDIFF constraint $\mathcal{C}_3$ also down- and up-locks its variables, i.e., x_1, x_2, and x_3. The resulting lock numbers are $\zeta_1^- = 1$, $\zeta_1^+ = 2$, $\zeta_2^- = 1$, $\zeta_2^+ = 2$, $\zeta_3^- = 3$, $\zeta_3^+ = 2$, $\zeta_4^- = 0$, $\zeta_4^+ = 1$, $\zeta_5^- = 2$, $\zeta_5^+ = 1$. Most interestingly, variable x_4 has no down-locks. If its objective function coefficient c_4 is non-negative, we can fix it to its lower bound. This reduction does not alter the feasibility status of the instance, and if the instance is feasible it preserves at least one optimal solution. It is performed by the *dual fixing* plugin, see Section 10.8.

3.3.4 Variable Aggregation

One of the main operations to simplify the transformed problem during presolving is the fixing and aggregation of variables. This can delete variables from the problem by replacing their occurrences in the constraints with the corresponding counterpart, either a fixed value or an affine linear combination of active problem variables. The fixings and aggregations are stored in a variable aggregation graph, which is used by the framework to automatically convert any operations on those variables to equivalent ones on active problem variables. The variable aggregation graph is a directed graph which is free of directed cycles. The sinks of this graph, i.e., the nodes which do not have outgoing arcs, represent either fixed or active problem variables.

The variable aggregation graph encodes an equation system in triangular form

$$\begin{aligned}
y_1 &= f_1(x) \\
y_2 &= f_2(x, y_1) \\
&\dots \\
y_k &= f_k(x, y_1, \dots, y_{k-1})
\end{aligned}$$

with f_i being affine linear functions on active problem variables $x \in \mathbb{R}^{n-k}$ and aggregated variables $y_1, \dots, y_{i-1} \in \mathbb{R}$. The automatic transformations applied by SCIP to represent a given affine linear form in terms of active problem variables is called *crushing*.

Definition 3.6 (crushed form). Let $\{y_i = f_i(x, y_1, \dots, y_{i-1}) \mid i = 1, \dots, k\}$ be an equation system in triangular form with affine linear functions f_i on active problem variables $x \in \mathbb{R}^{n-k}$ and aggregated variables $y \in \mathbb{R}^k$. Given an affine linear function $g = g(x, y)$, the *crushed form* $\tau(g) = \tau(g)(x)$ is defined by recursively substituting $f_i(x, y_1, \dots, y_{i-1})$ for all occurrences of y_i in $g(x, y)$, $i = k, \dots, 1$.

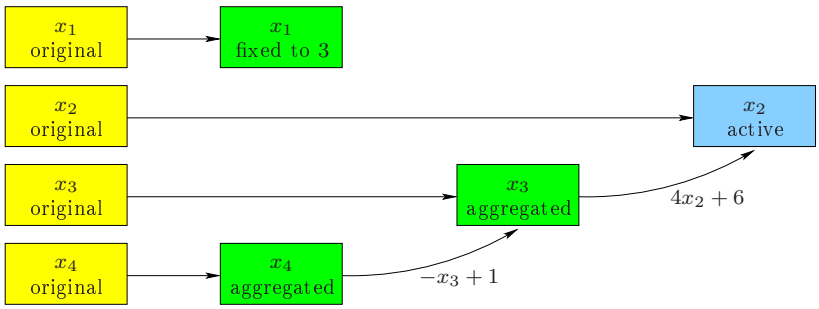

Figure 3.6. Variable aggregation graph of Example 3.8.

If the aggregations imply that two variables always take the same value in any feasible solution, we say that these variables are *equivalent*. Binary variables which always take opposite values are called *negated equivalent*.

Definition 3.7 (equivalence of variables). Given an aggregation system

$$\{y_i = f_i(x, y_1, \ldots, y_{i-1}) \mid i = 1, \ldots, k\},$$

two variables $z_1, z_2 \in \{x_1, \ldots, x_{n-k}, y_1, \ldots, y_k\}$ are called *equivalent* or *always equal*, denoted by $z_1 \stackrel{\star}{=} z_2$, if $\tau(z_1) = \tau(z_2)$. Two binary variables z_1, z_2 are called *negated equivalent* or *always unequal*, denoted by $z_1 \stackrel{\star}{\neq} z_2$, if $\tau(z_1) = \tau(1 - z_2)$.

In an algorithmic environment, the action of aggregating a variable y with an affine linear form $f(x)$ is denoted by $y :\stackrel{\star}{=} f(x)$.

Example 3.8. Consider the linear constraints

$$3x_1 = 9 \tag{3.5}$$
$$2x_1 + 4x_2 - x_3 = 0 \tag{3.6}$$
$$x_3 + x_4 = 1 \tag{3.7}$$

on integer variables $x_1, x_2, x_3, x_4 \in \mathbb{Z}$. The presolving of Constraint (3.5) fixes $x_1 :\stackrel{\star}{=} 3$. The linear constraint handler then replaces the occurrence of x_1 in (3.6) with its fixed value, resulting in $4x_2 - x_3 = -6$. Now, x_3 can be aggregated to $x_3 :\stackrel{\star}{=} 4x_2 + 6$. Additionally, Constraint (3.7) inserts the aggregation $x_4 :\stackrel{\star}{=} 1 - x_3$ into the aggregation graph.

Figure 3.6 shows the complete aggregation graph of this example. On the left hand side, the original problem variables are shown. They are linked to their transformed problem counterparts. The aggregations introduce additional links between the transformed variables.

Assume now, some constraint handler or cut separator adds the inequality

$$x_1 + 4x_2 + 3x_3 + 2x_4 \leq 23$$

to the LP relaxation. This inequality is constructed out of a mixture of original, fixed, aggregated, and active problem variables. Applying the aggregation graph, it is automatically transformed into a form only involving active problem variables. This results in the crushed form

$$\tau(x_1 + 4x_2 + 3x_3 + 2x_4) = 8x_2 + 11 \leq 23,$$

which is actually passed as $8x_2 \leq 12$ to the LP solver. From this inequality, we can derive the bound change $x_2 \leq 1$, which also automatically produces the corresponding bound changes $x_3 \leq 10$ and $x_4 \geq -9$.

3.3.5 Implication Graph and Clique Table

Atamtürk, Nemhauser, and Savelsbergh [24] proposed the notion of a *conflict graph* to store assignment pairs $(x_i = v_i, x_j = v_j)$ of binary variables $x_i, x_j \in \{0,1\}$ that cannot occur in any feasible solution. Such conflicting assignments can be detected in the presolving stage by constraint handlers or presolver plugins, in particular by the probing presolver, see Section 10.6. The conflict graph $G = (V, E)$ consists of vertices $V = \{x_j, \bar{x}_j \mid j = 1, \ldots, n\}$, and edges $E \subseteq \{uv \mid u, v \in V, u = 1 \rightarrow v = 0\}$.[3] Note that this condition is symmetric and hence G is undirected. Each edge of the graph represents one pair of conflicting variable assignments. The conflict graph defines a stable set relaxation of the problem instance, since variable assignments can only occur in a feasible solution if the corresponding vertices are not adjacent in the graph. Atamtürk, Nemhauser, and Savelsbergh use this relaxation to generate cutting planes which are known to be valid or facet defining for the stable set polytope, for example clique inequalities or odd-hole inequalities.

We take a slightly different approach. Instead of storing conflicting assignments in a conflict graph, we store implications in an *implication graph*. Additionally, we do not only store implications between binary variables, but also include implications between binary and arbitrary variables. Note also, that we use the term "*conflict graph*" for a different object which is constructed during conflict analysis, see Chapter 11.

Definition 3.9 (implication graph). Let CIP $= (\mathfrak{C}, I, c)$ be a CIP instance on variables $x \in \mathbb{R}^n$, $x_j \in \mathbb{Z}$ for $j \in I \subseteq N = \{1, \ldots, n\}$, and let $B \subseteq I$ be the indices of the binary variables $x_j \in \{0,1\}$. Then, the *implication graph* for CIP is the directed infinite graph $D = (V, A)$ with vertices

$$V = \{(x_j \leq v), (x_j \geq v) \mid j \in N, v \in \mathbb{R}\}$$

and arcs

$$\begin{aligned} A = &\{(x_i \leq 0, x_j \leq v) \mid i \in B, j \in N, v \in \mathbb{R}, x_i = 0 \rightarrow x_j \leq v\} \cup \\ &\{(x_i \leq 0, x_j \geq v) \mid i \in B, j \in N, v \in \mathbb{R}, x_i = 0 \rightarrow x_j \geq v\} \cup \\ &\{(x_i \geq 1, x_j \leq v) \mid i \in B, j \in N, v \in \mathbb{R}, x_i = 1 \rightarrow x_j \leq v\} \cup \\ &\{(x_i \geq 1, x_j \geq v) \mid i \in B, j \in N, v \in \mathbb{R}, x_i = 1 \rightarrow x_j \geq v\}, \end{aligned}$$

which represent the derivable implications $x_i = v_i \rightarrow x_j \leq v_j$ or $x_i = v_i \rightarrow x_j \geq v_j$ with $i \in B$, $j \in N$, $v_i \in \{0,1\}$, $v_j \in \mathbb{R}$.

Since one usually does not know all implications between the variables, the graph stored during the solving process is in general only a partial version of the full implication graph. Furthermore, we store only those nodes and arcs that are needed

[3] In general, we can only construct a subset of all conflict edges, since generating the full conflict graph is $\mathcal{NP}$-hard: deciding the feasibility of a binary programming instance is $\mathcal{NP}$-complete (see Garey and Johnson [92]), and for a given binary programming instance the conflict graph is the complete graph if and only if the instance is infeasible.

to represent the strongest implications for each binary variable assignment that are known to the solver, i.e.,

$$x_i = v_i \to x_j \leq \min\{v_j \mid x_i = v_i \to x_j \leq v_j\}$$

and

$$x_i = v_i \to x_j \geq \max\{v_j \mid x_i = v_i \to x_j \geq v_j\}.$$

Note that due to Condition (1.1) of Definition 1.6, there are no implications on continuous variables with strict inequalities in a constraint integer program. Therefore, the minimum and maximum above always exist or are infinite in which case the binary variable can be fixed to the opposite value.

The implication graph includes all conflicting assignments between binary variables. For such an assignment $x_i = 1 \to x_j = 0$, both arcs $(x_i \geq 1, x_j \leq 0)$ and $(x_j \geq 1, x_i \leq 0)$ are member of the implication graph. Implications $x_i = v_i \to x_j \leq v_j$ or $x_i = v_i \to x_j \geq v_j$ between binary variables x_i and non-binary variables x_j are only included unidirectional. The other direction is implicitly stored as a *variable bound* of x_j:

Definition 3.10 (variable bounds). Let x_i, $i \in I$, be an integer variable and x_j, $j \in N$, be an arbitrary variable of a constraint integer program. Valid inequalities

$$x_j \geq sx_i + d \quad \text{or} \quad x_j \leq sx_i + d$$

with $s, d \in \mathbb{R}$ are called *variable lower bounds* and *variable upper bounds* of x_j, respectively.

Note that in the definition of the variable bounds, the variable x_i does not need to be binary. If x_i is binary, however, the implications of x_i are related to the variable bounds of x_j.

Observation 3.11. Each implication on binary variables $x_i \in \{0, 1\}$ and arbitrary variables $x_j \in [l_j, u_j]$ gives rise to a variable bound of x_j:

$$\begin{aligned}
x_i = 0 \to x_j \leq v_j &\quad \Leftrightarrow \quad x_j \leq (u_j - v_j)x_i + v_j \\
x_i = 0 \to x_j \geq v_j &\quad \Leftrightarrow \quad x_j \geq (l_j - v_j)x_i + v_j \\
x_i = 1 \to x_j \leq v_j &\quad \Leftrightarrow \quad x_j \leq (v_j - u_j)x_i + u_j \\
x_i = 1 \to x_j \geq v_j &\quad \Leftrightarrow \quad x_j \geq (v_j - l_j)x_i + l_j
\end{aligned}$$

if the corresponding global bound l_j or u_j is finite.

SCIP stores the variable bounds in a similar data structure as the implications. This means, given a non-binary variable x_j, we can find all implications with x_j in the conclusion by inspecting the list of variable bounds of x_j. Whenever implications between binary variables are added, the arcs for both directions are added to the implication graph. If an implication between a binary variable x_i and a non-binary variable x_j is added, a corresponding variable bound for x_j is added to the variable bounds data structure. If a variable bound is added for x_j with x_i being binary, a corresponding implication between x_i and x_j is added to the implication graph. Furthermore, the implication graph is always maintained to be closed with respect to transitivity.

For certain types of problem instances, in particular for set partitioning and set packing instances, the explicit construction of the implication graph can consume huge amounts of memory. For example, consider the set partitioning constraint

$$\sum_{j=1}^{q} x_j = 1$$

with $x_j \in \{0,1\}$ for all $j = 1, \ldots, q$. In this case, every fixing of $x_j = 1$ leads to implications $x_j = 1 \to x_k = 0$ for all $k \neq j$. This means that each pair of variables in the constraint yields an arc in the implication graph, and thus, the number of arcs is quadratic in the number of variables. Typical set partitioning instances have a lot of variables, but only a few constraints. Take the instance `nw04` from Miplib 2003 [4, 6] as an example. It has 87482 variables, 36 constraints, and 636666 non-zero coefficients, which gives an average of 17685 variables per constraint. This leads to about 11 billion implications, and with 20 bytes used for each implication in SCIP's data structures, this results in 220 gigabyte memory consumption.

To avoid this situation, such sets of pairwise contradicting assignments of binary variables are stored in a separate table. This table is called *clique table*, since the variable assignments form a clique in the conflict graph as defined by Atamtürk, Nemhauser, and Savelsbergh [24]. We denote this table by $\mathcal{Q}$ and write $\mathcal{Q}(x_j = v)$, $v \in \{0,1\}$, to refer to the set of cliques the variable x_j ($v = 1$) or its negation $\bar{x}_j$ ($v = 0$) is member of.

In the example of `nw04`, we just have to store 36 cliques in the clique table, each of them having 17685 members on average. Using 12 bytes per element in the clique, this yields a memory consumption of about 8 Megabyte. The small disadvantage of the clique table is that we now have to scan two different data structures when we want to check for implications of a binary variable. In particular, this complicates the implementation of the clique cut separator, see Section 8.7.

3.3.6 Branching Tree

The subproblems that are processed during the branch-and-bound search are organized as a branching tree, see Section 2.1. The root node of the tree represents the global problem instance R. The partitioning of a problem Q into subproblems $Q_1, \ldots, Q_k$ by branching creates *child nodes* of Q. Except for the root node, each node Q has a unique *parent node* $p(Q)$. Child nodes Q_i, Q_j with the same parent node $p(Q_i) = p(Q_j)$ are called *siblings*. The nodes on the path from the parent node $p(Q_i)$ to the root node R are called *ancestors* of Q. If a subproblem is pruned (due to bounding, infeasibility, or optimality), it is removed from the tree. Additionally, its ancestors are removed recursively as long as they have no other children. If the root node is removed, the problem instance has been solved.

The search tree can be partitioned into *depth levels*, where the level $d(Q)$ of a node Q is the length of the shortest path from Q to the root node R. This means, the root node is in depth level $d(R) = 0$, and the children Q_i of a node Q are in depth level $d(Q_i) = d(Q) + 1$.

SCIP stores the subproblem information using *trailing*, which means to only store the differences to the parent node $p(Q)$ in the subproblem Q. Therefore, to switch from one subproblem Q to the next subproblem Q', one has to find the common ancestor $\hat{Q}$ of Q and Q' of maximal depth, undo the problem changes on the path from Q to $\hat{Q}$, and apply the changes on the path from $\hat{Q}$ to Q'. An alternative to *trailing* is to employ *copying* where the whole problem information is stored at each

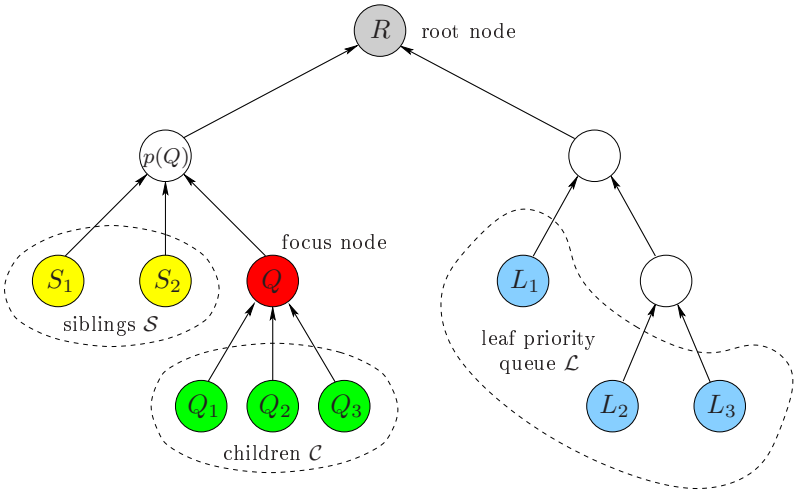

Figure 3.7. Branching tree data structure. Each node has a pointer to its parent. The focus node, its children and its siblings are stored in individual data structures. The other unprocessed leaves of the tree are stored in a priority queue.

subproblem. This would entail a faster switching between subproblems at a cost of a larger memory consumption. See Schulte [200] for a comparison of these strategies in constraint programming.

The unprocessed leaves of the search tree are stored in a priority queue $\mathcal{L}$—the *leaf priority queue*—with a priority function being defined by the node selection strategy in charge, see Section 3.1.7. The currently processed subproblem, its siblings, and its children are stored in separate data structures outside the queue, see Figure 3.7. The following information is attached to each node Q of SCIP's search tree:

- ▷ a pointer to the parent node $p(Q)$,
- ▷ the constraints that have been deleted at Q,
- ▷ the constraints that have been added to Q,
- ▷ the bounds that have been tightened at Q,
- ▷ a lower (dual) objective bound of Q,
- ▷ the depth $d(Q)$ of the node in the search tree,
- ▷ a successively assigned unique node number,
- ▷ a flag that indicates whether the node is active or not,
- ▷ the type of the node.

The path $\mathcal{A} = (\mathcal{A}_0, \ldots, \mathcal{A}_{d(Q)})$ from the root node $\mathcal{A}_0 = R$ to the currently processed subproblem $\mathcal{A}_{d(Q)} = Q$ is called *active path*. A node $\hat{Q} \in \mathcal{A}$ is called *active node*. Note that the root node R is always active.

We distinguish between the following types of nodes:

- ▷ the FOCUSNODE is the currently processed subproblem,

- ▷ a SIBLING is a sibling of the focus node that was not yet stored in the leaf priority queue,
- ▷ a CHILD is a child of the focus node,
- ▷ a LEAF is a leaf which is stored in the leaf priority queue,
- ▷ a JUNCTION is an already processed subproblem for which the LP relaxation was not solved, and
- ▷ a FORK is an already processed subproblem for which the LP relaxation was solved.

In fact, there are some more node types, namely PROBINGNODE, DEADEND, PSEUDOFORK, SUBROOT, and REFOCUSNODE. We omit these additional node types in the presentation, since they are only needed due to technical implementation issues.

Depending on the type of the node, additional node information is stored at a subproblem:

- ▷ the number of existing children (FOCUSNODE, JUNCTION, FORK),
- ▷ the columns and rows that have been added to the LP relaxation (FOCUSNODE, FORK),
- ▷ the LP warm start information (FOCUSNODE, FORK), and
- ▷ a pointer to the FORK parent, i.e., the ancestor of maximal depth for which the LP relaxation was solved (SIBLING, CHILD, LEAF).

The main operation that is supported by the branching tree is the switching between subproblems Q and Q', which is depicted in Algorithm 3.1. In order to find the common ancestor $\hat{Q}$ of Q and Q' in Step 1, we just have to follow the path from Q' to the root node R until an active node is discovered. Note that the loop always terminates, since the root node is active. Step 2 restores the ancestor $\hat{Q}$ by undoing all problem and LP changes on the path from $\hat{Q}$ to Q in reverse direction. Since the problem and LP changes from $\hat{Q}$ to Q' have to be applied in forward direction, and we can only traverse the tree in backward direction from the leaves to the root, we have to first construct the new active path $\mathcal{A}$ in Step 3. Afterwards, the path can be traversed in the desired direction in order to apply the problem changes and to construct subproblem Q'.

If the old focus node Q has children, i.e., a branching took place to solve the subproblem, it is converted into a FORK or JUNCTION node in Step 5. If the LP relaxation was solved, the current warm start information is retrieved from the LP solver, and the node type is set to FORK. If the LP relaxation of Q was not solved or if it was discarded due to numerical difficulties, the node type is set to JUNCTION. If the old focus node has no children, it is either infeasible, exceeds the primal bound, or was solved to optimality. In any case, it can be pruned from the tree, thereby reducing the number of live children of its parent $p(Q)$. If the parent now also has no more children, it can be deleted as well, and the deletion can be continued recursively.

Steps 6 to 8 update the sets of current siblings $\mathcal{S}$, children $\mathcal{C}$, and active leaves $\mathcal{L}$. If the new focus node Q' is a child of the old focus node Q, the former children become siblings and the former siblings are moved to the leaf priority queue. If Q' is a sibling of Q, the other siblings remain siblings, and the former children are

Algorithm 3.1 Node Switching

Input: current subproblem Q with siblings $\mathcal{S} = \{S_1, \ldots, S_l\}$ and children $\mathcal{C} = \{Q_1, \ldots, Q_k\}$, active path $\mathcal{A}$, and leaf priority queue $\mathcal{L}$; next subproblem Q' to be processed which is of type SIBLING, or CHILD, or which is the top-priority LEAF in $\mathcal{L}$.

Output: updated data structures such that the new current problem is Q'.

1. Find the common ancestor node $\hat{Q}$ of Q and Q' of maximal depth:
   (a) Set $\hat{Q} := Q'$.
   (b) While $\hat{Q}$ is not active, set $\hat{Q} := p(\hat{Q})$.
2. Undo all problem and LP changes on the path from Q to $\hat{Q}$:
   (a) Set $\tilde{Q} := Q$.
   (b) While $\tilde{Q} \neq \hat{Q}$:
      i. Remove the columns and rows from the LP relaxation that have been added at $\tilde{Q}$.
      ii. Add the constraints that have been deleted from $\tilde{Q}$.
      iii. Delete the constraints that have been added to $\tilde{Q}$.
      iv. Relax the bounds that have been tightened at $\tilde{Q}$.
      v. Set $\tilde{Q} := p(\tilde{Q})$.
3. Update the active path:
   (a) Set $\tilde{Q} := Q'$.
   (b) While $\tilde{Q} \neq \hat{Q}$: Set $\mathcal{A}_{d(\tilde{Q})} := \tilde{Q}$, and set $\tilde{Q} := p(\tilde{Q})$.
4. Apply all problem changes on the path from $\hat{Q}$ to Q':
   (a) For $\tilde{Q} = \mathcal{A}_{d(\hat{Q})}, \ldots, \mathcal{A}_{d(Q')}$:
      i. Tighten the bounds that have been tightened at $\tilde{Q}$.
      ii. Add the constraints that have been added to $\tilde{Q}$.
      iii. Delete the constraints that have been deleted from $\tilde{Q}$.
      iv. Add the columns and rows from the LP relaxation that have been added at $\tilde{Q}$.
5. If $\mathcal{C} \neq \emptyset$, convert Q into a FORK or JUNCTION node, depending on whether the LP relaxation has been solved or not. Otherwise, delete Q and all of its ancestors without live children.
6. If Q' is of type CHILD: Convert siblings $\mathcal{S}$ to type LEAF, set $\mathcal{L} := \mathcal{L} \cup \mathcal{S}$, convert children $\mathcal{C} \setminus \{Q'\}$ to type SIBLING, and set $\mathcal{S} := \mathcal{C} \setminus \{Q'\}$ and $\mathcal{C} := \emptyset$.
7. If Q' is of type SIBLING: Convert children $\mathcal{C}$ to type LEAF, and set $\mathcal{L} := \mathcal{L} \cup \mathcal{C}$, $\mathcal{S} := \mathcal{S} \setminus \{Q'\}$, and $\mathcal{C} := \emptyset$.
8. If Q' is of type LEAF: Convert siblings $\mathcal{S}$ and children $\mathcal{C}$ to type LEAF, and set $\mathcal{L} := (\mathcal{L} \setminus \{Q'\}) \cup \mathcal{S} \cup \mathcal{C}$, $\mathcal{S} := \emptyset$, and $\mathcal{C} := \emptyset$.
9. Convert Q' to type FOCUSNODE, and update $Q := Q'$.
10. Load the LP warm start information of the FORK parent of Q.

moved to the leaf priority queue. If the new focus node Q' is a leaf from the priority queue, it is not a direct relative of the former children and siblings. Therefore, both children and siblings must be moved to the leaf priority queue.

Step 9 installs node Q' as the new focus node Q. Finally in Step 10, the LP warm start information for the new focus node Q is loaded into the LP solver. For simplex solvers, this is usually the optimal simplex basis of the FORK parent, but since the warm start information is an abstract data type, which is implemented in the LP solver interface, each LP solver can store a different type of warm start information in the tree.

Note. The LP warm start information should be as compact as possible, since the warm start information of a subproblem Q must stay in memory as long as there is any active node left whose FORK parent is Q. A simplex basis can be stored using only two bits per row and column, since for each row (i.e., slack variable) and each column we only need to know whether it is basic, on its lower, or on its upper bound. Using this information, the corresponding basic solution can be recalculated by refactorizing the basis matrix and solving the corresponding equation systems.

Of course, it would improve the LP solving performance if we also stored the basis matrix factorization, since this would save the effort for recalculating the initial factorization. Unfortunately, the factorization usually consumes too much memory to store it in the search tree for every node. A reasonable tradeoff is to apply depth first search or plunging node selection strategies, see Section 3.1.7 and Chapter 6. They tend to choose children of the current node as the next subproblem to be processed, which means that the current factorization which is still loaded in the LP solver can be used further and does not need to be recalculated.

3.3.7 LP Relaxation

The LP relaxation stores a linear relaxation of the current subproblem. Like the stable set relaxation, given by the implication graph described in Section 3.3.5, it provides a global view on the problem and a way of sharing information between different plugins.

We make the following notational distinctions between the CIP and its LP relaxation. The CIP consists of *variables* and *constraints*. The variables are marked to be integer or continuous. The constraints are stored in constraint handler specific data structures. Their semantics is unknown to the framework and only implicitly given by the actions performed in the constraint handlers' callback methods. The LP relaxation consists of *columns* and *rows*. For each column, the lower and upper bounds are known. Every column belongs to exactly one CIP variable, but not every CIP variable needs to be represented by a column in the LP. The rows are defined as linear combinations of columns and have left and right hand sides as additional data. A single constraint like the TSP's NOSUBTOUR constraint (see Example 3.2 on page 24) can give rise to multiple rows in the LP, but rows can also live on their own, e.g., if they were created by a general purpose cut separator.

The LP relaxation is created and extended by variable pricing and cutting plane separation. Pricing of existing problem variables is performed automatically, while unknown variables have to be treated by problem specific variable pricer plugins, see Section 3.1.5. New columns enter the LP through the pricing storage described in Section 3.3.9. Cutting plane separation is performed by constraint handlers and separators, see Sections 3.1.1 and 3.1.3, respectively. These linear inequalities and

equations are passed to the LP relaxation via the separation storage explained in Section 3.3.8. Both variables and constraints can be marked to be *initial* which means that their corresponding columns and rows form the initial LP of the root node relaxation. Afterwards, further extensions are applied in the price-and-cut loop during the processing of the subproblems, see Section 3.2.8. The LP relaxation is automatically updated during the switching between subproblems in the search tree, see Section 3.3.6.

For any given subproblem, the user may choose whether the LP relaxation should be solved. By completely deactivating the LP relaxation, one can mimic a pure constraint programming or SAT solver, while solving the LP relaxation at every node is common for mixed integer programming solvers.

3.3.8 Separation Storage

Cutting planes are produced by constraint handlers and cut separators, see Sections 3.1.1 and 3.1.3, respectively. After adding cuts, the LP is resolved, and the separators are called again with the new LP solution. It turns out to be very inefficient to immediately resolve the LP after the first cutting plane was found. Instead, one performs cutting plane separation in rounds. In each round, various cutting planes are generated to cut off the current LP solution. Since one does not want to increase the size of the LP too much by adding all cutting planes that one can find, they are first collected in the *separation storage* from which only a subset of the available cutting planes is selected to enter the LP.

The selection of the cuts to be added to the LP is a crucial decision which affects the performance and the stability of the LP solving process in the subsequent calls. In SCIP, the cuts are selected with respect to three different criteria:

- ▷ the *efficacy* of the cuts, i.e., the distance of their corresponding hyperplanes to the current LP solution,
- ▷ the *orthogonality* of the cuts with respect to each other, and
- ▷ the *parallelism* of the cuts with respect to the objective function.

The first two have already been used by Balas, Ceria, and Cornuéjols [30] in the context of lift-and-project cuts, and by Andreello, Caprara, and Fischetti [13] for $\{0, \frac{1}{2}\}$-cuts.

It is tried to select a nearly orthogonal subset of cutting planes, which cut as deep as possible into the current LP polyhedron. Cutting planes are slightly preferred if they are closer to being parallel to the objective function. The user has the possibility to change the employed distance norm to measure the efficacy of the cuts. The default settings apply the Euclidean norm. The user can also adjust the importance of the three criteria with respect to each other. Computational results to evaluate the cutting plane selection can be found in Section 8.10.

Algorithm 3.2 shows the details of the selection procedure. For each cut $r \in \mathcal{R}$ that was found in the current separation round, we calculate the efficacy e_r and the objective function parallelism p_r in Step 1. Using an initial orthogonality value of $o_r = 1$, the initial score $s_r = s(e_r, p_r, o_r)$ is computed. The score function $s : \mathbb{R}^3 \to \mathbb{R}$ in SCIP combines the individual quality measures to produce a single value by calculating a weighted sum $s(e_r, p_r, o_r) = w_e e_r + w_p p_r + w_o o_r$ of the three operands with non-negative weights $w_e, w_p, w_o \in \mathbb{R}_{\geq 0}$. The weights themselves can be adjusted by the user. The default settings are $w_e = 1$, $w_p = 0.1$, and $w_o = 1$.

Algorithm 3.2 Cutting Plane Selection

Input: current LP solution $\check{x}$ and set $\mathcal{R}$ of generated cutting planes; a score function $s : \mathbb{R}^3 \to \mathbb{R}$, the minimal orthogonality minortho $\in [0, 1]$, and the maximal number maxsepacuts of separated cuts per round.

Output: updated LP relaxation.

1. For all $r \in \mathcal{R}$ with $r : \underline{\gamma}_r \leq d_r^T x \leq \overline{\gamma}_r$ calculate:
   (a) the efficacy $e_r := \max\{\underline{\gamma}_r - d_r^T\check{x}, d_r^T\check{x} - \overline{\gamma}_r\}/\|d_r\|$,
   (b) the objective parallelism $p_r := |d_r^T c| \,/\, (\|d_r\| \cdot \|c\|)$,
   (c) the initial orthogonality $o_r := 1$, and
   (d) the initial score $s_r := s(e_r, p_r, o_r)$.
2. While $\mathcal{R} \neq \emptyset$ and less than maxsepacuts cuts have been added to the LP:
   (a) Add cut $r^\star \in \mathcal{R}$ with largest score $s_{r^\star}$ to the LP. Set $\mathcal{R} := \mathcal{R} \setminus \{r^\star\}$.
   (b) For all cuts $r \in \mathcal{R}$:
      i. Update $o_r := \min\left\{o_r, 1 - |d_{r^\star}^T d_r| \,/\, (\|d_{r^\star}\| \cdot \|d_r\|)\right\}$.
      ii. If $o_r <$ minortho, set $\mathcal{R} := \mathcal{R} \setminus \{r\}$.
      Otherwise, update $s_r := s(e_r, p_r, o_r)$.

After calculating the initial score values, the cuts are consecutively passed to the LP relaxations in Loop 2 until the cut list is empty or a maximum number maxsepacuts of cuts has been added. In each iteration of the loop, a cut $r^\star$ with largest score $s_{r^\star}$ is selected in Step 2a and enters the LP. Afterwards, we update the orthogonalities o_r of the remaining cuts $r \in \mathcal{R}$ in Step 2b. A cut is discarded if its orthogonality falls below the threshold minortho. Otherwise, we recalculate its score s_r.

SCIP uses a default setting of minortho $= 0.5$. The number of cuts generated per round is restricted to maxsepacuts $= 2000$ in the root node and to maxsepacuts $= 100$ in subproblems. Using a minimal orthogonality minortho > 0 automatically removes all cuts that are dominated by or are equal to parallel cuts. Stronger domination criteria that take the current bounds of the variables into account are not applied. However, cuts can be marked to be "*removable*", which means that they will be eliminated from the LP if they are not satisfied with equality for a number of separation rounds or in the final LP solution. Thereby, dominated "removable" cuts are automatically deleted from the LP relaxation with a slight delay.

The results of computational experiments to evaluate various cut selection policies can be found in Section 8.10 in the context of cutting plane separation for mixed integer programs.

3.3.9 Pricing Storage

Like cutting plane separation, pricing of variables is performed in rounds. In every pricing round, the current problem variables which are not yet represented in the LP are inspected to check whether their inclusion in the LP relaxation would potentially decrease (i.e., improve) the LP objective value. Additionally, all activated pricer plugins are called to generate new variables. These candidate variables for entering the LP as columns are collected in a *pricing storage*.

In contrast to cutting planes, the CIP framework does not have enough informa-

tion about a variable to evaluate its effectiveness in the LP relaxation. In fact, it is unknown which constraints depend on the new variable and which role the variable plays for the semantics of the constraints. Therefore, the new variables are sorted in the pricing storage with respect to a *score* value, which must be provided by the external pricing algorithms. Usually, one uses the reduced costs of the variable as score value, but other—more problem specific—criteria are also possible.

After a pricing round is finished, the best maxpricevars variables in the pricing storage are added to the LP relaxation by creating corresponding columns. In the default settings, SCIP uses maxpricevars = 2000 at the root node and maxpricevars = 100 at subproblems. After adding the new columns, the LP is resolved and the next pricing round is performed. This process is iterated until no more improving variables can be found.

3.3.10 Cut Pool

Certain cut separation algorithms are computationally very expensive or produce cutting planes in a heuristic fashion. In this case, it might be desirable to keep the retrieved cutting planes even if they are useless for separating the current LP solution. The hope is that they might be applicable in later separation rounds or on other subproblems in the search tree. Since it is very expensive to generate them again or—due to the heuristic nature of the separation algorithm—we may fail to find them again, it can be useful to store those cutting planes for later use in a global *cut pool*.

The cut pool is a collection of globally valid LP rows augmented by a hash table to avoid multiple insertions of the same row. The rows in the cut pool are checked for violation during the price-and-cut loop of the node processing, see Section 3.2.8. If violated rows are found, they are added as ordinary cutting planes to the separation storage. In order to limit the size of the cut pool and the associated expenses for processing the rows during separation, we delete rows from the pool if they are not violated for cutagelimit consecutive violation checks. This parameter is set to cutagelimit = 100 by default.

Besides the global cut pool, the user can use additional cut pools for his own purposes. He can add and delete cuts from a pool, he can inspect the current contents of a pool, and he can separate the rows stored in the pool. Furthermore, the cutagelimit parameter can be set individually for each cut pool.

3.3.11 Solution Pool

In a simple branch-and-bound scheme as explained in Section 2.1, we only have to store the current best primal solution, the so-called *incumbent solution*. The incumbent is only used for pruning subproblems by bounding and to have the final optimal solution available when the search is completed.

For particular purposes, however, it is useful to have different feasible solutions at hand, even if some of them have worse objective values than others. For example, in many applications it is not completely clear which is the desired objective function for the model. In this situation, a user might want to obtain a larger set of "good" feasible solutions, which he can evaluate and compare based on his knowledge and experience with the underlying real-world problem.

Despite this practical reason, suboptimal solutions can also help to speed up the solving process. For example, a node selection strategy (see Section 3.1.7) might

want to consider regions of the search tree first, that are close to a number of feasible solutions. The hope is to find more and even better solutions in this region. A branching rule (see Section 3.1.6) might want to branch on variables first that are set to the same value in all feasible solutions found so far. If the subproblem becomes infeasible in the opposite branching direction, the variable can be fixed. The most important benefits of a large pool of different feasible solutions, however, are primal heuristics, see Section 3.1.8 and Chapter 9. In particular the *improvement heuristics* consider also suboptimal solutions as a starting point from which improved solutions can be found.

SCIP stores the best maxsol solutions with respect to the objective value in a sorted array which is called the *solution pool*. The first element of this array is the incumbent solution. In the default settings, the size of the solution pool is restricted to maxsol = 100.

3.3.12 Memory Management

The internal memory management of SCIP provides three different memory allocation techniques:

- ▷ standard memory management,
- ▷ block memory management, and
- ▷ memory buffers.

Depending on the type of data object and its life cycle, one of the three techniques should be selected to allocate the necessary memory.

Standard Memory Management

Standard memory management denotes the allocation and deallocation of memory with the standard methods `malloc()` and `free()` of the C programming language. In fact, in optimized compilation mode, SCIP's methods for standard memory management are only synonyms for these C methods. In debugging mode, however, standard memory management includes the maintenance of a list of currently allocated memory regions together with the source code lines at which each memory region was allocated. At the end of the program execution the list contains the memory regions that were not deallocated and can therefore detect memory leaks.

Block Memory Management

During a typical run of SCIP to solve a CIP instance, many data objects of the same type are allocated and deallocated from the memory heap of the system process. For example, there may exist thousands of cutting plane data structures or millions of branch-and-bound nodes. Using the standard `malloc()` and `free()` methods can lead to a substantial overhead caused by the free list management of this allocator and the operating system.

A common way of improving runtime performance in memory management is to replace the standard memory allocator with a program specific allocation method. A thorough review of allocator strategies can be found in Wilson et al. [215]. The block memory management of SCIP implements a *suballocator* (see [191]) using a *segregated free lists* scheme with *exact lists* (see Comfort [67]). For each memory block

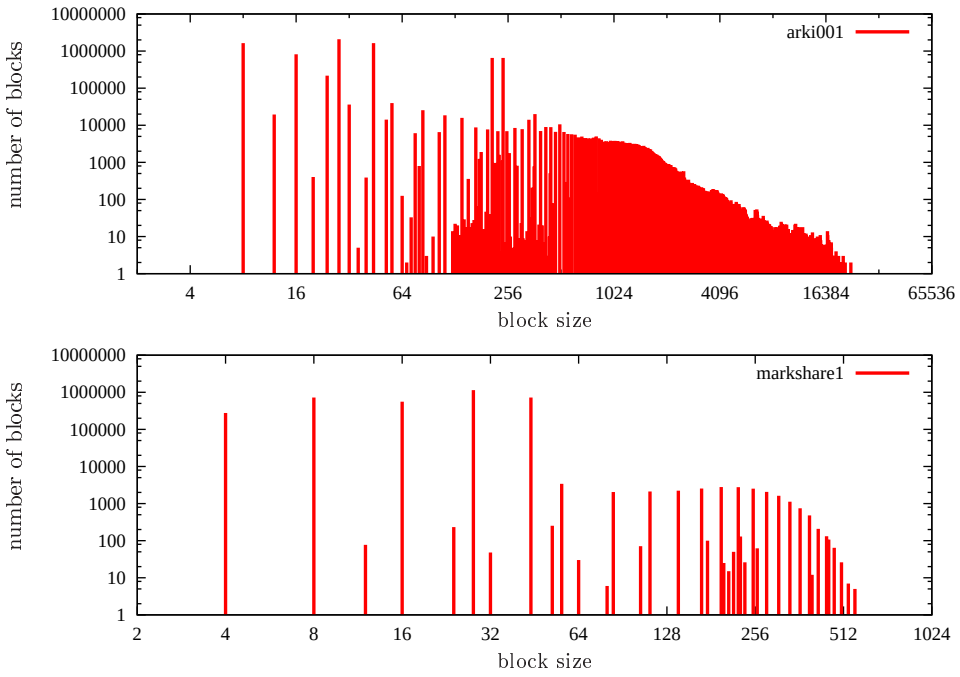

Figure 3.8. Histograms of block sizes allocated after processing 1 000 000 branch-and-bound nodes of the MIP instances `arki001` and `markshare1`.

size, a single-linked list of free memory blocks is maintained. The lists themselves are stored in a hash map referenced by the block size. If a memory block of a certain size is allocated, the first element in the corresponding free list is unlinked from the list and returned to the user. If the free list is empty, a new chunk of memory is allocated by calling `malloc()`. The chunk is split into blocks of the corresponding size, which are added to the free list. The sizes of the chunks grow exponentially with the number of chunks allocated for a given block size. Thereby, the number of `malloc()` calls is logarithmic in the maximal number of simultaneously live blocks.

If a memory block is no longer needed, the user has to call the deallocation method of the block memory allocator, which has to add the block to the free list of the corresponding block size. Therefore, the allocator has to know the size of the freed block. Many existing memory allocators for C replace the `malloc()` and `free()` calls by own implementations. This has the advantage that the user does not need to modify the source code. Since `free()` does not provide the size of the freed memory block as a parameter, the allocator has to retrieve the size by different means. Usually, the size of a block is recorded in a *header field*, which is an additional word located in front of the actual data block. This means, that the memory consumption is increased by one word per memory block.

Figure 3.8 shows typical memory allocation histograms as they appear during the solving process of MIP instances. Note that both axes are logarithmically scaled. The instance `arki001` is a medium sized MIP with 1388 variables and 1048 constraints. One can see that most memory blocks are of small size: 78 % of the allocated blocks are smaller or equal than 44 bytes, which is the size of the branch-and-bound node data structure in SCIP on a 32-bit processor. The average block size is 101.3 bytes. For the very small instance `markshare1` with 62 variables and 6 constraints, the allocation histogram shows an even larger tendency towards small

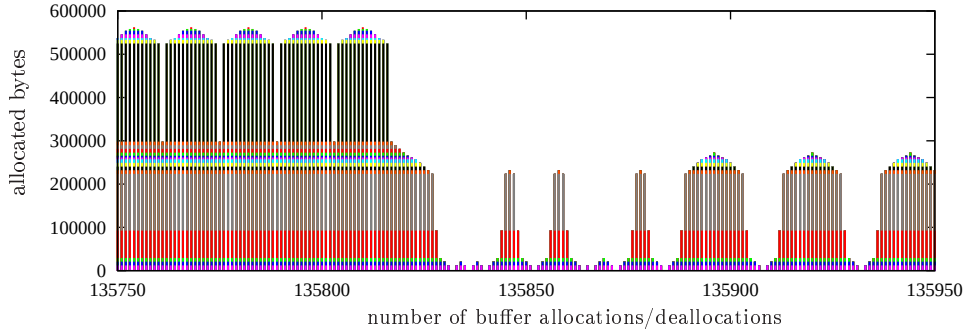

Figure 3.9. Extract of memory buffer allocation trace during processing the MIP instance `arki001`.

block sizes. Here, 99 % of the blocks are of size up to 44 bytes, and the average block size is 24.6 bytes. In this case, an overhead of one word (i.e., 4 bytes on a 32-bit processor) for the blocks would increase the memory consumption by 16 %.

For this reason, we did not implement the block memory allocator as a direct replacement of `malloc()` and `free()`. Instead, the user has to provide the deallocation method with the size of the memory block to be freed. This was never a problem in the whole implementation of SCIP, since the sizes of the data objects were always known, even for dynamically allocated arrays. In contrast, this redundancy helps to detect errors when dealing with dynamically growing data arrays, since in debug mode it is checked whether a freed block is actually a member of a memory chunk of the given block size.

Memory Buffers

It is very common that subroutines in SCIP or in user plugins need a certain amount of temporary memory for internal calculations that can be discarded after the subroutine exits. Often the size of the temporary memory depends on the problem instance or on dynamically changing parameters. For example, a branching rule might want to calculate preference values for all integer variables with a fractional value in the current LP solution and therefore needs a data array of length at least equal to the number of fractional variables. Such a memory area cannot be allocated statically on the stack, since its size is not known at compile time. Therefore, one has to dynamically allocate memory from the heap.

Since many subroutines are called very often during the solving process, it can be inefficient to allocate and deallocate the memory in each call with `malloc()` and `free()`. Instead, temporary memory should be allocated by using SCIP's *memory buffers*. Memory buffers are allocated or enlarged to fit the needs of an allocation request, but they are not returned to the operating system with a call to `free()`. Instead, SCIP keeps the unused buffers allocated to satisfy later requests for temporary memory.

The buffers are organized as a stack, which matches the typical allocation and deallocation behavior of temporary memory. Figure 3.9 shows an extract of the memory buffer allocation trace, which is generated during the solving process of the MIP instance `arki001`. The x axis counts the number of buffer operations, i.e., either allocations or deallocations. The height of a bar denotes the total size of the temporary memory that is currently in use. The color partitioning of a bar shows how large the individual buffers are. One can see that most of the time, the

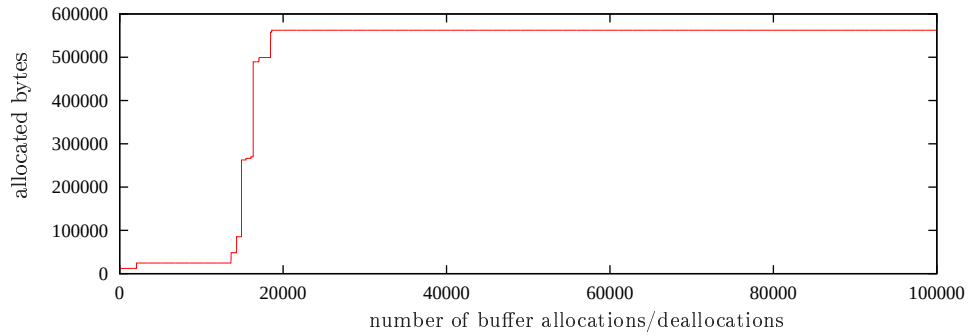

Figure 3.10. Total size of buffer memory allocated during processing the MIP instance `arki001`.

buffers are allocated in the same order and with the same size. This is because nested subroutines are always called in the same order. At buffer operation 135832, one set of nested subroutines was finished such that all buffers were deallocated. Afterwards, the program entered a different subroutine which produced a different buffer allocation scheme.

Figure 3.10 illustrates the progression of the total size of the memory buffers in the beginning of the solving process. Presolving is finished after 14 902 buffer operations using a total of 85 418 bytes in memory buffers. Afterwards, the branch-and-bound search and cutting plane generation starts, which drives the total size of the buffers to 562 473 bytes. In the remaining solving process, the buffers are never enlarged again, which means that no additional memory allocation with `malloc()` or `realloc()` has to be performed.

Table 3.1 gives a summary of the performance impact of block memory allocation and memory buffers; compare also the detailed Tables B.1 to B.10 in Appendix B, and see Appendix A for a description of the test sets and the computational environment. Column "no block" shows the results with disabled block memory, "no buffer" with disabled memory buffers, and "none" with both disabled. A disabled technique is replaced by standard `malloc()` and `free()` calls.

One can see that enabling both block memory and memory buffers yields the best performance on almost all of the test sets. While the speedup due to memory buffers does not seem to be significant, block memory allocation gives a substantial runtime improvement of up to 11 %.

	test set	no block	no buffer	none
time	MIPLIB	+8	+3	+6
	CORAL	+10	0	+11
	MILP	+7	−1	+7
	ENLIGHT	+3	−1	+7
	ALU	+4	+1	+6
	FCTP	+5	+4	+3
	ACC	+2	0	+2
	FC	+3	+3	+5
	ARCSET	+9	+1	+8
	MIK	+11	+1	+11
	total	+8	+1	+8

Table 3.1. Performance effect of different memory management techniques for solving MIP instances. The values denote the percental changes in the geometric mean of the runtime compared to the default settings with both memory management techniques enabled. Positive values represent a slowdown.

Two interesting cases are the ENLIGHT and ALU testsets, in which most instances are quite small in size but require many branching nodes to solve. Thus, a comparably large fraction of the time is spent on memory operations. One can see that disabling block memory allocation yields a small slowdown of 3 % and 4 %, respectively, on these two test sets, but disabling memory buffers does not make any significant difference. The comparison of columns "no block" and "none" shows, however, that with block memory turned off, memory buffers do have an impact on the performance. A possible explanation might be that with block memory, the buffer memory allocations are almost the only remaining memory operations left, and that the standard allocator of `malloc()` and `free()` behaves very similar to the memory buffer strategy. Without using block memory, temporary and long-term memory allocations with `malloc()` are interleaved, which might negatively influence the performance of the standard allocator.

Part II

Mixed Integer Programming

Chapter 4

Introduction

Integer programming and mixed integer programming emerged in the late 1950's and early 1960's when researchers realized that the ability to solve mixed integer programming models would have great impact for practical applications (see Markowitz and Manne [156] and Dantzig [74]). Gomory [104] and Martin [161] discovered the first algorithms that can solve integer programs to optimality in a finite number of steps. Further details on the history of integer programming can be found in Gomory [105].

Mixed integer programming has applications in a large variety of domains, including scheduling, project planning, public transport, air transport, telecommunications, economics and finance, timetabling, mining, and forestry. Many of these examples can be found in Heipcke [114].

As there is a lot of commercial interest in solving MIPs, it is not surprising that the development of MIP solvers soon became itself a commercial endeavor. Today's best tools for mixed integer programming are developed by commercial vendors, including Cplex [118], Lingo [148], and Xpress [76]. The source code of these solvers is proprietary, which imposes some difficulties for academic researchers to evaluate new ideas by computations within a state-of-the-art environment. Despite its integration of CP and SAT techniques into MIP solving, the development of SCIP can be seen as an attempt to provide the MIP research community with a freely available software, which is (almost) comparable to the current commercial codes in terms of performance. A benchmark comparison of Cplex and SCIP can be found in Appendix C.

In this part of the thesis we investigate the key ingredients of branch-and-bound based MIP solvers, discuss a number of different approaches and algorithms for each component, and present some new ideas. Chapter 5 evaluates different branching rules and subsumes a number of well-known strategies under a new and very general parameterized rule, the *reliability branching rule*. Chapter 6 presents and compares different strategies to select the next subproblem from the search tree to be processed. The selected node is then subject to the domain propagation algorithms, which are discussed in Chapter 7. A very brief overview of various cutting plane separation algorithms to strengthen the LP relaxation of the node is given in Chapter 8.

During the node solving process, primal heuristics are called at different points in order to generate feasible MIP solutions. A huge amount of proposals for MIP heuristics can be found in the literature. Chapter 9 presents the ones that are implemented in SCIP. Chapter 10 explains the presolving techniques that are used to simplify the problem instance and to extract additional information about the instance before the actual solving process commences. Finally, Chapter 11 generalizes the idea of *conflict analysis* in SAT solvers to mixed integer programming. Besides its value for mixed integer programming, we will see in Part III of the thesis that conflict analysis is a very important tool for solving the chip design verification problem by constraint integer programming.

Each chapter presents computational results to compare the effectiveness of the discussed algorithms and strategies. The test set of MIP instances that we used and the computational environment in which the experiments have been performed are described in Appendix A.

We conclude this introduction by recapitulating the basic definitions of mixed integer programming as they have been introduced in Section 1.3.

A mixed integer program (MIP) is defined as follows.

Definition (mixed integer program). Given a matrix $A \in \mathbb{R}^{m \times n}$, vectors $b \in \mathbb{R}^m$, and $c \in \mathbb{R}^n$, and a subset $I \subseteq N = \{1, \ldots, n\}$, the *mixed integer program* $\text{MIP} = (A, b, c, I)$ is to solve

$$\text{(MIP)} \quad c^\star = \min \{c^T x \mid Ax \leq b,\ x \in \mathbb{R}^n,\ x_j \in \mathbb{Z} \text{ for all } j \in I\}\,.$$

The vectors in the set $X_{\text{MIP}} = \{x \in \mathbb{R}^n \mid Ax \leq b,\ x_j \in \mathbb{Z} \text{ for all } j \in I\}$ are called *feasible solutions* of MIP. A feasible solution $x^\star \in X_{\text{MIP}}$ of MIP is called *optimal* if its objective value satisfies $c^T x^\star = c^\star$.

The bounds of the variables are denoted by $l_j \leq x_j \leq u_j$ with $l_j, u_j \in \mathbb{R} \cup \{\pm\infty\}$. Formally, they are part of the constraint system $Ax \leq b$, but in practice they are treated outside the coefficient matrix. Depending on the integrality status and the bounds of the variables, we define the following subsets of the variable indices $N = \{1, \ldots, n\}$:

$$\begin{aligned} \text{binary variables:} \quad & B := \{j \in I \mid l_j = 0 \text{ and } u_j = 1\} \\ \text{integer variables:} \quad & I \\ \text{continuous variables:} \quad & C := N \setminus I \end{aligned}$$

Specializations of MIPs are

- ▷ *linear programs* (LPs) with $I = \emptyset$,
- ▷ *integer programs* (IPs) with $I = N$,
- ▷ *mixed binary programs* (MBPs) with $B = I$, and
- ▷ *binary programs* (BPs) with $B = I = N$.

If we remove the integrality restrictions from an MIP, we obtain its *LP relaxation*:

Definition (LP relaxation of an MIP). Given a mixed integer program $\text{MIP} = (A, b, c, I)$, its *LP relaxation* is defined as

$$\text{(LP)} \quad \check{c} = \min \{c^T x \mid Ax \leq b,\ x \in \mathbb{R}^n\}\,.$$

$X_{\text{LP}} = \{x \in \mathbb{R}^n \mid Ax \leq b\}$ is the set of *feasible solutions* of the LP relaxation. An LP-feasible solution $\check{x} \in X_{\text{LP}}$ is called *LP-optimal* if $c^T \check{x} = \check{c}$.

The solution set of the LP relaxation defines a polyhedron $P = X_{\text{LP}}$. This *LP polyhedron* is a superset of its integer hull $P_I \subseteq P$, which is the convex hull

$$P_I = \text{conv}\{P \cap (\mathbb{Z}^I \times R^{N \setminus I})\} = \text{conv}\{X_{\text{MIP}}\}$$

of the MIP feasible solutions.

Chapter 5

Branching

Most of this chapter is joint work with Thorsten Koch and Alexander Martin. Parts of it were published in Achterberg, Koch, and Martin [5].

Since branching is in the core of any branch-and-bound algorithm, finding good strategies was important to practical MIP solving right from the beginning, see Bénichou et al. [39] or Mitra [165]. We refrain from giving details of all existing strategies, but concentrate on the most popular rules used in todays MIP solvers, in particular the ones that are available in SCIP. For a comprehensive study of branch-and-bound strategies we refer to Land and Powell [139], Linderoth and Savelsbergh [146], Fügenschuh and Martin [90], and the references therein.

The only way to split a problem Q within an LP based branch-and-bound algorithm is to branch on linear inequalities in order to keep the property of having an LP relaxation at hand. The easiest and most common inequalities are *trivial inequalities*, i.e., inequalities that split the feasible interval of a singleton variable, compare Figure 2.2 on page 17. To be more precise, if x_j, $j \in I$, is some integer variable with a fractional value $\check{x}_j$ in the current optimal LP solution, we obtain two subproblems: one by adding the trivial inequality $x_j \leq \lfloor \check{x}_j \rfloor$ (called the *left subproblem* or *left child*, denoted by Q_j^-) and one by adding the trivial inequality $x_j \geq \lceil \check{x}_j \rceil$ (called the *right subproblem* or *right child*, denoted by Q_j^+). This procedure of branching on trivial inequalities is also called *branching on variables*, because it only requires to change the bounds of variable x_j. Branching on more complicated inequalities or even splitting the problem into more than two subproblems are rarely incorporated into general MIP solvers, even though it can be effective in special cases, see, for instance, Borndörfer, Ferreira, and Martin [51], Clochard and Naddef [62], or Naddef [169]. SCIP supports general branching on constraints with an arbitrary number of children, but all of the branching rules included in the distribution branch on variables and create a binary search tree.

The basic algorithm for variable selection may be stated as follows:

Algorithm 5.1 Generic variable selection

Input: Current subproblem Q with an optimal LP solution $\check{x} \notin X_{\text{MIP}}$.
Output: An index $j \in I$ of an integer variable x_j with fractional LP value $\check{x}_j \notin \mathbb{Z}$.

1. Let $F = \{j \in I \mid \check{x}_j \notin \mathbb{Z}\}$ be the set of branching candidates.
2. For all candidates $j \in F$, calculate a score value $s_j \in \mathbb{R}$.
3. Return an index $j \in F$ with $s_j = \max_{k \in F}\{s_k\}$.

In the following we focus on the most common variable selection rules, which are all variants of Algorithm 5.1. The difference is how the score in Step 2 is computed.

The ultimate goal is to find a computationally inexpensive branching strategy that minimizes the number of branch-and-bound nodes that need to be evaluated.

Since no global approach is known, one tries to find a branching variable that is at least a good choice for the current branching. One way to measure the quality of a branching is to look at the changes in the objective function of the LP relaxations of the two children Q_i^- and Q_i^+ compared to the relaxation of the parent node Q. Recently, Patel and Chinneck [185] proposed to favor branchings that result in LP solutions $\check{x}_{Q_j^-}$ and $\check{x}_{Q_j^+}$ with a large Euclidean distance to the current LP solution $\check{x}_Q$. However, we follow the traditional way of trying to improve the dual bound.

In order to compare branching candidates, for each candidate the two objective function changes $\Delta_j^- := \check{c}_{Q_j^-} - \check{c}_Q$ and $\Delta_j^+ := \check{c}_{Q_j^+} - \check{c}_Q$ or estimates of these values are mapped on a single score value. This is typically done by using a function of the form

$$\text{score}(q^-, q^+) = (1 - \mu) \cdot \min\{q^-, q^+\} + \mu \cdot \max\{q^-, q^+\}, \tag{5.1}$$

see, for instance, Linderoth and Savelsbergh [146]. The *score factor* μ is some number between 0 and 1. It is usually an empirically determined constant, which is sometimes adjusted dynamically through the course of the algorithm (in SCIP we use a static value of $\mu = \frac{1}{6}$, which is also used in SIP [159]). In addition, SCIP features a new idea which is to calculate the score via a product

$$\text{score}(q^-, q^+) = \max\{q^-, \epsilon\} \cdot \max\{q^+, \epsilon\} \tag{5.2}$$

with $\epsilon = 10^{-6}$. This product is the default score function in SCIP, and the computational results in Section 5.11 show its superiority to the weighted sum of Equation (5.1). Bounding the values by ϵ is necessary to be able to compare two pairs (Δ_j^-, Δ_j^+) and (Δ_k^-, Δ_k^+) where one of the values is zero for each pair. There are a lot of MIP instances where such a behavior can be observed, typically for the downwards changes Δ^-.

In the forthcoming explanations all cases are symmetric for the left and right subproblem. Therefore we will only consider one direction most of the time, the other will be analogous.

5.1 Most Infeasible Branching

This still very common rule chooses the variable with fractional part closest to 0.5, i.e., $s_j = \phi(\check{x}_j) = \min\{\check{x}_j - \lfloor \check{x}_j \rfloor, \lceil \check{x}_j \rceil - \check{x}_j\}$. The heuristic reason behind this choice is that this selects a variable where the least tendency can be recognized to which "side" (up or down) the variable should be rounded. Unfortunately, as the numerical results in Section 5.11 indicate, the performance of this rule is in general not much better than selecting the variable randomly.

5.2 Least Infeasible Branching

In contrast to the *most infeasible branching* rule, the *least infeasible branching* strategy prefers variables that are close to integrality: $s_j = \max\{\check{x}_j - \lfloor \check{x}_j \rfloor, \lceil \check{x}_j \rceil - \check{x}_j\}$. Like *most infeasible branching*, this strategy yields a very poor performance.

5.3 Pseudocost Branching

This is a sophisticated rule in the sense that it keeps a history of the success of the variables on which already has been branched. This rule goes back to Bénichou et al. [39]. In the meantime various variations of the original rule have been proposed. In the following we present the one used in SCIP and SIP [159]. For alternatives see Linderoth and Savelsbergh [146].

Let ς_j^- and ς_j^+ be the objective gains per unit change in variable x_j at node Q after branching in the corresponding direction, that is

$$\varsigma_j^- = \frac{\Delta_j^-}{f_j^-} \quad \text{and} \quad \varsigma_j^+ = \frac{\Delta_j^+}{f_j^+} \tag{5.3}$$

with $f_j^+ = \lceil \check{x}_j \rceil - \check{x}_j$ and $f_j^- = \check{x}_j - \lfloor \check{x}_j \rfloor$. Let σ_j^+ denote the sum of ς_j^+ over all problems Q, where x_j has been selected as branching variable and the LP relaxation of Q_j^+ has already been solved and was feasible. Let η_j^+ be the number of these problems, and define σ_j^- and η_j^- to be the analogue values for the downwards branch. Then the pseudocosts of variable x_j are calculated as the arithmetic means

$$\Psi_j^- = \frac{\sigma_j^-}{\eta_j^-} \quad \text{and} \quad \Psi_j^+ = \frac{\sigma_j^+}{\eta_j^+}. \tag{5.4}$$

Using $s_j = \text{score}(f_j^- \Psi_j^-, f_j^+ \Psi_j^+)$ in Algorithm 5.1 yields what is called *pseudocost branching*.

Observe that at the beginning of the algorithm $\sigma_j^- = \eta_j^- = \sigma_j^+ = \eta_j^+ = 0$ for all $j \in I$. We call the pseudocosts of a variable $j \in I$ *uninitialized for the upward direction*, if $\eta_j^+ = 0$. Uninitialized upward pseudocosts are set to $\Psi_j^+ = \Psi_\varnothing^+$, where $\Psi_\varnothing^+$ is the average of the initialized upward pseudocosts over all variables. This average number is set to 1 in the case that all upward pseudocosts are uninitialized. We proceed analogously with the downward direction. The pseudocosts of a variable are called *uninitialized* if they are uninitialized in at least one direction.

5.4 Strong Branching

The idea of *strong branching* was developed in the context of the traveling salesman problem, see Applegate et al. [14]. Soon, it became a standard ingredient in mixed integer programming codes like Cplex. Strong branching means to test which of the fractional candidates gives the best progress in the dual bound before actually branching on any of them. This test is done by temporarily introducing an upper bound $x_j \le \lfloor \check{x}_j \rfloor$ and subsequently a lower bound $x_j \ge \lceil \check{x}_j \rceil$ for variable x_j with fractional LP value $\check{x}_j$ and solving the linear relaxations.

If we choose as candidate set the full set $F = \{j \in I \mid \check{x}_j \notin \mathbb{Z}\}$ and if we solve the resulting LPs to optimality, we call the strategy *full strong branching*. In other words, *full strong branching* can be viewed as finding the locally (with respect to the given score function) best variable to branch on. We will see in Section 5.11 that selecting this locally best variable usually works very well in practice w.r.t. the number of nodes needed to solve the problem instances.

Unfortunately the computation times per node of *full strong branching* are high. Accordingly, most branching rules presented in the literature, including *pseudocost*

branching, may be interpreted as an attempt to find a (fast) estimate of what *full strong branching* actually measures.

One possibility to speed up *full strong branching*, is to restrict the candidate set in some way, e.g., by considering only a subset $F' \subseteq F$ of the fractional variables. Another idea that can be found in the literature is to only perform a few simplex iterations to estimate the changes in the objective function for a specific branching decision. This seems reasonable, because usually the change of the objective function per iteration in the simplex algorithm decreases with an increasing number of iterations. Thus, the parameters of *strong branching* to be specified are the maximal size κ of the candidate set F', the maximum number γ of dual simplex iterations to be performed for each candidate variable, and a criterion according to which the candidate set is selected.

In SCIP as well as in SIP, the size of the candidate set is not fixed in advance to a specific (small) value, but the candidates are evaluated with a "look ahead" strategy: if no new best candidate was found for $\lambda = 8$ successive candidates, the evaluation process is stopped. By processing variables with largest *pseudocost* scores first, only the most promising candidates are evaluated. A maximum of $\kappa = 100$ strong branching candidate evaluations is imposed as a safeguard to avoid very expensive computations in exceptional situations.

The iteration limit for strong branching evaluations is set to $\gamma = 2\bar{\gamma}$ with a minimal value of 10 and a maximal value of 500 iterations, where $\bar{\gamma}$ is the average number of simplex iterations per LP needed so far. Note that for small or medium sized instances this number only protects from unexpected long simplex runs, and the candidate LPs will usually be solved to optimality. We observed that using such a large iteration limit typically does not produce a significant overhead. Instead, it often helps to produce better branching decisions or to derive variable fixings due to infeasible strong branching LPs.

5.5 Hybrid Strong/Pseudocost Branching

Even with the speedups indicated at the end of Section 5.4, the computational burden of *strong branching* is high, and the higher the speedup, the less precise the decisions are.

On the other hand, the weakness of *pseudocost branching* is that at the very beginning there is no information available, and s_j basically reflects only the fractionalities $\phi(\check{x}_j)$ for all variables $j \in F$. Many of the early nodes are located in the upper part of the search tree where the decisions have the largest impact on the structure of the tree and the subproblems therein. With *pseudocost branching*, these decisions are taken with respect to pseudocost values that are not useful yet.

To circumvent these drawbacks the positive aspects of *pseudocost* and *strong branching* are put together in the combination *hybrid strong/pseudocost branching*, where *strong branching* is applied in the upper part of the tree up to a given depth level d. For nodes with depth larger than d, *pseudocost branching* is used. This branching rule is available for example in LINGO [148]. In our implementation, we use $d = 10$.

Algorithm 5.2 Reliability branching

Input: Current subproblem Q with an optimal LP solution $\check{x} \notin X_{\text{MIP}}$.
Output: An index $j \in I$ of an integer variable x_j with fractional LP value $\check{x}_j \notin \mathbb{Z}$.

1. Let $F = \{j \in I \mid \check{x}_j \notin \mathbb{Z}\}$ be the set of branching candidates.
2. For all candidates $j \in F$, calculate the score $s_j = \text{score}(f_j^- \Psi_j^-, f_j^+ \Psi_j^+)$ and sort them in non-increasing order of their pseudocost scores.

   For at most κ candidates $j \in F$ with $\min\{\eta_j^-, \eta_j^+\} < \eta_{\text{rel}}$:
   (a) Perform a number of at most γ dual simplex iterations on subproblem Q_j^- and Q_j^+, respectively. Let $\tilde{\Delta}_j^-$ and $\tilde{\Delta}_j^+$ be the resulting gains in the objective value.
   (b) Update the pseudocosts Ψ_j^- and Ψ_j^+ with the gains $\tilde{\Delta}_j^-$ and $\tilde{\Delta}_j^+$.
   (c) Update the score $s_j = \text{score}(\tilde{\Delta}_j^-, \tilde{\Delta}_j^+)$.
   (d) If the maximum score $s^\star = \max_{k \in F}\{s_k\}$ has not changed for λ consecutive score updates, goto Step 3.
3. Return an index $j \in F$ with $s_j = \max_{k \in F}\{s_k\}$.

5.6 Pseudocost Branching with Strong Branching Initialization

The decisions of *pseudocost* as well as the ones of *hybrid strong/pseudocost branching* in the lower part of the tree are potentially based on uninitialized pseudocost values, leading to an inferior selection of branching variables.

The idea to avoid this risk, which goes back to Gauthier and Ribière [93] and which was further developed by Linderoth and Savelsbergh [146], is to use strong branching for variables with uninitialized pseudocosts and to use the resulting strong branching estimates to initialize the pseudocosts. In contrast to the fixed depth level of *hybrid strong/pseudocost branching*, this rule uses strong branching in a more dynamic way.

5.7 Reliability Branching

We generalize the idea of *pseudocost branching with strong branching initialization* by not only using strong branching on variables with uninitialized pseudocost values, but also on variables with *unreliable* pseudocost values. The pseudocosts of a variable x_j are called *unreliable*, if $\min\{\eta_j^-, \eta_j^+\} < \eta_{\text{rel}}$, with η_{rel} being the "reliability" parameter. We call this new branching rule *reliability branching*.

An outline of the selection of a branching variable with *reliability branching* is given in Algorithm 5.2 which implements Step 2 of Algorithm 5.1.

As in the *strong branching* rule we set the maximal number of strong branching initializations to $\kappa = 100$ and the maximal number of simplex iterations per subproblem to $\gamma = 2\bar{\gamma}$, bounded by $\gamma \geq 10$ and $\gamma \leq 500$.

The reliability parameter is usually set to $\eta_{\text{rel}} = 8$, but it is dynamically adjusted to control the total number of strong branching simplex iterations $\hat{\gamma}_{\text{SB}}$ compared to the total number of regular node LP simplex iterations $\hat{\gamma}_{\text{LP}}$. In the default

parameter settings our goal is to restrict the strong branching simplex iterations to a maximum of $\hat{\gamma}_{\text{SB}}^{\max} = \frac{1}{2}\hat{\gamma}_{\text{LP}} + 100000$. If they exceed $\hat{\gamma}_{\text{SB}} > \hat{\gamma}_{\text{SB}}^{\max}$, the reliability value is reduced to $\eta_{\text{rel}} = 0$ which turns off strong branching initializations such that *reliability branching* becomes equivalent to *pseudocost branching*. Within the interval $\hat{\gamma}_{\text{SB}} \in \left[\frac{1}{2}\hat{\gamma}_{\text{SB}}^{\max}, \hat{\gamma}_{\text{SB}}^{\max}\right]$ η_{rel} is linearly decreased from 8 to 1, with the extreme case $\eta_{\text{rel}} = 1$ corresponding to *pseudocost branching with strong branching initialization*. On the other hand, if $\hat{\gamma}_{\text{SB}}$ is very small, namely $\hat{\gamma}_{\text{SB}} < \frac{1}{2}\hat{\gamma}_{\text{LP}}$, η_{rel} is increased proportionally to $\frac{\hat{\gamma}_{\text{LP}}}{\hat{\gamma}_{\text{SB}}}$ such that *reliability branching* resembles *strong branching* in the limiting case of $\eta_{\text{rel}} \to \infty$.

5.8 Inference Branching

In a CSP or SAT instance where no objective function is available it does not make sense to base the branching decision on the change in the LP relaxation's objective value. Therefore one has to use a different measure to estimate the impact of a variable to the given problem instance. One idea is to select a branching variable that, after tightening its domain, produces the largest number of deductions on other variables.

Like with LP objective value based branching rules, the impact of a variable in terms of deductions can either be calculated directly in a strong branching fashion by explicitly propagating the bound changes of the branching candidates, or by collecting historical information similar to the pseudocost values. For example, the SAT solver SATZ [143, 144] takes the former approach. The *inference branching* rule of SCIP uses the latter idea.

The *inference value* of a variable x_j, $j \in I$, is defined analog to the pseudocosts of Equation (5.4) as

$$\Phi_j^+ = \frac{\varphi_j^+}{\nu_j^+} \tag{5.5}$$

where φ_j^+ is the total number of all inferences deduced after branching upwards on variable x_j, and ν_j^+ is the number of corresponding subproblems Q_j^+ for which domain propagation has already been applied. Note that ν_j^+ is very similar to the pseudocost counter η_j^+, but it needs not to be the same since there may be subproblems for which only domain propagation or only LP solving is applied. Additionally, pseudocosts are also collected by strong branching evaluations, while the inference history can be populated by probing and other presolving techniques, see Chapter 10.

Like *pseudocost branching*, the *inference branching* rule suffers from the fact that the most crucial decisions at the top of the search tree are based on very little information, since the inference values are collected during the search. Obviously, one could overcome this issue by combining explicit inference calculations and historical information analog to the *pseudocost branching with strong branching initialization* or even the *reliability branching* rule. However, we do not take this approach in SCIP. For general integer variables we define an uninitialized inference value to be zero. For (the usually more important) binary variables we use the information of the implication graph and clique table, see Section 3.3.5, to define replacements for uninitialized inference values. Let $D = (V, A)$ be the current implication graph as defined in Definition 3.9 and let $\mathcal{Q}(x_j = 1)$ be the set of cliques the variable is contained in as positive literal. Now, if an inference value of a binary variable is

uninitialized, i.e., $\nu_j^+ = 0$, we define

$$\Phi_j^+ = \left|\delta_D^-(x_j = 1)\right| + 2 \cdot \left|\mathcal{Q}(x_j = 1)\right|.$$

Note that the factor 2 applied to the number of cliques is an underestimate of the actual implications represented by the cliques since all cliques in the clique table are at least of cardinality 3.

If probing is used as a presolver, see Section 10.6, the implication graph gets populated by all implications of binary variables. Therefore, one can see probing in this regard as a strong branching initialization of the inference values at the root node for all binary variables. In this sense, our approach of dealing with uninitialized inference values is very similar to *pseudocost branching with strong branching initialization*.

5.9 Hybrid Reliability/Inference Branching

The *hybrid reliability/inference branching* rule combines the selection criteria of *reliability branching* and *inference branching*. Additionally, in the presence of conflict analysis, see Chapter 11, we include the score values of a *variable state independent decaying sum (VSIDS)* branching strategy as it is used in SAT solvers, see Moskewicz et al. [168]. This rule prefers variables that have been used in the conflict graph analysis to produce recent conflict constraints. Finally, we include a score value which is based on the number of infeasible or bound-exceeding subproblems (i.e., the number of cutoffs) that have been generated due to branching on the respective variable.

Let s_j^{reli}, s_j^{infer}, s_j^{conf}, and s_j^{cutoff} be the individual score values for the *reliability branching*, *inference branching*, *conflict branching*, and *cutoff branching* rules, respectively. The problem with combining these values into a single score is that they operate on completely different scales. In particular, the scale of s^{reli} is highly dependent on the problem instance, namely the objective function. Therefore one has to apply a normalization step to transform the individual score values onto a unified scale, for which we use the function

$$g : \mathbb{R}_{\geq 0} \to [0, 1), \quad g(x) = \frac{x}{x+1}.$$

As one can see in Figure 5.1 the function $g(\cdot)$ has its dynamic range roughly in the region between 0 and 4. It seems reasonable to first scale the different score values such that they are mapped into the dynamic range of $g(\cdot)$ and apply $g(\cdot)$ afterwards. Therefore, we combine the four values with the formula

$$s_j = \omega^{\text{reli}} g\left(\frac{s_j^{\text{reli}}}{s_\varnothing^{\text{reli}}}\right) + \omega^{\text{infer}} g\left(\frac{s_j^{\text{infer}}}{s_\varnothing^{\text{infer}}}\right) + \omega^{\text{conf}} g\left(\frac{s_j^{\text{conf}}}{s_\varnothing^{\text{conf}}}\right) + \omega^{\text{cutoff}} g\left(\frac{s_j^{\text{cutoff}}}{s_\varnothing^{\text{cutoff}}}\right)$$

in which the $s_\varnothing$ values are the current average values over all variables in the problem instance. The weights are set to $\omega^{\text{reli}} = 1$, $\omega^{\text{infer}} = \omega^{\text{cutoff}} = 10^{-4}$, and $\omega^{\text{conf}} = 10^{-2}$.

Besides the different calculation of the score values, the *hybrid reliability/inference branching* rule is equal to *reliability branching* as shown in Algorithm 5.2. Thus, one can view *reliability branching* as a special case of *hybrid reliability/inference branching* with $\omega^{\text{reli}} = 1$ and $\omega^{\text{infer}} = \omega^{\text{conf}} = \omega^{\text{cutoff}} = 0$. Since $g(\cdot)$ is strictly monotone, the application of $g(\cdot)$ to s^{reli} does not modify the branching variable selection.

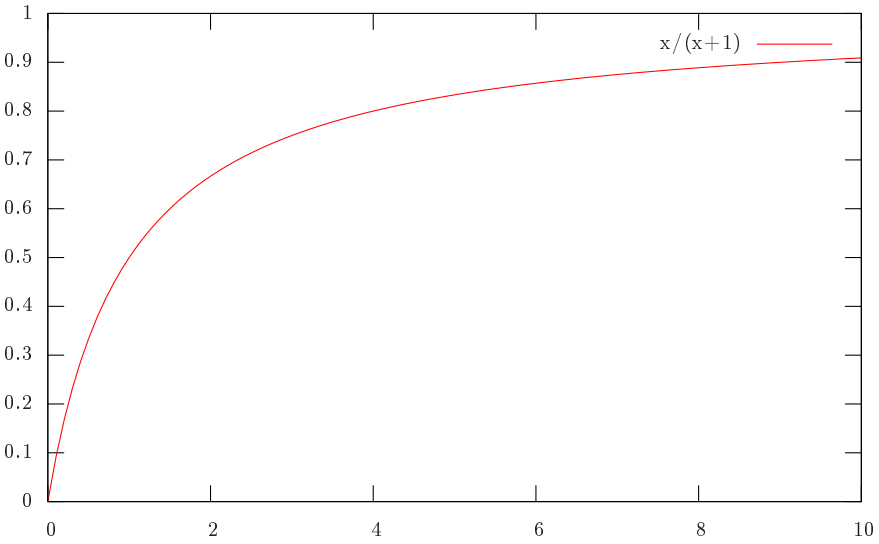

Figure 5.1. Function $g : \mathbb{R}_{\geq 0} \to [0, 1)$ to map branching scores into the unit interval.

5.10 Branching Rule Classification

Some of the proposed branching rules can be adjusted with parameter settings. All of the strategies using strong branching include the simplex iteration limit γ and the look ahead value λ. The *hybrid strong/pseudocost branching* exhibits an additional depth parameter d, while the *reliability branching* comes along with the reliability parameter η_{rel}.

It is interesting to note that depending on the parameter settings, the branching rules have interrelations as illustrated in Figure 5.2.

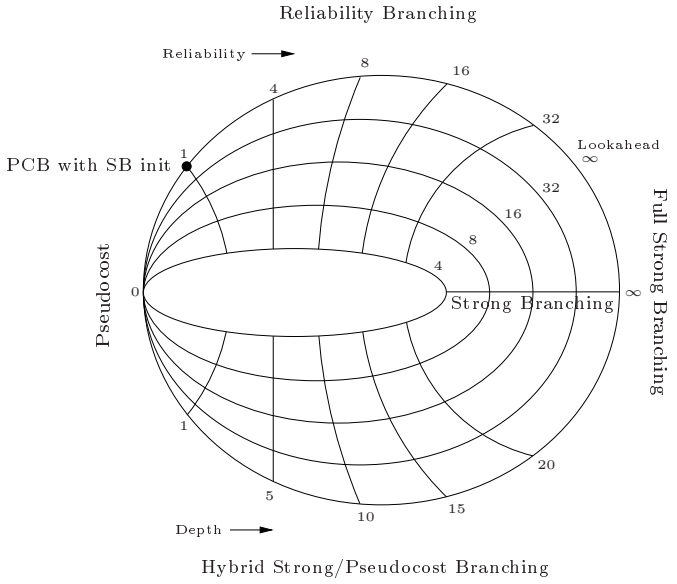

Figure 5.2. Interrelations between branching rules and their parameters.

	test set	random	most inf	least inf	pseudocost	full strong	strong	hybr strong	psc strinit	reliability	inference
time	MIPLIB	+139	+139	+266	+16	+92	+38	+20	+5	−1	+101
	CORAL	+332	+314	+575	+40	+97	+59	+27	+2	+7	+177
	MILP	+81	+86	+109	+23	+107	+44	+43	+9	+6	+20
	ENLIGHT	+115	−40	+149	−27	+45	+11	+9	+27	+5	−70
	ALU	+1271	+1991	+1891	+619	+180	+13	+11	+55	+36	−35
	FCTP	+288	+267	+379	+35	+36	+25	+4	+14	+2	+187
	ACC	+52	+85	+138	−41	+174	+82	+153	+11	+84	−24
	FC	+912	+1152	+837	+98	+14	+18	+14	−5	−2	+188
	ARCSET	+1276	+1114	+1296	+106	+112	+72	+38	+18	−1	+317
	MIK	+10606	+10606	+9009	+102	+59	+8	+11	+35	+2	+5841
	total	**+226**	**+219**	**+341**	**+33**	**+95**	**+44**	**+30**	**+8**	**+6**	**+95**
nodes	MIPLIB	+475	+341	+1096	+87	−65	−62	−18	+13	−7	+269
	CORAL	+694	+517	+1380	+79	−79	−68	−12	+18	+16	+329
	MILP	+194	+187	+306	+71	−72	−59	+41	+40	+7	+76
	ENLIGHT	+163	−29	+219	+3	−83	−85	+1	+23	−8	−49
	ALU	+6987	+5127	+9084	+1659	−60	−78	−31	+120	+17	+6
	FCTP	+511	+443	+931	+103	−73	−68	+6	+39	0	+364
	ACC	+393	+513	+1422	+88	−95	−52	+392	+31	+404	+33
	FC	+5542	+5060	+6039	+603	−81	−73	−28	+54	0	+1137
	ARCSET	+3219	+2434	+3573	+248	−60	−51	−6	+37	0	+742
	MIK	+8994	+7397	+9195	+123	−90	−86	+1	+32	−1	+4652
	total	**+543**	**+428**	**+976**	**+98**	**−75**	**−65**	**+3**	**+27**	**+9**	**+217**

Table 5.1. Performance effect of different branching rules for solving MIP instances. The values denote the percental changes in the shifted geometric mean of the runtime (top) and number of branching nodes (bottom) compared to the default *hybrid reliability/inference branching* rule. Positive values represent a deterioration, negative values an improvement.

Hybrid strong/pseudocost branching with $d = 0$ as well as *reliability branching* with $\eta_{\text{rel}} = 0$ coincide with pure *pseudocost branching*. With a static value of $\eta_{\text{rel}} = 1$, *reliability branching* is equal to *pseudocost branching with strong branching initialization*. If the depth d and the reliability η_{rel} are increased, the number of strong branching evaluations also increases, and with $d = \eta_{\text{rel}} = \infty$, both strategies converge to pure *strong branching*. Additionally, if the look ahead parameter is set to $\lambda = \infty$ and the maximal number of simplex iterations and candidates are also chosen as $\gamma = \kappa = \infty$, *strong branching* becomes *full strong branching*.

5.11 Computational Results

Table 5.1 summarizes the performance impact of the different branching rules presented in this Chapter. More detailed results can be found in Tables B.11 to B.20 in Appendix B. The test sets and the experimental setup is described in Appendix A.

The *least infeasible branching* rule gives the worst results, both for the time and the number of nodes. It is even worse than *random branching*. This suggests that *most infeasible branching* should be much better than *random branching*, but this is not the case. Since *pseudocost branching* has almost the same computational overhead but yields much better results, there is no reason at all to employ *most infeasible branching* for general mixed integer programming.

Full strong branching and *strong branching* need by far the smallest number of branching nodes over all strategies. One reason for this is that the solving of the strong branching sub-LPs is sometimes able to detect an infeasibility and can thereby tighten the bound of the corresponding variable. This infeasibility detection is not

counted as a search node in the statistics. Nevertheless, the results show that the "local greedy" procedure of *full strong branching* is a good strategy to produce a small search tree.

Unfortunately, the node reductions achieved by the extensive use of strong branching do not justify the runtime costs: on the diverse test sets MIPLIB, CORAL, and MILP, *full strong branching* is around 100 % slower while *strong branching* is still 50 % slower than *hybrid reliability/inference branching*. Although not that prominent, the effect is clearly visible on the other test sets as well, which consist of instances of a single problem class each.

Out of the strategies that combine pseudocosts and strong branching, namely *hybrid strong/pseudocost branching* ("hybr strong"), *pseudocost branching with strong branching initialization* ("psc strinit"), and *reliability branching*, the latter is the most successful on our test sets. *Hybrid strong/pseudocost branching* usually needs fewer nodes, but this cannot compensate the higher computational costs of strong branching applied up to depth $d = 10$ of the search tree. In contrast, in *reliability branching* the node reduction due to the more extensive use of strong branching pays off: compared to *pseudocost branching with strong branching initialization*, it also leads to a reduction in the runtime for most of the test sets.

The *inference branching* rule is usually inferior to *reliability branching*. However, it is the winner on the ENLIGHT, ALU, and ACC instances. The ENLIGHT test set consists of instances of a combinatorial game, in which the objective function does not play a significant role. The chip verification instances of the ALU test set only contain a completely artificial objective function. The instances of the ACC test set model a basketball scheduling problem (see Nemhauser and Trick [173]) which basically is a pure feasibility problem without objective function. In all cases, it is not surprising that pseudocosts do not yield a good evaluation of the branching candidates and that the number of inferences is a better choice. At least for the ALU and ACC instances, the incorporation of the inference history into the *reliability branching* rule is able to transfer some of the benefits of *inference branching* to the default *hybrid reliability/inference branching* rule. On the other test sets, *reliability branching* performs equally well.

BRANCHING SCORE FUNCTION

Table 5.2 summarizes the benchmarks to compare various branching score functions of type (5.1) against the default SCIP product score function (5.2). Detailed results can be found in Tables B.21 to B.30 in Appendix B.

Using the weighted sum score function

$$\text{score}(q^-, q^+) = (1 - \mu) \cdot \min\{q^-, q^+\} + \mu \cdot \max\{q^-, q^+\},$$

with a weight of $\mu = 0$ as suggested by Bénichou et al. [39] and Beale [37] means to choose a branching variable for which the minimum of the two individual score values q^- and q^+ is as large as possible. In the default *hybrid reliability/inference branching* rule, the largest contribution to the total score comes from the pseudocost estimates $\tilde{\Delta}^- = f_j^- \Psi_j^-$ and $\tilde{\Delta}^+ = f_j^+ \Psi_j^+$. Thus, using the weight $\mu = 0$ basically means to select a branching variable for which the smaller estimated objective increase is maximal. The idea behind this choice is to balance the search tree and to improve the global dual bound as fast as possible.

The other extreme case is to use $\mu = 1$. The rationale behind this setting is to drive one of the two children to infeasibility as fast as possible in order to restrict

	test set	min ($\mu = 0$)	weighted ($\mu = \frac{1}{6}$)	weighted ($\mu = \frac{1}{3}$)	avg ($\mu = \frac{1}{2}$)	max ($\mu = 1$)
time	MIPLIB	+26	+12	+28	+30	+53
	CORAL	+25	+20	+27	+49	+113
	MILP	+18	+10	+36	+35	+58
	ENLIGHT	+34	−19	−1	−9	+115
	ALU	+88	+66	+85	+101	+187
	FCTP	+72	−2	+6	+26	+56
	ACC	+43	+70	+29	+41	+50
	FC	+58	−7	−6	−4	−2
	ARCSET	+35	+11	+22	+32	+62
	MIK	+134	+13	+31	+52	+169
	total	**+29**	**+14**	**+29**	**+37**	**+75**
nodes	MIPLIB	+24	+21	+46	+73	+92
	CORAL	+25	+48	+53	+102	+199
	MILP	+19	+26	+69	+68	+97
	ENLIGHT	+12	−13	+15	+2	+116
	ALU	+85	+104	+96	+165	+357
	FCTP	+71	+8	+12	+34	+68
	ACC	+100	+390	+222	+235	+391
	FC	+147	−12	−6	−35	−10
	ARCSET	+57	+20	+37	+60	+109
	MIK	+131	+16	+34	+63	+193
	total	**+31**	**+34**	**+54**	**+76**	**+130**

Table 5.2. Performance effect of different branching score functions for solving MIP instances. The values denote the percental changes in the shifted geometric mean of the runtime (top) and number of branching nodes (bottom) compared to the default product score function (5.2). Positive values represent a deterioration, negative values an improvement.

the growth of the search tree. In the best case, one child turns out to be infeasible which means that we have avoided the node duplication for this branching step.

The results for the two settings (columns "min" and "max") show that improving the global dual bound and balancing the tree with $\mu = 0$ is much more successful than the infeasibility idea of $\mu = 1$. The "min" approach is also superior to using the average value as shown in column "avg", which was proposed by Gauthier and Ribière [93]. However, the best setting for the weight μ is located between 0 and $\frac{1}{2}$, which was also reported in earlier computational studies, see Linderoth and Savelsbergh [146]. They found the value of $\mu = \frac{1}{3}$ to be the most successful. In contrast, the value $\mu = \frac{1}{6}$ that Martin [159] used in SIP seems to be the best choice for SCIP on the considered test sets.

Although we tried several values for the weight μ, none of them can compete against the product score function (5.2), which is the default strategy in SCIP. The product is clearly superior to all variants of the weighted sum score function. Even the best of them is outperformed by more than 10 %. As to the author's knowledge, using a product based score function is a new idea that has not been proposed previously in the literature.

Chapter 6

Node Selection

After a subproblem has been processed, the solving process can continue with any subproblem that is a leaf of the current search tree. The selection of the subproblem that should be processed next has two usually opposing goals within the MIP branch-and-bound search:

1. finding good feasible MIP solutions to improve the primal (upper) bound, which helps to prune the search tree by bounding, and

2. improving the global dual (lower) bound.

Besides by employing primal heuristics, feasible MIP solutions can be found as solutions to LP relaxations of subproblems that happen to be integral. It can be observed, see for example Linderoth and Savelsberg [146], that integral LP solutions are typically found very deep in the search tree. Therefore, to quickly identify feasible solutions of a MIP instance, strategies like *depth first search* seem to be the most natural choice. The second goal, however, is completely disregarded by *depth first search*, since the nodes with the best (i.e., smallest) lower bound are usually close to the root node of the search tree. To improve the global dual bound as fast as possible, one should use *best first search* which is to always select a leaf with the currently smallest dual objective value. Trying to achieve both goals at the same time leads to a mixture of the two strategies, which is called *best first search with plunging*.

A variant of *best first search* is the so-called *best estimate search*. This strategy does not select the node with the best dual bound, but the one with the best estimated objective value of the feasible solutions contained in the corresponding subtree. The estimate is calculated from the dual bound and the fractionalities and pseudocost values of the variables (see Section 5.3). The goal of *best estimate search* is to find a good, preferably optimal feasible solution as soon as possible. Naturally, this strategy can also be combined with *depth first search*, which leads to *best estimate search with plunging*. Finally, one can combine all the three strategies into hybrid versions, for example by applying the selection rules in an interleaving fashion or by using weighted combinations of the individual node selection score values.

In the following sections, we will take a closer look at the different strategies and evaluate them by computational experiments.

6.1 Depth First Search

Depth first search was proposed by Little et al. [149] for the traveling salesman problem and by Dakin [71] for mixed integer programming. This node selection rule always chooses a child of the current node as the next subproblem to be processed. If the current node is pruned and therefore has no children, the search backtracks to the most recent ancestor that has another unprocessed child left and selects one

of its children. Thus, one selects always a node from the leaf queue with maximal depth in the search tree.

Depth first search is the preferred strategy for pure feasibility problems like SAT or CSP. Since these problems do not have an objective function, the solving process is solely focused on finding a feasible solution, and there is no need to increase a lower objective bound. But besides its ability to identify feasible solutions, *depth first search* has a second advantage: the next subproblem to be processed is almost always very similar to the current one. Therefore, the subproblem management is reduced to a minimum. In particular, only very small changes have to be applied to the LP relaxation. If the branching is performed on variables by splitting a bound of an integer variable into two parts, the only update in the LP is one bound change of a single variable. This can be done very efficiently in simplex solvers. Most notably, the current basis matrix factorization remains valid for the child problem, thereby saving the time for basis refactorization.

A third advantage of *depth first search* is its small memory consumption. If the current node is in depth d, then the search tree consists of at most $b \cdot d + 1$ nodes that are not yet pruned, where b is the maximal number of children of a node. This means, the number of nodes we have to store for a 2-way branching scheme never exceeds $2d_{\max} + 1$, with $d_{\max}$ being the maximal depth of the search tree. For binary or mixed binary programs, the maximal depth is bounded by $d_{\max} \leq |B|$ if we branch on variables, which means the memory consumption for node data structures is linear in the number of binary variables.

Using *depth first search* as node selection strategy leaves open only one additional choice, namely which of the unsolved children of the current node should be processed first. SAT solvers like BERKMIN [100] try to use this freedom of choice to balance their conflict clause databases (see Chapter 11) with respect to the appearance of the branching variable and its negation in the conflict clauses. This is basically achieved by first selecting the fixing of the branching variable that appears in the larger number of conflict clauses, since new conflict clauses derived in this subtree can only contain the negation of the branching variable.

The idea of Martin [159] for mixed integer programming (using the usual branching on integer variables with fractional LP value) is to select the branch that pushes the LP value of the variable further away from its value in the root node's LP solution. Say, for example, that we branched on variable x_j with current LP solution $\check{x}_j = 2.3$, which has an LP solution value of $(\check{x}_R)_j = 2.6$ in the root node's relaxation. Martin's idea is that on the path to the current node the value of the variable has the tendency to be pushed towards 2, and that the branch $x_j \leq 2$ should therefore be inspected first. Although we did not verify this by thorough computational studies, we experienced that this child selection strategy yields very good results and therefore adopted it in SCIP.

6.2 BEST FIRST SEARCH

Best first search aims at improving the global dual bound as fast as possible by always selecting a subproblem with the smallest dual bound of all remaining leaves in the tree. As a side-effect, this strategy leads to a minimal number of nodes that need to be processed, given that the branching rule is fixed. Note that *best first search* is not uniquely defined as there may be multiple nodes with equal dual bound.

Proposition 6.1. Given an instance of a constraint integer program and a fixed branching strategy in the branch-and-bound Algorithm 2.1, there exists a node selection strategy of *best first search* type which solves the instance in a minimal number of nodes.

Proof. Due to the fixed branching strategy, the search tree is uniquely defined, including all nodes that could be pruned by bounding if the optimal solution value $c^\star$ was known. A node selection strategy σ defines an order of the nodes in the tree such that for all nodes Q_i with parent $p(Q_i)$ we have $\sigma(p(Q_i)) < \sigma(Q_i)$. Let $\underline{\check{c}}_{Q_i}$ be the lower bound of node Q_i before it is processed. Certainly, $\underline{\check{c}}_{Q_i} \geq \check{c}_{p(Q_i)}$ with $\check{c}_Q$ being the optimal value of a relaxation Q_{relax} of subproblem Q, for example, the LP relaxation of Q.

Let $\sigma^\star$ be an optimal node selection strategy with respect to the number of nodes that have to be processed. Assume that $\sigma^\star$ is not of *best first search* type. Then there are node indices i, j with $\sigma(Q_i) < \sigma(Q_j)$ and $\underline{\check{c}}_{Q_i} > \underline{\check{c}}_{Q_j}$. Since Q_i is processed in the optimal node selection strategy, the optimal value for the CIP must be $c^\star \geq \underline{\check{c}}_{Q_i} > \underline{\check{c}}_{Q_j}$. Therefore, Q_j cannot be pruned by bounding and must also be processed. Thus, we can exchange Q_i and Q_j in the node selection order without increasing the number of processed nodes. By iteratively applying this exchange procedure, $\sigma^\star$ can be converted into a *best first search* node selection strategy that is still optimal. □

Note. Although there always exists an optimal *best first search* node selection strategy, not every *best first search* rule processes a minimal number of nodes. As an example, assume that $\mathcal{L} = \{Q_1, Q_2\}$ is the current list of open subproblems, and both subproblems have a lower bound of $\underline{\check{c}}_{Q_1} = \underline{\check{c}}_{Q_2} = \underline{\check{c}} < \hat{c}$ with $\hat{c}$ being the value of the current incumbent solution. Furthermore, suppose that we apply a best first node selection strategy that picks Q_1 as the next subproblem. If after the processing of Q_1 the relaxation value is $\check{c}_{Q_1} > \underline{\check{c}}_{Q_2}$, but processing of Q_2 leads to $\check{c}_{Q_2} = \underline{\check{c}}_{Q_2}$ and the detection of a solution with value $c^\star = \check{c}_{Q_2}$, then we have processed Q_1 unnecessarily: if we had processed Q_2 first, we would have found the solution and pruned Q_1 due to $\underline{\check{c}}_{Q_1} \geq c^\star$ without processing it.

The best first node selection strategy can be efficiently implemented by storing the leaves of the search tree in a priority queue, see Section 3.3.6. The question remains which node one should select if there are multiple nodes that have dual bounds equal to the global dual bound. In order to find good feasible solutions early, we select the node with better estimate, see Section 6.4 below. If there are still ties left, we try to stay close to the previous subproblem and favor child nodes of the current node over its siblings, and siblings over the remaining leaves of the tree.

6.3 Best First Search with Plunging

As said in the introduction of this chapter, *best first search* leads to a small number of processed nodes, while *depth first search* tends to produce feasible solutions earlier and speeds up the node solving process due to the closer resemblance of successive subproblems. The idea of plunging is to mix both strategies in order to combine their benefits. As long as the current node has unprocessed children, one of them is selected as the next node. Otherwise, plunging continues with one of the current

node's siblings. If no more unprocessed children or siblings are available, the current plunge is completed, and a leaf from the tree with best dual bound is selected.

The branching tree data structures of SCIP are specifically tailored to support this strategy, see Section 3.3.6. The children and the siblings of the current node are stored separately from the remaining leaves and can therefore easily be accessed and identified. The remaining leaves are stored in a priority queue which enables efficient access to the best node corresponding to a certain ordering. For *best first search* the ordering is defined by the dual bounds of the leaves.

Again, the question remains which child or sibling should be processed next during plunging. Usually, the child nodes inherit the dual bound of their parent node, which means that they cannot be differentiated with respect to their dual bounds. As plunging is mainly focused on finding feasible solutions, we apply—as in *depth first search*—the LP solution guided rule of Martin [159]. We use Martin's rule even if strong branching (see Section 5.4) produced different dual bounds for the child nodes, which would enable a best first selection.

The largest disadvantage of *depth first search* is its high risk of processing superfluous nodes that would have been pruned if a good solution was known earlier. Plunging has the same property, although to a much smaller extent, since after each plunge the search continues with a node of smallest lower bound. Nevertheless, it might be profitable to prematurely abort a plunge if the local lower bound approaches the primal bound. This would avoid the processing of some of the superfluous nodes, but the disadvantage is that one may miss small improvements in the primal bound.

The strategy of SCIP is to mainly use plunging for its effect of faster node processing due to close resemblance of subproblems. The identification of primal solutions is left to the primal heuristics, namely the diving heuristics which basically continue the plunging outside the branching tree, but apply different variable and value selection rules that are particularly tailored for finding feasible solutions; see Chapter 9 for an overview of the primal heuristics included in SCIP. During each plunge, we perform a certain minimal number of plunging steps, but we abort the plunging after a certain total number of steps or if the local relative gap

$$\gamma(Q) = \frac{\check{c}_Q - \underline{\check{c}}}{\hat{c} - \underline{\check{c}}}$$

of the current subproblem Q exceeds the threshold $\gamma_{\max} = 0.25$. Here, $\check{c}_Q$ is the lower bound of the subproblem, $\underline{\check{c}}$ is global lower bound, and $\hat{c}$ is the global upper bound, i.e., the value of the incumbent solution or infinity. The minimal and maximal number of plunging steps are adjusted dynamically to be equal to $0.1 \cdot d_{\max}$ and $0.5 \cdot d_{\max}$, respectively, with $d_{\max}$ being the maximum depth of all processed nodes. This means that in the beginning of the search almost no plunging is performed, but the extent of plunging is increased during the course of the algorithm.

6.4 BEST ESTIMATE SEARCH

The nodes selected by *best first search* have good dual bounds, but their LP solutions are usually far away from integrality. *Depth first search* quickly leaves the region of good objective values but might be able to find feasible solutions faster. Therefore, none of the two node selection rules aims at finding *good* solutions. This is addressed by *best estimate search*. This rule calculates estimate values e_Q for the feasible

solutions that might be contained in the subtrees represented by the leaves Q of the tree, and it selects a node that minimizes this estimate. The estimate combines information about the dual bound and the integrality of the LP solution.

The literature basically proposes two different estimate schemes. The *best projection* criterion of Bénichou et al. [39] calculates an estimate

$$e_Q^{\text{proj}} = \check{c}_Q + \left(\frac{\hat{c} - \check{c}_R}{\phi(\check{x}_R)}\right) \phi(\check{x}_Q),$$

in which $\check{c}_Q$ is the dual bound of the current subproblem, $\check{c}_R$ the dual bound of the root node, $\hat{c}$ the incumbent solution value, $\check{x}_Q$ the current LP solution, and $\check{x}_R$ the LP solution at the root node. The fractionality $\phi(\check{x})$ of a vector $\check{x} \in \mathbb{R}^n$ is defined as $\phi(\check{x}) = \sum_{j=1}^{n} \phi(\check{x}_j)$ with $\phi(\check{x}_j) = \min\{\check{x}_j - \lfloor \check{x}_j \rfloor, \lceil \check{x}_j \rceil - \check{x}_j\}$. The interpretation of the best projection estimate is that one calculates the objective value increase per unit decrease in the fractionality of the LP solution for the root node and assumes that the current LP solution can be driven to integrality with the same objective value increase per unit of fractionality. Note that the best projection method needs a globally valid upper bound.

The *best estimate* rule of Forrest et al. [88] employs the pseudocost values of the variables (see Section 5.3) to estimate the increase in the objective value. This pseudocost-based estimate is defined as

$$e_Q = \check{c}_Q + \sum_{j \in I} \min\left\{\Psi_j^- f_j^-, \Psi_j^+ f_j^+\right\}$$

with $f_j^- = \check{x}_j - \lfloor \check{x}_j \rfloor$ and $f_j^+ = \lceil \check{x}_j \rceil - \check{x}_j$ being the distances to the nearest integers for the current LP solution value $\check{x}_j$ for variable x_j, and Ψ_j^- and Ψ_j^+ being the pseudocost values of variable x_j for rounding downwards and upwards, respectively. Assuming that the pseudocosts are reliable indicators for the per unit objective value increase for shifting a variable downwards or upwards, the best estimate rule calculates the estimated minimum value of a rounded solution.

Linderoth and Savelsbergh [146] give computational indication that the *best estimate* rule is superior to the *best projection* rule. Therefore, we only implemented the *best estimate* rule.

6.5 Best Estimate Search with Plunging

As for the *best first search* node selection, we can combine *best estimate search* with *depth first search* by a plunging strategy. Again, child or sibling nodes are selected until either all of them have been pruned or the plunge is aborted due to the criteria presented in Section 6.3. As before, the goal is to transfer the node processing speedup of *depth first search* regarding the closely resembled successive subproblems to the more sophisticated *best estimate search* strategy.

6.6 Interleaved Best Estimate/Best First Search

The aim of *best estimate search* is to quickly find good feasible solutions. After an optimal solution has been found, the node selection strategy has (despite its interaction with other solver components) no more impact on the number of nodes

that have to be processed, since the remaining nodes of the tree, defined by the branching strategy, have to be processed anyway and the order does not matter. Due to time restrictions, however, the user might not want to wait until the optimality of the solution has been proven, but is already satisfied with a given quality guarantee. Therefore, the progression of the global dual bound plays a significant role. In this regard, *best estimate search* can perform very poor. Suppose the node with the best dual bound has a large fractionality measure. That would lead to a rather large estimate for the node, which means that the node is not processed for a very long time. The global dual bound would stay at the same level as long as this node is not touched.

The solution to this problem is to interleave best first and *best estimate search* and combine this with plunging. The resulting strategy proceeds as best estimate with plunging, but every bestfreq plunge we choose a node with the best dual bound instead of one with a best estimate as the next subproblem. We use bestfreq = 10 as default value in our implementation.

6.7 Hybrid Best Estimate/Best First Search

A second approach of increasing the importance of the global dual bound in the best estimate node selection rule is to calculate the node selection score as a weighted sum of the *best estimate* and *best first* scores. In this rule we are selecting a node Q that minimizes

$$\omega\, e_Q + (1-\omega)\check{c}_Q \tag{6.1}$$

with $\omega \in [0,1]$. Again, this node selection strategy is combined with plunging. We chose $\omega = 0.1$ as the default, which means that a larger weight is put on the dual bound and the estimate e_Q is only influencing the decision among nodes with very similar dual bounds.

6.8 Computational Results

In this section we present computational results to compare the various node selection strategies on several sets of mixed integer programming instances. The test sets are described in Appendix A. Detailed results can be found in Tables B.31 to B.40 in Appendix B.

Table 6.1 summarizes the benchmark results. As expected, pure *best first search* ("bfs") yields the smallest search trees. Note, however, that Proposition 6.1 cannot be applied since the branching strategy is affected by the order in which the nodes are processed. In particular, the pseudocosts at a certain node will vary for different node selection rules. Thus, it is not always the case that *best first search* needs the fewest branching nodes.

The second expected behavior can also be clearly observed: *depth first search* ("dfs") produces much larger search trees than *best first search* but can compensate this disadvantage by a faster node processing time. Overall, the performances of *best first* and *depth first search* are roughly similar.

The column labeled "bfs/plunge" shows that combining *best first search* and *depth first search* into the *best first search with plunging* strategy indeed yields the desired result: although not as small as for pure *best first search*, the search trees are much

	test set	dfs	bfs	bfs/plunge	estimate	estim/plunge	hybrid
time	MIPLIB	+22	+28	+3	+14	−3	+8
	CORAL	+49	+38	+2	+19	−3	−5
	MILP	+12	+20	+1	+8	+5	+3
	ENLIGHT	+53	+74	+53	0	−11	+25
	ALU	−40	+101	+25	+113	+25	+3
	FCTP	+59	+25	+9	+22	−2	+7
	ACC	+102	−7	−7	−7	+8	+7
	FC	−4	+14	+2	+6	+1	+5
	ARCSET	+152	+53	+11	+42	+5	+13
	MIK	−7	+44	+3	+39	−2	+6
	total	**+28**	**+30**	**+4**	**+16**	**0**	**+3**
nodes	MIPLIB	+64	−18	+3	−15	−3	+7
	CORAL	+143	−28	0	−9	+3	−9
	MILP	+41	−12	+2	−9	+9	+8
	ENLIGHT	+109	+16	+36	+2	+7	+13
	ALU	−43	+50	+29	+87	+17	+6
	FCTP	+74	−6	+2	−5	−3	+2
	ACC	+425	−30	−7	−29	+25	+27
	FC	−9	−31	+9	−32	+9	+12
	ARCSET	+249	+6	0	+19	+7	+3
	MIK	+19	+3	−4	+6	+2	0
	total	**+78**	**−17**	**+3**	**−9**	**+4**	**+2**

Table 6.1. Performance effect of different node selection strategies for solving MIP instances. The values denote the percental changes in the shifted geometric mean of the runtime (top) and number of branching nodes (bottom) compared to the default *interleaved best estimate/best first search* strategy. Positive values represent a deterioration, negative values an improvement.

smaller than for *depth first search*. Since most of the speedup of *depth first search* for processing the nodes carries over to *plunging*, the combined node selection rule is superior to the individual strategies.

The *best estimate search* rule ("estimate") turns out to be slightly faster than pure *best first search*. A possible explanation is that *best estimate search* implicitly follows the idea of plunging: usually, nodes Q that are located deeper in the tree have a smaller fractionality $\phi(\check{x}_Q)$ and, related to that, a smaller "estimate penalty" $e_Q - \check{c}_Q$. Therefore, it is more likely than for *best first search* that a child or sibling of the previous node is selected as next subproblem. Nevertheless, performing the plunging explicitly as in *best first search with plunging* is superior to the implicit plunging of *best estimate search*. This can also be seen in the column "estim/plunge": *best estimate search with plunging* is much faster than pure *best estimate search*. Even more, it slightly outperforms *best first search with plunging*.

As indicated by the mostly positive values in the table, the default *interleaved best estimate/best first search* is the best overall strategy. It needs fewer nodes than *best estimate search with plunging* and achieves very similar runtimes. On the ALU testset, however, it performs much better. The reason might be that the ALU instances are infeasible MIPs, and estimate based node selection rules, which are tailored to find feasible solutions earlier, are therefore not suited for these instances.

Like the default *interleaved* node selection rule, *hybrid best estimate/best first search* ("hybrid") combines all three ideas, *depth first*, *best first*, and *best estimate search*. It is, however, slower than the interleaved approach and can only achieve a performance similar to *best first search with plunging*. As in *best estimate search* and its plunging variant, processing of a node with a dual bound equal to the global dual bound might be delayed very long with the *hybrid* rule. This can slow down the decrease of the optimality gap, and the user has to wait longer until a specified solution quality is proven. Therefore, *interleaved best estimate/best first search*

seems to be the better choice even though it might be possible to improve the performance of *hybrid best estimate/best first search* by altering the weight $\omega \in [0, 1]$ in Equation (6.1).

Note. In the version of SCIP used in this thesis, there is a significant performance bottleneck associated with long leaf queues and node selection rules that are not based on *best first search*. Thus, the performance of *best estimate*, *best estimate with plunging*, *hybrid best estimate/best first search*, and the default *interleaved best estimate/best first search* suffers from that issue on instances that take many nodes and produce large leaf queues during the run. In particular, many of the instances in ENLIGHT, ALU, and MIK are of this type. Therefore, the "true" performance of the estimate based node selection rules on these test sets would be better than the values in Table 6.1 indicate.

CHILD SELECTION

As the default node selection strategy involves plunging, we have to define a child selection rule in order to decide which of the two children of the current node should be processed next. The main goal of the child selection is to pick a direction which leads to the finding of a feasible solution, preferably of small objective value. We compare the following strategies:

- ▷ *Downwards selection* always chooses the downwards branch $x_j \leq \lfloor \check{x}_j \rfloor$.
- ▷ *Upwards selection* always chooses the upwards branch $x_j \geq \lceil \check{x}_j \rceil$.
- ▷ *Pseudocost selection* prefers the direction that yields a smaller pseudocost estimate $f_j^- \Psi_j^-$ or $f_j^+ \Psi_j^+$ for the LP objective value deterioration, see Section 5.3. The idea is to guide the search into the area of better objective values.
- ▷ *LP value selection* means to round the branching variable x_j to the integer that is closer to its current LP solution value $\check{x}_j$. Thus, it selects the downwards branch if $f_j^- = \check{x}_j - \lfloor \check{x}_j \rfloor \leq \frac{1}{2}$ and the upwards branch otherwise.
- ▷ *Root LP value selection* denotes the idea of Martin [159] which we already mentioned in Section 6.1. It compares the current LP value $\check{x}_j$ of the branching variable to its LP value $(\check{x}_R)_j$ in the root node and supports the movement of the value in its current direction: if $\check{x}_j \leq (\check{x}_R)_j$, the variable is branched downwards. Otherwise, the upwards branch is inspected first.
- ▷ *Inference selection* chooses the direction in which the branching variable has a larger inference history value Φ_j^- or Φ_j^+. These values denote the average number of deductions derived from branching the variable into the respective direction (see Section 5.8). The hope is that the branch with larger inference history value produces more domain propagations and thus a smaller subproblem for which it is easier to either find a feasible solution or prove the infeasibility.
- ▷ *Hybrid inference/root LP value selection* is a combination of the *inference* and *root LP value* selection rules. It chooses the downwards branch if

$$(\Phi_j^- + \epsilon) \cdot ((\check{x}_R)_j - \check{x}_j + 1) \geq (\Phi_j^+ + \epsilon) \cdot (\check{x}_j - (\check{x}_R)_j + 1) \tag{6.2}$$

	test set	down	up	pseudocost	LP value	root LP value	inference
time	MIPLIB	+23	0	+12	+7	0	+4
	CORAL	+10	−11	+6	−2	−5	−4
	MILP	+14	−5	+10	+12	−8	−5
	ENLIGHT	−1	+3	−2	−11	−8	+3
	ALU	−12	−3	+11	+5	+41	−12
	FCTP	+4	−7	+4	+6	−6	−2
	ACC	−19	−1	+20	−15	+13	−7
	FC	+1	−3	−5	−3	−4	−1
	ARCSET	−2	+3	+6	+1	+6	+4
	MIK	−8	+8	+19	+17	+2	−1
	total	**+11**	**−5**	**+9**	**+4**	**−3**	**−2**
nodes	MIPLIB	+18	−1	−8	+1	−2	−4
	CORAL	+14	−19	−1	−9	−14	+1
	MILP	+8	−9	+10	+15	−9	−12
	ENLIGHT	−4	+20	−3	−9	−3	+7
	ALU	−6	−20	+27	+3	+29	+6
	FCTP	+2	−11	−2	−1	−9	−2
	ACC	−13	+35	+64	−18	+21	+27
	FC	+4	−1	−9	+1	−7	+3
	ARCSET	+5	−2	+5	0	+7	+3
	MIK	+13	−5	+12	+20	+12	−5
	total	**+10**	**−9**	**+2**	**+1**	**−6**	**−3**

Table 6.2. Performance effect of different child selection strategies for solving MIP instances. The values denote the percental changes in the shifted geometric mean of the runtime (top) and number of branching nodes (bottom) compared to the default *hybrid inference/root LP value selection*. Positive values represent a deterioration, negative values an improvement.

and the upwards branch otherwise. Here, $\Phi_j^-, \Phi_j^+ \in \mathbb{R}_{\geq 0}$ are the inference history values for the downwards and upwards direction, respectively, and $\epsilon = 10^{-9}$ is the zero tolerance. The selection Inequality (6.2) means that a variable that has moved at least by one unit downwards or upwards from the root node to the current subproblem will be pushed further into that direction, independent from the inference values. On the other hand, the inference values dominate the selection for variables with current LP values $\check{x}_j$ that are close to their root LP values $(\check{x}_R)_j$.

Table 6.2 depicts a summary of the experiments. Appendix B provides detailed results in Tables B.41 to B.50. The comparison of the pure downwards ("down") and upwards ("up") preferences shows that consequentially branching upwards is clearly superior, both in the solving time and the number of branching nodes. A possible explanation is that most MIP instances contain binary variables that represent decisions for which the "yes" case (i.e., setting $x_j = 1$) has a much larger impact on the model as the "no" case (i.e., $x_j = 0$). It seems natural that taking the crucial decisions early by fixing a variable to $x_j = 1$ gives better chances to end up with a feasible solution. In contrast, postponing the crucial decisions by first ruling out some options with $x_j = 0$ most probably leads to a situation in which we cannot satisfy all constraints with the remaining alternatives.

The bad performance of the *pseudocost selection* rule ("pseudocost") is somewhat surprising as it seems to be the natural choice in the spirit of best first and best estimate search. On the other hand, many MIP models have costs associated to setting a binary variable to $x_j = 1$. Thus, upwards pseudocosts tend to be larger than downwards pseudocosts, such that the *pseudocost selection* rule is more similar to *downwards selection* than to *upwards selection*. Therefore, the above explanation for the inferior performance of *downwards selection* also applies to the pseudocost

based rule.

The *LP value selection*, although better than *pseudocost* and *downwards selection*, is also inferior to the *upwards selection* rule. In contrast, the *root LP value selection* strategy of Martin [159] is comparable to *upwards selection* but has the advantage that it is insensitive against complementation: if we complement all integer variables of a model by setting $x'_j := l_j + u_j - x_j$ for $j \in I$ (assuming that the bounds are finite), the *upwards selection* rule would turn into the inferior *downwards selection* while the *root LP value selection* strategy would produce the same results as before—at least if all other components of the solver would behave equivalently in the complemented variable space.

The *inference selection* rule is another alternative to *upwards selection* of similar performance. As the *root LP value selection*, it is invariant under variable complementation. However, it is more tailored to pure feasibility problems without meaningful objective function since the inference values are based on feasibility arguments. In contrast, the LP solution values include both feasibility and optimality considerations. The effect can be seen on the ALU and ACC instances. The ALU test set consists of infeasible MIPs with an artificial objective function, while ACC is a collection of basically pure feasibility problems without objective function. Here, *inference selection* performs much better than *root LP value selection*. For the other test sets, however, *inference selection* is slightly inferior to *root LP value selection*.

The default strategy, *hybrid inference/root LP value selection*, also achieves similar performance as *upwards selection*, *root LP value selection*, and *inference selection*. As one can see from the negative values in the columns of the other three rules, however, it is slightly inferior. The attempt to combine the positive effects of *root LP value selection* and *inference selection* did not succeed. It might be that a different way to combine the two ideas results in an improved performance. In particular, it seems that Inequality (6.2) is biased too much towards the inference history values Φ_j^- and Φ_j^+. Therefore, one should increase the summand ϵ in the inequality to a much larger value in order to put more weight on the root LP value difference.

Chapter 7

Domain Propagation

Domain propagation denotes the task of tightening the domains of variables by inspecting the constraints and the current domains of other variables at a local subproblem in the search tree. In the MIP community, this process is usually called *node preprocessing*. In fact, one can see domain propagation as a restricted version of presolving, see Chapter 10. The main restriction for the operations applied to local nodes is that they must not modify the constraints. In particular, the deletion of variables is not allowed. Instead, one only tightens the domains of the variables, since this can be done without a large bookkeeping and LP management overhead.

Since the LP relaxation is not able to handle holes inside a domain, MIP solvers are only using bound propagation, i.e., one tries to deduce tighter lower and upper bounds for the variables.

Besides the integrality restrictions, there is only one type of constraints in a mixed integer program, namely the linear constraints. Therefore, the domain propagation methods implemented in the linear constraint handler are a superset of the methods for the more specialized constraint classes like the knapsack or the set covering constraints. The structure of these specific constraints can, however, be exploited in order to implement more efficient domain propagation algorithms.

In addition to the constraint based (primal) domain propagation techniques, SCIP features two dual domain reduction methods that are driven by the objective function, namely the *objective propagation* and the *root reduced cost strengthening*.

7.1 Linear Constraints

In SCIP we treat linear constraints in the form

$$\underline{\beta} \leq a^T x \leq \overline{\beta}$$

with the left and right hand sides $\underline{\beta}, \overline{\beta} \in \mathbb{R} \cup \{\pm\infty\}$ and $a \in \mathbb{R}^n$ being the coefficients of the constraint. Obviously, equations can be modeled by $\underline{\beta} = \overline{\beta}$. For inequalities one has typically either $\underline{\beta} = -\infty$ or $\overline{\beta} = +\infty$, but so-called *ranged rows* with both sides being finite and $\underline{\beta} < \overline{\beta}$ are also possible.

Bound propagation for linear constraints in SCIP is performed as explained in the "basic preprocessing techniques" of Savelsbergh [199]. The main idea is very simple, but the implementation gets a little more involved if infinite bounds, numerical issues, and runtime performance have to be considered.

The most important notion in this regard is the concept of *activity bounds*:

Definition 7.1 (activity bounds). Given a linear constraint $\underline{\beta} \leq a^T x \leq \overline{\beta}$, let

$$\underline{\alpha} := \min\{a^T x \mid \tilde{l} \leq x \leq \tilde{u}\} \quad \text{and} \quad \overline{\alpha} := \max\{a^T x \mid \tilde{l} \leq x \leq \tilde{u}\}$$

be the minimal and maximal activity $a^T x$ of the linear constraint with respect to the local bounds $\tilde{l} \leq x \leq \tilde{u}$ of the current subproblem Q. The values $\underline{\alpha}$ and $\overline{\alpha}$ are

called *activity bounds*. Furthermore, let

$$\underline{\alpha}_j := \min\{a^T x - a_j x_j \mid \tilde{l} \le x \le \tilde{u}\} \quad \text{and} \quad \overline{\alpha}_j := \max\{a^T x - a_j x_j \mid \tilde{l} \le x \le \tilde{u}\}$$

be the *activity bound residuals* for the scalar product $a^T x$ over all variables but x_j.

The activity bounds and the residuals can be easily calculated by inserting the lower or upper bounds of the variables into the product $a^T x$, depending on the sign of the coefficients a_j. Note that they can be infinite if the local bounds $\tilde{l}$, $\tilde{u}$ of the variables are infinite.

The propagations are based on the following observations:

1. If $\underline{\beta} \le \underline{\alpha}$, the left hand side is redundant and can be replaced by $-\infty$ without changing the set of feasible solutions in the subproblem or worsening the dual bound of the LP relaxation.
2. If $\overline{\alpha} \le \overline{\beta}$, the right hand side is redundant and can be replaced by $+\infty$.
3. If $\underline{\beta} \le \underline{\alpha}$ and $\overline{\alpha} \le \overline{\beta}$, the constraint is redundant and can be removed.
4. If $\underline{\beta} > \overline{\alpha}$ or $\underline{\alpha} > \overline{\beta}$, the constraint cannot be satisfied within the local bounds and the current subproblem is infeasible.
5. For all $j = 1, \ldots, n$ we have

$$\frac{\underline{\beta} - \overline{\alpha}_j}{a_j} \le x_j \le \frac{\overline{\beta} - \underline{\alpha}_j}{a_j} \quad \text{if } a_j > 0 \quad \text{and}$$
$$\frac{\overline{\beta} - \underline{\alpha}_j}{a_j} \le x_j \le \frac{\underline{\beta} - \overline{\alpha}_j}{a_j} \quad \text{if } a_j < 0,$$

   and we can tighten the bounds of x_j accordingly. If x_j is an integer variable, $j \in I$, the lower bounds can be rounded up and the upper bounds can be rounded down.

Inside domain propagation, we only apply Reductions 3 to 5. Despite removing some degeneracies in the LP there is no benefit in relaxing the constraint sides, but the additional management overhead would be considerable. The redundancy detection of Reduction 3, however, is useful since we can completely ignore such constraints in future propagations within the whole subtree defined by the current subproblem.

It is easy to see that the repeated application of Reductions 4 and 5 suffices to obtain bound consistency, see Definition 2.7 on page 22. Although it is very likely that the following propositions are old results, we did not find them in the literature. Therefore, we provide proofs.

Proposition 7.2. A linear constraint $\underline{\beta} \le a^T x \le \overline{\beta}$ on variables $x \in \mathbb{R}^n$, $x_j \in \mathbb{Z}$ for $j \in I$, with bounds $\tilde{l} \le x \le \tilde{u}$, $\tilde{l}_j, \tilde{u}_j \in \mathbb{R}$, $\tilde{l}_j, \tilde{u}_j \in \mathbb{Z}$ for $j \in I$, is bound consistent if and only if Reductions 4 and 5 cannot be applied to detect infeasibility or to tighten a variable's domain.

Proof. Since all domains of the variables are non-empty, bound consistency implies that there is a vector $\hat{x} \in [\tilde{l}, \tilde{u}]$ with $\underline{\beta} \le a^T \hat{x} \le \overline{\beta}$. Therefore, $\underline{\beta} \le a^T \hat{x} \le \overline{\alpha}$ and $\underline{\alpha} \le a^T \hat{x} \le \overline{\beta}$, which means that Reduction 4 cannot be applied. Assume that bound

consistency holds, but Reduction 5 can be applied on variable x_j. Consider the case $a_j > 0$ and $(\underline{\beta} - \overline{\alpha}_j)/a_j > \tilde{l}_j$. Bound consistency yields a feasible solution $\hat{x}$ with $\hat{x}_j = \tilde{l}_j$. Then we have

$$a_j\tilde{l}_j < \underline{\beta} - \overline{\alpha}_j \leq \underline{\beta} - (a^T\hat{x} - a_j\hat{x}_j) = \underline{\beta} - a^T\hat{x} + a_j\tilde{l}_j \leq a_j\tilde{l}_j,$$

which is a contradiction. The other three cases can be shown analogously.

Now suppose that neither Reduction 4 nor Reduction 5 can be applied. In order to prove bound consistency, we have to show that for each bound $\tilde{l}_j$, $\tilde{u}_j$, $j \in N$, there exists a (potentially fractional) *support vector* $\hat{x} \in [\tilde{l}, \tilde{u}]$ with $\hat{x}_j = \tilde{l}_j$ or $\hat{x}_j = \tilde{u}_j$, respectively, that satisfies the constraint. Consider variable x_j with $a_j > 0$ and its lower bound $\tilde{l}_j$. Let

$$x_k^{\min} := \begin{cases} \tilde{l}_j & \text{if } k = j \\ \tilde{l}_k & \text{if } a_k \geq 0 \\ \tilde{u}_k & \text{if } a_k < 0 \end{cases} \quad \text{and} \quad x_k^{\max} := \begin{cases} \tilde{l}_j & \text{if } k = j \\ \tilde{u}_k & \text{if } a_k \geq 0 \\ \tilde{l}_k & \text{if } a_k < 0 \end{cases}.$$

Since Reduction 4 is not applicable, we have

$$a^T x^{\min} = a_j\tilde{l}_j + \underline{\alpha}_j = \underline{\alpha} \leq \overline{\beta},$$

and because Reduction 5 is not applicable, it follows

$$a^T x^{\max} = a_j\tilde{l}_j + \overline{\alpha}_j \geq \underline{\beta}.$$

If one of $x^{\min}$ or $x^{\max}$ is contained in $[\underline{\beta}, \overline{\beta}]$, it is a valid support vector for $\tilde{l}_j$, and we are done. Otherwise, the only remaining possibility is

$$a^T x^{\min} < \underline{\beta} \leq \overline{\beta} < a^T x^{\max}.$$

In this case, consider the affine linear function

$$\alpha(t) = a^T(x^{\min} + t(x^{\max} - x^{\min})).$$

This is a continuous function $\alpha : [0, 1] \to \mathbb{R}$ with $\alpha(0) < \underline{\beta} \leq \overline{\beta} < \alpha(1)$. Therefore, there exists $t^\star \in (0, 1)$ with $\underline{\beta} \leq \alpha(t^\star) \leq \overline{\beta}$, and the vector

$$x^\star = x^{\min} + t^\star(x^{\max} - x^{\min})$$

is a valid support vector for $\tilde{l}_j$. The other cases for upper bounds $\tilde{u}_j$ and negative coefficients a_j follow with analogous reasoning. □

In the case that one of the constraint sides $\underline{\beta}$ or $\overline{\beta}$ is infinite, we can even achieve the stronger notion of interval consistency by applying Reductions 4 and 5:

Corollary 7.3. A linear constraint $\underline{\beta} \leq a^T x$ or $a^T x \leq \overline{\beta}$ on variables $x \in \mathbb{R}^n$, $x_j \in \mathbb{Z}$ for $j \in I$, with bounds $\tilde{l} \leq x \leq \tilde{u}$, $\tilde{l}_j, \tilde{u}_j \in \mathbb{R}$, $\tilde{l}_j, \tilde{u}_j \in \mathbb{Z}$ for $j \in I$, is interval consistent if and only if Reductions 4 and 5 cannot be applied to detect infeasibility or to tighten a variable's domain.

Proof. In the proof of Proposition 7.2 the case

$$a^T x^{\min} < \underline{\beta} \leq \overline{\beta} < a^T x^{\max}$$

cannot appear. If $\underline{\beta} = -\infty$, the vector $x^{\min}$ is integral for $j \in I$ and supports $\tilde{l}_j$. If $\overline{\beta} = +\infty$, the vector $x^{\max}$ is integral for $j \in I$ and supports $\tilde{l}_j$. □

Unfortunately, there is little hope for an efficient algorithm to achieve interval consistency in the general case of linear constraints:

Proposition 7.4. Deciding interval consistency for linear constraints of the form $\underline{\beta} \leq a^T x \leq \overline{\beta}$ on variables $x_j \in [l_j, u_j]$ and $x_j \in \mathbb{Z}$ for $j \in I$ is $\mathcal{NP}$-complete.

Proof. We provide a reduction from the subset sum problem which is $\mathcal{NP}$-complete (see Garey and Johnson [92]). Given a set of integers $a_j \in \mathbb{Z}_{>0}$, $j = 1, \ldots, n$, and an integer $b \in \mathbb{Z}$, the task of the subset sum problem is to decide whether there is a subset $S \subseteq N = \{1, \ldots, n\}$ with $\sum_{j \in S} a_j = b$. For $b = 0$ or $b = a^T \mathbb{1}$, the instance has the trivial solutions $S = \emptyset$ or $S = N$, respectively, and for $b < 0$ or $b > a^T \mathbb{1}$, the instance is obviously infeasible. Thus, we assume $0 < b < a^T \mathbb{1}$.

Given such an instance (a, b) of the subset sum problem, consider the linear constraint

$$0 \leq -by - (a^T \mathbb{1})z + a^T x \leq 0 \tag{7.1}$$

with domains $y, z, x_j \in \{0, 1\}$, $j \in N$. The vectors $(y = 0, z = 0, x = 0)$ and $(y = 0, z = 1, x = \mathbb{1})$ are feasible integral solutions that support the lower bounds $l_y = l_z = l_{x_j} = 0$, $j \in N$, and the upper bounds $u_z = u_{x_j} = 1$, $j \in N$, respectively. Thus, Constraint (7.1) is interval consistent if and only if there exists a feasible integral solution $(y^\star, z^\star, x^\star)$ with $y^\star = 1$. Such a solution must have $z^\star = 0$, because $b > 0$. Therefore, interval consistency of Constraint (7.1) is equivalent to the existence of $x^\star \in \{0, 1\}^n$ with $a^T x^\star = b$, which in turn is equivalent to the existence of $S \subseteq N$ with $\sum_{j \in S} a_j = b$. □

Propositions 7.2 and 7.4 provide the basic theoretical background for domain propagation of linear constraints. In the remaining part of the section, we will focus on the implementational issues.

With respect to performance, the most crucial parts are to update the activity bounds $\underline{\alpha}$ and $\overline{\alpha}$ instead of recalculating them from scratch at every node and to only process those constraints where the activity bounds have been changed since the last propagation round. In order to accomplish these goals, the linear constraint handler interacts with an event handler (see Section 3.1.10) to update $\underline{\alpha}$ and $\overline{\alpha}$ and to mark each constraint that is affected by a bound change of a variable.

The contributions of infinite bounds $\tilde{l}_j$ and $\tilde{u}_j$ to $\underline{\alpha}$ and $\overline{\alpha}$ are accumulated in separate counters. These counters are updated whenever a bound of a variable switches between a finite and an infinite value. If the counter is positive, the actual value stored in $\underline{\alpha}$ or $\overline{\alpha}$ is ignored and instead treated as infinity.

Tightened bounds deduced by one constraint are used in the domain propagation of other constraints to deduce further bound tightenings. Due to this iterative nature, numerical rounding errors can easily accumulate and produce wrong results, in particular if coefficients of very different magnitude are involved in the calculations. To avoid numerical errors, we slightly relax the newly calculated bounds $\tilde{l}_j$ and $\tilde{u}_j$ by using

$$\tilde{l}_j \leftarrow 10^{-5} \lfloor 10^5 \tilde{l}_j + \hat{\delta} \rfloor \quad \text{and} \quad \tilde{u}_j \leftarrow 10^{-5} \lceil 10^5 \tilde{u}_j - \hat{\delta} \rceil \tag{7.2}$$

with $\hat{\delta} = 10^{-6}$ being the feasibility tolerance parameter. This operation rounds (within the feasibility tolerance) the values to the five most significant digits after the decimal point.

A risk of wasting time on iterated domain propagation originates from general integer variables with large domains. Consider the artificial example

$$0.2 \leq x - y \leq 0.8$$

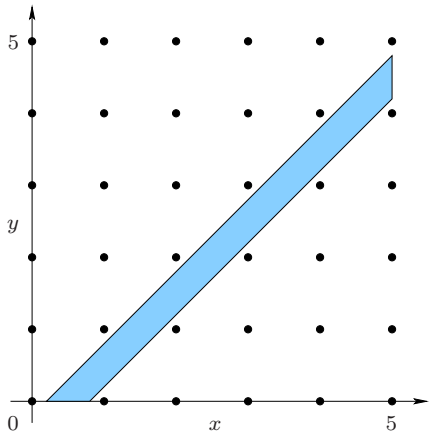

Figure 7.1. Feasible region of the linear constraint $0.2 \leq x - y \leq 0.8$ and bounds $0 \leq x, y \leq 5$.

with two variables $x, y \in \{0, \ldots, 1000\}$, and pretend that presolving did not find the obvious reduction. The feasible region of the constraint (disregarding the integrality conditions) is a larger version of to the one that is illustrated in Figure 7.1. Using the left hand side of the constraint, domain propagation would tighten the upper bound of y to $y \leq 999$. Afterwards, the right hand side can be used to deduce $x \leq 999$, which in turn leads to $y \leq 998$ due to the left hand side. This can be continued until the infeasibility of the problem is detected after 1000 iterations.

To avoid such long chains of small domain changes, we enforce a minimum size for the interval that is cut off from the domain: bound changes $\tilde{l}_j \to \tilde{l}'_j$ and $\tilde{u}_j \to \tilde{u}'_j$ are only accepted if they change the bound from an infinite to a finite value or if they satisfy

$$\begin{aligned} \tilde{l}'_j &> \tilde{l}_j + 0.05 \cdot \max\left\{\min\{\tilde{u}_j - \tilde{l}_j, |\tilde{l}_j|\}, 1\right\} \quad \text{or} \\ \tilde{u}'_j &< \tilde{u}_j - 0.05 \cdot \max\left\{\min\{\tilde{u}_j - \tilde{l}_j, |\tilde{u}_j|\}, 1\right\}, \end{aligned} \tag{7.3}$$

respectively. Note that this restriction still allows for bound changes on variables with only one finite bound like the commonly used non-negative variables $0 \leq x_j \leq +\infty$. The bound change is accepted as long as it is large enough relative to the width of the domain or the magnitude of the bound.

The whole domain propagation procedure is summarized in Algorithm 7.1. Each linear constraint possesses a "propagated" flag which is initially set to 0. It marks constraints that have not been affected by bound changes since the last propagation round. Consequently, if the flag is set to 1, we can skip the domain propagation for this constraint in Step 1. Step 2 marks the constraint as propagated by setting the "propagated" flag to 1. The flag will be automatically reset to 0 by the associated event handler (see Algorithm 7.2) whenever a bound of a variable with a non-zero coefficient $a_j \neq 0$ is modified, in particular if the linear constraint propagation itself modifies the bounds of the involved variables. Since the flag is set to 1 in Step 2 prior to the actual propagation of the constraint, it is possible that the propagations of the constraint trigger another propagation round on the same constraint.

Step 3 performs the possible bound strengthenings for variables with positive coefficient $a_j > 0$ as described previously. Step 4 treats the variables with negative coefficients. Finally, Steps 5 and 6 check for the infeasibility and redundancy of the constraint.

As already said, the domain propagation of linear constraints closely interacts

Algorithm 7.1 Domain Propagation for Linear Constraints

Input: Linear constraint $\underline{\beta} \leq a^T x \leq \overline{\beta}$, current local bounds $\tilde{l} \leq x \leq \tilde{u}$, and current activity bounds $\underline{\alpha}$ and $\overline{\alpha}$.

Output: Tightened local bounds for x.

1. If the constraint is already marked as propagated, abort.
2. Mark the constraint as propagated.
3. For all variables x_j with $a_j > 0$:
   (a) Calculate residual activities $\underline{\alpha}_j := \underline{\alpha} - a_j\tilde{l}_j$ and $\overline{\alpha}_j := \overline{\alpha} - a_j\tilde{u}_j$.
   (b) If $\underline{\alpha}_j > -\infty$ and $\overline{\beta} < +\infty$:
      i. Set $\tilde{u}'_j := (\overline{\beta} - \underline{\alpha}_j)/a_j$.
      ii. Set $\tilde{u}'_j := 10^{-5}\lceil 10^5\tilde{u}'_j - \hat{\delta}\rceil$.
      iii. If $j \in I$, set $\tilde{u}'_j := \lfloor \tilde{u}'_j + \hat{\delta}\rfloor$.
      iv. If $\tilde{u}'_j < \tilde{u}_j - 0.05 \cdot \max\left\{\min\{\tilde{u}_j - \tilde{l}_j, |\tilde{u}_j|\}, 1\right\}$, tighten $\tilde{u}_j := \tilde{u}'_j$.
   (c) If $\overline{\alpha}_j < +\infty$ and $\underline{\beta} > -\infty$:
      i. Set $\tilde{l}'_j := (\underline{\beta} - \overline{\alpha}_j)/a_j$.
      ii. Set $\tilde{l}'_j := 10^{-5}\lfloor 10^5\tilde{l}'_j + \hat{\delta}\rfloor$.
      iii. If $j \in I$, set $\tilde{l}'_j := \lceil \tilde{l}'_j - \hat{\delta}\rceil$.
      iv. If $\tilde{l}'_j > \tilde{l}_j + 0.05 \cdot \max\left\{\min\{\tilde{u}_j - \tilde{l}_j, |\tilde{l}_j|\}, 1\right\}$, tighten $\tilde{l}_j := \tilde{l}'_j$.
4. For all variables x_j with $a_j < 0$:
   (a) Calculate residual activities $\underline{\alpha}_j := \underline{\alpha} - a_j\tilde{u}_j$ and $\overline{\alpha}_j := \overline{\alpha} - a_j\tilde{l}_j$.
   (b) If $\underline{\alpha}_j > -\infty$ and $\overline{\beta} < +\infty$:
      i. Set $\tilde{l}'_j := (\overline{\beta} - \underline{\alpha}_j)/a_j$.
      ii. Set $\tilde{l}'_j := 10^{-5}\lfloor 10^5\tilde{l}'_j + \hat{\delta}\rfloor$.
      iii. If $j \in I$, set $\tilde{l}'_j := \lceil \tilde{l}'_j - \hat{\delta}\rceil$.
      iv. If $\tilde{l}'_j > \tilde{l}_j + 0.05 \cdot \max\left\{\min\{\tilde{u}_j - \tilde{l}_j, |\tilde{l}_j|\}, 1\right\}$, tighten $\tilde{l}_j := \tilde{l}'_j$.
   (c) If $\overline{\alpha}_j < +\infty$ and $\underline{\beta} > -\infty$:
      i. Set $\tilde{u}'_j := (\underline{\beta} - \overline{\alpha}_j)/a_j$.
      ii. Set $\tilde{u}'_j := 10^{-5}\lceil 10^5\tilde{u}'_j - \hat{\delta}\rceil$.
      iii. If $j \in I$, set $\tilde{u}'_j := \lfloor \tilde{u}'_j + \hat{\delta}\rfloor$.
      iv. If $\tilde{u}'_j < \tilde{u}_j - 0.05 \cdot \max\left\{\min\{\tilde{u}_j - \tilde{l}_j, |\tilde{u}_j|\}, 1\right\}$, tighten $\tilde{u}_j := \tilde{u}'_j$.
5. If $\underline{\beta} > \overline{\alpha}$ or $\underline{\alpha} > \overline{\beta}$, the current subproblem is infeasible.
6. If $\underline{\beta} \leq \underline{\alpha}$ and $\overline{\alpha} \leq \overline{\beta}$, delete the constraint from the current subproblem.

with an event handler, which is shown in Algorithm 7.2. This event handler catches all bound changes that are applied on variables that appear with non-zero coefficient $a_j \neq 0$ in the constraint. Whenever such a bound change was performed, the event handler updates the activity bounds and marks the constraint such that it will be propagated again in the next propagation round.

Algorithm 7.2 Event Handler for Linear Constraints

Input: Linear constraint $\underline{\beta} \leq a^T x \leq \overline{\beta}$, a variable x_j for which the bounds have been changed from $[\tilde{l}_j, \tilde{u}_j]$ to $[\tilde{l}'_j, \tilde{u}'_j]$, and current activity bounds $\underline{\alpha}$ and $\overline{\alpha}$.

Output: Updated activity bounds $\underline{\alpha}$ and $\overline{\alpha}$.

1. If $a_j > 0$:
   (a) Update $\underline{\alpha} := \underline{\alpha} + a_j(\tilde{l}'_j - \tilde{l}_j)$.
   (b) Update $\overline{\alpha} := \overline{\alpha} + a_j(\tilde{u}'_j - \tilde{u}_j)$.
2. If $a_j < 0$:
   (a) Update $\underline{\alpha} := \underline{\alpha} + a_j(\tilde{u}'_j - \tilde{u}_j)$.
   (b) Update $\overline{\alpha} := \overline{\alpha} + a_j(\tilde{l}'_j - \tilde{l}_j)$.
3. Mark the constraint as not propagated.

7.2 Knapsack Constraints

Binary knapsack constraints are of the form

$$a^T x \leq \overline{\beta} \tag{7.4}$$

with $a \in \mathbb{Z}^B_{\geq 0}$, $x_j \in \{0, 1\}$ for $j \in B$, and $\overline{\beta} \in \mathbb{Z}_{\geq 0}$. In common terminology, the coefficients a_j are called *weights*, and the right hand side $\overline{\beta}$ is the *capacity* of the knapsack. The constraint requires to select a subset of the *items* x_j such that their total weight $a^T x$ does not exceed the capacity $\overline{\beta}$.

As all other constraint types used in mixed integer programming, knapsack constraints are a special case of the linear constraints and could be treated by the same algorithms. Since only binary variables are involved and the coefficients and right hand side are integers, specialized data structures and algorithms can improve the memory and runtime performance. In particular, the weights and capacity are stored as integers instead of floating point values, and the calculations are executed in integer arithmetic.

Note that knapsack constraints cover more general constraints than those of the form given in Equation (7.4): every linear constraint with only one finite side that consists of only binary variables and rational coefficients can be transformed into a knapsack constraint by

- ▷ multiplying the constraint with -1 if $\underline{\beta} > -\infty$ and $\overline{\beta} = +\infty$,
- ▷ scaling it with the smallest common multiple of the coefficients' denominators,
- ▷ complementing variables with negative coefficients by $\bar{x}_j := 1 - x_j$, and
- ▷ rounding down the right hand side.

SCIP automatically performs these transformations in the presolving of linear constraints, see Section 10.1, such that all knapsack constraints are represented in their standard form (7.4).

The following observations help to improve the domain propagation for knapsack constraints:

Algorithm 7.3 Domain Propagation for Knapsack Constraints

Input: Knapsack constraint $a^T x \leq \overline{\beta}$, current local bounds $\tilde{l} \leq x \leq \tilde{u}$, and current minimal activity $\underline{\alpha}$.

Output: Tightened local bounds for x.

1. If the constraint is already marked as propagated, abort.
2. Mark the constraint as propagated.
3. If $\underline{\alpha} > \overline{\beta}$, the current subproblem is infeasible.
4. Make sure that the coefficients are sorted such that $a_{j_1} \geq a_{j_2} \geq \ldots$.
5. Set $w_0 := 0$.
6. For $k = 1, \ldots, |B|$:
   (a) If $a_{j_k} \leq \overline{\beta} - \underline{\alpha}$, break the loop.
   (b) If $\tilde{l}_{j_k} = 0$, set $\tilde{u}_{j_k} := 0$ and $w_0 := w_0 + a_{j_k}$.
7. If $\sum_{j \in B} a_j - w_0 \leq \overline{\beta}$, delete the constraint from the current subproblem.

1. The only propagation that can be applied is

$$\tilde{l}_j = 0 \ \wedge \ \underline{\alpha} + a_j > \overline{\beta} \ \ \rightarrow \ \ \tilde{u}_j = 0.$$

2. If we cannot fix a variable x_j with $\tilde{l}_j = 0$ to zero, it is not possible to apply propagations on variables x_k with $a_k \leq a_j$.

Due to Observation 1, we do not need to track the maximal activity $\overline{\alpha}$ for knapsack constraints. The maximal activity can be used to detect redundancy of the constraint, but this does not justify the additional overhead for tracking its value. Observation 2 indicates that we should sort the variables by non-increasing weight a_j and stop the propagation process if we reach an index j where $\underline{\alpha} + a_j \leq \overline{\beta}$. Note that the propagation only fixes variables to zero and thus the minimal activity $\underline{\alpha}$ does not change during the propagation. Therefore, the propagation can be implemented as a simple scan through the weights which fixes all variables to zero for which $\tilde{l}_j = 0$ and $a_j > \overline{\beta} - \underline{\alpha}$. It is shown in Algorithm 7.3.

Steps 1 and 2 check and set the "propagate" flag as in the linear constraint propagation. Step 3 checks for infeasibility due to the exceedance of the capacity by the variables currently fixed to one. Step 4 sorts the variables by non-increasing weight. Usually, this has only to be performed once during the whole solving process. It might, however, happen in presolving that some coefficients are modified, see Section 10.2. Since domain propagation is called as a subroutine of the presolving algorithm, the sorting may therefore be performed multiple times.

The local variable w_0 sums up the weights of variables that are fixed to zero. It is initialized in Step 5 and updated during the propagation loop in Step 6. Note, however, that it is not calculated exactly if the loop is aborted prematurely. Therefore, the redundancy detection in Step 7 can miss certain cases. The advantage of this approach is that we usually can abort the propagation loop at the first iteration and the work spent in the domain propagation algorithm is very low.

As the linear constraint handler, the knapsack constraint handler interacts with an event handler that tracks the bound changes of the involved variables, see Algorithm 7.4. In contrast to the event handler for linear constraints, it only needs

Algorithm 7.4 Event Handler for Knapsack Constraints

Input: Knapsack constraint $a^T x \leq \bar{\beta}$, a variable x_j for which the lower bound has been changed from $\tilde{l}_j$ to $\tilde{l}'_j$, and current minimal activity $\underline{\alpha}$.

Output: Updated minimal activity $\underline{\alpha}$.

1. Update $\underline{\alpha} := \underline{\alpha} + a_j(\tilde{l}'_j - \tilde{l}_j)$.
2. Mark the constraint as not propagated.

to catch changes on the lower bounds, because all weights are non-negative and the maximal activity does not need to be updated. The event handler updates the minimal activity as usual and marks the constraint as not being propagated.

7.3 Set Partitioning and Set Packing Constraints

Set partitioning and set packing constraints are used to model restrictions in which from a certain set of items exactly one or at most one, respectively, has to be selected. Such constraints are very common in several applications, for example to model the graph coloring problem (see Mehrotra and Trick [163], or Hansen, Labbé, and Schindl [113]). They can be stated as

$$\sum_{j \in S} x_j = 1 \quad \text{(set partitioning)}$$

and

$$\sum_{j \in S} x_j \leq 1 \quad \text{(set packing)},$$

with binary variables $x_j \in \{0, 1\}$, $j \in S$, and $S \subseteq B$. As for knapsack constraints, scaling of the equation or inequality and complementing some of the binary variables may help to convert a general linear constraint into the form of a set partitioning or set packing constraint.

As they are specializations of linear constraints, set partitioning and set packing constraints could be dealt with by the linear constraint handler. However, since all coefficients of included variables are 1 and the left and right hand sides are also fixed, the constraint data only consists of the set of included variables and can therefore be stored much more compactly than in the linear constraint handler. More important, domain propagation can be implemented more efficiently.

In the set packing case, there is only one possibility of domain propagation, namely

$$x_k = 1 \quad \rightarrow \quad \forall j \in S \setminus \{k\} : x_j = 0.$$

For set partitioning, the additional propagation rule

$$\forall j \in S \setminus \{k\} : x_j = 0 \quad \rightarrow \quad x_k = 1$$

has to be considered. In order to apply these rules, we only have to count the number of variables currently fixed to zero and one, respectively. This is done with the help of an event handler. Having these numbers at hand, the propagation is very easy, as can be seen in Algorithm 7.5.

Algorithm 7.5 Domain Propagation for Set Partitioning/Packing Constraints

Input: Set partitioning constraint $\sum_{j \in S} x_j = 1$ or set packing constraint $\sum_{j \in S} x_j \leq 1$, current local bounds $\tilde{l} \leq x \leq \tilde{u}$, current number of variables x_j, $j \in S$, fixed to zero (F_0) and one (F_1).

Output: Tightened local bounds for x.

1. If $F_1 \geq 2$, the current subproblem is infeasible.
2. If $F_1 = 1$, fix all variables $j \in S$ with $\tilde{l}_j = 0$ to zero by assigning $\tilde{u}_j := 0$.
3. If the constraint is of set partitioning type:
   (a) If $F_0 = |S|$, the current subproblem is infeasible.
   (b) If $F_0 = |S| - 1$, fix the variable $j \in S$ with $\tilde{u}_j = 1$ to one by assigning $\tilde{l}_j := 1$.
4. If $F_1 = 1$, delete the constraint from the current subproblem.

A safeguard against useless propagations as in the linear and knapsack constraint handers is unnecessary, since the checks for propagation potential in Steps 1 to 4 are not time consuming anyway. For both, set partitioning and set packing constraints, two variables fixed to one render the constraint infeasible and the subproblem can be pruned. If exactly one of the involved variables is fixed to one, all others must be zero. The corresponding propagations are applied in Step 2. As said before, set partitioning constraints yield an additional propagation rule, which is applied in Step 3: if all variables are fixed to zero, the constraint is infeasible; if all but one of the variables are fixed to zero, the remaining one can be fixed to one. Finally, if now exactly one variable is fixed to one, the constraint is redundant and can be deleted from the current subproblem in Step 4.

Algorithm 7.6 illustrates the associated event handler. It just updates the F_0 and F_1 counters depending on the changing of the local bounds.

Algorithm 7.6 Event Handler for Set Partitioning/Packing Constraints

Input: Set partitioning constraint $\sum_{j \in S} x_j = 1$ or set packing constraint $\sum_{j \in S} x_j \leq 1$, a variable x_j, $j \in S$, for which the local bounds have been changed from $[\tilde{l}_j, \tilde{u}_j]$ to $[\tilde{l}'_j, \tilde{u}'_j]$, and current number of variables fixed to zero (F_0) and one (F_1).

Output: Updated minimal activity $\underline{\alpha}$.

1. If $\tilde{l}'_j > \tilde{l}_j$, increase $F_1 := F_1 + 1$.
2. If $\tilde{l}'_j < \tilde{l}_j$, decrease $F_1 := F_1 - 1$.
3. If $\tilde{u}'_j < \tilde{u}_j$, increase $F_0 := F_0 + 1$.
4. If $\tilde{u}'_j > \tilde{u}_j$, decrease $F_0 := F_0 - 1$.

7.4 Set Covering Constraints

Set covering constraints are the third type of subset selection constraints, namely the one that demands that from a set of items at least one has to be selected. It can be represented as

$$\sum_{j \in S} x_j \geq 1$$

with binary variables $x_j \in \{0,1\}$, $j \in S$, and $S \subseteq I$. Again, scaling and complementation may be applied to reach this standard form.

The only propagation rule that can be used for set covering constraints is

$$\forall j \in S \setminus \{k\} : x_j = 0 \quad \rightarrow \quad x_k = 1.$$

This rule already appeared in Section 7.3 for set partitioning constraints. We treat set covering constraints in a separate section, since their propagation mechanism substantially differs from the one used for set partitioning and set packing constraints. It is specifically tailored to efficiently handle large numbers of constraints. These can, for example, result from the use of conflict analysis, see Chapter 11.

Set covering constraints are equivalent to clauses $\ell_1 \vee \ldots \vee \ell_{|S|}$ of the satisfiability problem, see Definition 1.3 on page 10. Moskewicz et al. [168] invented a simple and efficient scheme to propagate SAT clauses, which they called *two watched literals scheme*. The main observation is that an implication can be derived from a clause only if all but one of the literals are fixed to 0. Thus, a clause only needs to be considered for domain propagation after the number of literals fixed to 0 increased from $|S| - 2$ to $|S| - 1$. If this is the case, the remaining unfixed literal is implied and can be fixed to 1.

Instead of using a counter F_0 for the number of variables fixed to zero as we did for the set partitioning and set packing constraints, it suffices for SAT clauses (and thus for set covering constraints) to only watch the state of two arbitrarily chosen literals of the constraint. As long as both literals remain unfixed, we do not need to process the constraint since no propagation can be applied. If one of the watched literals is fixed to 0, we inspect the other literals of the clause. If at least one of the other literals is fixed to 1, the constraint is redundant and can be removed from the subproblem. If at least one of them is unfixed, we stop watching the fixed literal and instead use the unfixed literal as new watched literal. After this switch we have again two watched literals, which are unfixed. Finally, if all literals except the other watched literal are fixed to 0, the other watched literal is implied and can be fixed to 1.

Moskewicz et al. report a significant speedup for their SAT solver Chaff compared to the then state-of-the-art solvers Grasp [157] and Sato [223]. Besides a different branching rule, the two watched literals scheme is the key element for the performance improvements of Chaff. The main advantage of this propagation scheme is that—particularly for long clauses—one can simply ignore most of the fixings applied to the involved variables. Despite the inspection of the literals after a watched literal was fixed to 0, the runtime of the domain propagation procedure is independent from the length of the clause. Another advantage is that we do not have to perform any bookmarking when the search backtracks to an ancestor of the current node, i.e., a more global subproblem. If the two watched literals are unfixed in the local subproblem, they remain unfixed in the more global subproblem. If one or both watched literals are fixed in the local subproblem, they will be the first literals for which the fixings will be undone on the backtracking path to the

more global subproblem. Therefore, there is no need to switch the watched variables during backtracking.

We implemented the two watched literals scheme for set covering constraints in SCIP using an event handler. The difference to the event handler for set partitioning and set packing constraints is that we do not catch the bound change events for all variables contained in the constraint, but only for two watched variables x_{w_1}, x_{w_2}, $w_1, w_2 \in S$. As explained above, the selection of the watched variables changes during the course of the search.

If a watched variable is fixed to zero, we have to search for a replacement, i.e., a variable of the constraint that is unfixed. Here, we can select any of the unfixed variables as new watched variable. An obvious choice would be to just select the first unfixed variable that is found, since then we can immediately abort the search loop over the variables. We performed computational experiments which, however, indicate that a different strategy yields a better overall performance. The rationale of this strategy is to select a variable such that we do not have to search for another watched variable replacement as long as possible. Inside a search tree traversed in a breadth first fashion like with the best first or best estimate node selection rule (see Chapter 6) we have a good indication of which bound changes will be applied in the future, namely the branchings that have been performed on the processed subproblems and which of these branchings already generated offspring. It makes sense to avoid variables for which the down branch (i.e., the fixing to zero) has already been evaluated frequently, since in any subtree defined by this branching the variable is fixed to zero and we will have to search for a replacement for the watched variable whenever we enter such a subtree.

Algorithm 7.7 summarizes the domain propagation for set covering constraints. The initial Step 1 disables the further propagation of the constraint. It will be reenabled by the associated event handler, see Algorithm 7.8 below. Steps 2 and 3 examine the local bounds of the watched variables. If one of the variables is fixed to one, the constraint is redundant. If both variables are unfixed, nothing has to be done. Step 4 initializes the indices w_1' and w_2' of the new watched variables. If one of the watched variables is still unfixed, we keep it as watched variable in order to reduce the event swapping overhead, and by setting the number of branchings $\beta_i := -1$, we make sure that it is not overwritten in the following search loop. If x_{w_2} is unfixed but x_{w_1} is fixed, we swap the indices such that we start the loop with $\beta_1 \leq \beta_2$.

In the search loop of Step 5, we check the remaining variables for potential use as watched variable. The cases of fixed variables are treated in Steps 5a and 5b. If we encounter an unfixed variable, we make sure that our current watched variable candidates w_1' and w_2' are the unfixed variables with the least number of processed downwards branchings β.

After the search loop the result is evaluated. Step 6 detects infeasibility of the current subproblem if all variables of the constraint are fixed to zero. If all but one of the variables are fixed to zero, the according propagation is applied at Step 7. If we found at least two unfixed variables, we apply the event swapping in Step 8 such that the associated event handler presented in Algorithm 7.8 now reacts to bound change events for $x_{w_1'}$ and $x_{w_2'}$.

The event handler for set covering constraints as shown in Algorithm 7.8 is very simple: it just reactivates the propagation for the constraint if one of the watched variables is fixed to zero.

Algorithm 7.7 Domain Propagation for Set Covering Constraints

Input: Set covering constraint $\sum_{j \in S} x_j \geq 1$, current local bounds $\tilde{l} \leq x \leq \tilde{u}$, current watched variable indices $w_1, w_2 \in S$, $w_1 \neq w_2$.

Output: Tightened local bounds for x, new watched variables w_1, w_2.

1. Disable the propagation of the constraint.
2. If $\tilde{l}_{w_1} = 1$ or $\tilde{l}_{w_2} = 1$, the constraint is redundant and can be removed from the current subproblem.
3. If $\tilde{u}_{w_1} = 1$ and $\tilde{u}_{w_2} = 1$, stop.
4. If $\tilde{u}_{w_1} = 1$, set $w'_1 := w_1$ and $\beta_1 := -1$. Else, set $w'_1 := -1$ and $\beta_1 := \infty$.
   If $\tilde{u}_{w_2} = 1$, set $w'_2 := w_1$ and $\beta_2 := -1$. Else, set $w'_2 := -1$ and $\beta_2 := \infty$.
   If $\beta_2 < \beta_1$, swap w'_1 and w'_2, and swap β_1 and β_2.
5. For all $j \in S \setminus \{w'_1, w'_2\}$:
   (a) If $\tilde{l}_j = 1$, the constraint is redundant and can be removed from the current subproblem. Stop.
   (b) If $\tilde{u}_j = 0$, continue Loop 5 with the next j.
   (c) Let β be the current number of evaluated downwards branchings on x_j.
   (d) If $\beta_1 \leq \beta < \beta_2$, set $w'_2 := j$ and $\beta_2 := \beta$.
   (e) If $\beta < \beta_1$, set $w'_2 := w'_1$, $\beta_2 := \beta_1$, $w'_1 := j$, and $\beta_1 := \beta$.
6. If $w'_1 = -1$, the current subproblem is infeasible.
7. If $w'_2 = -1$, fix x_{w_1} to one by assigning $\tilde{l}_{w_1} := 1$.
8. If $w'_1, w'_2 \geq 0$, switch the watched variables to be $w_1 := w'_1$ and $w_2 := w'_2$.

7.5 Variable Bound Constraints

Variable bound constraints in SCIP are defined as

$$\underline{\beta} \leq x_i + a_j x_j \leq \overline{\beta}$$

with $x_j \in \mathbb{Z}$, $a_j \in \mathbb{R} \setminus \{0\}$, and $\underline{\beta}, \overline{\beta} \in \mathbb{R} \cup \{\pm\infty\}$. This is a generalization of the common variable upper bounds $x_i \leq u'_i x_j$ and variable lower bounds $x_i \geq l'_i x_j$ with $x_j \in \{0, 1\}$. Variable upper bounds are a well-known tool to model fixed charge problems like the fixed charge network flow problem, see, e.g., Padberg, Roy, and Wolsey [181]. For general mixed integer programs they are computationally important as they can be used to derive complemented mixed integer rounding cuts (see Marchand and Wolsey [155]) and flow cover inequalities (see Gu, Nemhauser, and Savelsbergh [110]). See Wolter [218] for the details on the implementation of these cutting plane separators in SCIP.

The domain propagation method applied to variable bound constraints is ba-

Algorithm 7.8 Event Handler for Set Covering Constraints

Input: Set covering constraint $\sum_{j \in S} x_j \geq 1$, a watched variable x_j, $j \in S$, for which the local upper bound has been changed from $\tilde{u}_j = 1$ to $\tilde{u}'_j = 0$.

1. Enable the propagation of the constraint.

sically the same as for the general linear constraints of Section 7.1. As for the previously described specializations of linear constraints, the advantage of an individual constraint handler for variable bound constraints is the possibility of a more compact data representation and an easier and faster propagation method.

The following propagations are applied:

$$x_i \geq \underline{\beta} - a_j \tilde{u}_j \quad \text{for } a_j > 0 \quad \text{and} \quad x_i \geq \underline{\beta} - a_j \tilde{l}_j \quad \text{for } a_j < 0, \tag{7.5}$$

$$x_i \leq \overline{\beta} - a_j \tilde{l}_j \quad \text{for } a_j > 0 \quad \text{and} \quad x_i \leq \overline{\beta} - a_j \tilde{u}_j \quad \text{for } a_j < 0, \tag{7.6}$$

$$x_j \geq \left\lceil \frac{\underline{\beta} - \tilde{u}_i}{a_j} \right\rceil \quad \text{for } a_j > 0 \quad \text{and} \quad x_j \leq \left\lfloor \frac{\underline{\beta} - \tilde{u}_i}{a_j} \right\rfloor \quad \text{for } a_j < 0, \tag{7.7}$$

$$x_j \leq \left\lfloor \frac{\overline{\beta} - \tilde{l}_i}{a_j} \right\rfloor \quad \text{for } a_j > 0 \quad \text{and} \quad x_j \geq \left\lceil \frac{\overline{\beta} - \tilde{l}_i}{a_j} \right\rceil \quad \text{for } a_j < 0. \tag{7.8}$$

Since these calculations are not as involved as the ones for general linear constraints, we do not need to apply a conservative rounding as in Equation (7.2) on page 86 for the propagation of linear constraints. However, the risk of numerous iterated small tightenings of domains of general integer or continuous variables is also present for variable bound constraints. Therefore, we use the same limits on the minimal fraction to cut off from domains as for linear constraints, see Inequalities (7.3).

Algorithm 7.9 recapitulates the propagation procedure for variable bound constraints. As for linear constraints, a propagation mark controls whether the constraint has to be considered again for propagation. This mark is reset by the associated event handler as can be seen in Algorithm 7.10.

7.6 Objective Propagation

The propagations described in the previous sections are based on primal feasibility reasoning: a bound is tightened because setting a variable to a value outside the tightened bounds leads to an infeasible subproblem. The *objective propagation* of this section and the *root reduced cost strengthening* of the following section take the dual point of view. They infer bounds that are valid due to optimality considerations: if the variable takes a value outside the tightened bounds, the solution cannot be better than the current incumbent.

Let $\hat{c} = c^T \hat{x}$ be the objective value of the current incumbent solution $\hat{x} \in \mathbb{R}^n$, i.e., the smallest objective value of all feasible solutions found so far. Then, the objective function can be used to rule out inferior solutions by propagating

$$\sum_{j \in N} c_j x_j \leq \hat{c} - \check{\delta} \tag{7.9}$$

with $\check{\delta} \in \mathbb{R}_{>0}$ being the *dual feasibility tolerance* or *optimality tolerance*. This objective constraint is a regular linear constraint, and we can use Algorithm 7.1 to propagate it.

The *objective cutoff*, i.e., the right hand side $\hat{c}$ of Inequality (7.9), can be tightened if the objective function value is always integral. A sufficient condition for objective integrality is that

- ▷ all objective coefficients are integral: $c_j \in \mathbb{Z}$ for all $j \in N$, and
- ▷ all objective coefficients for continuous variables are zero: $c_j = 0$ for all $j \in C$.

Algorithm 7.9 Domain Propagation for Variable Bound Constraints

Input: Variable bound constraint $\underline{\beta} \leq x_i + a_j x_j \leq \overline{\beta}$ and current local bounds $x_i \in [\tilde{l}_i, \tilde{u}_i]$ and $x_j \in [\tilde{l}_j, \tilde{u}_j]$.

Output: Tightened local bounds for x_i and x_j.

1. If the constraint is already marked as propagated, abort.
2. Mark the constraint as propagated.
3. Set $\tilde{l}'_i := \tilde{l}'_j := -\infty$ and $\tilde{u}'_i := \tilde{u}'_j := +\infty$.
4. If $\underline{\beta} > -\infty$:
   (a) If $a_j > 0$, set $\tilde{l}'_i := \underline{\beta} - a_j \tilde{u}_j$ and $\tilde{l}'_j := (\underline{\beta} - \tilde{u}_i)/a_j$.
   (b) If $a_j < 0$, set $\tilde{l}'_i := \underline{\beta} - a_j \tilde{l}_j$ and $\tilde{u}'_j := (\underline{\beta} - \tilde{u}_i)/a_j$.
5. If $\overline{\beta} < +\infty$:
   (a) If $a_j > 0$, set $\tilde{u}'_i := \overline{\beta} - a_j \tilde{l}_j$ and $\tilde{u}'_j := (\overline{\beta} - \tilde{l}_i)/a_j$.
   (b) If $a_j < 0$, set $\tilde{u}'_i := \overline{\beta} - a_j \tilde{u}_j$ and $\tilde{l}'_j := (\overline{\beta} - \tilde{l}_i)/a_j$.
6. If $i \in I$, round $\tilde{l}'_i := \lceil \tilde{l}'_i \rceil$ and $\tilde{u}'_i := \lfloor \tilde{u}'_i \rfloor$.
   Round $\tilde{l}'_j := \lceil \tilde{l}'_j \rceil$ and $\tilde{u}'_j := \lfloor \tilde{u}'_j \rfloor$.
7. If $\tilde{l}'_i > \tilde{l}_i + 0.05 \cdot \max\Big\{\min\big\{\tilde{u}_i - \tilde{l}_i, |\tilde{l}_i|\big\}, 1\Big\}$, tighten $\tilde{l}_i := \tilde{l}'_i$.
   If $\tilde{u}'_i < \tilde{u}_i - 0.05 \cdot \max\Big\{\min\big\{\tilde{u}_i - \tilde{l}_i, |\tilde{u}_i|\big\}, 1\Big\}$, tighten $\tilde{u}_i := \tilde{u}'_i$.
   If $\tilde{l}'_j > \tilde{l}_j + 0.05 \cdot \max\Big\{\min\big\{\tilde{u}_j - \tilde{l}_j, |\tilde{l}_j|\big\}, 1\Big\}$, tighten $\tilde{l}_j := \tilde{l}'_j$.
   If $\tilde{u}'_j < \tilde{u}_j - 0.05 \cdot \max\Big\{\min\big\{\tilde{u}_j - \tilde{l}_j, |\tilde{u}_j|\big\}, 1\Big\}$, tighten $\tilde{u}_j := \tilde{u}'_j$.
8. If both constraint sides are redundant, i.e.,
   (a) $\underline{\beta} = -\infty$, or $a_j > 0$ and $\tilde{l}_i + a_j \tilde{l}_j \geq \underline{\beta}$, or $a_j < 0$ and $\tilde{l}_i + a_j \tilde{u}_j \geq \underline{\beta}$, and
   (b) $\overline{\beta} = +\infty$, or $a_j > 0$ and $\tilde{u}_i + a_j \tilde{u}_j \leq \overline{\beta}$, or $a_j < 0$ and $\tilde{u}_i + a_j \tilde{l}_j \leq \overline{\beta}$,
   then the constraint can be deleted from the current subproblem.

Besides this automatic detection, SCIP provides a method to set the integrality status of the objective function. This is useful for models where the objective integrality is implicit and cannot be determined directly from the objective coefficients.

If we know that the objective value is always integral, we can apply *integral cutoff tightening* and propagate

$$\sum_{j \in N} c_j x_j \leq \hat{c} - (1 - \check{\delta}).$$

Furthermore, if the objective coefficients for continuous variables are zero and the ones for integer variables are rational numbers with reasonably small denominators,

Algorithm 7.10 Event Handler for Variable Bound Constraints

Input: Variable bound constraint $\underline{\beta} \leq x_i + a_j x_j \leq \overline{\beta}$, a variable x_k, $k \in \{i, j\}$, for which a bound has been changed.

1. Mark the constraint as not propagated.

Algorithm 7.11 Root Reduced Cost Strengthening

Input: Current incumbent value $\hat{c}$, root node LP objective $\check{c}_R$, solution $\check{x}_R$, and reduced costs $\check{r}_R$, and global bounds $l \leq x \leq u$.

Output: Tightened global bounds for x.

1. If $\hat{c}$ has not changed since the last call, stop.
2. For all variables x_j with $(\check{r}_R)_j > 0$:
   (a) Set $u'_j := (\check{x}_R)_j + (\hat{c} - \check{c}_R)/(\check{r}_R)_j$.
   (b) If $j \in I$, set $u'_j := \lfloor u'_j + \hat{\delta} \rfloor$.
   (c) If $u'_j < u_j$, tighten $u_j := u'_j$.
3. For all variables x_j with $(\check{r}_R)_j < 0$:
   (a) Set $l'_j := (\check{x}_R)_j + (\hat{c} - \check{c}_R)/(\check{r}_R)_j$.
   (b) If $j \in I$, set $l'_j := \lceil l'_j - \hat{\delta} \rceil$.
   (c) If $l'_j > l_j$, tighten $l_j := l'_j$.

one can multiply Inequality (7.9) with the smallest common multiple of the denominators, divide the resulting integral values by their greatest common divisor, and subtract $1 - \check{\delta}$ from the resulting right hand side. Note that this *rational cutoff tightening* is new in SCIP 0.90i and not included in version 0.90f which we used for the MIP benchmarks.

7.7 Root Reduced Cost Strengthening

In the design of SCIP, the well-known *reduced cost strengthening* procedure (see Nemhauser and Wolsey [174]) is implemented as a cutting plane separator, compare Section 8.8. It tightens the local bounds $\tilde{l} \leq x \leq \tilde{u}$ of the variables by comparing their reduced cost values $\check{r}$ in the current LP solution with the objective value $\hat{c}$ of the incumbent solution and the objective value $\check{c}$ of the LP relaxation:

$$x_j \geq \tilde{l}_j + \frac{\hat{c} - \check{c}}{\check{r}_j} \quad \text{if } \check{r}_j > 0,$$
$$x_j \leq \tilde{u}_j + \frac{\hat{c} - \check{c}}{\check{r}_j} \quad \text{if } \check{r}_j < 0.$$

The *root reduced cost strengthening* propagator introduced in this section provides the same reasoning for global bounds which are tightened using root node LP information. If the root node LP has been solved, we store the final objective value $\check{c}_R$, the LP solution vector $\check{x}_R$, and the reduced cost vector $\check{r}_R$. Then, each time when a new incumbent solution has been found we reapply root node reduced cost strengthening in order to tighten the global bounds.

The procedure is illustrated in Algorithm 7.11. Step 1 checks whether the incumbent solution has been improved since the last application of the propagator. If this is the case, Steps 2 and 3 reapply the reduced cost strengthening at the root node. Note that the calculations of the new bounds in Steps 2a and 3a do not depend on

	test set	none	aggr linear	no obj prop	no root redcost
time	MIPLIB	+6	+3	−3	−2
	CORAL	+15	−8	−4	−7
	MILP	+13	−3	+5	+1
	ENLIGHT	+104	−42	0	−1
	ALU	+222	−36	0	+1
	FCTP	−7	0	0	0
	ACC	+31	−18	−1	−1
	FC	+42	+3	+40	+3
	ARCSET	+13	+1	+1	0
	MIK	+19	−15	+21	+1
	total	**+17**	**−6**	**+1**	**−2**
nodes	MIPLIB	+1	0	−4	−3
	CORAL	+12	−10	−5	−8
	MILP	+20	−9	+15	0
	ENLIGHT	+183	−59	+2	0
	ALU	+283	−24	0	0
	FCTP	−5	−2	0	0
	ACC	+25	−5	0	0
	FC	+75	0	+63	+4
	ARCSET	+10	+2	+1	−4
	MIK	+66	−34	+38	−1
	total	**+20**	**−9**	**+4**	**−3**

Table 7.1. Performance effect of domain propagation techniques for solving MIP instances. The values denote the percental changes in the shifted geometric mean of the runtime (top) and number of branching nodes (bottom) compared to the default settings in which all domain propagators are called at every node except the linear propagation, which is only applied at every fifth depth level. Positive values represent a deterioration, negative values an improvement.

the current global bounds.[1] Therefore, there is no risk of iterated small reductions as in the linear constraint propagation (compare Figure 7.1 on page 87), and we can safely accept even small domain reductions.

7.8 Computational Results

We ran benchmarks to asses the impact of applying domain propagation techniques to the local subproblems of a MIP search tree. Table 7.1 gives a summary of the results. Further details can be found in Tables B.51 to B.60 in Appendix B. The test instances and the computational environment are described in Appendix A.

The column "none" shows the performance change for deactivating all local domain propagation algorithms. One can see that domain propagation is able to reduce the runtime on all test sets except FCTP. The most notable effect can be observed for the ENLIGHT and ALU instances, for which the runtime doubles and triples, respectively, if domain propagation is turned off. As already mentioned in Sections 5.11 and 6.8, these two problem classes are of a combinatorial type and their objective functions are not very important or, in the case of ALU, completely artificial. This suggests that for such instances, LP based techniques are not that strong and CP algorithms are more important. The computational results support this hypothesis.

During the development of the domain propagation algorithms, our impression was that the propagation of linear constraints as given in Algorithm 7.1 is too expensive to be applied at every node in the search tree. Therefore, the default settings

[1] Actually, for a variable with non-zero reduced costs, the root LP solution $(\check{x}_R)_j$ is equal to the global lower or upper bound as they have been set at the time the root LP was solved.

are to invoke the algorithm only at every fifth depth level. The column "aggr linear" shows the effect of applying the linear domain propagation at every node. It turns out that our concerns are unjustified: applying linear domain propagation at every node improves the performance on almost all test sets. Again, the largest speedup can be observed on the ENLIGHT and ALU instances.

The columns "no obj prop" and "no root redcost" demonstrate the impact of the two dual propagation methods included in SCIP. Both techniques have only a very small effect. However, the *objective propagation* considerably improves the performance on the FC and MIK test sets. As can be seen in Table B.58, the runtime increases by at least 10 % for 13 out of the 20 FC instances if objective propagation is disabled. In contrast, not a single instance in this test set can be solved faster without objective propagation.

Chapter 8

Cut Separation

Cutting planes for integer and mixed integer programming have been studied since the late 1950's. One of the most fundamental work in this area has been conducted by Gomory [102, 103, 104] who proved that integer programs with rational data can be solved in a finite number of steps by a pure cutting plane approach without any branching. Unfortunately, numerical issues in Gomory's approach prevented pure cutting plane methods from being effective in practice.

With the work of Balas et al. [31] in 1996 and finally with the release of Cplex 6.5 in 1999, it became clear that cutting planes, in particular Gomory mixed integer cuts, are very efficient if combined with branch-and-bound. The resulting algorithm is called *cut-and-branch* or *branch-and-cut*, depending on whether cutting planes are only generated at the root node or also at local subproblems in the search tree, see Section 2.2. Bixby et al. [46] report a speed-up of a factor of 22.3 from Cplex 6.0 to Cplex 6.5, with cutting planes providing a major contribution to this success.

Since the theory of cutting planes is very well covered in the literature (see, e.g., a recent survey by Klar [132]), this chapter focuses on the cutting plane separation methods that are available in SCIP. We will, however, only briefly sketch the algorithms and the underlying theory. A detailed description, including the theory and computational experiments that compare various SCIP parameter settings and different implementations, can be found in the diploma thesis of Wolter [218]. Additional information about computationally important cutting planes can, for example, be found in Marchand et al. [154] and in Fügenschuh and Martin [90].

Some of the results presented here, namely the sections about knapsack cover cuts, mixed integer rounding cuts, strong Chvátal-Gomory cuts, implied bound cuts, and clique cuts are joint work with Kati Wolter. The implementation of Gomory mixed integer cuts, knapsack cover cuts, and reduced cost strengthening is based on Alexander Martin's implementation of SIP [159]. The SCIP algorithms for flow cover cuts have been implemented by Kati Wolter.

8.1 Knapsack Cover Cuts

The knapsack polytope is one of the most thoroughly studied polyhedrons in mathematical programming, see for instance Balas [28], Hammer, Johnson, and Peled [112], Padberg [179, 180], Wolsey [216], Balas and Zemel [35, 36], Weismantel [212], or Martin and Weismantel [160]. Knapsack cover cuts, which are a particular family of valid inequalities for the knapsack polytope, are probably one of the first cutting planes that have been incorporated into commercial MIP solvers. For example, they are available in Cplex since version 3.0. Crowder, Johnson, and Padberg [70] were the first to successfully apply lifted knapsack cover inequalities to solve several binary programs that were, at the time, considered to be intractable.

In its most basic version, the knapsack cover cut can be derived as follows.

Definition 8.1 (knapsack cover). Consider a knapsack inequality $a^T x \leq \bar{\beta}$ with $a \in \mathbb{Z}^B_{\geq 0}$, $\bar{\beta} \in \mathbb{Z}_{\geq 0}$, and binary variables $x \in \{0,1\}^B$. Then, $V \subseteq B$ is called *cover* if $\sum_{j \in V} a_j > \bar{\beta}$. A cover V is called *minimal cover* if $\sum_{j \in V \setminus \{k\}} a_j \leq \bar{\beta}$ for all $k \in V$.

Knapsack covers directly lead to valid inequalities for the binary knapsack polytope $P_K := \text{conv}(X_K)$ with $X_K := \{x \in \{0,1\}^B \mid a^T x \leq \bar{\beta}\}$, see [28, 112, 179, 216]:

Proposition 8.2. Given a cover V, the corresponding *cover inequality*

$$\sum_{j \in V} x_j \leq |V| - 1 \tag{8.1}$$

is valid for the knapsack polytope P_K. Furthermore, if V is a minimal cover, Inequality (8.1) is facet defining for $P'_K := \text{conv}\left(X_K \cap \{x \mid x_j = 0 \text{ for all } j \in B \setminus V\}\right)$.

In order to convert the facet defining cover inequality (8.1) of the lower dimensional P'_K into an inequality which defines a facet or a high dimensional face of P_K, we have to lift the variables in $B \setminus V$, see Padberg [179]. Lifting is a technique that may introduce non-zero coefficients d_j for $j \in B \setminus V$ to Inequality (8.1), such that the resulting cut

$$\sum_{j \in V} x_j + \sum_{j \in B \setminus V} d_j x_j \leq |V| - 1$$

is still valid for P_K.

In addition to this *up-lifting* step, Wolsey [216] observed that one can also apply a *down-lifting* procedure. He proposed to partition the binary variables into three sets $B = V \cup L_0 \cup L_1$, such that V is a minimal cover for

$$\sum_{j \in V} a_j x_j \leq \bar{\beta} - \sum_{j \in L_1} a_j.$$

The resulting cover inequality (8.1) is facet defining for

$$P''_K := \text{conv}\left(X_K \cap \{x \mid x_j = 0 \text{ for all } j \in L_0\} \cap \{x \mid x_j = 1 \text{ for all } j \in L_1\}\right),$$

but in general invalid for the original knapsack polytope P_K. Down-lifting can be applied to the variables $j \in L_1$ to convert it into a valid inequality for P_K, and up-lifting for $j \in L_0$ can be used to strengthen the cutting plane. Note that every lifting sequence results in a valid cut, but the coefficients d_j of the cut depend on the order in which the variables are lifted. Thus, this type of lifting is called *sequence dependent* lifting. Using so-called superadditive functions, one can also perform *sequence independent* lifting, see Gu, Nemhauser, and Savelsbergh [111].

SCIP generates lifted knapsack cover cuts within the knapsack constraint handler. The procedure is sketched in Algorithm 8.1. The variables x_j that have a value of $\check{x}_j = 1$ in the current LP solution are selected for down-lifting and put to L_1 in Step 1. Then, in Step 2 we calculate a minimal cover on the reduced knapsack inequality. Using dynamic programming techniques (see, for example, Keller et al. [129]), we can find a most violated minimal cover in pseudo-polynomial time, with the runtime depending on the capacity $\bar{\beta} - \sum_{j \in L_1} a_j$ of the reduced knapsack. If this capacity is too large, we revert to the greedy heuristic of Dantzig [74]. It may happen that we do not find a violated cover inequality, in which case we have to abort the separation.

Algorithm 8.1 Separation of Lifted Knapsack Cover Cuts

Input: Knapsack constraint $a^T x \leq \overline{\beta}$ and current LP solution $\check{x}$.
Output: Cutting planes $d^T x \leq \overline{\gamma}$.

1. Set $L_1 := \{j \in B \mid \check{x}_j = 1\}$.
2. Calculate a minimal cover $V \subseteq B \setminus L_1$ for the knapsack constraint

$$\sum_{j \in B \setminus L_1} a_j x_j \leq \overline{\beta} - \sum_{j \in L_1} a_j.$$

   If $\overline{\beta} - \sum_{j \in L_1} a_j \leq 10000$, find a most violated minimal cover inequality by dynamic programming. Otherwise, use a greedy heuristic. If no violated cover inequality can be found, stop.
3. Set $L_0 := (B \setminus L_1) \setminus V$. Sort V and L_0 by non-increasing solution value $\check{x}_j$, breaking ties by non-increasing weight a_j.
4. While $V \neq \emptyset$:
   (a) Let x_k, $k \in V$, be a variable with smallest LP solution value $\check{x}_k$. Set $V := V \setminus \{k\}$ and $L_0 := L_0 \cup \{k\}$.
   (b) Generate the trivial cardinality inequality

$$\sum_{j \in V} x_j \leq |V|. \tag{8.2}$$

   (c) Strengthen Inequality (8.2) by up-lifting the variables $j \in L_0$ with $\check{x}_j > 0$.
   (d) Make Inequality (8.2) valid for P_K by down-lifting the variables $j \in L_1$.
   (e) Strengthen Inequality (8.2) by up-lifting the variables $j \in L_0$ with $\check{x}_j = 0$.
   (f) If the resulting inequality is violated by $\check{x}$, add it to the separation storage.

The initial up-lifting candidates L_0 consist of the remaining variables, which are neither in the down-lifting set L_1 nor in the cover V. Starting with this initial partition, Loop 4 performs the actual lifting and cutting plane generation. In fact, we do not only generate lifted minimal cover inequalities but also *lifted extended weight inequalities* as introduced by Weismantel [212].

The first Step 4a removes an item with smallest LP value from the cover and moves it to the up-lifting candidates. Then, we start with the trivial cardinality inequality (8.2) and lift in the variables of L_0 and L_1 as indicated in Steps 4c to 4e. The variables are lifted ordered by non-increasing LP value $\check{x}_j$ with ties broken by non-increasing weight a_j. In the first round of Loop 4, this means that the removed cover variable x_k immediately gets a lifting coefficient of $d_k = 1$ since the initial set V was a cover. Thereby, we restore the cover inequality and obtain a lifted cover inequality after the subsequent lifting of the remaining variables. In the successive rounds of the loops, we generate lifted extended weight inequalities.

Cutting plane separation for the knapsack polytope can also be applied on general linear constraints $\mathcal{C}_i : \underline{\beta} \leq a^T x \leq \overline{\beta}$ with $a \in \mathbb{R}^n$ and $x \in \mathbb{Z}^I \times \mathbb{R}^C$ by relaxing the constraint into a binary knapsack constraint. This relaxation is performed in the separation method of the linear constraint handler, and Algorithm 8.1 is called as a subroutine. For the relaxation, we have various options. In the default settings, we

proceed as follows.

If the dual solution of $\mathcal{C}_i$ is $\check{y}_i < 0$, we generate cuts for the right hand side inequality $a^T x \leq \overline{\beta}$. If $\check{y}_i > 0$, we generate cuts for the left hand side inequality $-a^T x \leq -\underline{\beta}$. If $\check{y}_i = 0$, we ignore the constraint for cut separation. This reflects the fact that the left and right hand sides of the constraint restrain the LP objective value if the dual solution is $\check{y}_i > 0$ or $\check{y}_i < 0$, respectively.

Afterwards, we replace all continuous and general integer variables x_j, $j \in N \setminus B$, with binary variables or constants: if possible, we substitute them for variable lower bounds $x_j \geq s x_k + d$ or variable upper bounds $x_j \leq s x_k + d$ involving binary variables x_k, depending on the sign of their coefficients a_j, see Section 3.3.5. If no suitable variable bound is available, we substitute the non-binary variables for their global bounds l_j or u_j.

Finally, we produce a rational representation of the resulting coefficients and multiply the relaxed constraint with the smallest common multiple of the denominators to obtain integral coefficients. In the conversion of the floating point values to rational numbers, we allow to relax the coefficients by 10 % in order to get smaller denominators.

8.2 Mixed Integer Rounding Cuts

Mixed integer rounding cuts (MIR cuts) as introduced by Nemhauser and Wolsey [175] can be formulated as follows:

Proposition 8.3 (MIR inequality). Given a linear constraint $a^T x \leq \overline{\beta}$ on variables $x \in \mathbb{Z}^I_{\geq 0} \times \mathbb{R}^C_{\geq 0}$, the *mixed integer rounding inequality*

$$\sum_{j \in I} \left(\lfloor a_j \rfloor + \frac{\max\{f_j - f_0, 0\}}{1 - f_0} \right) x_j \; + \; \sum_{j \in C} \frac{\min\{a_j, 0\}}{1 - f_0} x_j \; \leq \; \lfloor \overline{\beta} \rfloor \tag{8.3}$$

with $f_j := a_j - \lfloor a_j \rfloor$ and $f_0 := \overline{\beta} - \lfloor \overline{\beta} \rfloor$ is valid for the mixed knapsack polyhedron $P_{MK} := \operatorname{conv}(X_{MK})$ with $X_{MK} := \left\{ x \in \mathbb{Z}^I_{\geq 0} \times \mathbb{R}^C_{\geq 0} \mid a^T x \leq \overline{\beta} \right\}$.

Marchand [153] and Marchand and Wolsey [155] applied the MIR procedure to separate *complemented mixed integer rounding cuts* for mixed integer programs. The implementation of this separator in SCIP, which is depicted in Algorithm 8.2, is very similar to the approach of Marchand and Wolsey.

The aggregation heuristic of Step 1 calculates a score value for each row in the LP, which depends on the dual solution, the density, and the slack of the row. Only rows with a maximal density of 5 % and a maximal slack of 0.1 are considered. Then, the rows are sorted by non-increasing score value, and they are successively used as starting row for the aggregation until certain work limits have been reached.

Given a starting row, the aggregation procedure adds other rows to the starting row in order to eliminate continuous variables. In particular, we try to remove continuous variables with an LP solution $\check{x}_j$ that is far away from the bounds l_j and u_j. Under those variables that have approximately the same distance to their bounds, we prefer variable eliminations that can be achieved by adding a row with large score value. In every iteration of the aggregation loop, the current aggregation is passed to Steps 2 to 5 in order to generate a cutting plane. The aggregation loop for the current starting row is aborted if a violated cut has been found or six rows have been aggregated, including the starting row.

Algorithm 8.2 Separation of Complemented MIR Cuts

Input: LP relaxation $Ax \leq b$ of MIP, global bounds $l \leq x \leq u$, and current LP solution $\check{x}$.

Output: Cutting planes $d^T x \leq \overline{\gamma}$.

1. Aggregate linear constraints to obtain a single linear inequality $a^T x \leq \overline{\beta}$.
2. Transform the variables to the canonical form $x' \geq 0$ by either
   - ▷ shifting to their lower bound $x_j \geq l_j$: $x'_j := x_j - l_j$,
   - ▷ complementing to their upper bound $x_j \leq u_j$: $x'_j := u_j - x_j$,
   - ▷ substituting with a variable lower bound $x_j \geq sx_k + d$, $k \in I$: $x'_j := x_j - (sx_k + d)$, or
   - ▷ substituting with a variable upper bound $x_j \leq sx_k + d$, $k \in I$: $x'_j := (sx_k + d) - x_j$.
3. Divide the resulting inequality $a'^T x' \leq \overline{\beta}'$ by $\delta = \pm 1$, $\delta = \pm \max\{|a'_j| \mid j \in I\}$, and $\delta \in \{\pm a'_j \mid j \in I \text{ and } 0 < \check{x}'_j < u'_j\}$, and generate the corresponding MIR inequalities (8.3). Choose $\delta^\star$ to be the δ for which the most violated MIR inequality has been produced.
4. In addition to $\delta^\star$, check whether the MIR inequalities derived from dividing $a'^T x' \leq \overline{\beta}'$ by $\frac{1}{2}\delta^\star$, $\frac{1}{4}\delta^\star$, and $\frac{1}{8}\delta^\star$ yield even larger violations.
5. Finally, select the most violated of the MIR inequalities, transform it back into the space of problem variables x, substitute slack variables, and add it to the separation storage.

The bound substitution in Step 2 always selects the bound that is closest to the current LP solution $\check{x}_j$. If the LP value of a variable bound is at least as close as the global bound, we prefer variable bounds over global bounds.

Marchand and Wolsey [155] propose to augment the cutting plane separation procedure with a final step of additional variable complementation in order to find an even more violated cut. This complementation would be performed after Step 4. We do not follow this approach, since it turned out to be inferior in early benchmark tests. Instead, we slightly extend the bound substitution heuristic of Step 2: if the resulting right hand side $\overline{\beta}'$ is integral, we complement one additional variable or uncomplement one of the complemented variables to obtain a fractional right hand side.

8.3 Gomory Mixed Integer Cuts

Besides lift-and-project cuts (see Balas, Ceria, and Cornuéjols [29, 30]), Gomory mixed integer cuts (GMI cuts) have been the first general purpose cutting planes that were successfully employed within a branch-and-cut framework to solve mixed integer programs, see Balas et al. [31]. They have been discovered by Gomory [103, 104] in 1960, but despite their theoretical value, namely providing a method to solve integer programs with rational data, they have been regarded as computationally useless. This reputation changed with the work of Balas et al., and nowadays GMI cuts seem

to be one of the most important cutting planes employed in mixed integer solvers, see Bixby et al. [46].

Gomory mixed integer cuts can be stated as follows [103, 31]:

Proposition 8.4 (Gomory mixed integer cuts). Let $\mathcal{B} \subseteq N$ be an optimal simplex basis for the LP relaxation of a MIP in equality form

$$c^\star = \min\{c^T x \mid Ax = b, x \geq 0, x_j \in \mathbb{Z} \text{ for all } j \in I \subseteq N\},$$

and let $\mathcal{N} := N \setminus \mathcal{B}$ denote the index set of non-basic variables. Furthermore, let $\bar{a} = (A_{\mathcal{B}}^{-1}A)_i$. be row i of the simplex tableau, $\bar{a}_0 = (A_{\mathcal{B}}^{-1}b)_i$ be the right hand side of this tableau row, and $f_j = \bar{a}_j - \lfloor \bar{a}_j \rfloor$ be the fractional parts of the tableau row entries, $j \in \mathcal{N} \cup \{0\}$. Then, if x_i, $i \in \mathcal{B} \cap I$, is a basic integer variable with fractional LP solution $\check{x}_i = \bar{a}_0 \notin \mathbb{Z}$, the *Gomory mixed integer cut*

$$\sum_{j \in \mathcal{N} \cap I} \min\left\{\frac{f_j}{f_0}, \frac{1-f_j}{1-f_0}\right\} x_j \; + \sum_{j \in \mathcal{N} \cap C} \max\left\{\frac{\bar{a}_j}{f_0}, \frac{-\bar{a}_j}{1-f_0}\right\} x_j \;\geq\; 1 \tag{8.4}$$

is valid for the MIP and cuts off the fractional LP solution $\check{x}$.

The following proposition shows that GMI cuts are a subclass of MIR cuts and can be derived by employing the MIR procedure of Proposition 8.3 to the simplex tableau equation.

Proposition 8.5. The Gomory mixed integer cut (8.4) is equivalent to the mixed integer rounding cut (8.3) applied to the simplex tableau row

$$x_i + \sum_{j \in \mathcal{N}} \bar{a}_j x_j = \bar{a}_0 \tag{8.5}$$

for a non-basic integer variable x_i, $i \in \mathcal{N} \cap I$, with fractional $\bar{a}_0 \notin \mathbb{Z}$.

Proof. Multiplying the GMI cut (8.4) by $-f_0$ yields the equivalent constraint

$$\sum_{j \in \mathcal{N} \cap I} \max\left\{-f_j, -f_0\frac{1-f_j}{1-f_0}\right\} x_j \; + \sum_{j \in \mathcal{N} \cap C} \min\left\{-\bar{a}_j, \frac{\bar{a}_j f_0}{1-f_0}\right\} x_j \;\leq\; -f_0. \tag{8.6}$$

Adding Inequality (8.6) to the feasible Equation (8.5) yields

$$x_i + \sum_{j \in \mathcal{N} \cap I} \max\left\{\bar{a}_j - f_j, \bar{a}_j - f_0\frac{1-f_j}{1-f_0}\right\} x_j \; + \sum_{j \in \mathcal{N} \cap C} \min\left\{0, \bar{a}_j + \frac{\bar{a}_j f_0}{1-f_0}\right\} x_j \;\leq\; \bar{a}_0 - f_0,$$

which is equivalent to

$$x_i + \sum_{j \in \mathcal{N} \cap I} \left(\lfloor \bar{a}_j \rfloor + \max\left\{0, \frac{f_j - f_0}{1-f_0}\right\}\right) x_j \; + \sum_{j \in \mathcal{N} \cap C} \min\left\{0, \frac{\bar{a}_j}{1-f_0}\right\} x_j \;\leq\; \lfloor \bar{a}_0 \rfloor.$$

This final inequality is exactly same as the MIR cut for the tableau row (8.5) in the direction $x_i + \sum_{j \in \mathcal{N}} \bar{a}_j x_j \leq \bar{a}_0$. □

Following Proposition 8.5, the Gomory mixed integer cut separator of SCIP just applies Steps 1, 2, and 5 of the c-MIR cut separation Algorithm 8.2. For each fractional integer basic variable x_i, the aggregation of Step 1 is performed using the weights given by the corresponding row $(A_{\mathcal{B}}^{-1})_i$. in the basis inverse, such that the aggregation produces Equation 8.5, which may include slack variables. The bound substitution of Step 2 only complements those variables that are at their upper bounds.

8.4 Strong Chvátal-Gomory Cuts

Letchford and Lodi [142] proposed a method to strengthen Chvátal-Gomory cuts [60], which are closely related to Gomory fractional cuts [102]. Given a linear inequality $a^T x \leq \bar{\beta}$ on non-negative integer variables $x_j \in \mathbb{Z}_{\geq 0}$, $j \in I = N$, the Chvátal-Gomory cut (CG cut) can be derived by rounding down the coefficient vector and the right hand side:

$$\sum_{j \in I} \lfloor a_j \rfloor x \leq \lfloor \bar{\beta} \rfloor. \tag{8.7}$$

It is easy to see that this inequality is valid, since after rounding down the coefficients of the inequality the activity is always integral, which also allows for rounding down the right hand side. One way to obtain a stronger cut is to instead apply the mixed integer rounding procedure given in Inequality (8.3). The MIR cut always dominates the CG cut, since their right hand sides are equal and the coefficients of the MIR cut are greater or equal to the coefficients of the CG cut. Additionally, the MIR procedure can be applied to constraints that include continuous variables.

Letchford and Lodi suggest a different strengthening of the CG cut for inequalities containing only integer variables, which is given by the following theorem [142]:

Theorem 8.6 (strong Chvátal-Gomory cut). Consider the inequality $a^T x \leq \bar{\beta}$ on non-negative integer variables $x_j \in \mathbb{Z}_{\geq 0}$, $j \in I = N$. Suppose that $f_0 = \bar{\beta} - \lfloor \bar{\beta} \rfloor > 0$ and let $k \geq 1$ be the unique integer such that $\frac{1}{k+1} \leq f_0 < \frac{1}{k}$. Partition N into classes $N_0, \ldots, N_k$ as follows. Let $N_0 = \{j \in N \mid f_j \leq f_0\}$ and, for $p = 1, \ldots, k$, let $N_p = \{j \in N \mid f_0 + \frac{1}{k}(p-1)(1-f_0) < f_j \leq f_0 + \frac{1}{k} p(1-f_0)\}$. The inequality

$$\sum_{p=0}^{k} \sum_{j \in N_p} \Big(\lfloor a_j \rfloor + \frac{p}{k+1} \Big) x_j \;\leq\; \lfloor \bar{\beta} \rfloor \tag{8.8}$$

is valid for $P_{MK} := \mathrm{conv}(X_{MK})$ with $X_{MK} := \big\{ x \in \mathbb{Z}^N_{\geq 0} \mid a^T x \leq \bar{\beta} \big\}$ and dominates the CG cut (8.7).

Note. There is no dominance relation between strong CG cuts and MIR cuts.

In the same way as Gomory mixed integer cuts are MIR cuts for a row of the simplex tableau, Gomory fractional cuts are CG cuts for a tableau row. Following this similarity, our implementation of the strong CG cut separator generates cuts of type (8.8) for simplex tableau rows that belong to fractional integer variables. Thus, our strong CG cuts are actually "strong Gomory fractional cuts".

As for the MIR cuts, we have to perform a bound substitution and complementation step in order to achieve the standard form $x \geq 0$ for the bounds. Since strong CG cuts cannot handle continuous variables, we have to choose the complementation of continuous variables in such a way that their resulting coefficient in the base inequality is non-negative. If this is possible, we can relax the inequality by removing the continuous variables. If this is not possible, no strong CG cut can be generated.

Note. Both the MIR procedure of Proposition 8.3 and the strong CG procedure of Theorem 8.6 can be expressed as the application of a superadditive function to a linear inequality. They only differ in the shape of this function.

As Gomory mixed integer cuts are special types of MIR cuts, the "strong Gomory fractional cuts" produced by our strong CG cut separator are special types of the

more general strong CG cuts. Thus, it might make sense to also generate strong CG cuts in the fashion of the c-MIR separator in Algorithm 8.2 by applying an aggregation heuristic that tries to eliminate continuous variables. Overall, instead of having three different separators, one probably should combine the strong CG cut separator with the MIR and GMI cut separators. The combined strong CG and MIR cut separator would generate both the MIR and strong CG inequality in Steps 3 and 4 of Algorithm 8.2 and keep the one with larger violation. In the same fashion, the combined strong Gomory fractional and GMI cut separator would generate both type of cuts for each simplex tableau row belonging to a fractional integer variables and keep the more violated one.

8.5 Flow Cover Cuts

Flow cover cuts are based on the polyhedral study of the *0-1 single node flow problem*. They have been introduced by Padberg, van Roy, and Wolsey [181] and generalized by van Roy and Wolsey [208]. Later, Gu, Nemhauser, and Savelsbergh [110] applied a lifting procedure to strengthen flow cover cuts. Marchand [153] showed that by applying complemented mixed integer rounding on a particular mixed knapsack relaxation of the 0-1 single node flow set, one can obtain cuts that are equivalent to the lifted flow cover cuts.

The 0-1 single node flow set is defined as

$$X_{SNF} := \left\{(x,y) \in \{0,1\}^n \times \mathbb{R}^n_{\geq 0} \mid \sum_{j \in N^+} y_j - \sum_{j \in N^-} y_j \leq \bar{\beta},\ y_j \leq s_j x_j \text{ for all } j \in N\right\}.$$

This structure can be typically found in fixed charge network flow problems, which was the actual origin for the flow cover cuts [181]. In this application, the continuous variables y_j correspond to the flow over arcs j of maximal capacity s_j, and a non-zero flow over an arc entails a fixed charge cost that is triggered by the binary variable x_j. The sets N^+ and N^- consist of the incoming and outgoing arcs of a node, respectively. Then, the single node flow set imposes a constraint on the flow balance in the node.

Despite their very specific origin, flow cover inequalities are a class of cutting planes that can be applied to general mixed integer programs, because one can convert any constraint of a mixed binary program into the required form. General integer variables of a MIP have to be relaxed to continuous variables.

Given a single node flow set X_{SNF}, a *flow cover* is defined as a pair (V^+, V^-) with $V^+ \subseteq N^+$, $V^- \subseteq N^-$, and

$$\sum_{j \in V^+} s_j - \sum_{j \in V^-} s_j = \bar{\beta} + \lambda$$

with $\lambda > 0$. Thus, a flow cover is the continuous analogon to a knapsack cover, compare Definition 8.1. It reflects a configuration in which not all of the continuous variables can take their maximum value, i.e., not all arcs of the flow cover can be used with full capacity. The flow cover cuts that can be derived basically state that one has to either use fewer in-flow or more out-flow. For further details, we refer to the literature mentioned above and to the diploma thesis of Kati Wolter [218].

The SCIP version of the flow cover cut separator is an implementation of Marchand's c-MIR approach [153]. We convert each individual row of the LP relaxation

into a single node flow set and try to find a flow cover that will lead to a violated flow cover inequality. Then, following Marchand's suggestions, we try different subset selections and scaling factors, apply the c-MIR procedure for each of the choices, and add the most violated cut to the separation storage. The details of the implementation are described by Wolter [218].

8.6 Implied Bound Cuts

Implied bound cuts have been first used by Savelsbergh [199] and incorporated in Minto [172, 171]. The cut separator inspects the implication graph (see Section 3.3.5) and produces cutting planes to enforce implications that are violated by the current LP relaxation. Since the implications are consequences of the linear constraints and the integrality conditions, they can only be violated by the current LP solution if they contain an integer variable with fractional value. Therefore, we only have to scan a small part of the implication graph, which makes the implied bound cut separator very fast.

The most prominent application of the implied bound cut separator is the "on-demand" disaggregation of aggregated precedence relations, as it is shown in the following example.

Example 8.7. Consider the linear constraint

$$x_1 + x_2 + x_3 + x_4 + x_5 - 5y \leq 0 \tag{8.9}$$

with binary variables $x_j, y \in \{0,1\}$, $j = 1,\ldots,5$. This encodes the implications $y = 0 \rightarrow x_j = 0$ for all $j = 1,\ldots,5$. However, the direct representation of these implications as a system of linear inequalities

$$x_j - y \leq 0 \text{ for all } j = 1,\ldots,5 \tag{8.10}$$

yields a strictly stronger LP relaxation. For example, the fractional basic solution $x_1 = \ldots = x_4 = 1$, $x_5 = 0$, $y = 0.8$ is feasible for the aggregated Inequality (8.9) but violates System (8.10). The main disadvantage of System (8.10) is that it contains five instead of only one inequality, which usually slows down the LP solving. Therefore, the common approach is to initially use the aggregated Inequality (8.9) and let violated inequalities of System (8.10) be separated as implied bound cuts.

8.7 Clique Cuts

In the same paper where he introduced the implied bound cuts, Savelsbergh [199] proposed to derive clique inequalities to strengthen the LP relaxation of a MIP. The theoretical foundation of clique inequalities are described in Johnson and Padberg [125].

A clique inequality has the form

$$\sum_{j \in Q} x_j \leq 1$$

with $Q \subseteq B \cup \bar{B}$ being a subset of the binary variables $B \subseteq I \subseteq N$ and their complements $\bar{B}$. It expresses the logical constraint that at most one of the (complemented) variables in Q must be set to 1.

In order to separate clique inequalities one considers a stable set relaxation of the MIP. This consists of a graph $G = (V, E)$ with $V = B \cup \bar{B}$ and edges e_{uv} for all pairs of (complemented) binary variables for which we know that they cannot be both set to 1 at the same time. This graph is commonly called *conflict graph* in the MIP community, see for instance Atamtürk, Nemhauser, and Savelsbergh [24]. It is constructed during presolving, in particular by probing, see Section 10.6. Each clique in the conflict graph gives rise to a clique inequality, which is valid for the associated stable set polytope and thus also valid for the MIP. Other valid inequalities for the stable set polytope like, for example, odd-hole inequalities can also be used to strengthen the LP relaxation of the MIP, but it turned out that in practice, only the clique inequalities are generally useful for general mixed integer programming.

For the separation of violated clique inequalities one uses the LP values of the binary variables as weights for the nodes V in the conflict graph and searches for cliques with total weight larger than 1. Since the *maximum weighted clique* problem is $\mathcal{NP}$-hard [92], one has to resort to heuristics in order to efficiently separate clique cuts.

SCIP stores the knowledge about the incompatibility of pairs of (complemented) binary variables in the implication graph and in the clique table, see Section 3.3.5. Violated clique cuts are separated using the TCLIQUE algorithm of Borndörfer and Kormos [52]. This is a branch-and-bound based method that uses a list coloring relaxation for the bounding step. TCLIQUE is an exact algorithm and therefore able to find the most violated clique cut. However, we use it only in a heuristic fashion and do not enumerate the full branch-and-bound search tree to decrease the running time of the clique separator.

8.8 REDUCED COST STRENGTHENING

A simplex based LP solver provides reduced cost values for each column, which yield a lower bound on the change in the objective value for a unit change in the column's solution value. This information can be used for non-basic columns to tighten the opposite bound, a procedure that is called *reduced cost strengthening* (see Nemhauser and Wolsey [174]).

Since it does not cut off the current fractional LP solution, reduced cost strengthening is not a cutting plane separation method in its classical sense. However, as the point in time when the strengthening is applied is exactly the same as for cutting plane separators, namely after the solving of an LP relaxation, it is treated as a separator plugin in SCIP.

The procedure is very simple. Let $\check{x} \in \mathbb{R}^n$ be an optimal solution to the current LP relaxation with value $\check{c} = c^T\check{x}$, and let $\check{r}$ be the associated reduced cost vector. Furthermore, suppose an incumbent solution $\hat{x} \in \mathbb{Z}^I \times \mathbb{R}^C$ with value $\hat{c} = c^T\hat{x}$ is available. Then we can tighten the bounds for non-basic variables x_j, $j \in \mathcal{N}$, as follows:

- ▷ If $r_j > 0$ we have $x_j = \tilde{l}_j$ and we can deduce $x_j \leq \tilde{l}_j + \frac{1}{r_j}(\hat{c} - \check{c})$.
- ▷ If $r_j < 0$ we have $x_j = \tilde{u}_j$ and we can deduce $x_j \geq \tilde{u}_j + \frac{1}{r_j}(\hat{c} - \check{c})$.
- ▷ If $j \in I$, we can round down the upper bound and round up the lower bound.

Note that the tightened bounds are only valid for the current subproblem Q and the underlying subtree. Global reduced cost fixing can only be applied at the root node.

Since improved incumbent solutions found during the search can yield tighter global bounds due to additional root node reduced cost strengthening, the procedure is repeated for the root node each time a new incumbent has been found. This is the task of the *root reduced cost strengthening* propagator of Section 7.7.

An issue with the locally valid domain reductions produced by the reduced cost strengthening separator is the additional bookkeeping overhead. Since it does not cut off the current LP solution, reduced cost strengthening has no direct effect on the dual bound and often restricts only an "uninteresting" part of the problem. Useful consequences can only follow from subsequent domain propagations that exploit the integrality information. Therefore, we install the new bound only if the variable is of integer type or if at least 20 % of the local domain of a continuous variable is cut off. Additionally, we demand for continuous variables that the new bound cuts off parts of the "active region" of the variable, which is the smallest interval that contains all LP solution values $\check{x}_j$ that have ever been produced during the branch-and-bound search.

8.9 Cut Selection

Almost as important as the finding of cutting planes is the selection of the cuts that actually should enter the LP relaxation. In early years of cutting plane algorithms, one passed the cuts one by one to the LP and immediately resolved the LP after the addition of each cut. This was inspired by Gomory's algorithm [102] for solving integer programs by a pure cutting plane approach. Nowadays, we know that it is much better to add cutting planes in rounds, see for instance Balas et al. [31]. This means, after the solving of one LP we generate many different cuts which all cut off the current fractional LP solution. However, as the computational experiments of Section 8.10 show, adding all of these cuts to the LP does not yield the best overall performance. Therefore, a selection criterion is needed in order to identify a "good" subset of the generated cuts.

Balas, Ceria, and Cornuéjols [30] and Andreello, Caprara, and Fischetti [13] proposed to base the cut selection on *efficacy* and *orthogonality*. The efficacy is the Euclidean distance of the cut hyperplane to the current LP solution, and an orthogonality bound makes sure that the cuts added to the LP form an almost pairwise orthogonal set of hyperplanes. SCIP follows these suggestions. The detailed procedure to select the cuts is described in Section 3.3.8 on page 48.

8.10 Computational Results

In the previous sections we outlined the SCIP implementation of various cutting plane separation algorithms. In this section we evaluate their performance impact by computational experiments. Following the analysis of Bixby et al. [46], we investigate three different situations: first, we compare the default cut-and-branch algorithm to pure branch-and-bound without cutting plane separation in order to measure the overall impact of cutting planes. Second, we disable only one of the separators at a time and compare each of these settings to the defaults in which all cutting plane separators are enabled. This shows the contribution of the individual cuts to the performance of cut-and-branch. Third, we enable only one of the separators and measure the difference to pure branch-and-bound in order to assess the power of the

	test set	none	no knap	no c-MIR	no Gom	no SCG	no flow	no impl	no cliq	no rdcost	cons Gom
time	MIPLIB	+443	+5	+35	−15	−5	−9	+1	−6	+21	−6
	CORAL	+51	+4	+3	+12	−3	−1	−12	+2	+1	−15
	MILP	+6	+4	+18	−4	−7	−4	−9	−7	+5	−5
	ENLIGHT	−47	+51	−33	−40	−13	−30	−49	−3	−1	−4
	ALU	−21	+4	−16	−17	+16	−6	−34	−15	+11	−16
	FCTP	+152	+12	+9	−5	−1	+14	0	+3	0	−8
	ACC	+26	−1	−2	−28	+13	−2	+38	+150	−6	+8
	FC	+2433	+11	+99	−17	+5	+22	+5	0	+1	−9
	ARCSET	+104	−2	+50	+17	+1	−5	+12	0	+11	+9
	MIK	+10606	0	+131	+46	+6	+32	+24	−2	+404	+17
	total	**+117**	**+5**	**+18**	−4	−4	−3	**−7**	−1	**+11**	**−8**
nodes	MIPLIB	+2958	+13	+157	−1	−2	+3	+10	+7	+20	+2
	CORAL	+290	+6	+15	+54	+1	+6	−19	+12	+3	0
	MILP	+81	+9	+48	+3	−5	−3	−4	+5	+12	−4
	ENLIGHT	+34	+108	−21	−18	+13	−15	−31	0	+6	+19
	ALU	+7	−13	−14	−25	+4	+18	−50	−14	+14	−27
	FCTP	+898	+28	+25	+21	−2	+26	0	0	+3	+10
	ACC	+164	0	0	−22	+31	0	+98	+304	+9	+45
	FC	+81239	+39	+855	−7	+16	+169	+17	0	+1	−11
	ARCSET	+717	0	+141	+96	+4	−9	+28	0	+27	+57
	MIK	+8575	0	+211	+64	+5	+39	+32	0	+237	+23
	total	**+526**	**+10**	**+63**	**+16**	0	**+5**	−4	**+10**	**+14**	**+2**

Table 8.1. Performance effect of disabling individual cutting plane separators for solving MIP instances. The values denote the percental changes in the shifted geometric mean of the runtime (top) and number of branching nodes (bottom) compared to the default settings in which all cut separators are enabled. Positive values represent a deterioration, negative values an improvement.

cut separators in an isolated environment.

Table 8.1 shows the summary for the first two experiments. Detailed results can be found in Tables B.61 to B.70 in Appendix B. Column "none" gives the performance ratios for disabling all separators. One can see that cutting planes yield an overall performance improvement of more than a factor of 2. The largest deterioration can be observed for the MIK instances. Here, the solving time increases by more than a factor of 100 if cutting plane separation is disabled. In fact, none of the 41 instances in the MIK test set can be solved by pure branch-and-bound within the time limit of one hour, while the *total time* to solve all 41 instances with cut-and-branch is only 44 minutes with a geometric mean of 30 seconds per instance.

The only test sets for which cutting planes increase the solving time are the ENLIGHT and ALU instances. As noted earlier, they are basically pure feasibility problems with unimportant or artificial objective functions. Cutting planes focus on the region in the LP polyhedron where the LP solver has located an optimal solution. If they can only cut off a relatively small part of the polyhedron, they are usually not of much use for instances with artificial objective function since there would be almost no gain in driving the LP solution to a different region. An exception would be the case that the new LP solution is integral, but since there are only very few feasible solutions in the ENLIGHT instances and no solutions at all in the ALU instances, this is unlikely or impossible, respectively. Hence, it is no surprise that cutting planes deteriorate the performance, because they increase the size of the LP relaxations that are solved in the subproblems.

The remaining columns, except "cons Gom", show the outcomes of the second experiment which is to disable individual cut separators while leaving the other separators active. The first observation is that the sum of the changes in the runtime is much smaller than the deterioration arising from disabling all separators. The same holds for the number of branching nodes. This means, the strengths of the individual

cuts overlap: the deactivation of one class of cutting planes can be compensated by a different cut separator. This behavior is supported by theoretical observations. For example, Nemhauser and Wolsey [175] showed that the elementary closures of mixed integer rounding cuts and Gomory mixed integer cuts are identical. Cornuéjols and Li [69] provide a detailed overview of this and other relations between elementary closures. Another example for the overlap of cutting plane classes are the flow cover cuts of SCIP, which are generated as a special case of complemented mixed integer rounding. Nevertheless, such a "special case" cut separation algorithm is not necessarily superfluous since most of the cuts, in particular c-MIR cuts, are separated in a heuristic fashion.

The second observation is that disabling any of the separators except the implied bound and strong CG cut separators leads to an overall increase in the number of branching nodes. This does not, however, come along with an increase in the runtime in every case. By looking at the individual columns in Table 8.1 one can observe that the performance degrades only for disabling knapsack cuts ("no knap"), c-MIR cuts ("no c-MIR"), and reduced cost strengthening ("no rdcost"). Although for disabling each of the other cuts there is a test set with moderate or large deterioration, the totals show a small improvement compared to the default settings with all cut separators enabled. The most surprising result is the one for the Gomory mixed integer cuts ("no Gom"). Bixby et al. [46] detected these cuts to be the most effective in Cplex 6.5 with a sizable winning margin to knapsack cover cuts. At this time, however, Cplex did not contain complemented mixed integer rounding cuts.

Wolter [218] observed that the implementation of Gomory cuts in an earlier version of SCIP was inferior to the one of Cplex and CBC with respect to the objective gap closed at the root node. Therefore, we adjusted the parameter settings of the Gomory cut separator to be more aggressive and to produce more cuts, which resolved the issue of the inferior root node objective gap. However, the results of Table 8.1 show that we overshot the mark: replacing the aggressive Gomory cut separator of the default settings by a more conservative version ("cons Gom"), which adds only those cuts that appear to be numerically stable, yields a performance improvement that goes beyond the one of just deactivating the aggressive Gomory cut separation ("no Gom").

Table 8.2 comprises the summary of the third benchmark, which is to compare pure branch-and-bound with activating a single cutting plane separator. Detailed results can be found in Tables B.71 to B.80 in Appendix B. It is noticeable that all cutting plane classes help to reduce the number of branching nodes and also—with the exception of the clique cuts—to improve the runtime performance. As in the previous experiment, the complemented mixed integer rounding cuts are the clear winner. The Gomory and flow cover cuts, however, come in second and third place in Table 8.2 although their activation slightly deteriorates the overall performance if they are combined with all other separation algorithms, see Table 8.1. We take this as another evidence for their functionality being sufficiently covered by the c-MIR cuts. The conservative Gomory cut separator ("cons Gom") yields, if used in an isolated fashion, a similar performance improvement as its more aggressive variant ("Gomory"). Nevertheless, we already saw in Table 8.1 that this situation changes considerably if all cut separators are used simultaneously. Here, the conservative version improves the overall performance, while the aggressive variant leads to a deterioration.

	test set	knapsack	c-MIR	Gomory	strong CG	flow cover	impl bds	clique	redcost	cons Gom
time	MIPLIB	−35	−72	−50	−28	−40	−16	0	−20	−48
	CORAL	+15	−18	−25	+13	+7	−5	−1	−5	−27
	MILP	+5	−12	+4	+2	−4	−2	+11	−2	+8
	ENLIGHT	0	−65	+52	−35	−50	+2	0	+30	+9
	ALU	−3	−10	+16	−25	−7	0	−14	+8	−32
	FCTP	−52	−22	+15	+2	−57	+1	+2	+1	−3
	ACC	−1	+2	+54	−15	0	−43	−39	−3	+9
	FC	−90	−95	−7	+2	−55	+1	+2	−2	−7
	ARCSET	−2	−38	−36	−28	+7	0	−2	+17	−45
	MIK	0	0	−69	0	−65	0	0	−44	−67
	total	**−11**	**−40**	**−22**	**−6**	**−18**	**−7**	+1	**−7**	**−24**
nodes	MIPLIB	−51	−89	−70	−44	−57	−23	−7	−24	−61
	CORAL	+15	−39	−49	+19	−10	−15	−11	−15	−36
	MILP	+2	−29	−5	−4	+2	−8	−4	−8	−1
	ENLIGHT	0	−53	−7	−61	−41	0	0	+14	−30
	ALU	−14	−39	−21	−57	−2	−17	−30	+4	−54
	FCTP	−75	−45	−10	0	−82	0	0	0	−15
	ACC	0	0	+22	−50	0	−58	−50	−8	+2
	FC	−95	−99	−38	0	−75	−1	−1	−5	−25
	ARCSET	+1	−56	−66	−51	−6	0	+1	−11	−64
	MIK	−1	+2	−69	−3	−62	+1	+2	−19	−66
	total	**−21**	**−62**	**−44**	**−16**	**−29**	**−15**	**−9**	**−14**	**−36**

Table 8.2. Performance effect of enabling individual cutting plane separators for solving MIP instances. The values denote the percental changes in the shifted geometric mean of the runtime (top) and number of branching nodes (bottom) compared to a pure branch-and-bound approach for which all cut separators are disabled. Positive values represent a deterioration, negative values an improvement.

GENERATING CUTS AT LOCAL NODES

The default settings turn SCIP into a cut-and-branch algorithm, which means that cuts are only generated at the root node. The advantage of this approach is the smaller subproblem LP management overhead: in switching from one subproblem to the next, we only have to update the bounds of the columns in the LP while the rows stay untouched. There is, however, one exception: the reduced cost strengthening. Since this separator is a very special case that only produces bound changes, it does not introduce significant overhead to the LP management and is therefore called at every node in the tree.

In the following we evaluate a branch-and-cut approach for which the local LP relaxations are extended by additional cutting planes. Apart from the bound changes due to reduced cost strengthening, all separated cuts are globally valid. We consider six different settings and provide a summary of the benchmarks in Table 8.3. The details can be found in Tables B.81 to B.90.

The column labeled "all (1)" gives the results for calling all cut separators at every local node. Indeed, this gives a large reduction of 70 % in the average number of branching nodes needed to solve the instances. However, the runtime overhead is significant: overall, the average time to solve the instances is more than twice as large as for the default cut-and-branch.

In order to reduce the overhead in the runtime, we experimented with alternative parameter settings for which cuts are only separated at a local node if the node's dual bound is equal to the global dual bound. The idea is that for these nodes it is most important to improve the dual bound since they are defining the current global dual bound. Note that with the *best first search* node selection, this approach would also generate cutting planes at every local node. However, since the default

	test set	all (1)	all (1*)	all (10*)	impl bds (1*)	knapsack (1*)	impl/knap (1*)
time	MIPLIB	+162	+45	+7	−4	−3	−3
	CORAL	+171	+28	+2	−1	−3	−7
	MILP	+95	+37	+17	−1	0	+1
	ENLIGHT	+136	+66	+11	−18	−13	−6
	ALU	+196	+29	+2	−13	−25	+14
	FCTP	+123	+21	+8	+1	−5	−5
	ACC	+9	+9	−1	+28	−2	+25
	FC	+62	+6	+1	0	+1	+6
	ARCSET	+631	+69	+6	+1	+7	+8
	MIK	+56	−29	−12	+2	−23	−18
	total	**+137**	**+33**	**+7**	**−2**	**−3**	**−2**
nodes	MIPLIB	−63	−29	−7	−7	−5	−10
	CORAL	−73	−45	−13	−5	−5	−12
	MILP	−70	−38	−10	0	−4	−5
	ENLIGHT	−95	−53	+5	−33	−11	−30
	ALU	−47	+1	+3	−17	−16	+21
	FCTP	−72	−22	−1	0	−9	−8
	ACC	−45	−41	−23	+43	0	+43
	FC	−78	−36	−2	−4	−7	−9
	ARCSET	−45	−11	−6	−1	−3	−4
	MIK	−91	−69	−30	0	−42	−43
	total	**−70**	**−38**	**−10**	**−4**	**−6**	**−9**

Table 8.3. Performance effect of separating cuts at local nodes. The values denote the percental changes in the shifted geometric mean of the runtime (top) and number of branching nodes (bottom) compared to the default cut-and-branch approach in which cuts are only separated at the root node. Positive values represent a deterioration, negative values an improvement.

node selection rule performs plunging (see Chapter 6), cuts are separated to a lesser extent. The results are shown in column "all (1*)". As one can see, some of the node reduction could be preserved, but the runtime overhead is much smaller than for "all (1)". Unfortunately, branch-and-cut with this more conservative local cut generation is still inferior to cut-and-branch.

Column "all (10*)" denotes the setting in which we produce cuts at a local node if its dual bound is equal to the global dual bound and its depth in the search tree is divisible by 10. This yields an even smaller degree of local cut generation, but as before, the results remain unsatisfactory.

Finally, we investigate the generation of only one type of cuts at local nodes instead of calling all separation algorithms simultaneously. In particular, we applied the relatively cheap implied bound cut ("impl bds (1*)") and knapsack cover cut separation ("knapsack (1*)") at the local nodes defining the global dual bound. Here, the results are much more promising: for both separators applied individually, the number of nodes and the runtime is slightly smaller than with pure cut-and-branch. Unfortunately, separating both types of cutting planes simultaneously at local nodes, as shown in column "impl/knap (1*)", does not combine their benefits: although the number of nodes decreases, the average solving time is not smaller than for calling only one of the two cut separators alone.

Cut Selection

After having investigated the impact of the individual cutting plane separators, we now focus on the cut selection.

Cutting planes are generated in rounds, see Section 3.2.8, and all cuts of a single round cut off the current LP solution. Then, some of these cuts are added to the

	test set	one per round	take all	no obj paral	no ortho
time	MIPLIB	+97	+105	+4	+28
	CORAL	+98	+38	−8	+19
	MILP	+27	+45	−7	+11
	ENLIGHT	+85	−17	−3	+8
	ALU	−3	+1	+18	−16
	FCTP	+7	+69	+6	+17
	ACC	+1853	+351	−22	+313
	FC	+80	+694	+7	+102
	ARCSET	+35	+49	+6	+1
	MIK	+247	+676	+7	+8
	total	**+79**	**+71**	**−3**	**+22**
nodes	MIPLIB	+3	−8	+2	−7
	CORAL	+3	−7	−9	−3
	MILP	−4	−1	−8	−21
	ENLIGHT	+95	−29	−2	+10
	ALU	−13	−24	+18	−25
	FCTP	−10	−7	+10	+2
	ACC	+258	+128	−32	+85
	FC	+140	−24	+6	+11
	ARCSET	+50	−42	+12	0
	MIK	+312	+27	+4	−6
	total	**+11**	**−5**	**−5**	**−8**

Table 8.4. Performance effect of different cutting plane selection strategies for solving MIP instances. The values denote the percental changes in the shifted geometric mean of the runtime (top) and number of branching nodes (bottom) compared to the default procedure given in Section 3.3.8. Positive values represent a deterioration, negative values an improvement.

LP, and the modified LP is solved afterwards, followed by the next round of cutting plane separation.

The task of the cutting plane selection is to decide which of the cuts of a single separation round should enter the LP and which should be discarded. Algorithm 3.2 on page 49 shows the procedure that is employed by SCIP. The computational results for some variants of cutting plane selection can be found in the summary Table 8.4 and in the detailed Tables B.91 to B.100.

The columns "one per round" and "take all" denote the trivial strategies of adding only the most violated (in the Euclidean norm) or all of the generated cuts, respectively. As the results show, both rules are clearly inferior to the more involved selection that is applied in the default settings. The columns "no obj paral" and "no ortho" evaluate the two supplementary criteria *objective parallelism* and *orthogonality* of the cut selection Algorithm 3.2. In the former settings, we disabled the parallelism measurement by setting $w_p = 0$. For the latter, we set $w_o = 0$ and minortho $= 0$. One can see that the parallelism to the objective function does not play a significant role; disabling this criterion even yields a slight performance improvement. In contrast, the orthogonality within the set of selected cuts seems to be crucial: selecting the cuts in this way improves the overall performance by 22 %. The largest impact can be seen on the ACC and FC instances, for which the runtimes quadruple and double, respectively, if the orthogonality calculation is disabled.

Chapter 9

Primal Heuristics

The branch-and-bound algorithm to solve mixed integer programs is a so-called *complete* procedure. This means, apart from numerical issues, it is guaranteed to find the optimal solution for every problem instance in a finite amount of time. However, it is a very expensive method and has a worst case runtime which is exponential in the size of the problem instance.

In contrast, primal heuristics are *incomplete* methods. They try to find feasible solutions of good quality in a reasonably short period of time, but there is no guarantee that they will succeed in finding a solution, least of all an optimal solution. Nevertheless, primal heuristics have a significant relevance as supplementary procedures inside a complete MIP solver: they help to find good feasible solutions early in the search process. An early discovering of a feasible solution has the following advantages:

- ▷ It proves that the model is feasible, which is an indication that there is no error in the model.
- ▷ A user may already be satisfied with the quality of the heuristic solution, such that he can abort the solving process at an early stage.
- ▷ Feasible solutions help to prune the search tree by bounding, thereby reducing the work of the branch-and-bound algorithm.
- ▷ The better the current incumbent is, the more reduced cost fixing and other dual reductions can be applied to tighten the problem formulation, see Sections 7.6 and 7.7.

There exists a large variety of mixed integer programming heuristics proposed in the literature, including Hillier [116], Balas and Martin [33], Saltzman and Hillier [197], Glover and Laguna [95, 96, 97], Løkketangen and Glover [150], Glover et al. [98], Nediak and Eckstein [170], Balas et al. [32, 34], Fischetti and Lodi [85], Fischetti, Glover, and Lodi [84], Bertacco, Fischetti, and Lodi [40], Danna, Rothberg, and Le Pape [72], and Achterberg and Berthold [2]. Some of the heuristics available in SCIP are direct implementations of ideas found in the literature, some are variations of the proposals, and some of them are newly developed techniques.

The primal heuristics of SCIP can be grouped into four categories:

- ▷ Rounding heuristics try to round the fractional values of an LP solution such that the rounded integral vector stays feasible for the constraints.
- ▷ Diving heuristics start from the current LP solution and iteratively fix an integer variable to an integral value and resolve the LP.
- ▷ Objective diving heuristics are similar to diving heuristics, but instead of fixing the variables by modifying the bounds, they drive the variables into a desired direction by modifying their objective coefficients.

▷ Improvement heuristics consider one or more primal feasible solutions that have been previously found and try to construct an improved solution with better objective value.

We describe the individual heuristics contained in these classes only very briefly in Sections 9.1 to 9.4 and present computational results in Section 9.5. Detailed descriptions of the algorithms and an in-depth analysis of their computational impact can by found in the diploma thesis of Timo Berthold [41].

RENS, *Octane*, and all of the improvement heuristics in Section 9.4 have been implemented by Timo Berthold. The implementation of the *feasibility pump* is joint work with Timo Berthold. The remaining heuristics have been implemented by the author of this thesis.

9.1 Rounding Heuristics

Rounding heuristics start with a fractional vector $\check{x} \in \mathbb{R}^n$ that satisfies the linear constraints $A^T x \leq b$ and bounds $l \leq x \leq u$ of the MIP but violates some of the integrality restrictions. Usually, we take the optimal solution of the LP relaxation Q_{LP} of the current subproblem Q as starting point. Now, the task of a classical rounding heuristic is to round the fractional values $\check{x}_j$, $j \in F := \{j \in I \mid \check{x}_j \notin \mathbb{Z}\} \subseteq I$, of the integer variables down or up such that the final integral vector $\tilde{x} \in \mathbb{Z}^I \times \mathbb{R}^C$ still satisfies all linear constraints. More involved heuristics of this type also modify continuous variables or integer variables that already have integral values in order to restore linear feasibility which has been lost due to the rounding of fractional variables.

9.1.1 RENS

Let $\check{x} \in \mathbb{R}^n$ be an optimal solution of the current subproblem's LP relaxation, and let $F \subseteq I$ be the set of integer variables with fractional value. Then, there are $2^{|F|}$ possible roundings of $\check{x}$. The *relaxation enforced neighborhood search* (RENS) heuristic, which was invented by Berthold [41], constructs a sub-MIP

$$(\text{MIP}') \quad c^\star = \min \{c^T x \mid Ax \leq b,\ x \in \mathbb{R}^n,\ x_j \in \{\lfloor \check{x}_j \rfloor, \lceil \check{x}_j \rceil\} \text{ for all } j \in I\}$$

and solves it with SCIP in order to find the best possible feasible rounding if one exists. Note that in this sub-MIP all integer variables with integral values are fixed, and the ones with fractional values are restricted to be rounded down or up. This means, we can convert all integer variables into binary variables during presolving, see Section 10.5, to obtain a mixed binary program.

The issue with RENS is that the sub-MIP might still be very expensive to solve. Therefore, we only apply RENS at the root node of the search tree and abort the sub-MIP solving process after either a total of 5000 nodes has been processed or no improvement of the sub-MIP incumbent has been found for 500 consecutive nodes. Additionally, we skip the heuristic if $|F| > \frac{1}{2}|I|$ since this suggests that the sub-MIP is not sufficiently easier than the original MIP. Apart from the number of integral variables, which is an indicator for the "IP complexity" of the model, the *total* number of variables is also relevant for the solving speed of the sub-MIP since it gives a hint of the "LP complexity" of the instance. Therefore, we compare the total number of variables of the sub-MIP after presolving to the total number of

variables in the presolved version of the original instance. We only proceed with the sub-MIP solving process if the number of variables has been reduced by at least 25 %.

9.1.2 Simple Rounding

As the name suggests, *simple rounding* is a very simple and fast heuristic to round a fractional solution to a feasible integral vector. The idea is based on the *variable lock* numbers, see Definition 3.3 on page 38. Consider a variable x_j, $j \in F \subseteq I$, with fractional LP solution $\check{x}_j$. If $\zeta_j^- = 0$, we can safely set $\tilde{x}_j := \lfloor \check{x}_j \rfloor$ without violating any linear constraint. On the other hand, if $\zeta_j^+ = 0$, we can set $\tilde{x}_j := \lceil \check{x}_j \rceil$. The heuristic will succeed if all fractional variables $j \in F$ have either $\zeta_j^- = 0$ or $\zeta_j^+ = 0$.

Since the simple rounding heuristic is very fast, it is applied after the solving of every LP. This includes the intermediate LPs in the cutting plane separation loop and the LPs that are solved by other heuristics such as the diving heuristics of Section 9.2.

9.1.3 Rounding

The *rounding* heuristic is more involved than the *simple rounding* heuristic, but the solutions found by rounding are a superset of the ones that can be found by *simple rounding*. The rounding is applied to all fractional variables $j \in F$, even if this leads to an infeasibility in the linear constraints. After having generated an infeasibility it tries to select the rounding of the next variable such that the infeasibility is reduced or eliminated.

In a first step, the heuristic sets $\tilde{x} := \check{x}$ and calculates the activities $\alpha_i(\tilde{x}) = (a^i)^T \tilde{x}$ of the linear constraints. Then, it iterates over the set F of fractional variables. If the current activity vector is feasible, i.e., $\underline{\beta} \leq \alpha(\tilde{x}) \leq \overline{\beta}$, we select a fractional variable with the largest number of variable locks $\max\{\zeta_j^-, \zeta_j^+\}$ and round it into the "more feasible" direction, i.e., downwards if $\zeta_j^- \leq \zeta_j^+$ and upwards otherwise. The rationale behind this choice is to avoid the roundings that can break the feasibility of many constraints.

If the current activity vector violates one or more constraints, we select one of the constraints and try to find a fractional variable that can be rounded in a direction such that the violation of the constraint is decreased. If there is a choice, we select a variable with a minimum number of variable locks in the corresponding direction in order to decrease the feasibility of as few other constraints as possible. If no rounding can decrease the violation of the infeasible constraint, we abort.

In the default settings, rounding is also called after the solving of every LP in the search tree, but not on the LPs that are solved inside other heuristics.

9.1.4 Shifting

The *shifting* heuristic is equal to the *rounding* heuristic, but it tries to continue in the case that no rounding can decrease the violation of an infeasible constraint. In this case, we shift the value of a continuous variable or an integer variable with integral value in order to decrease the violation of the constraint. In the process, we have to make sure to not run into a cycle. Therefore, we penalize the repeated shifting in opposite directions of a variable, we randomize the selection of the infeasible row

for which the violation should be reduced, and we abort after 50 successive shiftings have been performed without decreasing the number of violated rows or the number of fractional variables.

As *shifting* is already quite time consuming, we only call it in every 10'th depth level of the search tree.

9.1.5 INTEGER SHIFTING

Integer shifting is closely related to *shifting*, but it deals differently with continuous variables. The first step is to relax all constraints in order to remove the continuous variables. For each individual constraint $a^T x \leq \bar{\beta}$ this is performed by moving the minimal contribution $a_j x_j$ of the continuous variables x_j, $j \in C$, to the right hand side. This is performed by substituting the variable with its lower bound l_j if $a_j \geq 0$ and with its upper bound u_j if $a_j < 0$.

Then, we apply the *shifting* heuristic in order to find an integer solution for the relaxed inequality system. If this was successful, we go back to the original MIP, fix all integer variables to their values in the solution for the relaxed system, and solve the resulting LP to get optimal values for the continuous variables. If the LP is feasible, we have found a feasible solution for the original MIP.

9.1.6 OCTANE

Octane is a heuristic for pure binary programs which is due to Balas et al. [32]. The name is an acronym for "octahedral neighborhood search". The idea is to associate every vertex of the hypercube $[0,1]^n$ with the corresponding facet of the hypercube's dual polytope, the n-dimensional octahedron. Then, starting from the point in the octahedron that corresponds to the LP solution, rays are shot in various directions, and it is tested whether one of the facet defining hyperplanes that are hit by some ray corresponds to a feasible point $\tilde{x} \in \{0,1\}^n$, $A\tilde{x} \leq b$. It can be exploited that successive facet hyperplanes hit by a ray correspond to 0/1 vectors that only differ in one entry, which makes the updating procedure in the ray tracing algorithm very efficient.

As Balas et al. proposed, we apply *Octane* only on the fractional subspace $\mathbb{R}^F$, i.e., all variables x_j, $j \in I \setminus F$, with integral value are fixed to their current values. This approach significantly speeds up the computations without decreasing the heuristic's ability to find good solutions too much. Applying *Octane* only on the fractional variables means to search for a rounded solution $\tilde{x}$ of $\check{x}$. Therefore, this version of *Octane* is indeed a rounding heuristic in the classical sense.

9.2 DIVING HEURISTICS

The general principle of diving heuristics is illustrated in Algorithm 9.1. Starting with an optimal solution of the current subproblem's LP relaxation, we round a fractional variable, propagate the bound change, and resolve the LP. Since the LP basis stays dual feasible after changing the bounds of the variables, the resolve can be efficiently conducted using the dual simplex algorithm. The rounding and resolving procedure is iterated until either the LP becomes infeasible or an integral solution is found. One level of backtracking may optionally be applied in Step 8.

Of course, we use additional conditions to control the termination of the diving

Algorithm 9.1 Generic Diving Heuristic

Input: Optimal LP solution $\check{x}$ of current subproblem.

Output: If available, one or more feasible integral solutions.

1. Set $\tilde{x} := \check{x}$.
2. If $F := \{j \in I \mid \tilde{x}_j \notin \mathbb{Z}\} = \emptyset$, stop and return the feasible integral solution $\tilde{x}$.
3. Apply the *simple rounding* heuristic of Section 9.1.2 on $\tilde{x}$ to potentially produce an intermediate feasible integral solution.
4. Choose a fractional variable x_j, $j \in F$, and a rounding direction.
5. If down rounding is selected, tighten $\tilde{u}_j := \lfloor \tilde{x}_j \rfloor$. Otherwise, tighten $\tilde{l}_j := \lceil \tilde{x}_j \rceil$.
6. Call domain propagation to propagate the tightened bound.
7. Resolve the LP relaxation with the new bounds.
8. (optional) If the LP is infeasible, undo the previous propagations, apply the opposite rounding, propagate, and resolve the LP again.
9. If the LP is still infeasible, stop with a failure. Otherwise, let $\tilde{x}$ be the new optimal solution and goto Step 2.

loop in order to avoid too expensive dives. In SCIP, the main abort criterion is based on the total number of simplex iterations used for the LP resolves in Steps 7 and 8. For most heuristics we demand that this number must stay below 5 % of the current total number of simplex iterations used for solving the regular LP relaxations of the branch-and-bound nodes.

The only difference in the following implementations of the generic diving heuristic is how the selection in Step 4 is conducted. A common feature that is shared by all diving heuristics is that variables x_j are avoided in the selection that have $\zeta_j^- = 0$ or $\zeta_j^+ = 0$, since they can be rounded to integral values anyway by the *simple rounding* heuristic in Step 3. Additionally, we usually prefer binary variables over general integer variables, since in most MIP models the binary variables represent the most crucial decisions.

9.2.1 Coefficient Diving

Coefficient Diving selects a variable x_j in Step 4 of Algorithm 9.1 that minimizes $\min\{\zeta_j^-, \zeta_j^+\}$ but, as noted above, has $\min\{\zeta_j^-, \zeta_j^+\} \geq 1$. The variable is rounded in the direction of the smaller variable lock number. As a tie breaker, we chose a variable that has the smallest distance from its fractional value to the rounded value.

The rationale behind this selection is that this rounding reduces the feasibility of only a small number of constraints and thereby hopefully leads to a small number of violated constraints that have to be fixed by the LP resolve.

9.2.2 Fractionality Diving

In *fractionality diving* we select a variable with minimal fractionality $\phi(\tilde{x}_j) = \min\{\tilde{x}_j - \lfloor \tilde{x}_j \rfloor, \lceil \tilde{x}_j \rceil - \tilde{x}_j\}$, which is rounded to the nearest integer. This selection rule seems to be a very natural choice as it tries to produce a rounding that stays close to the LP solution.

9.2.3 GUIDED DIVING

Guided diving was invented by Danna, Rothberg, and Le Pape [72]. As it needs a feasible integral solution as input, it can also be viewed as an improvement heuristic. However, it perfectly fits into the generic diving scheme of Algorithm 9.1 and is therefore presented here.

Danna et al. propose to use *guided diving* inside the branch-and-bound search tree as a child selection rule during a plunge, compare Chapter 6: the variable to branch on is selected by the regular branching rule, but the child to process next is chosen such that the value of the branching variable is driven into the direction of the variable's value $\hat{x}_j$ in the current incumbent solution. The hope is that there are more feasible solutions in the vicinity of the current incumbent, and therefore, the search should be guided to that area.

In contrast, *guided diving* as implemented in SCIP works outside the tree just like all other diving heuristics. Therefore, we have the additional choice of selecting the variable to round. Similar to *fractionality diving*, we select the variable that is closest to its value in the incumbent solution.

9.2.4 LINE SEARCH DIVING

The *line search diving* heuristic converts the child selection idea of Martin [159], see Section 6.1, into a diving heuristic: if a variable x_j with value $(\check{x}_R)_j$ in the root node of the search tree has now a value $\tilde{x}_j < (\check{x}_R)_j$, the value of the variable seems to be pushed downwards on the path from the root node to the current node. Therefore, the idea is to further reinforce this pushing by rounding the variable down.

The geometric interpretation is as follows. We connect the root node LP solution $\check{x}_R$ with the current solution $\tilde{x}$ by a line and extend this line until we hit an integer value for one of the fractional integer variables. The first variable for which this happens is rounded into the corresponding direction. Algebraically spoken, we round the variable which has the smallest distance ratio $\frac{\tilde{x}_j - \lfloor \tilde{x}_j \rfloor}{(\check{x}_R)_j - \tilde{x}_j}$ for $\tilde{x}_j < (\check{x}_R)_j$ and $\frac{\lceil \tilde{x}_j \rceil - \tilde{x}_j}{\tilde{x}_j - (\check{x}_R)_j}$ for $\tilde{x}_j > (\check{x}_R)_j$.

9.2.5 PSEUDOCOST DIVING

In Step 4 of Algorithm 9.1, *pseudocost diving* selects the variable with respect to the pseudocost values Ψ_j^- and Ψ_j^+ that have been collected during the search process. The pseudocosts give an estimate for each integer variable x_j on how much the LP objective value increases per unit change of the variable, see Section 5.3.

For each fractional variable x_j, $j \in F$, we decide whether we want to round it down or up. The primary criterion is similar as in the *line search diving* heuristic: if the difference of the current value $\tilde{x}_j$ to the root LP value $(\check{x}_R)_j$ gives a strong indication that the variable is pushed to a certain direction, we select this direction for the rounding. To be more specific, if $\tilde{x}_j < (\check{x}_R)_j - 0.4$, we choose downwards rounding, if $\tilde{x}_j > (\check{x}_R)_j + 0.4$, we select upwards rounding. If the difference to the root solution does not yield a decision, we look at the fractional part of the variable. If $\tilde{x}_j - \lfloor \tilde{x}_j \rfloor < 0.3$ we round down, if $\tilde{x}_j - \lfloor \tilde{x}_j \rfloor > 0.7$, we round up. If this still does not lead to a conclusion, we round down if $\Psi_j^- < \Psi_j^+$ and round up otherwise.

After having decided for each variable in which direction it should be rounded,

we select a variable for the actual rounding that maximizes

$$\sqrt{\lceil \tilde{x}_j \rceil - \tilde{x}_j} \cdot \frac{1+\Psi_j^+}{1+\Psi_j^-} \text{ (downwards)} \quad \text{or} \quad \sqrt{\tilde{x}_j - \lfloor \tilde{x}_j \rfloor} \cdot \frac{1+\Psi_j^-}{1+\Psi_j^+} \text{ (upwards)},$$

respectively. This measure combines the fractionality of the variable with a pseudo-cost ratio. It prefers variables that are close to their rounded value and for which the estimated objective change per unit is much smaller in the selected direction than in the opposite direction.

9.2.6 Vector Length Diving

The *vector length diving* heuristic is specifically tailored towards set covering and set partitioning models, although it can also be applied to general MIPs. The idea is to choose a rounding that covers the largest number of constraints with the smallest possible objective value deterioration. The rounding direction for each variable is selected to be opposite to the objective function direction, i.e., we round up if $c_j \geq 0$ and down otherwise. Since set covering or partitioning models usually have $c \geq 0$ we will always fix the binary variables in these models to 1.

In order to decide which fractional variable we want to round, we look at the ratio

$$\frac{f_j^- c_j}{|A_{\cdot j}| + 1} \text{ (downwards)} \quad \text{or} \quad \frac{f_j^+ c_j}{|A_{\cdot j}| + 1} \text{ (upwards)}$$

of the direct objective increase and the number of constraints the variable is contained in. Since we always round up for set covering or partitioning models, the objective increase would be $f_j^+ c_j$, and the length of the sparse column vector $|A_{\cdot j}|$ indicates how many constraints would be covered by the fixing of the variable to 1. Therefore, we select a variable with smallest ratio in order to minimize the costs per covered constraint.

9.3 Objective Diving Heuristics

In contrast to the "hard rounding" of diving heuristics that is conducted by tightening a bound of a variable, objective diving heuristics apply "soft rounding" by increasing or decreasing the objective coefficient of the variable in order to push it into the desired direction without actually forcing it to the rounded value. Such a soft rounding does not entail the risk of obtaining an infeasible LP. The downside is that we can no longer apply domain propagation, and that the size of the LP is not implicitly reduced as it is in regular diving due to the fixings of variables to their lower or upper bounds. Furthermore, we have to deactivate the objective limit imposed by the current incumbent, because the objective function does no longer coincide with the objective of the MIP. Additionally, we have to find means to avoid cycling.

The general approach to objective diving is similar to Algorithm 9.1 for regular diving. In Step 5, the tightening of the local bounds is replaced by a corresponding change in the objective function, the domain propagation of Step 6 and the backtracking of Step 8 are skipped, and the infeasibility of the LP is no longer an abort criterion. Note that the LPs can be resolved efficiently with the primal simplex algorithm, since a change in the objective function does not destroy primal feasibility of a previously optimal simplex basis.

9.3.1 OBJECTIVE PSEUDOCOST DIVING

Objective pseudocost diving shares the ideas of *pseudocost diving* but primarily uses soft rounding via the objective function instead of hard rounding with bound changes. The variable and direction are selected according to the same criterion as in *pseudocost diving*. If the variable x_j should be rounded down, we change its objective coefficient to $c_j := 1000(d+1) \cdot |c_j|$ with d being the current diving depth, i.e., the number of LP resolves that have been performed during the dive. If the variable should be rounded up, we use the negative of this value.

For each variable, we remember whether it was already soft-rounded downwards or upwards. If a variable should be rounded a second time, we apply a hard rounding into the other direction by changing its lower or upper bound. The rationale behind this approach is the following: if a variable could not be driven, for example, to its lower bound by dramatically increasing its objective coefficient, we take this as an indication that fixing it to its lower bound would probably generate an infeasible LP. Therefore, we apply a hard rounding into the opposite direction and round the variable upwards.

After imposing a bound change, we resolve the LP with the dual simplex algorithm. In this case, it may happen that the modified LP turns out to be infeasible, which leads to the termination of the heuristic.

9.3.2 ROOT SOLUTION DIVING

The *root solution diving* is the objective diving analogon to the *line search diving* heuristic. As in *objective pseudocost diving* we augment the soft roundings with hard roundings in order to avoid cycling. However, the approach to combine the objective and bound changes is slightly different.

In each iteration, we scale the current objective function with a factor of 0.9, thereby slowly fading out the original objective coefficients. A soft rounding is applied by increasing or decreasing the objective coefficient of the rounding variable by $0.1 \cdot \max\{|\check{c}|, 1\}$, with $\check{c}$ being the objective value of the LP solution at the subproblem where the dive was started. We count the number of downwards and upwards soft roundings for each variable, and if the difference of these counters becomes larger or equal to 10, we permanently apply a hard rounding into the preferred direction by modifying the lower or upper bound of the variable. If at any time a variable that has already been soft-rounded becomes integral, we fix the variable to this value by changing both of its bounds.

9.3.3 FEASIBILITY PUMP

The *feasibility pump* is a sophisticated objective diving heuristic that was invented by Fischetti, Glover, and Lodi [84] for pure integer programs and generalized by Bertacco, Fischetti, and Lodi [40] to mixed integer programs. Achterberg and Berthold [2] proposed a slight modification of the heuristic which they call *objective feasibility pump* and that yields feasible solutions of better objective value.

Starting as usual from the optimal solution $\check{x}$ of the current subproblem's LP relaxation, the solution is rounded to a vector $\tilde{x} = [\check{x}]$, with $[\cdot]$ defined by

$$[x]_j := \begin{cases} \lfloor x_j + 0.5 \rfloor & \text{if } j \in I \\ x_j & \text{if } j \in C = N \setminus I. \end{cases} \tag{9.1}$$

If $\tilde{x}$ is not feasible, an additional LP is solved in order to find a new point in the LP polyhedron

$$P := \{x \in \mathbb{R}^n \mid Ax \leq b,\ l \leq x \leq u\}$$

which is, w.r.t. the integer variables I, closest to $\tilde{x}$, i.e., that minimizes

$$\Delta(x, \tilde{x}) := \sum_{j \in I} |x_j - \tilde{x}_j|.$$

The procedure is iterated by using this point as new solution $\check{x} \in P$. Thereby, the algorithm creates two sequences of points: one with points $\check{x}$ that fulfill the inequalities, and one with points $\tilde{x}$ that fulfill the integrality requirements. The algorithm terminates if the two sequences converge or if a predefined iteration limit is reached.

In order to determine a point

$$\check{x} := \operatorname{argmin}\{\Delta(x, \tilde{x}) \mid x \in P\} \tag{9.2}$$

in P, which is nearest to $\tilde{x}$, the following LP is solved:

$$\begin{aligned} \min \quad & \sum_{j \in I : \tilde{x}_j = l_j} (x_j - l_j) + \sum_{j \in I : \tilde{x}_j = u_j} (u_j - x_j) + \sum_{j \in I : l_j < \tilde{x}_j < u_j} d_j \\ \text{s.t.} \quad & Ax \leq b \\ & d \geq x - \tilde{x} \\ & d \geq \tilde{x} - x \\ & l \leq x \leq u. \end{aligned} \tag{9.3}$$

The auxiliary variables d_j are introduced to model the nonlinear function $d_j = |x_j - \tilde{x}_j|$ for integer variables x_j, $j \in I$, that are not equal to one of their bounds in the rounded solution $\tilde{x}$.

The implementation of the *feasibility pump* in SCIP is slightly different from the one proposed by Bertacco, Fischetti, and Lodi [40]. First, it involves the modifications proposed in [2] to better guide the search into the area of solutions with good objective value. Second, it does not add auxiliary variables d_j as in System (9.3) for general integer variables, because adding and deleting columns produces overhead for the LP solving. Instead, we set the objective coefficient to $+1$, -1, or 0, depending on whether we want to round the variable down or up, or leave it on its integral value. Of course, such an objective modification may lead to overshooting the rounded value, and an objective of 0 does not necessarily mean that the variable will stay at its current value. As a third modification, we skip the expensive enumeration phase which is performed in stage 3 of the original *feasibility pump* algorithm.

9.4 Improvement Heuristics

Starting from one or more feasible solutions, the goal of improvement heuristics is to find feasible solutions with a better objective value. Besides the *one opt* heuristic, all improvement heuristics implemented in SCIP are solving a sub-MIP.

The first to use sub-MIP solves inside a MIP heuristic have been Fischetti and Lodi [85] with their *local branching*, although they did not think of it as a heuristic but as a different way of branching which is influenced by the current incumbent. Danna, Rothberg, and Le Pape [72] adopted the idea of solving sub-MIPs for

heuristic purposes and invented the *relaxation induced neighborhood search* (RINS) method, which they integrated into CPLEX. Rothberg [194] continued to pursue this approach and developed *mutation* and *crossover*, which form the basis of CPLEX' very successful "solution polishing". *Crossover* was independently developed by Timo Berthold who implemented this heuristic in SCIP.

From the improvement heuristics, only *one opt* and *crossover* are active by default. Berthold [41] observed that *local branching* deteriorates, but any of RINS, *crossover*, and *mutation* improves the performance of SCIP. However, combinations of these heuristics produce worse results on his test set. In total, *crossover* turned out to be the most effective of the four individual sub-MIP improvement heuristics.

9.4.1 ONE OPT

The idea of *one opt* is very simple: given a feasible solution $\hat{x}$, the value of a variable x_j can be decreased for $c_j > 0$ or increased for $c_j < 0$ if the resulting solution $\tilde{x}$ is still feasible, i.e., $A\tilde{x} \leq b$ and $l \leq \tilde{x} \leq u$. The modified solution would then have an improved objective value.

The version of *one opt* as implemented in SCIP first calculates a value $\delta_j \in \mathbb{Z}_{\geq 0}$ for each variable x_j with $j \in I$ and $c_j \neq 0$. This value denotes how far the variable can be shifted into the desired direction. If more than one variable can be shifted, they are sorted by non-decreasing objective improvement capability $|c_j \delta_j|$ and consecutively shifted until no more improvements can be obtained. Finally, the integer variables are fixed to their resulting values, and an LP is solved to obtain best possible values for the continuous variables.

9.4.2 LOCAL BRANCHING

Local branching was proposed by Fischetti and Lodi [85]. It is based on the observation that often primal feasible solutions have additional solutions in their vicinity. Therefore, Fischetti and Lodi implemented a branching scheme on top of a MIP solver that would produce the case distinction $\sum_{j \in I} |x_j - \hat{x}_j| \leq k$ and $\sum_{j \in I} |x_j - \hat{x}_j| \geq k + 1$, $k \in \mathbb{Z}_{>0}$, with the former case being enumerated first.

By dropping the requirement that the branch $\sum_{j \in I} |x_j - \hat{x}_j| \leq k$ has to be enumerated exhaustively, the *local branching* scheme can be easily converted into a primal improvement heuristic. However, some control of the neighborhood size k is necessary. The SCIP implementation of *local branching* starts with $k = 18$. This value is increased by 50 % if a sub-MIP has been enumerated completely and does not contain a better solution than the current incumbent. The neighborhood is decreased by 50 % if the sub-MIP solve was aborted due to the node limit that was applied.

9.4.3 RINS

The *relaxation induced neighborhood search* (RINS) solves sub-MIPs that are defined by the current incumbent solution $\hat{x}$ and the LP optimum $\check{x}$ of the current subproblem. RINS was invented by Danna, Rothberg, and Le Pape [72].

The sub-MIP that is solved in RINS is defined as

$$\min\{c^T x \mid Ax \leq b,\ l \leq x \leq u,\ x_j \in \mathbb{Z} \text{ for all } j \in I,\\ x_j = \hat{x}_j \text{ for all } j \in I \text{ with } \hat{x}_j = \check{x}_j\},$$

which means that an integer variable is fixed if both the incumbent and the current LP solution agree on a common value. As in the *local branching* heuristic, the difficult part is to control the calling frequency and the sub-MIP node limit that is applied to avoid spending too much time on the heuristic. Similar to the diving heuristics, the general approach in SCIP is to avoid spending more than a certain fraction (10 % in the case of RINS) of the total number of branching nodes for solving the sub-MIPs.

9.4.4 Mutation

Mutation and *crossover* are the two building blocks of a genetic algorithm. Rothberg [194] applied this idea to mixed integer programming to develop an improvement heuristic. The *mutation* part as proposed by Rothberg is to start from a feasible solution and fix a certain fraction of the general integer variables to their values. The actual variables to fix are selected randomly. Then, a sub-MIP is solved to identify the optimal values for the remaining variables. Rothberg proposes to adjust the fixing rate dynamically. However, the current implementation in SCIP chooses a static fixing rate of 80 %.

9.4.5 Crossover

A *crossover* of two or more feasible solutions is performed by fixing all integer variables on which the solutions agree to their specific values and leaving the other variables free in their global bounds. Thus, the more solutions participate in the crossover, the fewer variables become fixed and the larger is the subspace that will be enumerated in the sub-MIP. Like *mutation*, *crossover* was developed by Rothberg [194]. Independently, it was invented by Berthold [41] and implemented in SCIP.

The *crossover* implementation of SCIP uses three solutions to define the sub-MIPs. By default, SCIP keeps the best 100 feasible solutions in a solution pool sorted by non-decreasing objective value. After a solution has been found that takes one of the first three slots in the pool, the crossover is performed on the best three solutions. In the subsequent calls, a random selection of three solutions is used which is biased towards the solutions of better objective value.

9.5 Computational Results

The following computational results assess the impact of the various primal heuristics presented in the previous sections. We discuss the benchmarks only briefly, since a very detailed computational study of the MIP heuristics of SCIP can be found in Berthold [41].

In the default parameter settings, all primal heuristics except *Octane*, *local branching*, RINS, and *mutation* are enabled. Table 9.1 yields the results for disabling all heuristics of a certain category: for the values in column "no round", we disabled

	test set	none	no round	no diving	no objdiving	no improvement
time	MIPLIB	+39	+10	+5	+17	0
	CORAL	−9	−4	−3	−6	−3
	MILP	+7	+5	+2	+3	+2
	ENLIGHT	−8	−2	−13	+14	+1
	ALU	−13	−1	−2	+3	0
	FCTP	+3	+1	+4	−2	+4
	ACC	+35	−3	−9	+62	−2
	FC	+82	+21	+3	−1	0
	ARCSET	+2	+1	−1	+1	+1
	MIK	+481	+23	+2	+4	+2
	total	**+14**	**+3**	**0**	**+4**	**0**
nodes	MIPLIB	+111	+26	−3	+30	−1
	CORAL	+18	−3	−1	−3	−6
	MILP	+42	+9	+5	+9	+3
	ENLIGHT	−13	0	−17	+19	0
	ALU	−12	0	−3	−4	0
	FCTP	+84	+35	0	−2	+4
	ACC	+151	0	−5	+158	0
	FC	+774	+139	−2	−2	−10
	ARCSET	+53	+10	+5	+4	+1
	MIK	+379	+24	+10	0	+1
	total	**+59**	**+11**	**0**	**+11**	**−1**

Table 9.1. Performance effect of different classes of primal heuristics for solving MIP instances. The values denote the percental changes in the shifted geometric mean of the runtime (top) and number of branching nodes (bottom) compared to the default settings in which all heuristics except *Octane* are enabled. Positive values represent a deterioration, negative values an improvement.

all rounding heuristics, for column "no diving" we disabled all diving heuristics, column "no objdiving" shows the results for disabling the objective diving heuristics, and for column "no improvement", we disabled the improvement heuristics. More detailed results can be found in Tables B.101 to B.110 in Appendix B.

As one can see, the performance impact of the heuristics is rather small. Even if all heuristics are turned off (column "none"), the average solving time only increases by 14 %. The numbers for disabling individual heuristic classes reveal a similar behavior as the one that we already observed for cutting plane separation, see Section 8.10: the sum of the degradations is significantly smaller than the performance degradation for turning off all heuristics. A possible explanation is that the same solution can be found by different heuristics, and thus, the absence of one class of heuristics can be compensated by the remaining heuristics.

Tables 9.2, 9.3, and 9.4 show the results for disabling individual heuristics. Additionally, column "octane" of Table 9.2 indicates the results for enabling *Octane*, and column "no backtrack" of Table 9.3 shows the impact of disabling the backtracking Step 8 in the diving Algorithm 9.1.

The totals show that the most important heuristics are the *feasibility pump*, RENS, and *line search diving* with performance impacts of 7 %, 6 %, and 4 %, respectively. The other heuristics have almost no impact on the overall solving time, although they can influence the performance on individual test sets. Paradoxically, large differences for the diving (Table 9.3) and objective diving (Table 9.4) heuristics can be observed on the ENLIGHT, ALU, and ACC test sets, although none of the heuristics finds a solution to any of these instances. Still, the heuristics implicitly modify the path of the search since their application can, for example, lead to a different alternative optimal LP solution at the current node, thereby influencing the branching decision. Additionally, diving heuristics gather pseudocost and inference statistics, see Section 5.3, which further affects the branching variable selection.

	test set	no RENS	no simple rnd	no rounding	no shifting	no int shifting	octane
time	MIPLIB	+2	−2	−2	−1	+2	+2
	CORAL	+6	0	−1	−2	−2	0
	MILP	+10	0	−2	−5	+1	0
	ENLIGHT	−2	−1	−2	−2	−1	−2
	ALU	+1	−1	0	−1	+1	+2
	FCTP	+4	+12	+2	0	+3	0
	ACC	0	−1	−1	−2	0	−2
	FC	+21	+3	0	−3	0	+1
	ARCSET	+3	+2	+1	+4	+2	0
	MIK	+9	+9	0	0	+2	0
	total	**+6**	**0**	**−1**	**−2**	**0**	**+1**
nodes	MIPLIB	+19	−1	−1	+1	+3	−1
	CORAL	+11	0	+2	−4	−6	0
	MILP	+17	+1	0	−9	+3	0
	ENLIGHT	0	0	0	0	0	0
	ALU	0	0	0	0	0	0
	FCTP	+34	+13	0	0	+1	0
	ACC	0	0	0	0	0	0
	FC	+149	+8	+3	−6	−11	0
	ARCSET	+18	−1	+2	0	−1	0
	MIK	+13	+1	0	0	−1	0
	total	**+17**	**0**	**+1**	**−4**	**−1**	**0**

Table 9.2. Performance effect of individual rounding heuristics for solving MIP instances. The values denote the percental changes in the shifted geometric mean of the runtime (top) and number of branching nodes (bottom) compared to the default settings in which all heuristics except *Octane* are enabled. Positive values represent a deterioration, negative values an improvement.

	test set	no coef	no frac	no guided	no linesearch	no pscost	no veclen	no backtrack
time	MIPLIB	0	−6	−2	−1	−4	−3	−4
	CORAL	−8	0	−6	+4	−3	−7	−3
	MILP	−2	+1	+10	+6	0	+2	+3
	ENLIGHT	+4	−4	+2	+13	+18	+9	+19
	ALU	+36	−1	0	+8	−2	+5	−5
	FCTP	−3	+4	−3	+2	+1	0	−2
	ACC	+19	+13	−2	+15	+42	+16	+12
	FC	−1	0	0	0	+1	+1	0
	ARCSET	+5	−3	+1	+1	+10	−5	+6
	MIK	+2	+1	0	+4	+2	0	+2
	total	**−2**	**−1**	**0**	**+4**	**0**	**−2**	**0**
nodes	MIPLIB	+1	−3	0	0	−3	−3	−6
	CORAL	−13	0	−4	+4	−5	−7	+2
	MILP	−7	+4	+16	+10	0	+2	+5
	ENLIGHT	+3	+6	+2	+13	+19	+12	+11
	ALU	+52	+3	0	+22	+6	+11	−2
	FCTP	−3	+1	−4	0	+1	−2	−3
	ACC	+72	+48	0	+48	+73	+50	+26
	FC	−2	−1	0	0	+1	+1	−1
	ARCSET	+6	−6	0	+2	+8	−8	+11
	MIK	0	0	0	0	−1	−2	+4
	total	**−4**	**+1**	**+3**	**+6**	**0**	**−1**	**+2**

Table 9.3. Performance effect of individual diving heuristics for solving MIP instances. The values denote the percental changes in the shifted geometric mean of the runtime (top) and number of branching nodes (bottom) compared to the default settings in which all heuristics except *Octane* are enabled. Positive values represent a deterioration, negative values an improvement.

	test set	no obj pscost diving	no rootsol diving	no feaspump
time	MIPLIB	−3	+1	+19
	CORAL	−4	−3	0
	MILP	+6	+2	−1
	ENLIGHT	+15	+4	+3
	ALU	+15	+13	+10
	FCTP	−1	0	+2
	ACC	+9	+15	+107
	FC	−1	0	0
	ARCSET	+6	+8	+8
	MIK	−1	+2	+1
	total	**+1**	**+1**	**+7**
nodes	MIPLIB	−1	+3	+37
	CORAL	−1	−3	−3
	MILP	+12	+7	−2
	ENLIGHT	+10	+1	+5
	ALU	+18	+19	+17
	FCTP	−2	+1	0
	ACC	+24	+28	+239
	FC	0	0	0
	ARCSET	+6	+7	+9
	MIK	0	−1	0
	total	**+4**	**+3**	**+10**

Table 9.4. Performance effect of individual objective diving heuristics for solving MIP instances. The values denote the percental changes in the shifted geometric mean of the runtime (top) and number of branching nodes (bottom) compared to the default settings in which all heuristics except *Octane* are enabled. Positive values represent a deterioration, negative values an improvement.

Table 9.5 compares the improvement heuristics. The columns "no oneopt" and "no crossover" correspond to disabling the respective heuristics. The other three columns show the results for replacing *crossover* by *local branching*, RINS, or *mutation*, respectively. This means, we disabled *crossover* in these settings and enabled the respective non-default improvement heuristic. It turns out that none of the heuristics has a significant impact on the average performance on our test sets, with the exception that applying *local branching* slows down the solving of the MILP, FCTP, and FC instances, while RINS seems to help a little on the CORAL test set.

FINDING SOLUTIONS EARLY

Although we have seen that primal heuristics do not help much to reduce the time to solve MIP instances to optimality, their application may have a significant benefit: heuristics can help to find good feasible solutions early in the search process.

In order to study the contribution of primal heuristics in this regard, we conducted another computational experiment. Instead of measuring the time until an optimal solution has been found and its optimality has been proven, we terminate a run if a certain solution quality has been achieved. We quantify the quality of a solution by the primal-dual gap

$$\gamma = \begin{cases} 0 & \text{if } |\hat{c} - \check{\underline{c}}| \leq \epsilon, \\ (\hat{c} - \check{\underline{c}}) \,/\, |\check{\underline{c}}| & \text{if } \hat{c} \cdot \check{\underline{c}} > 0, \\ \infty & \text{otherwise,} \end{cases}$$

with $\hat{c}$ being the objective value of the current incumbent and $\check{\underline{c}}$ being the current global dual bound. All of our previous benchmark runs have been executed until $\gamma = 0$. In contrast, Table 9.6 shows the average time and number of nodes needed

	test set	no oneopt	no crossover	local branching	RINS	mutation
time	MIPLIB	−1	−4	0	−1	−1
	CORAL	+1	−2	+1	−5	−3
	MILP	0	+1	+5	+2	+1
	ENLIGHT	−3	−1	+1	+1	−2
	ALU	−1	0	−1	0	0
	FCTP	+3	+2	+6	−1	0
	ACC	−2	−1	−1	−2	−1
	FC	0	+1	+12	+2	+2
	ARCSET	+3	−4	−1	+4	+1
	MIK	+3	−4	+2	−1	+3
	total	**0**	**−1**	**+2**	**−1**	**−1**
nodes	MIPLIB	−4	−2	−4	−1	−2
	CORAL	−1	−4	−4	−9	−4
	MILP	+2	+2	+1	0	+1
	ENLIGHT	0	0	0	0	0
	ALU	0	0	0	0	0
	FCTP	0	+3	+3	−5	+3
	ACC	0	0	0	0	0
	FC	−10	+1	+1	+1	+1
	ARCSET	−1	−1	−1	−3	−1
	MIK	0	−1	−1	−1	−1
	total	**−1**	**−1**	**−2**	**−3**	**−1**

Table 9.5. Performance effect of individual improvement heuristics for solving MIP instances. The values denote the percental changes in the shifted geometric mean of the runtime (top) and number of branching nodes (bottom) compared to the default settings in which all heuristics except *Octane* are enabled. Positive values represent a deterioration, negative values an improvement.

to reach $\gamma = 0.05$ and $\gamma = 0.20$, respectively. We compare both enabling and disabling the default primal heuristics with solving the instances to optimality using the default settings.

The first observation is that a user who only wants to get a solution with a value that is guaranteed to be at most 5 % or 20 %, respectively, worse than the optimal solution value receives his answer much faster: to reach 5 % gap, the average runtime with enabled heuristics ("all (5 %)") reduces by 57 %, and to reach 20 % gap ("all (20 %)"), the reduction is even 74 %. This means, the average time to find a solution and to prove that its value is within 20 % of the optimal value is only one fourth of the time needed to solve the instances to optimality.

In this experiment, the primal heuristics show their potential. If they are disabled (columns "none"), the runtime reduction drops from 57 % to 41 % for 5 % gap, and from 74 % to 56 % gap for reaching 20 % gap. Thus, the average time to reach the desired solution quality increases by 37 % and 69 %, respectively. The effect gets even more prominent by looking at the average number of nodes: many of the values in the "all (20 %)" column are close or equal to −100, which indicates that for many or all of the instances of these test sets, the required gap is already achieved at the root node. In contrast, this is not the case if the heuristics are disabled.

	test set	all (5 %)	none (5 %)	all (20 %)	none (20 %)
time	MIPLIB	−62	−34	−72	−48
	CORAL	−59	−57	−73	−65
	MILP	−48	−36	−75	−62
	ENLIGHT	0	−10	−2	−11
	ALU	+1	−14	−1	−14
	FCTP	−65	−56	−95	−70
	ACC	+1	+34	−3	+37
	FC	−29	+51	−52	+43
	ARCSET	−77	−57	−89	−62
	MIK	−96	−41	−98	−67
	total	**−57**	**−41**	**−74**	**−56**
nodes	MIPLIB	−89	−39	−95	−64
	CORAL	−83	−64	−93	−78
	MILP	−51	−27	−88	−75
	ENLIGHT	0	−13	−3	−18
	ALU	0	−12	0	−12
	FCTP	−82	−25	−100	−54
	ACC	0	+151	0	+151
	FC	−69	+571	−100	+531
	ARCSET	−97	−73	−100	−82
	MIK	−100	−57	−100	−80
	total	**−80**	**−41**	**−94**	**−68**

Table 9.6. Impact of primal heuristics for reaching a gap of 5 % or 20 %, respectively. The values denote the percental changes in the shifted geometric mean of the runtime (top) and number of branching nodes (bottom) compared to solving the instances to optimality with the default settings. Positive values represent a deterioration, negative values an improvement.

Chapter 10

Presolving

Presolving is a way to transform the given problem instance into an equivalent instance that is (hopefully) easier to solve. Since many MIP instances appearing in practice[1] contain lot of irrelevant data that only slow down the solving process, all existing competitive MIP solvers feature some form of presolving. The most fundamental presolving concepts for mixed integer programming are described in Savelsbergh [199]. Additional information can be found in Fügenschuh and Martin [90].

The task of presolving is threefold: first, it reduces the size of the model by removing irrelevant information such as redundant constraints or fixed variables. Second, it strengthens the LP relaxation of the model by exploiting integrality information, e.g., to tighten the bounds of the variables or to improve coefficients in the constraints. Third, it extracts information such as implications or cliques from the model which can later be used, for example for branching or cutting plane separation.

Applying the domain propagation of Chapter 7 to the global problem instance R already yields a simple form of presolving, namely the tightening of the variables' global bounds $l \leq x \leq u$. In addition to this, presolving employs more sophisticated rules, which may alter the structure of the problem.

We distinguish between *primal* and *dual* presolving reductions. Primal reductions are solely based on feasibility reasoning, while dual reductions consider the objective function. The latter may exclude feasible solutions from the problem instance, as long as at least one optimal solution remains.

Sections 10.1 to 10.4 describe the presolving algorithms that are specialized to a certain type of constraints. These are implemented in the constraint handlers. Afterwards in Sections 10.5 to 10.8, we deal with general purpose presolving algorithms that can be applied to any constraint integer program independently from the type of the involved constraints. Section 10.9 presents *restarts*, a technique applied in SAT solvers that is new for the MIP community. The computational experiments of Section 10.10 evaluate the impact of presolving on the overall solving process.

10.1 Linear Constraints

Recall that linear constraints in SCIP are defined as

$$\underline{\beta} \leq a^T x \leq \overline{\beta}$$

with the left and right hand sides $\underline{\beta}, \overline{\beta} \in \mathbb{R} \cup \{\pm\infty\}$ and coefficients $a \in \mathbb{R}^n$.

The presolving of linear constraints is depicted in Algorithm 10.1. It commences in Step 1 by looking at the individual constraints one at a time. For each constraint,

[1] in particular those that have been automatically generated, for example using a modeling language such as Ampl [89] or Zimpl [133, 134]

Algorithm 10.1 Presolving for Linear Constraints

1. For all linear constraints $\underline{\beta} \leq a^T x \leq \overline{\beta}$:
   (a) Normalize the constraint using Algorithm 10.2.
   (b) If $a_j \in \mathbb{Z}$ for all $j \in N$ and $a_j = 0$ for all $j \in C$, set $\underline{\beta} := \lceil \underline{\beta} \rceil$ and $\overline{\beta} := \lfloor \overline{\beta} \rfloor$.
   (c) Tighten the bounds of the variables by calling domain propagation Algorithm 7.1.
   (d) If $\underline{\alpha} > \overline{\beta}$ or $\overline{\alpha} < \underline{\beta}$, the problem instance is infeasible.
   If $\underline{\alpha} \geq \underline{\beta}$, set $\underline{\beta} := -\infty$.
   If $\overline{\alpha} \leq \overline{\beta}$, set $\overline{\beta} := +\infty$.
   If $\underline{\beta} = -\infty$ and $\overline{\beta} = +\infty$, delete the constraint.
   (e) If the constraint is a set partitioning constraint $\sum_{j \in S} x_j = 1$ or a set packing constraint $\sum_{j \in S} x_j \leq 1$, $x_j \in \{0, 1\}$ for all $j \in S$, add clique S to the clique table.
   (f) For all $j \in I$ with $a_j > 0$, $\underline{\alpha} + a_j \geq \underline{\beta}$, and $\overline{\alpha} - a_j \leq \overline{\beta}$:
      i. Set $a'_j := \max\{\underline{\beta} - \underline{\alpha}, \overline{\alpha} - \overline{\beta}\}$.
      ii. Set $\underline{\beta} := \underline{\beta} - (a_j - a'_j) l_j$ and $\overline{\beta} := \overline{\beta} - (a_j - a'_j) u_j$.
      iii. Set $a_j := a'_j$.
   For all $j \in I$ with $a_j < 0$, $\underline{\alpha} - a_j \geq \underline{\beta}$, and $\overline{\alpha} + a_j \leq \overline{\beta}$:
      i. Set $a'_j := \min\{\underline{\alpha} - \underline{\beta}, \overline{\beta} - \overline{\alpha}\}$.
      ii. Set $\underline{\beta} := \underline{\beta} - (a_j - a'_j) u_j$ and $\overline{\beta} := \overline{\beta} - (a_j - a'_j) l_j$.
      iii. Set $a_j := a'_j$.
   (g) If the constraint or the bounds of the variables have been modified in Steps 1a to 1f, and if this loop has not already been executed 10 times, goto Step 1a.
   (h) If the constraint is an equation, i.e., $\underline{\beta} = \overline{\beta}$, call Algorithm 10.3.
   (i) Call the dual aggregation Algorithm 10.4.
2. If no reductions have been found yet in the current presolving round, call the constraint pair presolving Algorithm 10.5.
3. If no reductions have been found yet in the current presolving round, call the dual bound reduction Algorithm 10.6.
4. If no reductions have been found yet in the current presolving round, call Algorithm 10.7 for each linear constraint to upgrade it into a constraint of a more specific constraint type.

the first operation is the constraint normalization of Step 1a as illustrated in Algorithm 10.2. In the normalization, we remove fixed variables by subtracting their contribution to the activity $a^T x$ of the constraint from the left and right hand sides. Aggregated and multi-aggregated variables $x_k \stackrel{\star}{=} \sum_{j \in N} s_j x_j + d$ are substituted for their defining affine linear expression, again subtracting the constant $a_k d$ from the left and right hand sides. Afterwards, we multiply the constraint by $+1$ or -1 in order to reach a standard form, which simplifies the upgrading of the constraint into a more specialized constraint type, see below.

In the next step of the normalization, we try to scale the constraint to obtain integral coefficients. Since we are dealing with floating point arithmetic, this scaling

Algorithm 10.2 Normalization of Linear Constraints

Input: Linear constraint $\underline{\beta} \leq a^T x \leq \overline{\beta}$ and global bounds $l \leq x \leq u$.

Output: Normalized constraint.

1. Remove fixed variables: if $l_j = u_j$, set $\underline{\beta} := \underline{\beta} - a_j l_j$, $\overline{\beta} := \overline{\beta} - a_j l_j$, and $a_j := 0$.
2. Replace aggregated and multi-aggregated variables by their representing affine linear sum of active problem variables.
3. Multiply the constraint with $+1$ or -1 using the following rules in the given order until the sign is uniquely determined (i.e., if neither or both signs would satisfy the rule, proceed with the next rule):
   (a) non-negative right hand side: $\overline{\beta} \geq 0$,
   (b) finite right hand side: $\overline{\beta} < \infty$,
   (c) larger absolute value of right hand side: $|\overline{\beta}| > |\underline{\beta}|$,
   (d) more positive coefficients: $|\{j \mid a_j > 0\}| > |\{j \mid a_j < 0\}|$,
   (e) use the sign $+1$.
4. Identify a rational representation of the coefficients. If the smallest common multiple of the denominators is not too large, scale the constraint to obtain integral coefficients.
5. If all coefficients are integral, divide them by their greatest common divisor.

involves numerical issues. First, the identification of the rational representation of each coefficient is performed using the Euclidean algorithm, but in order to avoid too large factors we restrict it to only succeed if a rational number $\frac{p_j}{q_j}$, $p_j \in \mathbb{Z}$, $q_j \in \mathbb{Z}_{>0}$, can been found with $|\frac{p_j}{q_j} - a_j| \leq \epsilon := 10^{-9}$, $|p_j| \leq p_{\max} = 10^6$, and $q_j \leq q_{\max} := \frac{\hat{\delta}}{\epsilon} = 1000$. Second, the constraint is multiplied with the smallest common multiple of the denominators q_j, but this is only performed if $s := \operatorname{scm}(q_1, \ldots, q_n) \leq q_{\max}$ and $\max\{|s \cdot a_j|\} \leq p_{\max}$. Finally, if the coefficients have already been integral or if the multiplication with the smallest common multiple of the denominators was successful, we divide the integral coefficients by their greatest common divisor.

After the normalization has been performed, we proceed with Step 1b of the presolving Algorithm 10.1, which is to tighten the left and right hand sides: if all coefficients a_j are integral and all variables x_j with $a_j \neq 0$ are of integer type, we can round up a fractional left hand side and round down a fractional right hand side. This may already lead to the detection of infeasibility if $\underline{\beta} > \overline{\beta}$ after rounding.

Step 1c applies the domain propagation algorithm of Section 7.1 in order to tighten the global bounds of the variables. Afterwards, we can sometimes detect infeasibility or redundancy in Step 1d by inspecting the final constraint activity bounds $\underline{\alpha} = \min\{a^T x \mid l \leq x \leq u\}$ and $\overline{\alpha} = \max\{a^T x \mid l \leq x \leq u\}$: if the constraint cannot be satisfied within the activity bounds, the whole instance is infeasible. If one of the sides can never be violated, it can be removed by setting it to infinity. If both sides have been removed, the constraint is redundant and can be deleted from the problem instance.

Step 1e checks whether the constraint has the special form of a set partitioning or set packing constraint. Note that this form can sometimes be achieved by complementing binary variables. If the constraint is a set partitioning or set packing constraint, we add the corresponding clique $\sum_{j \in S} x_j \leq 1$ to the clique table, see

Section 3.3.5. It may be possible to extract cliques from other constraints as well, but this has not yet been implemented in SCIP.

The coefficient tightening of Step 1f aims to modify the constraint such that the set of feasible integral solutions to the constraint stays unchanged, but the set of fractional solutions is reduced. The reduction considers integer variables for which the sides of the constraint become redundant if the variable is not on its lower or upper bound, respectively. If this is the case, we can reduce the coefficient a_j of the variable and the sides $\underline{\beta}$ and $\overline{\beta}$ in order to obtain the same redundancy effect if the variable is not on its bounds and the same restriction on the other variables if the variable is set to one of its bounds.

Steps 1a to 1f are repeated as long as they modify the constraint or 10 rounds of the loop have been executed. One could refrain from executing the above steps in a loop and rely on the fact that all presolving methods are called cyclical by the outer presolving loop anyway, see Section 3.2.5. However, we perform this internal loop for performance reasons: since the data for the constraint and its variables are currently stored in the first or second level cache of the CPU, it makes sense to continue working on them immediately. The limit of 10 iterations is imposed to ensure that we do not get stuck in a series of very small changes, while a different presolving method could find a large reduction in one step.

10.1.1 PRESOLVING OF EQUATIONS

For linear equations, we can apply further presolving techniques, which are given in Algorithm 10.3. We consider two cases: equations with two coefficients and equations with more than two coefficients. In the case of two coefficients, Step 1 tries to represent one variable x_k as an affine linear term of the other variable x_j, i.e., as

$$x_k :\overset{\star}{=} -\frac{a_j}{a_k} x_j + \frac{\overline{\beta}}{a_k}. \tag{10.1}$$

If x_k is a continuous variable this is performed in Step 1a. Note that aggregating a variable x_k with $x_k :\overset{\star}{=} sx_j + d$ means to delete the variable x_k from the set of active problem variables and to update the bounds of x_j:

- ▷ If $s > 0$, update $l_j := \max\{l_j, \frac{l_k - d}{s}\}$ and $u_j := \min\{u_j, \frac{u_k - d}{s}\}$.
- ▷ If $s < 0$, update $l_j := \max\{l_j, \frac{u_k - d}{s}\}$ and $u_j := \min\{u_j, \frac{l_k - d}{s}\}$.
- ▷ If $j \in I$, set $l_j := \lceil l_j \rceil$ and $u_j := \lfloor u_j \rfloor$.

The tightened bounds of x_j can then be used to further tighten the bounds of x_k. If x_k is of integer type, the process can be iterated as long as the rounding provides further strengthening.

We can also perform Aggregation (10.1) if both variables are integers and $\frac{a_j}{a_k} \in \mathbb{Z}$, see Step 1b; in this case, the integrality of x_k is implied by the integrality of x_j. The bound strengthening loop of the aggregation procedure that is provided as infrastructure method of SCIP will automatically detect infeasibility if $\frac{\overline{\beta}}{a_k} \notin \mathbb{Z}$.

Steps 1c to 1g deal with the case that both variables are integers but $\frac{a_j}{a_k} \notin \mathbb{Z}$. This means the simple aggregation of Steps 1a and 1b does not work since the integrality of x_k would not be a consequence of the integrality of x_j and still has to be enforced. Therefore, x_k cannot be removed from the set of active problem variables which is the purpose of aggregating variables. Instead we try to find values

Algorithm 10.3 Presolving of Linear Equations

Input: Linear equation $a^T x = \overline{\beta}$ and global bounds $l \leq x \leq u$.

1. If there are exactly two non-zero coefficients $a_j, a_k \neq 0$, $j, k \in N$, $j \neq k$, try to aggregate one of the variables in the equation $a_j x_j + a_k x_k = \overline{\beta}$ (roles of j and k can be reversed):
   (a) If $k \in C$, aggregate $x_k :\overset{\star}{=} -\frac{a_j}{a_k} x_j + \frac{\overline{\beta}}{a_k}$ and stop. Otherwise, $j, k \in I$.
   (b) If $\frac{a_j}{a_k} \in \mathbb{Z}$, aggregate $x_k :\overset{\star}{=} -\frac{a_j}{a_k} x_j + \frac{\overline{\beta}}{a_k}$ and stop.
   (c) If $a_j \notin \mathbb{Z}$ or $a_k \notin \mathbb{Z}$, stop.
   (d) If $\overline{\beta} \notin \mathbb{Z}$, the constraint is infeasible. Stop.
   (e) Find a solution $(x_j^0, x_k^0) \in \mathbb{Z}^2$ to the equation $a_j x_j + a_k x_k = \overline{\beta}$, which does not necessarily have to respect the bounds of x_j and x_k.
   (f) Generate a new integer variable $y \in \mathbb{Z}$, initially with infinite bounds.
   (g) Aggregate $x_j :\overset{\star}{=} -a_k y + x_j^0$ and $x_k :\overset{\star}{=} a_j y + x_k^0$.
2. If the constraint has more than two non-zero coefficients, if there exists a variable x_k with $a_k \neq 0$ that does not appear in any other constraint, and if
   ▷ $k \in C$, or
   ▷ $\frac{a_j}{a_k} \in \mathbb{Z}$ for all $j \in N$, and $a_j = 0$ for all $j \in C$,
   then multi-aggregate $x_k :\overset{\star}{=} \frac{\overline{\beta}}{a_k} - \sum_{j \in N \setminus \{k\}} \frac{a_j}{a_k} x_j$ and replace the linear constraint by
   ▷ $\underline{\beta} - a_k u_k \leq \sum_{j \in N \setminus \{k\}} a_j x_j \leq \overline{\beta} - a_k l_k$, if $a_k > 0$, or
   ▷ $\underline{\beta} - a_k l_k \leq \sum_{j \in N \setminus \{k\}} a_j x_j \leq \overline{\beta} - a_k u_k$, if $a_k < 0$.
   If $\frac{\overline{\beta}}{a_k} \notin \mathbb{Z}$, the constraint is infeasible.

$a'_j, a'_k, x_j^0, x_k^0 \in \mathbb{Z}$ and create a new integer variable $y \in \mathbb{Z}$, such that $x_j :\overset{\star}{=} -a'_k y + x_j^0$ and $x_k :\overset{\star}{=} a'_j y + x_k^0$ are valid aggregations, which also yields a net reduction of one variable.

Note that usually the constraint normalization Algorithm 10.2 has already scaled the constraint to obtain integer coefficients $a_j, a_k \in \mathbb{Z}$. If this is not the case (because normalization failed due to numerical reasons), we have to stop. If both coefficients are integral but the right hand side is fractional, the constraint proves the infeasibility of the problem instance and we can abort in Step 1d.

Otherwise, all involved values are integers and we are searching in Step 1e for an integral solution (x_j^0, x_k^0) to the integral equation $a_j x_j + a_k x_k = \overline{\beta}$, i.e., we have to solve a *linear Diophantine equation*. Note that x_j^0 and x_k^0 do not need to satisfy the bounds of x_j and x_k. Since a_j and a_k are relatively prime due to Step 5 of the normalization Algorithm 10.2, such a solution always exists and can easily be found using the extended Euclidean algorithm.

Finally, we generate the new integer variable in Step 1f and perform the aggregation in Step 1g. Because a_j and a_k are relatively prime, the set of all integral solutions to the homogeneous equation $a_j x_j + a_k x_k = 0$ is given by

$$(x_j, x_k) \in \big\{ y \cdot (-a_k, a_j) \;\big|\; y \in \mathbb{Z} \big\}.$$

Hence, the aggregated variables $x_j :\overset{\star}{=} -a_k y + x_j^0$ and $x_k :\overset{\star}{=} a_j y + x_k^0$ cover all integral solutions of the inhomogeneous equation $a_j x_j + a_k x_k = \overline{\beta}$ for $y \in \mathbb{Z}$. Note that the bounds of x_j and x_k are automatically transformed into bounds of the new variable y during the aggregation.

The following example illustrates the reduction of Steps 1c to 1g:

Example 10.1 (aggregation of equations with two integer variables). Consider the equation $3x_1 + 8x_2 = 37$ with integer variables $x_1, x_2 \in \{0, \ldots, 5\}$. Neither Step 1a nor 1b of Algorithm 10.3 can be applied. We find the initial solution $(x_1^0, x_2^0) = (7, 2)$ and aggregate $x_1 :\overset{\star}{=} -8y + 7$ and $x_2 :\overset{\star}{=} 3y + 2$. During the first aggregation we calculate bounds $[\frac{7}{8} - \frac{1}{8} \cdot 5, \frac{7}{8} - \frac{1}{8} \cdot 0] = [\frac{1}{4}, \frac{7}{8}]$ for the integer variable y which can be rounded to the infeasible bounds $1 \leq y \leq 0$. Thus, we have detected the infeasibility of the constraint within the domains $x_1, x_2 \in \{0, \ldots, 5\}$.

Step 2 handles the case that the linear equation has more than two non-zero coefficients. Of course, we could also represent one of the variables as an affine linear combination of the others and remove it from the problem instance—at least if the variable is continuous or the affine linear combination is always integral. However, aggregating a variable means that we have to substitute it with its defining affine linear combination in all other constraints. This would add non-zero coefficients to the other constraints and make the coefficient matrix of the LP relaxation more dense, which is usually not beneficial in terms of LP solving performance and numerical stability.

Therefore, we perform such a *multi-aggregation* (an aggregation with more than one variable in the defining affine linear combination) only if the aggregated variable does not appear in other constraints. The typical case for such a substitution are slack variables that have been explicitly added to the model, as can been seen in the following example.

Example 10.2 (slack elimination). Consider the equation $4x_1 + 7x_2 + 3x_3 + s = 20$ with $s \geq 0$, and assume that s does not appear in other constraints. Then, Step 2 of Algorithm 10.3 would multi-aggregate $s :\overset{\star}{=} 20 - 4x_1 - 7x_2 - 3x_3$ and replace the equation with the inequality $4x_1 + 7x_2 + 3x_3 \leq 20$.

10.1.2 Dual Aggregation

Step 1i of the presolving Algorithm 10.1 for the current linear constraint performs a dual reduction that is shown in Algorithm 10.4. The basic idea of this dual reduction is the following: if a variable x_k has an objective coefficient $c_k \geq 0$, and if one side of the linear constraint is the only constraint which may block the setting of the variable to its lower bound, this side of the constraint will always be satisfied with equality. Therefore, we can multi-aggregate the variable if its bounds will always be satisfied by the aggregation and if it is either a continuous variable or the integrality condition will also always be satisfied. Analogously, the same can be applied for variables with $c_k \leq 0$ and a single constraint that blocks the setting of the variable to its upper bound.

Step 1 of Algorithm 10.4 treats the case in which we want to satisfy the left hand side with equality. The preconditions for this reduction are that

Algorithm 10.4 Dual Aggregation for Linear Constraints

Input: Linear constraint $\underline{\beta} \leq a^T x \leq \overline{\beta}$.

1. If $\underline{\beta} > -\infty$ and there is a variable x_k with
   (a) $\underline{\beta} - \overline{\alpha}_k \geq \min\{a_k l_k, a_k u_k\}$,
   (b) $\underline{\beta} - \underline{\alpha}_k \leq \max\{a_k l_k, a_k u_k\}$,
   (c) with
      ▷ $k \in C$ or
      ▷ $\frac{a_j}{a_k} \in \mathbb{Z}$ for all $j \in N$ and $a_j = 0$ for all $j \in C$,
   (d) and with
      ▷ $a_k > 0$, $c_k \geq 0$, and $\zeta_k^- = 1$, or
      ▷ $a_k < 0$, $c_k \leq 0$, and $\zeta_k^+ = 1$,

   aggregate $x_k :\overset{\star}{=} \frac{\underline{\beta}}{a_k} - \sum_{j \neq k} \frac{a_j}{a_k} x_j$ and delete the constraint. If $k \in I$ and $\frac{\underline{\beta}}{a_k} \notin \mathbb{Z}$, the constraint is infeasible.
2. If $\overline{\beta} < +\infty$ and there is a variable x_k with
   (a) $\overline{\beta} - \overline{\alpha}_k \geq \min\{a_k l_k, a_k u_k\}$,
   (b) $\overline{\beta} - \underline{\alpha}_k \leq \max\{a_k l_k, a_k u_k\}$,
   (c) with
      ▷ $k \in C$ or
      ▷ $\frac{a_j}{a_k} \in \mathbb{Z}$ for all $j \in N$ and $a_j = 0$ for all $j \in C$,
   (d) and with
      ▷ $a_k > 0$, $c_k \leq 0$, and $\zeta_k^+ = 1$, or
      ▷ $a_k < 0$, $c_k \geq 0$, and $\zeta_k^- = 1$,

   aggregate $x_k :\overset{\star}{=} \frac{\overline{\beta}}{a_k} - \sum_{j \neq k} \frac{a_j}{a_k} x_j$ and delete the constraint. If $k \in I$ and $\frac{\overline{\beta}}{a_k} \notin \mathbb{Z}$, the constraint is infeasible.

1. the resulting aggregation

$$x_k :\overset{\star}{=} \frac{\underline{\beta}}{a_k} - \sum_{j \neq k} \frac{a_j}{a_k} x_j$$

   satisfies the bounds of x_k for all values of x_j, $j \neq k$ (Conditions 1a and 1b),
2. either the variable is continuous or the aggregation is always integral (Condition 1c), and
3. the constraint at hand is the only constraint that blocks the setting of the variable to its best bound w.r.t. the objective function (Condition 1d).

Recall that the variable lock numbers ζ_k^- and ζ_k^+ queried in Condition 1d denote the number of constraints that block the shifting of the variable in the respective direction, compare Definition 3.3 on page 38. For $\underline{\beta} > -\infty$, the conditions $a_k > 0$ and $a_k < 0$ state that this constraint locks the variable in the corresponding direction and is therefore the reason for $\zeta_k^- = 1$ or $\zeta_k^+ = 1$, respectively.

Step 2 treats the case where we want to satisfy the right hand side of the constraint with equality. It applies the same reasoning as Step 1.

10.1.3 PRESOLVING OF CONSTRAINT PAIRS

After the individual constraint presolving in Step 1 of Algorithm 10.1 has been applied, the presolving continues with processing pairs of linear constraints as depicted in Algorithm 10.5. Since this pairwise comparison of constraints is quite expensive with its worst case runtime of $\mathcal{O}(nm^2)$ for dense coefficient matrices, we only apply it if no presolving method in the current presolving round found a reduction. The first step of the pairwise presolving algorithm is to calculate positive and negative signatures for all constraints. These are defined as follows:

Definition 10.3 (constraint signatures). Let $\mathcal{C}_i : \underline{\beta} \leq a^T x \leq \overline{\beta}$ be a linear constraint. Then, the bit vectors $\text{sig}_i^+, \text{sig}_i^- \in \{0,1\}^{64}$ with

$$(\text{sig}_i^+)_k = \begin{cases} 1 & \text{if } \exists j \in N : \max\{a_j l_j, a_j u_j\} > 0 \wedge (j \bmod 64) = k, \\ 0 & \text{otherwise} \end{cases}$$

$$(\text{sig}_i^-)_k = \begin{cases} 1 & \text{if } \exists j \in N : \min\{a_j l_j, a_j u_j\} < 0 \wedge (j \bmod 64) = k, \\ 0 & \text{otherwise} \end{cases}$$

for $k = 0, \ldots, 63$ are called *positive* and *negative signatures* of constraint $\mathcal{C}_i$.

The signatures are used in the remaining algorithm to quickly rule out uninteresting pairs of constraints where no presolving reduction can be applied. Note that they can be stored in 64 bit registers, and fast bit instructions can be used to compare the signature vectors of two constraints. Our experience is (although not supported with detailed computational results) that—with deactivated constraint aggregation, see below—this signature filter suffices to discard almost all constraint pairs on which no presolving can be applied, which significantly reduces the running time of the algorithm.

After calculating the signatures, we proceed with the pairwise comparison loop in Step 2. We perform a bookkeeping on the constraint modifications which allows us to skip pairs for which none of the two constraints has been altered since the last call to the comparison algorithm.

Step 2a compares the signatures and the sides of the constraints to check whether reductions are possible. This is the case if the constraint coefficients are equal (potentially after multiplying one of the constraints with -1), if one of the constraints dominates one of the sides, or if at least one of the constraints is an equation, which may yield the possibility to aggregate the constraints. If the positive and negative signatures of the constraints differ, the coefficients cannot be equal. If the positive signature of one constraint differs from the negative signature of the other constraint, the coefficients cannot be negated versions of each other. The domination of the left and right hand sides is defined as follows:

Definition 10.4 (domination of constraint sides). Let $\mathcal{C}_p : \underline{\beta}_p \leq (a^p)^T x \leq \overline{\beta}_p$ and $\mathcal{C}_q : \underline{\beta}_q \leq (a^q)^T x \leq \overline{\beta}_q$ be a pair of linear constraints defined on variables $x \in [l, u]$, $l, u \in \mathbb{R} \cup \{\pm\infty\}$. Then we say that the left hand side of $\mathcal{C}_p$ *dominates* the the left hand side of $\mathcal{C}_q$, if

$$\underline{\beta}_p \geq \underline{\beta}_q \quad \text{and} \quad \forall j \in N \, \forall x_j \in [l_j, u_j] : a_j^p x_j \leq a_j^q x_j.$$

The right hand side of $\mathcal{C}_p$ *dominates* the right hand side of $\mathcal{C}_q$, if

$$\overline{\beta}_p \leq \overline{\beta}_q \quad \text{and} \quad \forall j \in N \, \forall x_j \in [l_j, u_j] : a_j^p x_j \geq a_j^q x_j.$$

Algorithm 10.5 Pairwise Presolving of Linear Constraints

1. For all linear constraints $\mathcal{C}_i$ calculate the signature vectors $\mathrm{sig}_i^+, \mathrm{sig}_i^- \in \{0,1\}^{64}$.
2. For all pairs $(\mathcal{C}_p, \mathcal{C}_q)$, $p < q$, of linear constraints where at least one of the two has been modified since the last call to Algorithm 10.5:
   (a) Compare the signatures to check whether reductions are possible:

$$\begin{aligned}
\mathsf{coefseq} &:= (\mathrm{sig}_p^+ = \mathrm{sig}_q^+) \wedge (\mathrm{sig}_p^- = \mathrm{sig}_q^-) \\
\mathsf{coefsneg} &:= (\mathrm{sig}_p^+ = \mathrm{sig}_q^-) \wedge (\mathrm{sig}_p^- = \mathrm{sig}_q^+) \\
\mathsf{lhsdom}_p &:= (\underline{\beta}_p \geq \underline{\beta}_q) \quad \wedge (\mathrm{sig}_p^+ \leq \mathrm{sig}_q^+) \wedge (\mathrm{sig}_p^- \geq \mathrm{sig}_q^+) \\
\mathsf{lhsdom}_q &:= (\underline{\beta}_p \leq \underline{\beta}_q) \quad \wedge (\mathrm{sig}_p^+ \geq \mathrm{sig}_q^+) \wedge (\mathrm{sig}_p^- \leq \mathrm{sig}_q^+) \\
\mathsf{rhsdom}_p &:= (\overline{\beta}_p \leq \overline{\beta}_q) \quad \wedge (\mathrm{sig}_p^+ \geq \mathrm{sig}_q^+) \wedge (\mathrm{sig}_p^- \leq \mathrm{sig}_q^+) \\
\mathsf{rhsdom}_q &:= (\overline{\beta}_p \geq \overline{\beta}_q) \quad \wedge (\mathrm{sig}_p^+ \leq \mathrm{sig}_q^+) \wedge (\mathrm{sig}_p^- \geq \mathrm{sig}_q^+) \\
\mathsf{aggr} &:= (\underline{\beta}_p = \overline{\beta}_p) \quad \vee (\underline{\beta}_q = \overline{\beta}_q)
\end{aligned}$$

   If all of these Boolean values are 0, continue Loop 2 with the next pair.
   (b) For all $j \in N$, as long as one of the Boolean values of Step 2a is 1:
      i. If $a_j^p \neq a_j^q$, set $\mathsf{coefseq} := 0$.
      ii. If $a_j^p \neq -a_j^q$, set $\mathsf{coefsneg} := 0$.
      iii. If $a_j^p > a_j^q$ and $l_j < 0$ set $\mathsf{rhsdom}_p := 0$ and $\mathsf{lhsdom}_q := 0$.
      iv. If $a_j^p > a_j^q$ and $u_j > 0$ set $\mathsf{lhsdom}_p := 0$ and $\mathsf{rhsdom}_q := 0$.
      v. If $a_j^p < a_j^q$ and $l_j < 0$ set $\mathsf{lhsdom}_p := 0$ and $\mathsf{rhsdom}_q := 0$.
      vi. If $a_j^p < a_j^q$ and $u_j > 0$ set $\mathsf{rhsdom}_p := 0$ and $\mathsf{lhsdom}_q := 0$.
   (c) If $\mathsf{lhsdom}_p = 1$, set $\underline{\beta}_q := -\infty$. Else, if $\mathsf{lhsdom}_q = 1$, set $\underline{\beta}_p := -\infty$.
   If $\mathsf{rhsdom}_p = 1$, set $\overline{\beta}_q := +\infty$. Else, if $\mathsf{rhsdom}_q = 1$, set $\overline{\beta}_p := +\infty$.
   (d) If $\underline{\beta}_p = -\infty$ and $\overline{\beta}_p = +\infty$, delete $\mathcal{C}_p$ and continue Loop 2.
   If $\underline{\beta}_q = -\infty$ and $\overline{\beta}_q = +\infty$, delete $\mathcal{C}_q$ and continue Loop 2.
   (e) If $\mathsf{coefseq} = 1$, set $\underline{\beta}_p := \max\{\underline{\beta}_p, \underline{\beta}_q\}$, $\overline{\beta}_p := \min\{\overline{\beta}_p, \overline{\beta}_q\}$, delete $\mathcal{C}_q$, and continue Loop 2.
   If $\mathsf{coefsneg} = 1$, set $\underline{\beta}_p := \max\{\underline{\beta}_p, -\overline{\beta}_q\}$, $\overline{\beta}_p := \min\{\overline{\beta}_p, -\underline{\beta}_q\}$, delete $\mathcal{C}_q$, and continue Loop 2.
   (f) If $\underline{\beta}_p = \overline{\beta}_p$, select $k \in N$ to minimize

$$\omega^\star := \min\left\{\omega\left(a^q - \frac{a_k^q}{a_k^p}a^p\right) \mid k \in N : a_k^p, a_k^q \in \mathbb{Z} \setminus \{0\}\right\}.$$

   If $\omega^\star < \omega(a^q)$ and $\left\|a_j^q - \frac{a_k^q}{a_k^p}a^p\right\|_\infty \leq \left\|a_j^q\right\|_\infty$, replace $\mathcal{C}_q$ with $\mathcal{C}_q - \frac{a_k^q}{a_k^p}\mathcal{C}_p$, i.e., with

$$\underline{\beta}_q - \frac{a_k^q}{a_k^p}\underline{\beta}_p \leq \left(a^q - \frac{a_k^q}{a_k^p}a^p\right)^T x \leq \overline{\beta}_q - \frac{a_k^q}{a_k^p}\overline{\beta}_p.$$

   (g) If $\underline{\beta}_p \neq \overline{\beta}_p$ and $\underline{\beta}_q = \overline{\beta}_q$, perform an analogous calculation as in Step 2f to replace $\mathcal{C}_p$ with $\mathcal{C}_p - \frac{a_k^p}{a_k^q}\mathcal{C}_q$.

The following is a simple reformulation of the domination definition:

Observation 10.5. For constraints defined as in Definition 10.4 the following holds:

1. $\underline{\beta}_p$ dominates $\underline{\beta}_q$ if and only if $\underline{\beta}_p \geq \underline{\beta}_q$ and

$$\forall j \in N : \big((l_j < 0 \rightarrow a_j^p \geq a_j^q) \wedge (u_j > 0 \rightarrow a_j^p \leq a_j^q)\big).$$

2. $\overline{\beta}_p$ dominates $\overline{\beta}_q$ if and only if $\overline{\beta}_p \leq \overline{\beta}_q$ and

$$\forall j \in N : \big((l_j < 0 \rightarrow a_j^p \leq a_j^q) \wedge (u_j > 0 \rightarrow a_j^p \geq a_j^q)\big).$$

The purpose of identifying constraint side dominations is to remove a redundant side by applying the following proposition:

Proposition 10.6. Let $\mathcal{C}_p$ and $\mathcal{C}_q$ be defined as in Definition 10.4. Let $\mathcal{C}_q^{-\infty} : -\infty \leq (a^q)^T x \leq \overline{\beta}_q$ and $\mathcal{C}_q^{+\infty} : \underline{\beta}_q \leq (a^q)^T x \leq +\infty$ be relaxations of $\mathcal{C}_q$, and define $\mathfrak{C} := \{\mathcal{C}_p, \mathcal{C}_q\}$, $\mathfrak{C}^{-\infty} := \{\mathcal{C}_p, \mathcal{C}_q^{-\infty}\}$, and $\mathfrak{C}^{+\infty} := \{\mathcal{C}_p, \mathcal{C}_q^{+\infty}\}$. Then the following holds:

1. If $\underline{\beta}_p$ dominates $\underline{\beta}_q$, then $\mathfrak{C}(x) = \mathfrak{C}^{-\infty}(x)$ for all $x \in [l, u]$.
2. If $\overline{\beta}_p$ dominates $\overline{\beta}_q$, then $\mathfrak{C}(x) = \mathfrak{C}^{+\infty}(x)$ for all $x \in [l, u]$.

Proof. Recall that the function $\mathfrak{C}(x) : \mathbb{R}^n \rightarrow \{0, 1\}$ maps to 1 if and only if x satisfies all constraints $\mathcal{C} \in \mathfrak{C}$. Thus, in order to prove Statement 1 we have to show that the set of vectors which satisfy both constraints does not change if the left hand side of $\mathcal{C}_q$ is replaced by $-\infty$. Since $\mathfrak{C}^{-\infty}$ is a relaxation of $\mathfrak{C}$, $\mathfrak{C}(x) = 1$ implies $\mathfrak{C}^{-\infty}(x) = 1$ for all $x \in [l, u]$. To show the other direction, consider an arbitrary vector $x \in [l, u]$ with $\mathfrak{C}^{-\infty}(x) = 1$. Then,

$$\sum_{j \in N} a_j^q x_j \geq \sum_{j \in N} a_j^p x_j \geq \underline{\beta}_p \geq \underline{\beta}_q$$

holds as a direct consequence of Definition 10.4. Therefore, $\mathcal{C}_q$ is satisfied and we have $\mathfrak{C}(x) = 1$. Statement 2 can be shown analogously. □

The constraint signatures may give a simple proof for the non-dominance of the sides: if there is a potentially positive summand $a_j x_j$ in $\mathcal{C}_p$ for which the corresponding summand in $\mathcal{C}_q$ is always non-positive, or if there is a potentially negative summand in $\mathcal{C}_q$ for which the corresponding summand in $\mathcal{C}_p$ is always non-negative, constraint $\mathcal{C}_p$ cannot dominate the left hand side. Analogous reasoning applies to the other domination relations.

If the signatures allow for a reduction, we proceed with Step 2b, which compares the individual coefficients in the constraint to check exactly whether one of the possible reductions can be applied. If applicable, the removal of redundant sides is performed in Step 2c. If this resulted in a constraint with both sides being infinite, the constraint is deleted in Step 2d. If the coefficients of the constraints are either pairwise equal or pairwise negated, we can replace the two constraints by a single constraint. This means in particular that equations and ranged rows that have been disaggregated by the modeler are aggregated again into a single constraint.

Algorithm 10.6 Dual Bound Reduction for Linear Constraints

1. Calculate *redundancy bounds* $redl_j$ and $redu_j$ for all $j \in N$.
2. For all variables $j \in N$:
   (a) If $c_j \geq 0$ and x_j is only down-locked by linear constraints, update $u_j := \min\{u_j, redl_j\}$.
   (b) If $c_j \leq 0$ and x_j is only up-locked by linear constraints, update $l_j := \max\{l_j, redu_j\}$.

Steps 2f and 2g add an equation to the other constraint in order to reduce the *weighted support* of the constraint. We define the weighted support of a linear constraint to be

$$\begin{aligned}\omega(a) = \quad & \big|\{j \in B \mid a_j \neq 0\}\big| \\ & + 4\big|\{j \in I \setminus B \mid a_j \neq 0\}\big| \\ & + 8\big|\{j \in C \mid a_j \neq 0\}\big|,\end{aligned}$$

i.e., each binary variable in the constraint counts as 1, each general integer variable as 4, and each continuous variable counts as 8. The goal is to reduce the number of non-zero entries, in particular the ones that belong to non-binary variables. This can be useful to upgrade the constraint to a more specific type in Step 4 of Algorithm 10.1. The additional hope is that it also helps the aggregation heuristic of the complemented mixed integer rounding cut separator to find more cuts, and to enable the generation of flow cover cuts, see Sections 8.2 and 8.5.

For numerical reasons, we restrict the aggregation to apply the coefficient elimination only on those variables where both coefficients a^p_k and a^q_k have integral values. Additionally, we do not want the maximum norm of the coefficient vector to increase, since this may also lead to numerical issues, in particular after many aggregations applied to the same constraint. Note, however, that this aggregation of constraints is disabled in the default settings of SCIP. Without aggregation, Algorithm 10.5 can usually be executed much faster, since then we can ignore the `aggr` variable in Step 2a, which makes it more likely to skip the constraint pair.

10.1.4 Dual Bound Reduction

In Step 3 of the presolving Algorithm 10.1 for linear constraints, we perform another dual reduction that can be applied to all variables that only appear in linear constraints. Algorithm 10.6 illustrates the procedure.

Step 1 calculates so-called *redundancy bounds* for each variable $j \in N$, which are defined as follows:

Definition 10.7 (redundancy bounds). Given a set $\mathfrak{C} = \{\mathcal{C}_1, \ldots, \mathcal{C}_m\}$ of linear constraints $\mathcal{C}_i : \underline{\beta}_i \leq (a^i)^T x \leq \overline{\beta}_i$, the values $redl_j, redu_j \in \mathbb{R} \cup \{\pm\infty\}$ defined as

$$\begin{aligned} redl_j := \min\{x_j \in \mathbb{R} \mid\ & \underline{\alpha}^i_j + a^i_j x_j \geq \underline{\beta}_i \text{ for all } \mathcal{C}_i \text{ with } a^i_j > 0 \text{ and} \\ & \overline{\alpha}^i_j + a^i_j x_j \leq \overline{\beta}_i \text{ for all } \mathcal{C}_i \text{ with } a^i_j < 0\} \\ redu_j := \max\{x_j \in \mathbb{R} \mid\ & \underline{\alpha}^i_j + a^i_j x_j \geq \underline{\beta}_i \text{ for all } \mathcal{C}_i \text{ with } a^i_j < 0 \text{ and} \\ & \overline{\alpha}^i_j + a^i_j x_j \leq \overline{\beta}_i \text{ for all } \mathcal{C}_i \text{ with } a^i_j > 0\} \end{aligned}$$

are called *redundancy bounds* of variable j w.r.t. the linear constraints $\mathfrak{C}$. Here, $\underline{\alpha}_j^i$ and $\overline{\alpha}_j^i$ are the activity bound residuals of $(a^i)^T x$ w.r.t. x_j, see Definition 7.1 on page 83.

The purpose of calculating these values becomes clear with the following simple proposition:

Proposition 10.8. Let $\mathfrak{C}$ be a set of linear constraints that only have one finite side $\underline{\beta}$ or $\overline{\beta}$ each, and let $j \in N$. Then the following holds:

1. All constraints $\mathcal{C}_i \in \mathfrak{C}$ that down-lock variable x_j are redundant if $x_j \geq \mathit{redl}_j$.
2. All constraints $\mathcal{C}_i \in \mathfrak{C}$ that up-lock variable x_j are redundant if $x_j \leq \mathit{redu}_j$.

Proof. Consider the inequality $\underline{\beta} \leq a^T x$ and assume $a_j > 0$. Then, this constraint down-locks variable x_j, compare Definition 3.3 on page 38. For $x_j \geq \mathit{redl}_j$ we have $\underline{\alpha}_j + a_j x_j \geq \underline{\beta}$ by Definition 10.7. This is equivalent to

$$\forall \{x \in [l, u] \mid x_j \geq \mathit{redl}_j\} : a^T x \geq \underline{\beta},$$

which means that the constraint is redundant if $x_j \geq \mathit{redl}_j$. The remaining cases can be shown analogously. □

Proposition 10.8 is applied in Step 2 of Algorithm 10.6. If the objective coefficient c_j is non-negative and all constraints that become "less feasible" by decreasing x_j are already redundant for $x_j \geq \mathit{redl}_j$, then there is no reason to set x_j to a larger value than redl_j. In other words, if the problem instance is feasible and bounded, there is always an optimal solution $x^\star$ with $x_j^\star \leq \mathit{redl}_j$. Therefore, we can tighten the upper bound in Step 2a and still preserve at least one optimal solution. Step 2b is analogous.

10.1.5 Upgrading of Linear Constraints

The final Step 4 of the main presolving Algorithm 10.1 is to upgrade linear constraints into constraints of a more specific type. For this, the linear constraint handler provides a callback mechanism which other constraint handlers can use to take over linear constraints and convert them into equivalent constraints of their own type. The linear constraint handler calculates certain statistics for each constraint in order to simplify the upgrading process for the other constraint handlers. This is shown in Algorithm 10.7.

First, we call the normalization Algorithm 10.2 another time, since the constraint might have been modified since the last normalization in Step 1a of Algorithm 10.1. Then, we calculate a number of statistical values for the constraint, namely the number of positive and negative coefficients for binary, general integer, and continuous variables, the number of ± 1 coefficients, and the number of positive and negative integral and fractional coefficients. Finally, the upgrade methods of the constraint handlers that are hooked to the linear constraint upgrading mechanism are called in a specified order, see below. The first constraint handler that calls for the constraint transforms it into one of its own type, and the linear representation of the constraint is deleted.

The upgrade methods of other constraint handlers can use the calculated statistics, but can also consider the coefficient vector and left and right hand sides directly.

Algorithm 10.7 Upgrading of Linear Constraints

Input: Linear constraint $\underline{\beta} \leq a^T x \leq \overline{\beta}$.

1. Call Algorithm 10.2 to normalize the constraint.
2. Calculate the following statistics:

$$\begin{aligned}
\mathsf{nvars} &:= |\{j \in N \mid a_j \neq 0\}| \\
\mathsf{nposbin} &:= |\{j \in B \mid a_j > 0\}| \\
\mathsf{nnegbin} &:= |\{j \in B \mid a_j < 0\}| \\
\mathsf{nposgenint} &:= |\{j \in I \setminus B \mid a_j > 0\}| \\
\mathsf{nneggenint} &:= |\{j \in I \setminus B \mid a_j < 0\}| \\
\mathsf{nposcont} &:= |\{j \in C \mid a_j > 0\}| \\
\mathsf{nnegcont} &:= |\{j \in C \mid a_j < 0\}| \\
\mathsf{nposcoefone} &:= |\{j \in N \mid a_j = 1\}| \\
\mathsf{nnegcoefone} &:= |\{j \in N \mid a_j = -1\}| \\
\mathsf{nposcoefint} &:= |\{j \in N \mid a_j \in \mathbb{Z}_{>0} \setminus \{1\}\}| \\
\mathsf{nnegcoefint} &:= |\{j \in N \mid a_j \in \mathbb{Z}_{<0} \setminus \{-1\}\}| \\
\mathsf{nposcoeffrac} &:= |\{j \in N \mid a_j \in \mathbb{R}_{>0} \setminus \mathbb{Z}\}| \\
\mathsf{nnegcoeffrac} &:= |\{j \in N \mid a_j \in \mathbb{R}_{<0} \setminus \mathbb{Z}\}|
\end{aligned}$$

3. Call all registered constraint handlers in the given priority order until one of them took over the constraint. If successful, delete the linear constraint.

However, all constraint handlers for special types of linear constraints available in SCIP can decide the membership of the constraint to their class by inspecting the statistics only. They are called in the following order and verify the given statistics. Note that the tests check whether it is possible to reach the standard form of the constraint by complementing some of the binary variables and optionally by multiplying the constraint with -1.

1. set covering constraints: $\sum_{j \in S} x_j \geq 1$ with $S \subseteq B$
   - ▷ $\mathsf{nposbin} + \mathsf{nnegbin} = \mathsf{nvars}$,
   - ▷ $\mathsf{nposcoefone} + \mathsf{nnegcoefone} = \mathsf{nvars}$,
   - ▷ $\underline{\beta} = 1 - \mathsf{nnegcoefone}$ or $\overline{\beta} = \mathsf{nposcoefone} - 1$, and
   - ▷ $\underline{\beta} = -\infty$ or $\overline{\beta} = +\infty$.
2. set packing constraints: $\sum_{j \in S} x_j \leq 1$ with $S \subseteq B$
   - ▷ $\mathsf{nposbin} + \mathsf{nnegbin} = \mathsf{nvars}$,
   - ▷ $\mathsf{nposcoefone} + \mathsf{nnegcoefone} = \mathsf{nvars}$,
   - ▷ $\underline{\beta} = \mathsf{nposcoefone} - 1$ or $\overline{\beta} = 1 - \mathsf{nnegcoefone}$, and
   - ▷ $\underline{\beta} = -\infty$ or $\overline{\beta} = +\infty$.
3. set partitioning constraints: $\sum_{j \in S} x_j = 1$ with $S \subseteq B$

   - ▷ nposbin + nnegbin = nvars,
   - ▷ nposcoefone + nnegcoefone = nvars, and
   - ▷ $\underline{\beta} = \overline{\beta} =$ nposcoefone $- 1$ or $\underline{\beta} = \overline{\beta} = 1 -$ nnegcoefone.

4. knapsack constraints: $a^T x \leq \overline{\beta}$ with $a_j \in \mathbb{Z}_{\geq 0}$ for $j \in B$, $a_j = 0$ for $j \in N \setminus B$, and $\overline{\beta} \in \mathbb{Z}_{\geq 0}$
   - ▷ nposbin + nnegbin = nvars,
   - ▷ nposcoefone + nnegcoefone + nposcoefint + nnegcoefint = nvars, and
   - ▷ $\underline{\beta} = -\infty$ or $\overline{\beta} = +\infty$.

5. variable bound constraints: $\underline{\beta} \leq x_i + a_j x_j \leq \overline{\beta}$ with $i \in N \setminus B$ and $j \in I$
   - ▷ nvars = 2,
   - ▷ nposbin + nnegbin ≤ 1, and
   - ▷ nposcont + nnegcont ≤ 1.

Note that the knapsack constraint handler does not need to check whether the resulting right hand side after transforming the coefficients into $a \in \mathbb{Z}_{\geq 0}$ is integral and non-negative. This is already ensured by the presolving of linear constraints, compare Steps 1b and 1d of Algorithm 10.1.

10.2 KNAPSACK CONSTRAINTS

A binary knapsack constraint is a linear constraint

$$a^T x \leq \overline{\beta}$$

with $a \in \mathbb{Z}^B_{\geq 0}$, $x_j \in \{0, 1\}$ for $j \in B$, and $\overline{\beta} \in \mathbb{Z}_{\geq 0}$. The coefficients a_j are called *weights*, the right hand side $\overline{\beta}$ is called *capacity* of the knapsack.

To transform a linear inequality on binary variables with integral coefficients into a knapsack constraint, it is sometimes necessary to complement some of the binary variables using $\bar{x}_j = 1 - x_j$. If we want to access or modify the weight of a negated variable $\bar{x}_j$, we write $a_{\bar{j}}$. In terms of the active problem variables, such a "complemented" coefficient means to have a negative coefficient $a_j = -a_{\bar{j}}$ and an updated right hand side $\overline{\beta} - a_{\bar{j}}$. However, it is easier to think of x_j and $\bar{x}_j$ as being two different variables with both having non-negative coefficients. For ease of notation, we assume that there are no complemented variables in the initial form of the knapsack constraint.

Presolving for knapsack constraints is much easier than for linear constraints. On the one hand, we can usually rely on the fact that all presolving methods of the linear constraint handler have already been applied and the constraint was upgraded into a knapsack constraint afterwards. On the other hand, knapsack constraints are much simpler, since they are pure inequalities that consist of only binary variables and non-negative integral coefficients. There is one ingredient implemented in the knapsack presolving algorithm, however, that is not available for general linear constraints, at least not in the SCIP implementation: coefficient tightening and lifting using clique information.

Algorithm 10.8 describes the proceeding of the presolving for a single knapsack constraint, which is successively applied to all knapsack constraints of the problem

Algorithm 10.8 Presolving for Knapsack Constraints

Input: Knapsack constraint $a^T x \leq \overline{\beta}$ with item set $S := \{j \in N \mid a_j > 0\}$.

1. Remove fixed variables: if $l_j = u_j$, set $\overline{\beta} := \overline{\beta} - a_j l_j$, and $a_j := 0$.
2. Calculate $d := \gcd(a)$, and set $a_j := a_j/d$ and $\overline{\beta} := \lfloor \overline{\beta}/d \rfloor$.
3. Tighten the bounds of the variables by calling the domain propagation Algorithm 7.3. Remove variables from S that have been fixed to zero.
4. Sort the item set $S = \{j_1, \ldots, j_p\}$ by non-increasing weight $a_{j_1} \geq \ldots \geq a_{j_p}$.
5. Let k be the largest position in $S = \{j_1, \ldots, j_p\}$ with $a_{j_k} + a_{j_{k+1}} > \overline{\beta}$. If k exists, then do for all $l = k+1, \ldots, p$:
   (a) If $a_{j_k} + a_{j_l} > \overline{\beta}$ add the clique $x_{j_1} + \ldots + x_{j_k} + x_{j_l} \leq 1$ to the clique table.
6. Let $\overline{\alpha} := a^T \mathbb{1}$. For all $k = 1, \ldots, p$:
   (a) If $\overline{\alpha}_{j_k} := \overline{\alpha} - a_{j_k} \geq \overline{\beta}$, break the loop.
   (b) Set $\Delta := \overline{\beta} - \overline{\alpha}_{j_k}$, $a_{j_k} := a_{j_k} - \Delta$, $\overline{\beta} := \overline{\beta} - \Delta$, and $\overline{\alpha} := \overline{\alpha} - \Delta$.
7. Partition S into pairwise disjoint cliques $S = Q_1 \cup \ldots \cup Q_q$ using a greedy algorithm. For each clique Q_l let $a_{Q_l} := \max\{a_j \mid j \in Q_l\}$ be the maximum weight in the clique, and let $\overline{\alpha}^Q := \sum_{l=1}^{q} a_{Q_l}$ be the sum of the maximum clique weights. Sort the cliques by non-increasing a_{Q_l}. For all $l = 1, \ldots, q$:
   (a) If $\overline{\alpha}^Q_{Q_l} := \overline{\alpha}^Q - a_{Q_l} \geq \overline{\beta}$, break the loop.
   (b) Set $\Delta := \overline{\beta} - \overline{\alpha}^Q_{Q_l}$, $a_j := \max\{a_j - \Delta, 0\}$ for all $j \in Q_l$, $\overline{\beta} := \overline{\beta} - \Delta$, and $\overline{\alpha}^Q := \overline{\alpha}^Q - \Delta$.
   (c) If $\min\{a_j + a_k \mid j,k \in Q_l,\ j \neq k\} \leq \overline{\beta}$, add set packing constraint $\sum_{j \in Q_l} x_j \leq 1$ to the problem instance.

   Update S to account for the coefficients which have been set to zero.
8. For all binary variables x_k, $k \in B$, and values $v \in \{0,1\}$:
   (a) Calculate the clique residuals $C_l^{x_k=v} := C_l \setminus \{j \in C_l \mid x_k = v \rightarrow x_j = 0\}$ for all cliques C_l, $l = 1, \ldots, q$, in the clique partition of S.
   (b) Let $a_{Q_l^{x_k=v}} := \max\{a_j \mid j \in Q_l^{x_k=v}\}$ be the maximum weight in the clique residual, and let $\overline{\alpha}^{Q^{x_k=v}} := \sum_{l=1}^{q} a_{Q_l^{x_k=v}}$ be the sum of the maximum clique residual weights.
   (c) If $\overline{\alpha}^{Q^{x_k=v}} < \overline{\beta}$, set $q := q+1$, $\Delta := \overline{\beta} - \overline{\alpha}^{Q^{x_k=v}}$ and
      i. if $v = 1$, set $a_k := a_k + \Delta$, $Q_q := \{k\}$, and update $S := S \cup \{k\}$,
      ii. if $v = 0$, set $a_{\bar{k}} := a_{\bar{k}} + \Delta$, $Q_q := \{\bar{k}\}$, and update $S := S \cup \{\bar{k}\}$ with $a_{\bar{k}}$ being the coefficient for the complemented variable $\bar{x}_k$ of x_k.
9. For all pairs of complemented variables $k, \bar{k} \in S$:
   (a) If $a_k > a_{\bar{k}}$, set $\overline{\beta} := \overline{\beta} - a_{\bar{k}}$, $a_k := a_k - a_{\bar{k}}$, $a_{\bar{k}} := 0$, and $S := S \setminus \{\bar{k}\}$.
   (b) If $a_k < a_{\bar{k}}$, set $\overline{\beta} := \overline{\beta} - a_k$, $a_{\bar{k}} := a_{\bar{k}} - a_k$, $a_k := 0$, and $S := S \setminus \{k\}$.
   (c) If $a_k = a_{\bar{k}}$, set $\overline{\beta} := \overline{\beta} - a_k$, $a_k := 0$, $a_{\bar{k}} := 0$, and $S := S \setminus \{k, \bar{k}\}$.
10. Sort item set S by non-increasing weight a_j: $S = \{j_1, \ldots, j_q\}$.
    For all $k = 1, \ldots, q-1$:
    (a) If $a_{j_k} + a_{j_q} \leq \overline{\beta}$, break the loop.
    (b) Set $a_{j_k} := \overline{\beta}$. If $k = q-1$, set $a_{j_q} := \overline{\beta}$.

instance. Step 1 cleans up the constraint data by removing all fixed variables from the constraint. If a variable is fixed to 1, its weight has to be subtracted from the capacity. Note that additionally, all variables are replaced by their *unique representative*, which is either an active problem variable x_j, or the negation $\bar{x}_j$ of an active problem variable. In Step 2, the weights are divided by their greatest common divisor. Of course, the capacity has to be divided by the same value, and it can be rounded down afterwards.

In Step 3, we tighten the bounds of the variables by calling the domain propagation Algorithm 7.3 on page 90 and clean up the item set by removing the variables that have been fixed to zero in the propagation.

10.2.1 Clique Extraction

As all constraints in SCIP, a knapsack constraint is internally represented in a sparse fashion, i.e., only the non-zero coefficients and their corresponding indices are stored. The set S denotes this index set, and we make sure in Step 4 that it is stored in an order of non-increasing weights a_j. Then in Step 5, it is easy to extract all maximal cliques induced by the knapsack constraint in order to add them to the clique table of SCIP: they always consist of the variables with the k largest coefficients and one additional variable.

Definition 10.9 (maximal induced clique). Let $\mathcal{C} : \mathbb{R}^n \to \{0,1\}$ be a constraint on variables x_j, $j \in N$, and let $B \subseteq N$ be the subset of binary variables. Then, $Q \subseteq B$ is called *induced clique* of $\mathcal{C}$ if

$$\mathcal{C}(x) = 1 \quad \Rightarrow \quad \sum_{j \in Q} x_j \leq 1.$$

An induced clique Q of $\mathcal{C}$ is called *maximal induced clique* of $\mathcal{C}$ if

$$\mathcal{C}(x) = 1 \quad \not\Rightarrow \quad \sum_{j \in Q \cup \{q\}} x_j \leq 1$$

for all $q \in B \setminus Q$.

Proposition 10.10 (maximal induced cliques of knapsacks). Let $a^T x \leq \overline{\beta}$ be a knapsack constraint on binary variables x_j, $j \in S := \{1, \ldots, p\} \subseteq B$, with positive integral coefficients $a_1 \geq \ldots \geq a_p$ and $a_1 \leq \overline{\beta}$. Then, $Q \subseteq S$ with $|Q| \geq 2$ is a maximal induced clique of the knapsack constraint if and only if

1. $Q = \{1, \ldots, k\} \cup \{l\}$ with $1 \leq k < l \leq p$,
2. $a_k + a_l > \overline{\beta}$, and
3. $k = p - 1$ or $a_{k+1} + a_{k+2} \leq \overline{\beta}$.

Proof. Let $Q = \{1, \ldots, k\} \cup \{l\}$, $1 \leq k < l \leq p$, with $a_k + a_l > \overline{\beta}$. Then, for all $i, j \in Q$, $i \neq j$, we have

$$a_i + a_j \geq a_k + a_l > \overline{\beta},$$

which means that all variables in Q are pairwise contradictory and Q is a clique. If $k = p - 1$ then $Q = S$, and the clique is maximal because due to $a_1 \leq \overline{\beta}$ no variable $j \in B \setminus S$ (i.e., $a_j = 0$) can be a member of an induced clique. Otherwise, assume

$a_{k+1} + a_{k+2} \leq \overline{\beta}$, and suppose that Q is not a maximal induced clique. Then, there exists $l' \in S \setminus Q$ with $a_j + a_{l'} > \overline{\beta}$ for all $j \in Q$. In particular, $a_l + a_{l'} > \overline{\beta}$. From $l' \in S \setminus Q$ and $Q = \{1, \ldots, k\} \cup \{l\}$ it follows that either $l = k+1$ and $l' \geq k+2$ or $l \geq k+2$ and $l' \geq k+1$. In both cases we have

$$\overline{\beta} < a_l + a_{l'} \leq a_{k+1} + a_{k+2} \leq \overline{\beta},$$

which is a contradiction. This completes the proof of the maximality of clique Q.

To prove the other direction of the proposition, assume that Q is a maximal induced clique of the knapsack constraint and there are more than two variables after the first "hole" in the index set Q, i.e., there exist $k \in S \setminus Q$ and $l, l' \in Q$ with $k < l < l'$. Since Q is an induced clique, we have $a_j + a_{l'} > \overline{\beta}$ for all $j \in Q \setminus \{l'\}$. It follows that

$$a_j + a_k \geq a_j + a_{l'} > \overline{\beta}$$

for all $j \in Q \setminus \{l'\}$ and

$$a_k + a_{l'} \geq a_l + a_{l'} > \overline{\beta}.$$

Therefore, $Q \cup \{k\}$ is an induced clique, which contradicts the maximality of Q. Thus, Condition 1 holds. Condition 2 holds because Q is a clique, and Condition 3 follows from the maximality of Q. □

10.2.2 Coefficient Tightening

Steps 6 and 7 of Algorithm 10.8 tighten the weights of the knapsack constraint, which means to modify the weights and the capacity such that the set of feasible integral solutions to the knapsack constraint stays unaffected, but the LP relaxation is improved.

Step 6 performs the easy modifications that are already justified only by the weights and capacity. If the sum of all weights except a_k is at most the capacity $\overline{\beta}$, the constraint is redundant for $x_k = 0$. If this sum $\overline{\alpha}_k$ is smaller than $\overline{\beta}$, we can decrease the weight and the capacity by the difference $\Delta = \overline{\beta} - \overline{\alpha}_k$, since then, the constraint is also redundant for $x_k = 0$, and it has the same remaining capacity $\overline{\beta} - a_k$ for $x_k = 1$. However, the LP relaxation becomes stronger: if all other variables $j \neq k$ are set to $x_j = 1$, the LP value of x_k is forced to be 0 instead of having the possibility to be anywhere in $\left[0, \frac{\overline{\beta} - \overline{\alpha}_k}{a_k}\right]$.

Example 10.11 (coefficient tightening for knapsacks). Consider the knapsack constraint

$$12x_1 + 10x_2 + 7x_3 + 7x_4 + 5x_5 + 4x_6 + 3x_7 + x_8 \leq 42.$$

The sum of all weights is $\overline{\alpha} = 49$. The maximal residual activity bound for x_1 is $\overline{\alpha}_1 = 49 - 12 = 37 < 42$. Therefore, we can reduce a_1 and $\overline{\beta}$ by $42 - 37 = 5$ to obtain

$$7x_1 + 10x_2 + 7x_3 + 7x_4 + 5x_5 + 4x_6 + 3x_7 + x_8 \leq 37$$

and an updated weight sum of $\overline{\alpha} = 44$. For the next item x_2, the maximal residual activity bound is $\overline{\alpha}_2 = 44 - 10 = 34 < 37$. We can reduce a_2 and $\overline{\beta}$ by $37 - 34 = 3$ to obtain

$$7x_1 + 7x_2 + 7x_3 + 7x_4 + 5x_5 + 4x_6 + 3x_7 + x_8 \leq 34.$$

For x_3 we have $\overline{\alpha}_3 = 41 - 7 = 34 = \overline{\beta}$ and the algorithm aborts.

Step 7 applies the same reasoning as Step 6, but takes more information into account, namely the clique table provided by SCIP. Thus, the reductions of Step 7 are a superset of the ones of Step 6. However, if the knapsack consists of more than $|S| > 1000$ items, we skip Step 7 since it gets too expensive: the greedy algorithm which is used to partition S into cliques has a worst case runtime of $\mathcal{O}(|S|^2 \cdot |\mathcal{Q}|)$ with $\mathcal{Q}$ being the set of cliques stored in the clique table.

The reasoning for coefficient tightening using a clique partition $S = Q_1 \cup \ldots \cup Q_q$ is the following. First, it is clear that from each clique Q_l only one variable can be set to one. This means the maximum activity of the knapsack constraint is $\overline{\alpha}^Q$, which is the sum of the cliques' maximum weights. If there is a clique Q_l with residual activity bound $\overline{\alpha}^Q_{Q_l} := \overline{\alpha}^Q - a_{Q_l} \leq \overline{\beta}$, the knapsack constraint becomes redundant if we do not set any of the variables in Q_l to one. As in Step 6, if $\overline{\alpha}^Q_{Q_l} < \overline{\beta}$, we can decrease the weights of the clique elements and the capacity by the excess $\Delta := \overline{\beta} - \overline{\alpha}^Q_{Q_l}$ in order to tighten the LP relaxation. Note that this may reduce the weights of some variables to negative values. These weights can be replaced by zero, which further tightens the LP relaxation of the knapsack constraint.

Note that all reductions of Step 7 are only valid if the feasibility of the involved cliques is enforced. For a technical reason, the membership of the clique in the clique table does not suffice to ensure that it will not be violated by a solution. Therefore, we have to manually add corresponding set packing constraints in Step 7c to enforce cliques that are not implied anymore by the knapsack constraint itself. As the following example shows, this may lead to the complete disaggregation of an aggregated precedence constraint, which is usually not desirable in presolving: although the disaggregation tightens the LP relaxation, it can produce lots of additional constraints, which slow down the LP solving process. Therefore, Step 7 can be disabled via a parameter setting.

Example 10.12 (disaggregation of precedence constraints). Consider the constraint $qy + x_1 + \ldots + x_q \leq q$ with $y, x_1, \ldots, x_q \in \{0, 1\}$. The cliques extracted from this constraint are $\{y, x_l\}$, $l = 1, \ldots, q$. In the first round of the main presolving loop, Step 7 of Algorithm 10.8 partitions the indices into the cliques $Q_1 = \{y, x_1\}, Q_2 = \{x_2\}, \ldots, Q_q = \{x_q\}$. Note that the last $q - 1$ cliques are not maximal. The algorithm determines that $\overline{\alpha}^Q_{Q_1} = q - 1 < q$ which leads to the replacement of the knapsack constraint with

$$(q-1)y + x_2 + \ldots + x_q \leq q - 1 \quad \text{and} \quad y + x_1 \leq 1.$$

This is repeated in the subsequent presolving rounds with the other cliques $\{y, x_l\}$ such that, finally, the single knapsack constraint has been disaggregated into the q set packing constraints $y + x_l \leq 1$, $l = 1, \ldots, q$.

10.2.3 Clique Lifting

Step 8 of Algorithm 10.8 increases the weights of the knapsack items and lifts additional variables into the knapsack, i.e., assigns positive weights to variables x_j, $j \in B \setminus S$. This is performed by exploiting the implication graph and clique table of SCIP, see Section 3.3.5, and using the clique partition $S = Q_1 \cup \ldots \cup Q_q$ calculated in Step 7. As already said, calculating the clique partition can be expensive if there are many items in the knapsack. Therefore, we skip this step if $|S| > 1000$.

For each binary variable x_k, $k \in B$, and for their complements $\bar{x}_k$ we propagate the implications and cliques after tentatively fixing the (complemented) variable

to 1. This means, we remove all items of the cliques Q_l in the clique partition that are implied to be 0 by $x_k = 1$ or $x_k = 0$, which yields the clique residuals $Q_l^{x_k=1}$ and $Q_l^{x_k=0}$, respectively. Thus, if $x_k = v \in \{0,1\}$, the maximum activity $\overline{\alpha}^{Q^{x_k=v}}$ of the knapsack constraint is equal to the sum of the maximum weights $a_{Q_l^{x_k=v}}$ in the clique residuals. If this sum is smaller than the capacity $\overline{\beta}$, we can increase the weight of the (complemented) variable x_k ($\bar{x}_k$) by $\Delta := \overline{\beta} - \overline{\alpha}^{Q^{x_k=v}}$ in order to fill up the slack of the redundant inequality.

Note that the modification of the weight affects the clique partition and the maximum weights therein. For simplicity, we treat each modification as an addition of a new item with weight Δ to the knapsack and extend the clique partition by a new clique $Q_{q+1} = \{k\}$ or $Q_{q+1} = \{\bar{k}\}$. Thus, a previously modified weight affects the calculation of the subsequent weight liftings. Therefore, the procedure is sequence dependent. Currently, we just use the ordering of the binary variables as it is given in the data structures of SCIP.

Since the clique lifting of Step 8 might add complemented versions of variables that are already contained in the knapsack, we should clean up the constraint in the follow-up Step 9. A pair $(x_k, \bar{x}_k)$ of complemented variables will always contribute with at least the minimum $\min\{a_k, a_{\bar{k}}\}$ of their weights to the activity of the knapsack. Therefore, we can subtract this minimum from the capacity and the two weights, leaving at most one of the two items with positive weight in the knapsack.

Finally, Step 10 is a less expensive version of the clique lifting of Step 8, which only considers the cliques implied by the knapsack constraint itself. If the selection of one item forces all other variables in the knapsack to be zero, the weight of this item can be increased to be equal to the capacity. This property is very easy to verify, since we only have to check for each weight a_{j_k}, $k < q$, whether $a_{j_k} + a_{j_q} > \overline{\beta}$. Due to the processing of the items in a non-increasing order of their weights, we can immediately stop if the condition is no longer satisfied.

10.3 Set Partitioning, Set Packing, and Set Covering Constraints

Set partitioning, packing, and covering constraints ("sppc constraints") model restrictions which demand that exactly one, at most one, or at least one item of a set of items is selected. They are of the form

$$\sum_{j \in S} x_j = 1 \quad \text{(set partitioning)}$$
$$\sum_{j \in S} x_j \leq 1 \quad \text{(set packing)}$$
$$\sum_{j \in S} x_j \geq 1 \quad \text{(set covering)}$$

with $S \subseteq B$. As for knapsack constraints, some of the binary variables may be complemented in order to achieve these standard representations.

The presolving possibilities for sppc constraints are rather limited, since the structure of a single constraint is not very rich: the equation or inequality itself defines the only non-trivial facet of the associated polyhedron. This is because coefficient matrices $A \in \{0,1\}^{m \times n}$ with $m \leq 2$ are always totally unimodular. It is,

Algorithm 10.9 Presolving for Set Partitioning, Packing, and Covering Constraints

1. For all SPPC constraints $\mathcal{C}_i : \sum_{j \in S_i} x_j \diamond_i 1$, $\diamond_i \in \{=, \leq, \geq\}$:
   (a) Tighten the bounds of the variables by calling domain propagation Algorithms 7.5 and 7.7.
   (b) Remove the variables from S that have been fixed to zero.
2. For all SPPC constraints $\mathcal{C}_i$ calculate the positive signature $\text{sig}_i^+ \in \{0,1\}^{64}$.
3. For all pairs $(\mathcal{C}_p, \mathcal{C}_q)$, $p < q$, of SPPC constraints where at least one of the two has been modified since the last execution of this loop:
   (a) If $\text{sig}_p^+ \not\leq \text{sig}_q^+$ and $\text{sig}_p^+ \not\geq \text{sig}_q^+$, continue with the next pair.
   (b) If $S_p = S_q$:
      i. If $\diamond_p = \diamond_q$, delete constraint $\mathcal{C}_q$,
      ii. else if $\diamond_p =$ "$=$", delete constraint $\mathcal{C}_q$,
      iii. else if $\diamond_q =$ "$=$", delete constraint $\mathcal{C}_p$,
      iv. else set $\diamond_p :=$ "$=$" and delete constraint $\mathcal{C}_q$.
   (c) If $S_p \subset S_q$:
      i. If $\diamond_p \in \{=, \geq\}$ and $\diamond_q \in \{=, \leq\}$, set $x_j := 0$ for all $j \in S_q \setminus S_p$. Set $\diamond_p :=$ "$=$" and delete constraint $\mathcal{C}_q$.
      ii. If $\diamond_p \in \{=, \geq\}$ and $\diamond_q =$ "$\geq$", delete constraint $\mathcal{C}_q$.
      iii. If $\diamond_p =$ "$\leq$" and $\diamond_q \in \{=, \leq\}$, delete constraint $\mathcal{C}_p$.
   (d) If $S_p \supset S_q$:
      i. If $\diamond_p \in \{=, \leq\}$ and $\diamond_q \in \{=, \geq\}$, set $x_j := 0$ for all $j \in S_p \setminus S_q$. Set $\diamond_q :=$ "$=$" and delete constraint $\mathcal{C}_p$.
      ii. If $\diamond_p =$ "$\geq$" and $\diamond_q \in \{=, \geq\}$, delete constraint $\mathcal{C}_p$.
      iii. If $\diamond_p \in \{=, \leq\}$ and $\diamond_q =$ "$\leq$", delete constraint $\mathcal{C}_q$.

however, still possible to combine several constraints in order to strengthen them or to remove redundant constraints.

The presolving procedure is presented in Algorithm 10.9. The first step is to call the domain propagation procedures of Algorithms 7.5 and 7.7 for each SPPC constraint, see pages 92 and 95. This will already identify and delete redundant constraints from the problem instance. Afterwards, we remove variables from set S that are fixed to zero.

For each SPPC constraint $\mathcal{C}_i$, Step 2 calculates a signature vector $\text{sig}_i^+ \in \{0,1\}^{64}$, see Definition 10.3 on page 140. These signature vectors help to speed up the pairwise presolving loop of Step 3. Namely, if neither $\text{sig}_p^+ \leq \text{sig}_q^+$ nor $\text{sig}_p^+ \geq \text{sig}_q^+$, none of the two index sets S_p and S_q is included in the other, and we can skip the pair in Step 3a. Otherwise, we perform a complete comparison of the index sets in order to decide whether they are equal, one of them is a proper subset of the other, or they are not subsets of each other.

If the index sets are identical, we look at the type of the constraints in Step 3b. If the constraints have the same type, they are completely identical, and one of them can be discarded. If one of the constraints is a set partitioning constraint, it always dominates the other, and the other constraint can be deleted. In the remaining case, one is a set packing and the other is a set covering constraint, which means that they can be combined to a set partitioning constraint.

If $S_p \subset S_q$, we check for the constraint types in Step 3c. If $\mathcal{C}_p$ is a set partitioning

Algorithm 10.10 Presolving for Variable Bound Constraints

Input: Variable bound constraint $\underline{\beta} \leq x_i + a_j x_j \leq \overline{\beta}$.

1. Tighten the bounds of the variables by calling domain propagation Algorithm 7.9.
2. If $j \in B$, $\underline{\beta} > -\infty$, and $\overline{\beta} = +\infty$:
   (a) If $a_j > 0$ and $l_i > \underline{\beta} - a_j$, set $a_j := \underline{\beta} - l_i$.
   (b) If $a_j < 0$ and $l_i > \underline{\beta}$, set $a_j := a_j + l_i - \underline{\beta}$ and $\underline{\beta} := l_i$.
3. If $j \in B$, $\underline{\beta} = -\infty$, and $\overline{\beta} < +\infty$:
   (a) If $a_j < 0$ and $u_i < \overline{\beta} - a_j$, set $a_j := \overline{\beta} - u_i$.
   (b) If $a_j > 0$ and $u_i < \overline{\beta}$, set $a_j := a_j + u_i - \overline{\beta}$ and $\overline{\beta} := u_i$.
4. Add the variable bounds $x_i \geq -a_j x_j + \underline{\beta}$ and $x_i \leq -a_j x_j + \overline{\beta}$ to the variable bounds data structure of SCIP.

or set covering constraint, then at least one of the variables in S_p must be set to 1. This means we can fix the remaining variables in S_q to 0 if $\mathcal{C}_q$ is a set partitioning or set packing constraint. On the other hand, if $\mathcal{C}_q$ is a set covering constraint, it is redundant and can be deleted. In the case that $\mathcal{C}_p$ is a set packing constraint and $\mathcal{C}_q$ is of partitioning or packing type, constraint $\mathcal{C}_p$ is dominated by $\mathcal{C}_q$ and can be removed from the problem instance. Finally, Step 3d performs the same reductions as Step 3c with reversed roles of $\mathcal{C}_p$ and $\mathcal{C}_q$.

10.4 Variable Bound Constraints

Linear constraints of the form

$$\underline{\beta} \leq x_i + a_j x_j \leq \overline{\beta}$$

with $x_j \in \mathbb{Z}$, $a_j \in \mathbb{R}\backslash\{0\}$, and $\underline{\beta}, \overline{\beta} \in \mathbb{R} \cup \{\pm\infty\}$ are called *variable bound constraints*. The most common incarnation is the *variable upper bound* $x_i \leq u_i' x_j$ with continuous or integral x_i and binary x_j. A brief overview of the uses of such constraints can be found in Section 7.5.

As for the set partitioning, packing, and covering constraints, there are not many presolving opportunities for variable bound constraints. SCIP only applies the very basic domain propagation and coefficient tightening procedures as shown in Algorithm 10.10. Although it might be useful, we even refrain from comparing pairs of variable bound constraints, since such a pairwise comparison has usually already been performed by the linear constraint handler.

Step 1 calls the domain propagation Algorithm 7.9 to tighten the bounds of the involved variables. If the bounding variable x_j is binary and only one of the sides is finite, we can tighten the bounding coefficient a_j and the constraint side $\underline{\beta}$ or $\overline{\beta}$ in Steps 2 and 3, respectively. Namely, if the constraint is dominated by the global bounds of x_i in the non-restricting case of x_j, we can modify the coefficient and side to yield exactly the bound of x_i in the non-restricting case, but to behave unchanged in the restricting case.

Finally, Step 4 adds the variable bound information to the data structures of

Algorithm 10.11 Integer to Binary Conversion

1. For all general integer variables x_j, $j \in I \setminus B$, with $u_j = l_j + 1$:
   (a) Create a new binary variable $y \in \{0, 1\}$.
   (b) Aggregate $x_j :\overset{\star}{=} y + l_j$.

SCIP, see Section 3.3.5. This information can be useful for other components, for example the complemented mixed integer rounding or the flow cover cut separator, see Chapter 8.

10.5 Integer to Binary Conversion

Binary variables are more favorable than general integer variables for certain components of a MIP solver. For example, the probing of the following Section 10.6 is only applied to binary variables, and most of the specializations of linear constraints like the knapsack constraint deal with binary variables only and can therefore carry out their specialized algorithms only if all variables of the constraint are of binary type. Other examples of modules that work exclusively or at least better for binary variables are the flow cover cut separator of Section 8.5 and the conflict analysis explained in Chapter 11. Thus, it seems to make sense to convert general integer variables x_j with bounds $[l_j, l_j + 1]$ into binary variables by shifting them to the interval $[0, 1]$. This is carried out in Algorithm 10.11.

Despite the positive effects discussed above, there is also a reason for *not* converting general integer variables into binary variables: usually, the binary variables of the model describe qualitative "yes/no" decisions like, for example, whether a new factory should be built or not. On the other hand, general integers describe integral quantities like how many machines should be allocated to a factory. Hence, there is a structural difference between binary and general integer variables, which is captured by their type, even if a quantitative decision has only two options. In fact, many components like primal heuristics or branching rules consider the type of the variables to guide their decisions. Usually, binary variables are treated as more important and a decision on them is taken earlier. Thus, converting general integer variables into binary variables might "confuse" these components and thereby deteriorate the overall solving process.

The computational studies of Section 10.10 investigate whether the integer to binary conversion is useful. Unfortunately, it turns out that only very few instances are affected by this presolving operation, such that a definite conclusion cannot be drawn.

10.6 Probing

Probing denotes a very time-consuming but powerful preprocessing technique which evolved from the IP community, see Savelsbergh [199]. It consists of successively fixing each binary variable to zero and one and evaluating the corresponding subproblems by domain propagation techniques, see Chapter 7.

Let $x_k \in \{0, 1\}$, $k \in B$, be a binary variable, and let $x_j \in [l_j, u_j]$, $j \in N$,

denote some other (binary or non-binary) variable. Let l_j^0 and u_j^0 be the lower and upper bounds of x_j that have been deduced from $x_k = 0$. Let l_j^1 and u_j^1 be the corresponding bounds of x_j deduced from $x_k = 1$. The following observations can be made:

- ▷ If one of the fixings of x_k leads to an infeasible subproblem, x_k can be permanently fixed to the opposite value and removed from the problem instance.
- ▷ If $l_j^0 = u_j^0$ and $l_j^1 = u_j^1$, x_j can be aggregated as $x_j :\overset{\star}{=} l_j^0 + (l_j^1 - l_j^0)x_k$.
- ▷ $l'_j := \min\{l_j^0, l_j^1\}$ and $u'_j := \max\{u_j^0, u_j^1\}$ are valid global bounds of x_j.
- ▷ $x_k = 0 \to l_j^0 \leq x_j \leq u_j^0$ and $x_k = 1 \to l_j^1 \leq x_j \leq u_j^1$ are valid implications that can be stored in the implication graph of SCIP, see Section 3.3.5, and exploited during the solving process, e.g., in other preprocessing algorithms or in the branching variable selection.

The problematic issue with probing is its runtime complexity. In its full version, we have to execute the complete domain propagation loop twice for each binary variable. In order to avoid spending too much time on probing, one usually employs only a limited version of probing. One possible limitation is to restrict the number of rounds in each domain propagation loop. This can be done in SCIP via a parameter setting, but the default value of this parameter is to execute the domain propagation loop with an unlimited number of rounds, i.e., until no more deductions have been found.

Another possibility to speed up the algorithm is to not apply probing to all binary variables, but only to a subset of promising candidates. This approach is used in SCIP. The binary variables are sorted such that the most promising candidates are evaluated first. If the probing does not produce any useful information for some number of consecutive evaluated candidates, the probing algorithm is aborted. Additionally, we interrupt the probing loop after fixings or aggregations have been found in order to apply the other, less expensive presolving methods. The hope is that they are able to further reduce the size of the problem instance and to increase the chances for probing to find additional reductions. Afterwards, probing continues with the next candidate in its sorted list of binary variables.

Algorithm 10.12 depicts the details of the probing procedure. Note with default settings, probing is called in a "delayed" fashion, which means that it is skipped as long as other presolving components found reductions that trigger another round of presolving.

Step 1 of the algorithm checks whether the probing loop was aborted due to excessive useless probings, compare Step 4f. If this is the case, and no relevant changes have been applied to the instance by other presolving methods since the last probing call, we exit again. Otherwise, we reduce the "useless" counters by 10 % in Step 2 to allow for some more probings, even if the probing loop was aborted during the last call due to too many successive useless probings.

If the binary variables have not already been sorted, this is performed in Step 3. The score s_j of a variable x_j estimates the impact of fixing x_j to 0 or 1. It considers the following statistics:

- ▷ the number of constraints ζ_j^- and ζ_j^+ that get "less feasible" by setting x_j to 0 and 1, respectively (in a MIP, the sum $\zeta_j^- + \zeta_j^+$ is equal to the total number of constraints the variable is contained in, counting equations and ranged rows twice), see Definition 3.3 on page 38,

Algorithm 10.12 Probing

Input: Current values of counters startidx, nuseless, and ntotaluseless.

1. If probing was aborted in the last call, no variables have been fixed or aggregated, and no domains have been tightened since the last call, abort.
2. Set nuseless $:= 0.9 \cdot$ nuseless and ntotaluseless $:= 0.9 \cdot$ ntotaluseless.
3. If not already performed, generate a sorted list $S = \{k_1, \ldots, k_q\}$ of the binary variables x_{k_i}, $k_i \in B$: sort variables by non-increasing score

$$s_j = \zeta_j^- + \zeta_j^+ + \left|\delta_D^-(x_j = 0)\right| + \left|\delta_D^-(x_j = 1)\right| + 5\left|\mathcal{Q}(x_j = 0)\right| + 5\left|\mathcal{Q}(x_j = 1)\right|.$$

   with $D = (V, A)$ being the implication graph and $\mathcal{Q}$ being the clique table.
4. For $i =$ startidx$, \ldots, q$:
   (a) If x_{k_i} has been fixed or aggregated, continue Loop 4 with the next candidate.
   (b) For $v = 0, 1$:
      i. Tentatively fix $x_{k_i} := v$.
      ii. Propagate implications $\delta_D^-(x_{k_i} = v)$ and cliques $\mathcal{Q}(x_{k_i} = v)$ to obtain implied bounds $[\hat{l}^v, \hat{u}^v] \subseteq [l, u]$.
      iii. Call domain propagation to obtain implied bounds $[\tilde{l}^v, \tilde{u}^v] \subseteq [\hat{l}^v, \hat{u}^v]$.
      iv. Reset $l_{k_i} := 0$, $u_{k_i} := 1$, and undo all implied bound changes.
   (c) If propagation of $x_{k_i} = 0$ deduced a conflict, fix $x_{k_i} := 1$.
   If propagation of $x_{k_i} = 1$ deduced a conflict, fix $x_{k_i} := 0$.
   (d) For all $j = 1, \ldots, n$, $j \neq k_i$:
      i. Set $l_j := \min\{\tilde{l}_j^0, \tilde{l}_j^1\}$ and $u_j := \max\{\tilde{u}_j^0, \tilde{u}_j^1\}$.
      ii. If $\tilde{l}_j^0 = \tilde{u}_j^0$ and $\tilde{l}_j^1 = \tilde{u}_j^1$, aggregate $x_j \overset{\star}{:=} \tilde{l}_j^0 + (\tilde{l}_j^1 - \tilde{l}_j^0)x_{k_i}$.
      iii. If $\tilde{l}_j^0 > \hat{l}_j$, add $x_{k_i} = 0 \to x_j \geq \tilde{l}_j^0$ to the implication graph.
      If $\tilde{u}_j^0 < \hat{u}_j$, add $x_{k_i} = 0 \to x_j \leq \tilde{u}_j^0$ to the implication graph.
      If $\tilde{l}_j^1 > \hat{l}_j$, add $x_{k_i} = 1 \to x_j \geq \tilde{l}_j^1$ to the implication graph.
      If $\tilde{u}_j^1 < \hat{u}_j$, add $x_{k_i} = 1 \to x_j \leq \tilde{u}_j^1$ to the implication graph.
   (e) Increase nuseless and ntotaluseless. If a variable has been fixed or aggregated in Loop 4d, reset nuseless $:= 0$ and ntotaluseless $:= 0$. If a bound has been tightened in Loop 4d, reset ntotaluseless $:= 0$.
   (f) If nuseless ≥ 2000 or ntotaluseless ≥ 100, abort Loop 4.
   (g) If at least 50 variables have been fixed or aggregated in this probing call, interrupt Loop 4.
5. Set startidx $:= i + 1$.

- ▷ the number of implications that are triggered by $x_j = 0$ and $x_j = 1$, and
- ▷ the number of cliques $|\mathcal{Q}(x_j = v)|$ the variable is contained in as negative ($v = 0$) or positive ($v = 1$) literal, see Section 3.3.5.

We count each clique as 5 implications, which means we estimate the average clique size to be 6.

Step 4 represents the main probing loop in which the probing is applied to the individual candidates x_{k_i} of the sorted candidate list S. At first, we check in Step 4a whether the candidate is still an active variable. If it has been fixed or aggregated since the candidate list was generated, we skip the candidate. Step 4b performs the actual work of propagating the settings $x_{k_i} = 0$ and $x_{k_i} = 1$. Note that we first propagate the known implications in the implication graph and clique table in order to be able to identify unknown implications in the evaluation of Step 4(d)iii. Otherwise, we would generate the same implications over and over again and waste a considerable amount of time in the implication graph management. After having performed the implication and clique propagation we propagate the constraints.

Steps 4c and 4d evaluate the results of the probing. If one of the tentative fixings lead to an infeasibility, the probing variable can be fixed to the other value in Step 4c. If both probing directions produced a conflict, we conclude that the instance is infeasible and can abort the solving process. Note that if x_{k_i} has been fixed to $x_{k_i} := v$, we can immediately tighten the bounds of the other variables to $l := \tilde{l}^v$ and $u := \tilde{u}^v$. Otherwise, we inspect the bound deductions in Loop 4d. First, we can tighten the bounds of each variable x_j as indicated in Step 4(d)i. If this leads to a fixed variable $l_j = u_j$, we can skip the remaining steps and continue Loop 4d with the next variable. If the variable x_j is forced to one of its bounds in both probing directions, we can aggregate it in Step 4(d)ii. Otherwise, we compare the deduced bounds $[\tilde{l}_j^v, \tilde{u}_j^v]$ with the known implied bounds $[\hat{l}_j^v, \hat{u}_j^v]$ in Step 4(d)iii and add previously unknown implications to the implication graph of SCIP.

Step 4e updates the nuseless and ntotaluseless counters. We call a probing *useless* if it did not produce any fixings or aggregations of variables. If it even did not help to tighten a bound of a variable, we call it *totally useless*. We abort the probing loop in Step 4f if 2000 successive useless probings or 100 successive totally useless probings have been conducted. As mentioned above, we interrupt the probing process in Step 4g if it seems beneficial to further reduce the problem instance by applying other, less expensive presolving techniques. In the default settings of SCIP, probing is interrupted after a total number of 50 fixings or aggregations have been found in Loop 4. Finally, in Step 5 we record the index of the candidate with which we want to continue the probing loop in the next execution of the algorithm.

10.7 Implication Graph Analysis

As mentioned in the previous section, probing is a very time-consuming presolving technique and is therefore delayed until all other presolving components failed to find more reductions. Additionally, probing may be aborted prematurely. However, one part of probing can be implemented much more efficiently, namely the extraction of fixings and aggregations out of the implication graph. Therefore, this part of probing is replicated in an additional presolving plugin: the implication graph analysis.

The idea is to compare for each binary variable x_k, $k \in B$, the list of implications $\delta_D^-(x_k = 0)$ and $\delta_D^-(x_k = 1)$. Since these lists are sorted by the index of the implied

Algorithm 10.13 Implication Graph Analysis

1. For all binary variables x_k, $k \in B$, and all variables x_j, $j \in N$, that are implied variables in both implication lists, $\delta_D^-(x_k = 0)$ and $\delta_D^-(x_k = 1)$:
   (a) If $x_k = 0 \to x_j \geq l_j^0$ and $x_k = 1 \to x_j \geq l_j^1$, tighten $l_j := \min\{l_j^0, l_j^1\}$.
   (b) If $x_k = 0 \to x_j \leq u_j^0$ and $x_k = 1 \to x_j \leq u_j^1$, tighten $u_j := \max\{u_j^0, u_j^1\}$.
   (c) If $x_k = 0 \to x_j \leq l_j$ and $x_k = 1 \to x_j \geq u_j$, aggr. $x_j \overset{\star}{:=} l_j + (u_j - l_j)x_k$.
   (d) If $x_k = 0 \to x_j \geq u_j$ and $x_k = 1 \to x_j \leq l_j$, aggr. $x_j \overset{\star}{:=} u_j - (u_j - l_j)x_k$.

variable, they can be scanned in linear time w.r.t. the sum of their lengths in order to identify variables x_j that appear in both lists. The following conclusions can be made, which are similar to Steps 4(d)i and 4(d)ii of the probing Algorithm 10.12:

$$\begin{aligned}
x_k = 0 \to x_j \geq l_j^0 \ \wedge\ x_k = 1 \to x_j \geq l_j^1 \quad &\Rightarrow \quad x_j \geq \min\{l_j^0, l_j^1\} \\
x_k = 0 \to x_j \leq u_j^0 \ \wedge\ x_k = 1 \to x_j \leq u_j^1 \quad &\Rightarrow \quad x_j \leq \max\{u_j^0, u_j^1\} \\
x_k = 0 \to x_j = l_j \ \wedge\ x_k = 1 \to x_j = u_j \quad &\Rightarrow \quad x_j \overset{\star}{=} l_j + (u_j - l_j)x_k \\
x_k = 0 \to x_j = u_j \ \wedge\ x_k = 1 \to x_j = l_j \quad &\Rightarrow \quad x_j \overset{\star}{=} u_j - (u_j - l_j)x_k
\end{aligned}$$

The procedure is summarized in Algorithm 10.13. Note that the implication graph stores only non-redundant implications. This means the implied bounds are always tighter than the global bounds, and we do not have to check whether the minimum or maximum of the implied bounds used in Steps 1a and 1b is indeed better than the current global bound l_j or u_j, respectively.

10.8 Dual Fixing

Despite a few steps in the linear constraint presolving, all presolving techniques presented so far are so-called *primal presolving* algorithms. They are purely based on feasibility arguments and are therefore valid independently of the objective function.

In contrast, the reasoning of *dual presolving* techniques is based on optimality considerations. For example, if we can prove that for every optimal solution $x^\star$ there exists a solution $\hat{x}$ with the same objective value $c^\star = c^T x^\star = c^T \hat{x}$ in which a certain variable x_j has a specific value $\hat{x}_j = v$, we can fix $x_j := v$. Thereby, we may rule out a number of feasible solutions, but we are still sure that the feasibility status of the instance does not change and that an optimal solution for the presolved instance is also optimal in the original instance.

Dual presolving can be interpreted as primal presolving in the dual space. For example, like for primal variables, bound tightening can also be applied in the dual space for the dual variables and the reduced costs. If we find out that a dual variable y_i of a linear constraint $\mathcal{C}_i : a^T x \leq \overline{\beta}$ is always negative, we can conclude by the complementary slackness that the corresponding primal slack variable is always zero and the inequality is always satisfied with equality, i.e., $a^T x = \overline{\beta}$. As an additional example, if we can prove that a reduced cost r_j is always positive, we can again by the complementary slackness conclude that the primal variable x_j will always be at its lower bound, and we can fix $x_j := l_j$. Other dual reductions exploit *column domination* (which is the dual analogon of constraint domination, compare

Algorithm 10.14 Dual Fixing

1. For all variables x_j, $j \in N$:
   (a) If $c_j \geq 0$ and $\zeta_j^- = 0$, fix $x_j := l_j$.
   (b) If $c_j \leq 0$ and $\zeta_j^+ = 0$, fix $x_j := u_j$.
   (c) If the variable x_j has been fixed to an infinite value and $c_j \neq 0$, conclude that the instance is either unbounded or infeasible.

Step 2d of the pairwise linear constraint presolving Algorithm 10.5 on page 141) and *symmetric sets* of variables.

The common property of dual presolving algorithms is that they look at the *columns* $A_{\cdot j}$ of the coefficient matrix of the MIP. Unfortunately, the constraint based approach of SCIP does not support such a dual view of the problem instance, since the constraint data are stored in private data structures of the constraint handlers. This means, the necessary problem information is not accessible via framework methods, and even each constraint handler has only partial information about the problem.

A small step to remedy this situation is that SCIP explicitly collects a limited amount of dual information, which has to be provided by the constraint handlers, see Section 3.3.3. This information exists in the form of *variable locks*, see Definition 3.3 on page 38. For a MIP with inequality system $Ax \leq b$, the *down-locks* ζ_j^- count the number of negative entries in the column $A_{\cdot j}$, while the *up-locks* ζ_j^+ are the number of positive entries in $A_{\cdot j}$. More generally, for a constraint integer program the variable locks ζ_j^- and ζ_j^+ count the number of constraints that get "less feasible" by decreasing or increasing the value of x_j, respectively.

Having this information at hand, we can perform the so-called *dual fixing*, which means to fix a variable to its lower or upper bound whenever this is neither harmful to the objective function value nor to the feasibility of the constraints. This straightforward procedure is depicted in Algorithm 10.14.

For each variable x_j, Step 1a checks whether we can safely fix the variable to its lower bound without increasing the objective function value or increasing the violation of a constraint. Step 1b applies the same reasoning for the upper bound. It may happen that the variable gets fixed to an "infinite" value, since the bounds do not need to be finite. From a practical point of view, this is not a big issue: we just fix the variable to a very large value which is considered as infinity. Then, all constraints will be deleted as redundant that contain this variable. If the objective coefficient c_j of such a variable is non-zero, however, we end up with an infinite objective function value. This means, the instance is either unbounded or infeasible, depending on whether there is a feasible solution for the remaining constraints and variables. As other MIP solvers like Cplex [118] or Xpress [76], SCIP terminates in this situation with the undecided status "infeasible or unbounded". In order to decide the feasibility status of the instance, one can solve the model without objective function by setting $c := 0$.

10.9 RESTARTS

Restarts are a well-known and very important ingredient of modern SAT solvers like BERKMIN [100], MINISAT [82], or ZCHAFF [168]. Nevertheless, they have not been used so far for solving MIPs. Restarts differ from the classical presolving methods in that they are not applied *before* the branch-and-bound search commences. Instead, restarting means to abort a running search process in order to reapply presolving and to construct a new search tree from scratch. In the next solving pass, one exploits the information gathered during the previous pass. In this sence, one can view the previous search pass as a data collecting presolving step for the subsequent pass.

The most common implementation of restarts in SAT solvers is to interrupt each search pass after a certain number of *conflicts*, i.e., infeasible subproblems, have been detected. This number is increased after every restart, usually in an exponential fashion.

Note that SAT solvers based on conflict analysis (see Chapter 11) and restarts generate at least one conflict clause for each infeasible subproblem they encounter. The conflict clauses capture the reason for the infeasibility and prevent the search process from producing the same or a more general conflict again. Since SAT solvers proceed in a depth first fashion, the conflict clause database contains all information that are commonly represented as a search tree in branch-and-bound algorithms. Hence, discarding the current search tree does not lead to a loss of information: the knowledge about the search space that has already been inspected is still available in the form of conflict clauses. Therefore, restarts are an effective way to undo disadvantageous branching decisions and to increase the possibility to detect so-called *backdoor variables* (see Williams, Gomes, and Selman [213, 214]). These are variables which, if set to a fixed value, considerably reduce the difficulty of the resulting subproblems.

MIP solvers, however, differ from SAT solvers in two important properties: first, they process the nodes in a best first or similar order, thereby producing much longer lists of open subproblems during the search than depth first search based SAT solvers. Second, apart from the branching decisions and domain propagations, they store the LP warm start information in the tree. These data are very expensive to obtain since it requires the solving of the subproblems' LP relaxations. Thus, by discarding the search tree, MIP solvers waste much more information than SAT solvers, namely all the warm start LP bases that have been stored for the open leaves of the tree.

Therefore, we doubt that an extensive use of restarts improves the average runtime of a MIP solver. We performed preliminary computations which second this hypothesis. On the other hand, it is often the case that cutting planes and strong branching in the root node identify fixings of variables that have not been detected during presolving. These fixings can trigger additional presolve reductions after a restart, thereby simplifying the problem instance and improving its LP relaxation. The downside is that we have to solve the root LP relaxation again, which can sometimes be very expensive.

Nevertheless, the above observation leads to the idea of applying a restart directly after the root node has been solved and if a certain fraction of the integer variables have been fixed during the processing of the root node. In our implementation, a restart is performed if at least 5 % of the integer variables of the presolved model have been fixed.

	test set	none	no linear pairs	aggreg linear pairs	no knap disaggreg
time	MIPLIB	+65	−2	+2	−1
	CORAL	+93	−3	−5	+10
	MILP	+82	+4	+10	+8
	ENLIGHT	−7	−1	−2	−2
	ALU	+248	−6	−10	−1
	FCTP	+130	+3	+2	0
	ACC	+112	+7	+16	+19
	FC	+372	+1	+1	0
	ARCSET	+72	+24	−1	+2
	MIK	+458	+9	+2	−1
	total	**+91**	**+1**	**+2**	**+5**
nodes	MIPLIB	+163	+2	−2	+3
	CORAL	+126	−4	−6	+11
	MILP	+128	+10	+14	+1
	ENLIGHT	+50	0	0	0
	ALU	+398	−23	−43	0
	FCTP	+402	0	0	0
	ACC	+233	+35	+33	+62
	FC	+2589	−8	−4	0
	ARCSET	+114	+60	+8	0
	MIK	+204	−1	0	0
	total	**+162**	**+3**	**+1**	**+6**

Table 10.1. Performance effect of various presolving methods for solving MIP instances. The values denote the percental changes in the shifted geometric mean of the runtime (top) and number of branching nodes (bottom) compared to the default settings. Positive values represent a deterioration, negative values an improvement.

10.10 Computational Results

In this section we present benchmarks that evaluate various aspects of MIP presolving. Table 10.1 and 10.2 show the effect of disabling particular presolving methods. Detailed results can be found in Tables B.161 to B.180.

Column "none" of Table 10.1 yields the results for disabling presolving completely. Apart from the ENLIGHT instances, presolving significantly improves the performance on all test sets. Overall, disabling presolving almost leads to a doubling of the runtime and to an even larger increase in the number of branching nodes that have to be processed.

The remaining columns of Table 10.1 deal with certain subroutines in the linear and knapsack presolving. Disabling the presolving of pairs of linear constraints as depicted in Algorithm 10.5 yields the average performance ratios shown in column "no linear pairs". Besides the ARCSET, MIK, and ACC test sets, the impact of this method is rather weak.

The constraint aggregation Steps 2f and 2g of Algorithm 10.5 are disabled in the default settings. This decreases the runtime overhead of the pairwise presolving algorithm, since we can also rule out constraint pairs by the signature check in Step 2a that include equations. Activating this rather expensive step yields the results shown in column "aggreg linear pairs". The constraint aggregation can only improve the performance on the CORAL and ALU instances by a significant amount, even if we consider the number of nodes instead of the runtime. Overall, it leads to a slight deterioration.

The column labeled "no knap disaggreg" shows the benchmarks for disabling the knapsack clique disaggregation. This means, the coefficient tightening Step 7 of the knapsack presolving Algorithm 10.8 is skipped. Although it has been argued by other researchers, for example Bixby et al. [46] and Fügenschuh and Martin [90],

	test set	no int to binary	no probing	full probing	no impl graph	no dual fixing
time	MIPLIB	−3	+9	+16	−2	−1
	CORAL	−1	+20	+4	+5	+7
	MILP	+1	+12	+5	0	+3
	ENLIGHT	+35	+3	0	−2	−2
	ALU	−2	−2	−1	−22	−1
	FCTP	0	−4	0	+2	0
	ACC	+1	+100	+36	+1	0
	FC	0	+9	0	0	+4
	ARCSET	+7	−3	+2	+1	+7
	MIK	−3	+28	+3	0	+2
	total	**0**	**+14**	**+8**	**0**	**+3**
nodes	MIPLIB	−1	+16	−9	−2	+2
	CORAL	−1	+14	−17	+2	−6
	MILP	+4	+52	+5	0	−5
	ENLIGHT	+27	+44	0	0	0
	ALU	−2	−8	0	−37	0
	FCTP	0	−6	−1	0	0
	ACC	0	+286	+32	0	0
	FC	0	+15	0	0	+10
	ARCSET	+5	+13	+7	0	+11
	MIK	−2	+44	0	0	0
	total	**+1**	**+28**	**−5**	**−1**	**−2**

Table 10.2. Performance effect of generic presolving plugins for solving MIP instances. The values denote the percental changes in the shifted geometric mean of the runtime (top) and number of branching nodes (bottom) compared to the default settings. Positive values represent a deterioration, negative values an improvement.

that one should not disaggregate aggregated implied bound constraints and instead separate them on demand, our results show that—at least for SCIP and on our test sets—disaggregation of these constraints in the presolving improves the overall performance by 5 %. A possible explanation is that SCIP in its default settings only applies cutting plane separation at the root node and can therefore not generate missing implied bound cuts at local nodes. Additionally, the stronger LP relaxation due to immediate disaggregation leads to a more effective strong branching, since cutting planes are not separated for the strong branching LPs.

Table 10.3 shows the summary of our experiments on restarts. More detailed results can be found in Tables B.181 to B.190. As already said in Section 10.9, we restart the solving process after the root node has been processed and if at least 5 % of the integer variables of the presolved model have been fixed due to root node cuts or strong branching and the subsequent domain propagation. Note that for the nodes statistics, we count the total of the branching nodes in all passes. In the default settings, however, this results in only one additional node per restart since the restart is applied directly after the processing of the root node.

The results in column "no restart" for disabling the restarts show that they yield an average performance improvement of 8 %. An outstanding effect can be observed on the MIK instances for which disabling restarts multiplies the average runtime by 6. For most of these instances, the optimal solution can already be found at the root node of the first solving pass. This yields a significant amount of reduced cost strengthening and strong branching reductions, such that the subsequent presolving reduces the size of the problem instance by a large amount. For example, the instance `mik.500-5-75.1` has 500 integer variables in its original formulation, and the initial presolving can delete only one of them. After the primal heuristic RENS found the optimal solution at the root node, 349 additional integer variables could be fixed such that only 150 integer variables remained in the problem instance after

	test set	no restart	sub restart	aggr sub restart
time	MIPLIB	+3	+2	+4
	CORAL	−1	0	+2
	MILP	+10	+12	+14
	ENLIGHT	−1	−1	−1
	ALU	0	+1	0
	FCTP	+3	+2	+1
	ACC	+3	−1	+1
	FC	+1	+1	+2
	ARCSET	+9	−1	+1
	MIK	+500	0	+1
	total	**+8**	+4	**+6**
nodes	MIPLIB	−4	+3	+2
	CORAL	−4	+3	+2
	MILP	+19	+15	+21
	ENLIGHT	0	0	0
	ALU	0	0	−13
	FCTP	0	0	0
	ACC	0	0	0
	FC	+8	−2	−8
	ARCSET	+4	−6	−3
	MIK	+310	0	0
	total	**+7**	**+6**	**+6**

Table 10.3. Performance effect of using restarts for solving MIP instances. The values denote the percental changes in the shifted geometric mean of the runtime (top) and number of branching nodes (bottom) compared to the default settings. Positive values represent a deterioration, negative values an improvement.

the restart. Additionally, the second presolving reduced the number of rows in the LP from 324 to 249, although the dual bound at the root node improved.

We also experimented with applying a restart during the branch-and-bound search after a certain fraction of the integer variables has been globally fixed. Such global fixings can be identified, for example, by the *root reduced cost strengthening*, see Section 7.7, or if one of the two subtrees of the root node has been processed completely and has therefore been cut off from the tree.

Unfortunately, the results are inferior to the default settings. Column "sub restart" gives the results for applying an additional restart each time when the number of globally fixed integer variables exceeded 10 % of the total number of integer variables in the presolved model. The settings used to produce the results of column "aggr sub restart" are even more aggressive: here, the solving process is already restarted if 5 % of the integer variables have been globally fixed. A cursory inspection of the log files indicates that global variable fixings happen very infrequently during the processing of the subproblems, and if a significant amount of additional variables has been globally fixed, the search is usually almost finished such that a restart at this point is very disadvantegeous. Therefore, in order to make good use of delayed restarts, one has to invent different criteria for their application.

Chapter 11

Conflict Analysis

The branch-and-bound algorithm to solve mixed integer programs divides the given problem instance into smaller subproblems and iterates this process recursively until an optimal solution of the respective subproblem can be identified or the subproblem is detected to be infeasible or to exceed the primal bound. It seems that current state-of-the-art MIP solvers like Cplex [118], Lingo [148], SIP [159], or Xpress [76] simply discard infeasible and bound-exceeding subproblems without paying further attention to them.

The satisfiability problem can also be solved by a branch-and-bound decomposition scheme, which was originally proposed in this context by Davis, Putnam, Logemann, and Loveland [77, 78], hence the name *DPLL algorithm*. In contrast to MIP solvers, modern SAT solvers try to learn from infeasible subproblems, which is an idea due to Marques-Silva and Sakallah [157]. The infeasibilities are analyzed in order to generate so-called *conflict clauses*. These are implied clauses that help to prune the search tree. They also enable the solver to apply so-called *non-chronological backtracking*.

The idea of conflict analysis is to identify a reason for the infeasibility of the current subproblem and to exploit this knowledge later in the search. In SAT solving, such a reason is a subset of the current variable fixings that already suffices to trigger a chain of logical deductions that ends in a contradiction. This means, the fixing of the variables of this *conflict set* renders the problem instance infeasible. The *conflict clause* that can be learned from this conflict states that at least one of the variables in the conflict set has to take the opposite truth value. This clause is added to the clause database and can then be used at other subproblems to find additional implications in domain propagation, thereby pruning the search tree.

A similar idea of conflict analysis are the so-called *no-goods*, which emerged from the constraint programming community, see, e.g., Stallman and Sussman [201], Ginsberg [94], and Jiang, Richards, and Richards [124]. They can be seen as a generalization of conflict clauses to the domain of constraint programming.

In this chapter, we describe a generalization of conflict analysis to branch-and-bound based mixed integer programming and constraint integer programming. The same generalization was independently developed by Sandholm and Shields [198]. Parts of this chapter have been published in Achterberg [1].

Suppose a subproblem in the branch-and-bound search tree is detected to be infeasible or to exceed the primal bound. We will show that this situation can be analyzed with similar techniques as in SAT solving: a *conflict graph* is constructed from which *conflict constraints* can be extracted. These constraints can be used as cutting planes and in domain propagation to strengthen the relaxations of other subproblems in the tree.

Note that the term "*conflict graph*" is used differently in the SAT and MIP communities. In MIP solving, the conflict graph consists of implications between two binary variables each, see e.g., Atamtürk, Nemhauser, and Savelsbergh [24]. It represents a vertex-packing relaxation of the MIP instance and can, for instance,

be used to derive cutting planes like clique cuts, see Section 8.7. In SAT solving, and also in this chapter, the conflict graph represents the deductions that have been performed in order to prove the infeasibility of the current subproblem.

There are two main differences of MIP and SAT solving in the context of conflict analysis. First, the variables of an MIP need not to be of binary type; we also have to deal with general integer and continuous variables. Furthermore, the infeasibility of a subproblem in the MIP search tree usually has its sources in the linear programming (LP) relaxation of the subproblem. In this case, we first have to find a (preferably simple) reason for the LP's infeasibility before we can apply the SAT conflict analysis techniques for generating conflict constraints.

This chapter is organized as follows. Section 11.1 reviews conflict graph analysis of SAT solvers. For an infeasible subproblem, it is shown how to generate the conflict graph and how to extract valid conflict clauses out of this graph. Section 11.2 deals with the generalization of these techniques to mixed integer programming. We explain how infeasible and bound-exceeding linear programs can be analyzed in order to detect a conflict in the local bounds of the variables. This conflict is used as starting point to construct the conflict graph from which conflict constraints can be extracted with the techniques explained in Section 11.1. Additionally, we discuss how we generalize the notion of the conflict graph in the presence of non-binary variables. Experimental results in Section 11.3 demonstrate that conflict analysis leads to savings in the average number of branching nodes and the average time needed to solve MIPs. As the results of Chapter 17 show, conflict analysis is a key ingredient for solving the chip design verification problem with our constraint integer programming approach.

11.1 Conflict Analysis in SAT Solving

In this section we review the conflict analysis techniques used in SAT solving, see e.g., Marques-Silva and Sakallah [157] or Zhang et al. [224]. We recapitulate the definition of the satisfiability problem (SAT), see Section 1.2.

Definition (satisfiability problem). Let $\mathfrak{C} = \mathcal{C}_1 \wedge \ldots \wedge \mathcal{C}_m$ be a logic formula in conjunctive normal form (CNF) on Boolean variables $x_1, \ldots, x_n$. Each clause $\mathcal{C}_i = \ell_1^i \vee \ldots \vee \ell_{k_i}^i$ is a disjunction of literals. A literal $\ell \in L = \{x_1, \ldots, x_n, \bar{x}_1, \ldots, \bar{x}_n\}$ is either a variable x_j or the negation of a variable $\bar{x}_j$. The task of the *satisfiability problem* (SAT) is to either find an assignment $x^\star \in \{0,1\}^n$, such that the formula $\mathfrak{C}$ is satisfied, i.e., each clause $\mathcal{C}_i$ evaluates to 1, or to conclude that $\mathfrak{C}$ is unsatisfiable, i.e., for all $x \in \{0,1\}^n$ at least one $\mathcal{C}_i$ evaluates to 0.

The DPLL-algorithm to solve SAT problems fixes one of the binary variables to 0 or 1 at each node in the search tree. Then, *Boolean constraint propagation* (BCP) deduces further fixings by applying the domain propagation rule

$$\forall j \in \{1, \ldots, k_i\} \setminus \{p\} : \ell_j^i = 0 \quad \rightarrow \quad \ell_p^i = 1$$

on the clauses $\mathcal{C}_i = \ell_1^i \vee \ldots \vee \ell_{k_i}^i$, $i = 1, \ldots, m$, compare the domain propagation of set covering constraints in Section 7.4. This rule is triggered if the deletion of false literals reduces a still unsatisfied clause to a single literal, a so-called *unit clause*. In this case, the remaining literal can be fixed to 1. BCP is applied iteratively until no

more deductions can be found or a clause gets empty, i.e., all its literals are fixed to 0. The latter case is called a conflict, and conflict analysis can be performed to produce a *conflict clause*, which is explained in the following. Afterwards, the infeasible subproblem is discarded and the search continues by backtracking to a previous node and processing a different leaf of the branching tree.

11.1.1 Conflict Graph Analysis

The deductions performed in BCP can be visualized in a *conflict graph* $G = (V, A)$. The vertices $V = \{\ell_1, \ldots, \ell_k, \lambda\} \subset L \cup \{\lambda\}$ of this directed graph represent the current value assignments of the variables, with the special vertex λ representing the conflict. The arcs can be partitioned into $A = A_1 \cup \ldots \cup A_D \cup A_\lambda$. Each subset A_d, $d = 1, \ldots, D$, represents one deduction: whenever a clause $\mathcal{C}_i = \ell_1^i \vee \ldots \vee \ell_{k_i}^i \vee \ell_r^i$ became a unit clause in BCP with remaining unfixed literal ℓ_r^i, a set of arcs $A_d = \{(\bar{\ell}_1^i, \ell_r^i), \ldots, (\bar{\ell}_{k_i}^i, \ell_r^i)\}$ is created in order to represent the deduction $\bar{\ell}_1^i \wedge \ldots \wedge \bar{\ell}_{k_i}^i \rightarrow \ell_r^i$ that is implied by $\mathcal{C}_i$. The additional set of arcs $A_\lambda = \{(\bar{\ell}_1^\lambda, \lambda), \ldots, (\bar{\ell}_{k_\lambda}^\lambda, \lambda)\}$ represents clause $\mathcal{C}_\lambda$ that detected the conflict (i.e., the clause that became empty in BCP).

Example 11.1. Consider the CNF $\mathfrak{C} = \mathcal{C}_1 \wedge \ldots \wedge \mathcal{C}_{18}$ with the following clauses:

$\mathcal{C}_1 : x_1 \vee x_2$
$\mathcal{C}_2 : \bar{x}_2 \vee \bar{x}_3$
$\mathcal{C}_3 : \bar{x}_2 \vee \bar{x}_4 \vee \bar{x}_5$
$\mathcal{C}_4 : x_6 \vee \bar{x}_7$
$\mathcal{C}_5 : x_3 \vee x_5 \vee x_7 \vee x_8$
$\mathcal{C}_6 : x_3 \vee \bar{x}_8 \vee x_9$
$\mathcal{C}_7 : \bar{x}_{10} \vee x_{11}$
$\mathcal{C}_8 : \bar{x}_8 \vee x_{12} \vee x_{13}$
$\mathcal{C}_9 : x_{12} \vee x_{14}$
$\mathcal{C}_{10} : \bar{x}_8 \vee \bar{x}_{13} \vee \bar{x}_{14} \vee x_{15}$
$\mathcal{C}_{11} : \bar{x}_8 \vee \bar{x}_9 \vee \bar{x}_{15} \vee \bar{x}_{16}$
$\mathcal{C}_{12} : \bar{x}_{15} \vee x_{17}$
$\mathcal{C}_{13} : x_{16} \vee x_{18}$
$\mathcal{C}_{14} : \bar{x}_{17} \vee \bar{x}_{18}$
$\mathcal{C}_{15} : \bar{x}_{12} \vee x_{19}$
$\mathcal{C}_{16} : x_7 \vee \bar{x}_{19} \vee x_{20}$
$\mathcal{C}_{17} : x_{15} \vee \bar{x}_{20} \vee x_{21}$
$\mathcal{C}_{18} : \bar{x}_8 \vee \bar{x}_{20} \vee \bar{x}_{21}$

Suppose the fixings $x_1 = 0$, $x_4 = 1$, $x_6 = 0$, $x_{10} = 1$, and $x_{12} = 0$ were applied in the branching steps of the DPLL procedure. This leads to a conflict generated by constraint $\mathcal{C}_{14}$. The corresponding conflict graph is shown in Figure 11.1.

In the conflict graph, we distinguish between branching vertices V_B and deduced vertices $V \setminus V_B$. Branching vertices are those without incoming arcs. Each cut separating the branching vertices V_B from the conflict vertex λ gives rise to one distinct *conflict clause* (see Figure 11.1), which is obtained as follows.

Let $V = V_r \cup V_c$, $V_r \cap V_c = \emptyset$, be a partition of the vertices arising from a cut with $V_B \subseteq V_r$ and $\lambda \in V_c$. V_r is called *reason side*, and V_c is called *conflict side*. The reason side's frontier $V_f := \{\ell_p \in V_r \mid \exists (\ell_p, \ell_q) \in A,\ \ell_q \in V_c\}$ is called *conflict set*. Fixing the literals in V_f to 1 suffices to produce the conflict. Therefore, the *conflict clause* $\mathcal{C}_f = \bigvee_{\ell_j \in V_f} \bar{\ell}_j$ is valid for all feasible solutions of the SAT instance at hand.

Figure 11.1 shows different types of cuts (labeled 'A' to 'D'), leading to different conflict clauses. The cut labeled 'A' produces clause $\mathcal{C}_A = x_1 \vee \bar{x}_4 \vee x_6 \vee \bar{x}_{10} \vee x_{12}$ consisting of all branching variables. This clause does not help to prune the search tree, because the current subproblem is the only one where all branching variables are fixed to these specific values. The clause will never be violated again. Cut 'D' is not useful either, because clause $\mathcal{C}_D = \bar{x}_{17} \vee \bar{x}_{18}$ is equal to the conflict-detecting clause $\mathcal{C}_\lambda = \mathcal{C}_{14}$ and already present in the clause database. Therefore, useful cuts must be located somewhere "in between".

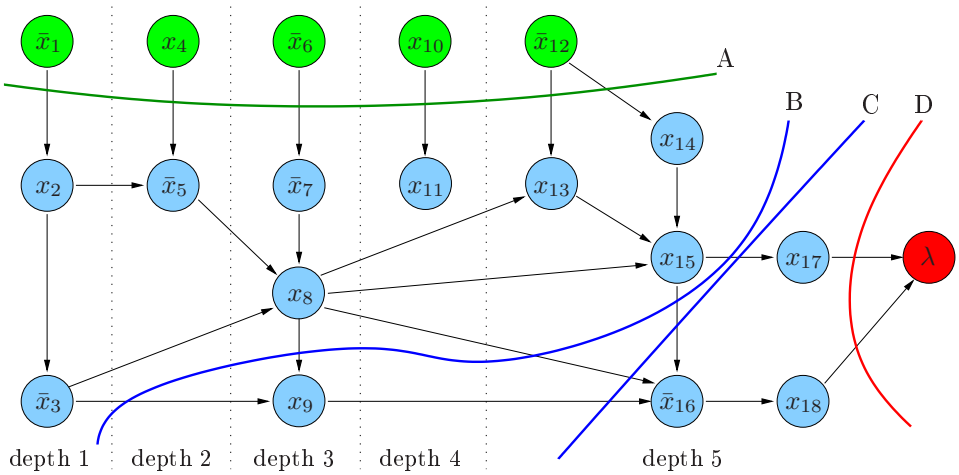

Figure 11.1. Conflict graph of Example 11.1. The vertices in the top row are branching decisions, the ones below are deductions. Each cut separating the branching vertices and the conflict vertex (λ) yields a *conflict clause*.

There are several methods for generating useful cuts. Two of them are the so-called *All-FUIP* and *1-FUIP* schemes which proved to be successful for SAT solving. These are explained in the following. We refer to Zhang et al. [224] for a more detailed discussion.

Each vertex in the conflict graph represents a fixing of a variable that was applied in one of the nodes on the path from the root node to the current node in the search tree. The *depth level* of a vertex is the depth of the node in the search tree at which the variable was fixed. In each depth level, the first fixing corresponds to a branching vertex while all subsequent fixings are deductions. In the example shown in Figure 11.1, there are 5 depth levels (excluding the root node) which are defined by the successive branching vertices $\bar{x}_1$, x_4, $\bar{x}_6$, x_{10}, and $\bar{x}_{12}$.

Definition 11.2 (unique implication point). A *unique implication point (UIP)* of depth level d is a vertex $\ell_u^d \in V$ representing a fixing in depth level d, such that every path from the branching vertex of depth level d to the conflict vertex λ goes through ℓ_u^d or through a *UIP* $\ell_{u'}^{d'}$ of higher depth level $d' > d$. The *first unique implication point (FUIP)* of a depth level d is the *UIP* $\ell_u^d \neq \lambda$ that was fixed last, i.e., that is closest to the conflict vertex λ.

Note that the *UIPs* of the different depth levels are defined recursively, starting at the last depth level, i.e., the level of the conflict. *UIPs* can be identified in linear time by a single scan through the conflict graph. In the example, the *FUIPs* of the individual depth levels are x_{15}, x_{11}, x_8, $\bar{x}_5$, and $\bar{x}_3$, respectively.

The *1-FUIP* scheme finds the first *UIP* in the last depth level. All literals that were fixed after this *FUIP* are put to the conflict side. The remaining literals and the *FUIP* are put to the reason side. In the example shown in Figure 11.1, the *FUIP* of the last depth level is x_{15}. The *1-FUIP* cut is the one labeled 'C'. It corresponds to the conflict clause $\mathcal{C}_C = \bar{x}_8 \vee \bar{x}_9 \vee \bar{x}_{15}$.

The *All-FUIP* scheme finds the first *UIP* of every single depth level. From each depth level, the literals fixed after their corresponding *FUIP* are put to the conflict side. The remaining literals and the *FUIPs* are put to the reason side. In the

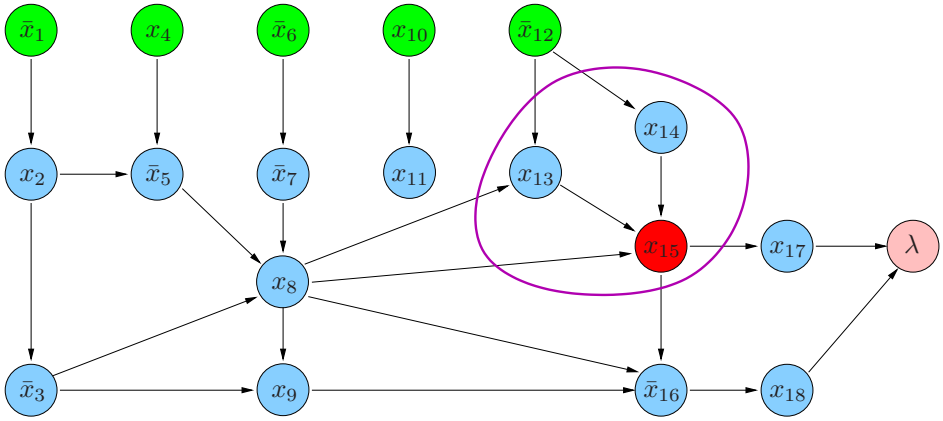

Figure 11.2. The cut separating the branching vertices (top row) and a deduced vertex (x_{15}) yields the reconvergence clause $\bar{x}_8 \vee x_{12} \vee x_{15}$.

example, this results in cut 'B' and the conflict clause $\mathcal{C}_B = x_3 \vee \bar{x}_8 \vee \bar{x}_{15}$.

11.1.2 Reconvergence Cuts

In the previous section it was shown that each cut separating the branching vertices from the conflict vertex gives rise to a conflict clause, which contains the literals of the reason side's frontier. By dropping the requirement that the cut must separate the conflict vertex from the branching vertices, we get a different class of cuts which are called *cuts not involving conflicts* (see Zhang et al. [224]). These cuts can also be used to derive valid clauses from the conflict graph. In order to apply *non-chronological backtracking*, which is explained in Section 11.1.3, one has to generate some of these cuts, in particular the *UIP reconvergence cuts* of the last depth level (see below).

Figure 11.2 gives an example of a cut not involving conflicts. In conflict graph analysis, the conflict vertex λ is substituted by an arbitrary vertex ℓ_u representing a literal. In the example, $\ell_u = x_{15}$ was chosen, which is the first unique implication point of the last depth level.

Each cut separating the branching vertices V_B from the vertex ℓ_u by partitioning the vertices V into $V_r \supseteq V_B$ and $V_c \ni \ell_u$ gives rise to a clause $\mathcal{C}_u = (\bigvee_{\ell_i \in V_f} \bar{\ell}_i) \vee \ell_u$. Again, V_f consists of the vertices at the reason side's frontier of the cut. However, such a clause is only useful if $V_c \cup V_f$ contains an *ℓ_u-reconvergence*, i.e., a vertex $\ell_i \in V_c \cup V_f$ with two different paths from ℓ_i to ℓ_u. Otherwise, it can be proven that all possible deductions of $\mathcal{C}_u$ can already be found by iterated BCP on the current clause database.

The cut shown in Figure 11.2 is a *UIP reconvergence cut*, which connects the two successive *UIPs* $\bar{x}_{12}$ and x_{15} of depth level 5: by applying all fixings of lower depth levels, $\mathcal{C}_u = \bar{x}_8 \vee x_{12} \vee x_{15}$ reduces to the implication $\bar{x}_{12} \rightarrow x_{15}$. Note that BCP can now also deduce $\bar{x}_{15} \rightarrow x_{12}$, which is not possible without using $\mathcal{C}_u$.

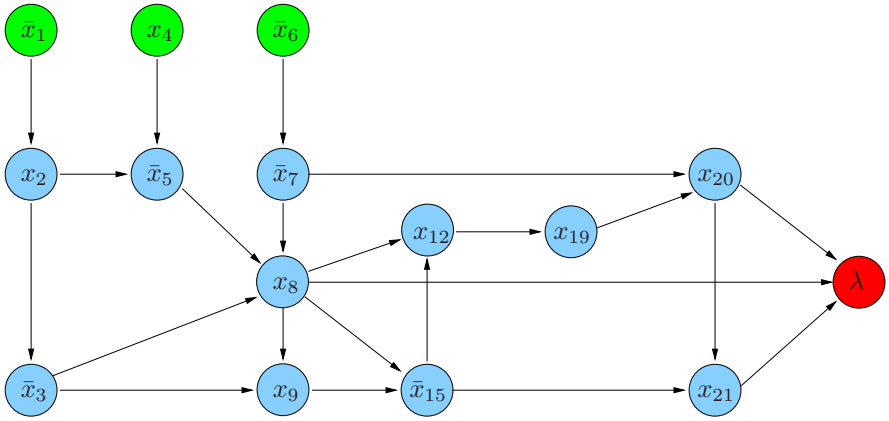

Figure 11.3. Reevaluation of the node in depth 3 after inserting conflict and reconvergence clauses again leads to a conflict.

11.1.3 NON-CHRONOLOGICAL BACKTRACKING

Suppose the conflict analysis procedure produced a clause with only one literal ℓ_u^d fixed at depth level d in which the conflict was detected. All other literals were fixed at depth levels smaller or equal to $d' < d$. If this clause would have been known earlier, the literal ℓ_u^d could already have been fixed to the opposite value in depth d'. Suppose the conflict analysis procedure also produced all *reconvergence clauses* necessary to connect ℓ_u^d to the branching vertex ℓ_b^d of depth d. Then, also the branching variable of depth d could have been fixed to the opposite value in depth d'.

Therefore, after having found such a conflict clause, the search tree's node in depth level d' can be reevaluated to apply the deductions leading to the opposite fixing of ℓ_b^d. Further deductions may lead to another conflict, thus rendering the whole subtree rooted in depth d' infeasible without evaluating its remaining leaves. Marques-Silva and Sakallah [157] empirically show that this so-called *non-chronological backtracking* can lead to large reductions in the number of evaluated nodes to solve SAT instances.

In our Example 11.1, the conflict analysis engine employed in SCIP produces the conflict clauses $\mathcal{C}_B = x_3 \vee \bar{x}_8 \vee \bar{x}_{15}$ and $\mathcal{C}_C = \bar{x}_8 \vee \bar{x}_9 \vee \bar{x}_{15}$. Additionally, the reconvergence clause $\mathcal{C}_R = \bar{x}_8 \vee x_{12} \vee x_{15}$ is added to the clause database. Evaluating the node in depth 3 again, $x_{15} = 0$ (using $\mathcal{C}_C$) and $x_{12} = 1$ (using $\mathcal{C}_R$) can be deduced, leading together with $\mathcal{C}_{15}, \ldots, \mathcal{C}_{18}$ to another conflict (see Figure 11.3). Therefore, the subtree with $x_1 = 0$, $x_4 = 1$, and $x_6 = 0$ can be pruned without evaluating the intermediate branching decisions (in this case $x_{10} = 0$ and $x_{10} = 1$).

11.2 CONFLICT ANALYSIS IN MIP

In this section we describe the generalization of the conflict analysis of Section 11.1 to mixed integer programming. Note that in this chapter we consider a mixed integer program in the *maximization* version:

$$\text{(MIP)} \quad \max\{c^T x \mid Ax \leq b,\ l \leq x \leq u,\ x_j \in \mathbb{Z} \text{ for all } j \in I\}$$

with $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^m$, $c, l, u \in \mathbb{R}^n$, and $I \subseteq N = \{1, \ldots, n\}$. A branch-and-bound based MIP solver decomposes the problem instance into subproblems typically by modifying the bounds l and u of the variables. These branching decisions may entail further deductions on the bounds of other variables, which are generated by domain propagation, see Chapter 7.

Suppose we detected a subproblem in the branch-and-bound search tree to be infeasible, either because a deduction leads to a variable with empty domain or because the LP relaxation is infeasible. To analyze this conflict, we proceed in the same fashion as in SAT solving: we construct a conflict graph, choose a cut in this graph, and produce a conflict constraint which consists of the variables in the conflict set, i.e., in the cut's frontier. Because an MIP may contain non-binary variables, we have to extend the concept of the conflict graph: it has to represent bound changes instead of fixings of variables. This generalization is described in Section 11.2.1.

A conflict in SAT solving is always detected due to a single clause that became empty during the Boolean constraint propagation process (see Section 11.1). This conflict-detecting clause provides the links from the vertices in the conflict graph that represent fixings of variables to the conflict vertex λ. In contrast, in an LP based branch-and-bound algorithm to solve mixed integer programs, infeasibility of a subproblem is almost always detected due to the infeasibility of its LP relaxation or due to the LP exceeding the primal bound. In this case the LP relaxation as a whole is responsible for the infeasibility. There is no single conflict-detecting constraint that defines the predecessors of the conflict vertex in the conflict graph. To cope with this situation, we have to analyze the LP in order to identify a subset of the bound changes that suffices to render the LP infeasible or bound-exceeding. The conflict vertex can then be connected to the vertices of this subset. Section 11.2.2 explains how to analyze infeasible LPs and how to identify an appropriate subset of the bound changes. The case of LPs having exceeded the objective bound is treated in Section 11.2.3.

Note that the LP analysis is related to the separation of *Dantzig cuts*, see Bowman and Nemhauser [53] or Rubin and Graves [195], which are known to be computationally ineffective. However, the latter include *all* non-basic variables of a fractional LP solution, while the LP analysis selects only a (hopefully very small) subset of the variables in an infeasible or bound-exceeding solution as starting point for the conflict graph analysis.

After the conflict graph has been constructed, we have to choose a cut in the graph in order to define the conflict set and the resulting conflict constraint. In the case of a binary program, i.e., $B = I = N$, $l = 0$, $u = 1$, the conflict graph can be analyzed by the same algorithms as described in Section 11.1 to produce a conflict clause $\mathcal{C}_f = \bigvee_{\ell_j \in V_f} \bar{\ell}_j$. This clause can be linearized by the set covering constraint

$$\sum_{j: x_j \in V_f} (1 - x_j) + \sum_{j: \bar{x}_j \in V_f} x_j \geq 1, \tag{11.1}$$

and added to the MIP's constraint set. However, in the presence of non-binary variables, the analysis of the conflict graph may produce a conflict set that contains bound changes on non-binary variables. In this case the conflict constraint cannot be linearized by the set covering constraint (11.1). Section 11.2.4 shows how non-binary variables can be incorporated into the conflict constraints.

11.2.1 GENERALIZED CONFLICT GRAPH

If general integer or continuous variables are present in the problem, a bound on a specific variable could have been changed more than once on the path from the root node to the current subproblem in the search tree. A local bound change on a non-binary variable can be both reason and consequence of a deduction, similar to a fixing of a binary variable. Therefore, we generalize the concept of the conflict graph: the vertices now represent bound changes instead of fixings. Note that there can now exist multiple vertices corresponding to the same non-binary variable in the conflict graph, each vertex representing one change of the variable's bounds.

Example 11.3. Consider the following constraints of an integer program with variables $x_1, \ldots, x_7 \in \{0,1\}$ and $z_1, \ldots, z_5 \in \mathbb{Z}_{\geq 0}$.

$$
\begin{aligned}
2x_1 \qquad\qquad\qquad\qquad\qquad\qquad + 3z_1 + 2z_2 \qquad\qquad &\leq 9 && (11.2)\\
+9x_2 \qquad\qquad\qquad\qquad\qquad\qquad - \ z_1 - 2z_2 \qquad\qquad &\leq 0 && (11.3)\\
-3x_2 + 5x_3 - 3x_4 \qquad\qquad\qquad\qquad\qquad\qquad &\leq 4 && (11.4)\\
-3x_2 \qquad + 9x_4 \qquad\qquad\qquad\qquad\qquad - 2z_3 \qquad &\leq 6 && (11.5)\\
+9x_5 \qquad\qquad\qquad\qquad - \ z_2 + 2z_3 \qquad &\leq 8 && (11.6)\\
-4x_6 - 7x_7 \qquad\qquad + 2z_3 \qquad &\leq 3 && (11.7)\\
+5x_7 \qquad - 2z_2 \qquad\qquad &\leq 2 && (11.8)\\
-\ x_5 \qquad + 5x_7 \qquad + 4z_2 - 5z_3 \qquad &\leq 2 && (11.9)\\
x_1 - \ x_2 + \ x_3 \qquad - 2x_5 + \ x_6 \qquad - \ z_1 - 2z_2 + \ z_3 - 2z_4 + 4z_5 &\leq 1 && (11.10)\\
+2x_2 \qquad - \ x_4 + 3x_5 - 2x_6 \qquad - \ z_1 + 5z_2 + \ z_3 + 2z_4 - 6z_5 &\leq 2 && (11.11)\\
-2x_1 \qquad - 2x_3 + \ x_4 + \ x_5 \qquad + \ z_1 + 2z_2 - 2z_3 + 2z_4 - 2z_5 &\leq 1 && (11.12)
\end{aligned}
$$

By the basic bound-strengthening techniques of Savelsbergh [199], see also Chapter 7, we can deduce upper bounds $z_1 \leq 3$, $z_2 \leq 4$, $z_3 \leq 6$, $z_4 \leq 23$, and $z_5 \leq 15$ on the general integer variables. Assume we branched on $x_1 = 1$. By applying bound-strengthening on constraint (11.2) we can deduce $z_1 \leq 2$ and $z_2 \leq 3$ (see Figure 11.4). Using constraint (11.3) and the new bounds on z_1 and z_2 it follows $x_2 = 0$. By inserting the bound on z_2 into constraint (11.6) we can also infer $z_3 \leq 5$. After branching on $x_3 = 1$ and $x_6 = 0$ and applying the deductions that follow from these branching decisions we arrive at the situation depicted in Figure 11.4 with the LP relaxation being infeasible. Note that the non-binary variables z_i appear more than once in the conflict graph. For example, the upper bound of z_3 was changed once and the lower bound was changed twice. The implications on variables z_4 and z_5 are not included in the figure. The bounds of these variables can be tightened to $7 \leq z_4 \leq 11$ and $4 \leq z_5 \leq 6$.

We use the following notation in the rest of the chapter. Let $\mathcal{B}_L = \{B_1, \ldots, B_K\}$ with hyperplanes

$$
\begin{aligned}
B_k = L_{j_k}^{\mu_k} &:= \{x \in \mathbb{R}^n \mid x_{j_k} \geq \mu_k\} \quad \text{or}\\
B_k = U_{j_k}^{\mu_k} &:= \{x \in \mathbb{R}^n \mid x_{j_k} \leq \mu_k\},
\end{aligned}
$$

with $1 \leq j_k \leq n$ and $l_{j_k} \leq \mu_k \leq u_{j_k}$ for $k = 1, \ldots, K$. The set $\mathcal{B}_L$ corresponds to the additional bounds imposed on the variables in the local subproblem. Thus, the subproblem is defined as

$$
\text{(MIP}') \quad \max \left\{ c^T x \mid Ax \leq b,\ l \leq x \leq u,\ x_j \in \mathbb{Z} \text{ for all } j \in I, x \in \bigcap_{B \in \mathcal{B}_L} B \right\}
$$

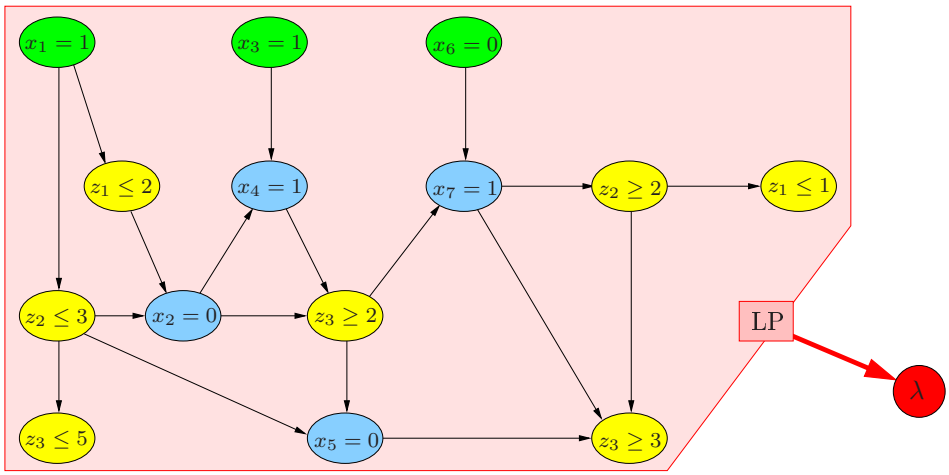

Figure 11.4. Conflict graph of Example 11.3. After applying the branching decisions $x_1 = 1$, $x_3 = 1$, $x_6 = 0$, and all inferred bound changes, the LP relaxation becomes infeasible. The implications on variables z_4 and z_5 are not included in the figure.

The vertices of the conflict graph correspond to the local bound changes $\mathcal{B}_L$. As before, the arcs of the graph represent the implications.

11.2.2 Analyzing Infeasible LPs

In order to analyze the conflict expressed by an infeasible LP, we have to find a subset $\mathcal{B}_C \subseteq \mathcal{B}_L$ of the local bound changes that suffice to render the LP (together with the global bounds and rows[1]) infeasible. If all these remaining bound changes are fixings of binary variables, this already leads to a valid inequality of type (11.1). Furthermore, even if bound changes on non-binary variables are present, such a subset can be used like the conflict-detecting clause in SAT to represent the conflict in the conflict graph. Analysis of this conflict graph may also lead to a valid inequality.

A reasonable heuristic to select $\mathcal{B}_C \subseteq \mathcal{B}_L$ is to try to make $|\mathcal{B}_C|$ as small as possible. This would produce a conflict graph with the least possible number of predecessors of the conflict vertex and thus (hopefully) a small conflict constraint. Unfortunately, the problem of finding the smallest subset of $\mathcal{B}_L$ with the LP still being infeasible is $\mathcal{NP}$-hard:

Definition 11.4 (minimal cardinality bound-IIS). Let $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^m$, and $F = \{x \in \mathbb{R}^n \mid Ax \leq b\}$. Let $\mathcal{B}_L = \{B_1, \ldots, B_K\}$ be additional bounds with $B_k = \{x \in \mathbb{R}^n \mid x_{j_k} \geq \mu_k\}$ or $B_k = \{x \in \mathbb{R}^n \mid x_{j_k} \leq \mu_k\}$, $1 \leq j_k \leq n$, for all $k = 1, \ldots, K$, such that $F \cap (\bigcap_{B \in \mathcal{B}_L} B) = \emptyset$. Then, the *minimal cardinality bound-IIS*[2] *problem* is to find a subset $\mathcal{B}_C \subseteq \mathcal{B}_L$ with

$$F \cap \Big(\bigcap_{B \in \mathcal{B}_C} B \Big) = \emptyset \quad \text{and} \quad |\mathcal{B}_C| = \min_{\mathcal{B} \subseteq \mathcal{B}_L} \Big\{ |\mathcal{B}| \;\Big|\; F \cap \Big(\bigcap_{B \in \mathcal{B}} B \Big) = \emptyset \Big\}.$$

[1] In a branch-and-cut framework, we have to either remove local cuts from the LP or mark the resulting conflict constraint being only locally valid at the depth level of the last local cut remaining in the LP. Removing local rows can of course render the LP feasible again, thus making conflict analysis impossible.

[2] IIS: irreducible inconsistent subsystem (an infeasible subsystem all of whose proper subsystems are feasible)

Proposition 11.5. The *minimal cardinality bound-IIS* problem is $\mathcal{NP}$-hard.

Proof. We provide a reduction from the *minimal cardinality IIS* problem, which is $\mathcal{NP}$-hard (see Amaldi, Pfetsch, and Trotter [11]). Given an instance $F' = (A, b)$ of the *minimal cardinality IIS* problem with $\{x \in \mathbb{R}^n \mid Ax \leq b\} = \emptyset$, the task is to find a minimal cardinality subset of the rows of $Ax \leq b$ that still defines an infeasible subsystem. Consider now the *minimal cardinality bound-IIS* problem instance

$$F = \{(x, s) \in \mathbb{R}^{n+m} \mid Ax + s = b\}$$

and $\mathcal{B}_L = \{B_1, \ldots, B_m\}$ with $B_i = \{(x, s) \mid s_i \geq 0\}$ for $i = 1, \ldots, m$. Then, for each subset $\mathcal{B} \subseteq \mathcal{B}_L$, the row index set $I_{\mathcal{B}} = \{i \mid B_i \in \mathcal{B}\}$ defines an infeasible subsystem of F' if and only if $F \cap (\bigcap_{B \in \mathcal{B}} B) = \emptyset$. Hence, there exists a one-to-one correspondence between the set of solutions of $(F, \mathcal{B}_L)$ and the one of F'. Because $|I_{\mathcal{B}}| = |\mathcal{B}|$, the optimal solution of $(F, \mathcal{B}_L)$ defines an optimal solution of F'. □

There are various heuristics for *minimal cardinality IIS* (see Pfetsch [186]). These can easily be specialized to the *minimal cardinality bound-IIS* problem. We implemented a preliminary version of a heuristic based on one of these methods which applies the Farkas lemma and works on the so-called *alternative polyhedron*, but the overhead in running time was very large. Therefore, we employ very simple heuristics using the LP information at hand, which are described in the following.

First, we will only consider the case with the global lower bounds l and local lower bounds $\tilde{l}$ being equal to $l = \tilde{l} = 0$. We further assume that each component of the global upper bounds u was tightened at most once to obtain the local upper bounds $\tilde{u} \leq u$. Thus, the set of local bound changes $\mathcal{B}_L$ consists of at most one bound change for each variable.

Suppose the local LP relaxation

$$\text{(P)} \quad \max\{c^T x \mid Ax \leq b,\ 0 \leq x \leq \tilde{u}\}$$

is infeasible. Then its dual

$$\text{(D)} \quad \min\{b^T y + \tilde{u}^T r \mid A^T y + r \geq c,\ (y, r) \geq 0\}$$

has an unbounded ray, i.e., $(\check{y}, \check{r}) \geq 0$ with $A^T \check{y} + \check{r} \geq 0$ and $b^T \check{y} + \tilde{u}^T \check{r} < 0$. Note that the dual LP does not need to be feasible.

We can aggregate the rows and bounds of the primal LP with the non-negative weights $(\check{y}, \check{r})$ to get the following proof of infeasibility:

$$0 \ \leq \ (\check{y}^T A + \check{r}^T)x \ \leq \ \check{y}^T b + \check{r}^T \tilde{u} \ < \ 0. \tag{11.13}$$

Now we try to relax the bounds as much as possible without loosing infeasibility. Note that the left hand side of Inequality (11.13) does not depend on $\tilde{u}$. Relaxing $\tilde{u}$ to some $\hat{u}$ with $\tilde{u} \leq \hat{u} \leq u$ increases the right hand side of (11.13), but as long as $\check{y}^T b + \check{r}^T \hat{u} < 0$, the relaxed LP

$$(\hat{P}) \quad \min\{c^T x \mid Ax \leq b,\ 0 \leq x \leq \hat{u}\}$$

is still infeasible with the same infeasibility proof $(\check{y}, \check{r})$. This leads to the heuristic Algorithm 11.1 to produce a relaxed upper bound vector $\hat{u}$ with the corresponding LP still being infeasible.

In the general case of multiple bound changes on a single variable, we have to process these bound changes step by step, always relaxing to the previously active

Algorithm 11.1 Infeasible LP Analysis

Input: An infeasible LP $\max\{c^T x \mid Ax \leq b,\ 0 \leq x \leq \tilde{u} \leq u\}$ with dual ray $(\check{y}, \check{r})$.

Output: Relaxed upper bounds $\hat{u} \geq \tilde{u}$ such that the LP is still infeasible.

1. Set $\hat{u} := \tilde{u}$, and calculate the infeasibility measure $d := \check{y}^T b + \check{r}^T \hat{u} < 0$.
2. Select a variable j with $\hat{u}_j < u_j$ and $d_j := d + \check{r}_j(u_j - \tilde{u}_j) < 0$. If no such variable exists, stop.
3. Set $\hat{u}_j := u_j$, update $d := d_j$, and go to Step 2.

bound. In the presence of non-zero lower bounds the reduced costs r may also be negative. In this case, we can split up the reduced cost values into $r = r^u - r^l$ with $r^u, r^l \geq 0$. It follows from the Farkas lemma that $r^u \cdot r^l = 0$. The infeasibility measure d of the dual ray has to be defined in Step 1 as $d := \check{y}^T b + (\check{r}^u)^T \hat{u} + (\check{r}^l)^T \hat{l}$. A local lower bound $\tilde{l}$ can be relaxed in the same way as an upper bound $\tilde{u}$, where u has to be replaced by l in the formulas of Steps 2 and 3.

Example 11.6 (continued from Example 11.3). After applying the deductions on the bounds of the variables in Example 11.3, the LP relaxation is infeasible. Let $y_{(i)}$ denote the dual variable of constraint (i) and r_j the reduced cost value of variable j. Then the dual ray $\check{y}_{(11.10)} = 2$, $\check{y}_{(11.11)} = 1$, $\check{y}_{(11.12)} = 1$, $\check{r}_{z_1} = 2$, $\check{r}_{z_2} = -3$, $\check{r}_{z_3} = -1$, and the remaining coefficients set to zero proves the infeasibility of the LP. In Step 1 of Algorithm 11.1 the infeasibility measure is calculated as

$$\begin{aligned} d &= \check{y}_{(11.10)} b_{(11.10)} + \check{y}_{(11.11)} b_{(11.11)} + \check{y}_{(11.12)} b_{(11.12)} + \check{r}^u_{z_1} \tilde{u}_{z_1} - \check{r}^l_{z_2} \tilde{l}_{z_2} - \check{r}^l_{z_3} \tilde{l}_{z_3} \\ &= \quad 2 \cdot 1 + \quad 1 \cdot 2 + \quad 1 \cdot 1 + \quad 2 \cdot 1 - \quad 3 \cdot 2 - \quad 1 \cdot 3 \\ &= -2. \end{aligned}$$

In Step 2, all local bounds except the upper bound of z_1 and the lower bounds of z_2 and z_3 can be relaxed to the corresponding global bounds, because their reduced cost values in the dual ray are zero. Additionally, the lower bound of z_3 can be relaxed from 3 to 2, which was the lower bound before $z_3 \geq 3$ was deduced. This relaxation increases d by 1 to $d = -1$. No further relaxations are possible without increasing d to $d \geq 0$. Thus, the local bounds $z_1 \leq 1$, $z_2 \geq 2$, and $z_3 \geq 2$ are identified as initial reason for the conflict, and the "global" arc from the LP to the conflict vertex in Figure 11.4 can be replaced by three arcs as shown in Figure 11.5. The *1-FUIP* scheme applied to the resulting conflict graph yields the cut labeled 'A' and the conflict constraint

$$(z_2 \leq 1) \vee (z_3 \leq 1).$$

Note that the involved variables z_2 and z_3 are non-binary. Section 11.2.4 shows how to proceed in this situation.

11.2.3 Analyzing LPs Exceeding the Primal Bound

In principle, the case of an LP exceeding the primal bound can be handled as in the previous section by adding an appropriate objective bound inequality to the constraint system. In the implementation, however, we use the dual solution directly

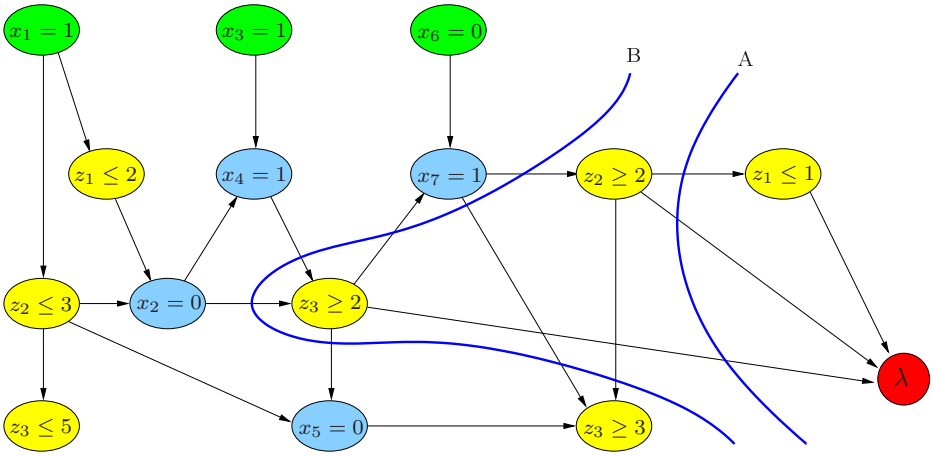

Figure 11.5. Conflict graph of Example 11.3 after the infeasible LP was analyzed. Cut 'A' is the *1-FUIP* cut. Cut 'B' was constructed by moving the non-binary variables of the conflict set of cut 'A' to the conflict side.

as a proof of objective bound excess. Then, we relax the bounds of the variables as long as the dual solution's objective value stays below the primal bound. Again, we describe the case with $l = \tilde{l} = 0$ and with at most one upper bound change per variable on the path from the root node to the local subproblem.

Suppose, the local LP relaxation

$$(P) \quad \max\{c^T x \mid Ax \leq b,\ 0 \leq x \leq \tilde{u}\}$$

exceeds (i.e., falls below) the primal objective bound $\hat{c}$. Then the dual

$$(D) \quad \min\{b^T y + \tilde{u}^T r \mid A^T y + r \geq c,\ (y, r) \geq 0\}$$

has an optimal solution $(\check{y}, \check{r})$ with $b^T\check{y} + \tilde{u}^T\check{r} \leq \hat{c}$. Note that the variables' upper bounds $\tilde{u}$ do not affect dual feasibility. Thus, after relaxing the upper bounds $\tilde{u}$ to a vector $\hat{u}$ with $\tilde{u} \leq \hat{u} \leq u$ that also satisfies $b^T\check{y} + \hat{u}^T\check{r} \leq \hat{c}$, the LP's objective value stays below the primal objective bound.

After relaxing the bounds, the vector $(\check{y}, \check{r})$ is still feasible, but not necessarily optimal for the dual LP. We may resolve the dual LP in order to get a stronger dual bound which can be used to relax further local upper bounds.

Algorithm 11.2 summarizes this procedure. As for the analysis of infeasible LPs, it is easy to generalize Algorithm 11.2 to be able to handle multiple bound changes on a single variable and non-zero lower bounds. Again, multiple bound changes have to be processed step by step, and non-zero lower bounds may lead to negative reduced cost values.

11.2.4 Conflict Constraints with Non-binary Variables

Despite the technical issue of dealing with bound changes instead of fixings in the conflict graph, there is also a principle obstacle in the presence of non-binary variables, which is the construction of the conflict constraint if non-binary variables appear in the conflict set.

The conflict graph analysis yields a conflict set, which is a subset $\mathcal{B}_f \subseteq \mathcal{B}_L$ that together with the global bounds l and u suffices to render the current subproblem

Algorithm 11.2 Bound Exceeding LP Analysis

Input: A bound exceeding LP $\max\{c^T x \mid Ax \leq b,\ 0 \leq x \leq \tilde{u} \leq u\}$, a primal objective bound $\hat{c}$, and a dual feasible solution $(\check{y}, \check{r})$ with $b^T\check{y} + \tilde{u}^T\check{r} \leq \hat{c}$.

Output: Relaxed upper bounds $\hat{u} \geq \tilde{u}$ such that the LP still exceeds the primal objective bound.

1. Set $\hat{u} := \tilde{u}$.
2. Calculate the bound excess measure $d := b^T\check{y} + \hat{u}^T\check{r} - \hat{c} \leq 0$.
3. Select a variable j with $\hat{u}_j < u_j$ and $d_j := d + \check{r}_j(u_j - \tilde{u}_j) \leq 0$. If no such variable exists, go to Step 5.
4. Set $\hat{u}_j := u_j$, update $d := d_j$, and go to Step 3.
5. (optional) If at least one upper bound was relaxed in the last iteration, resolve the dual LP to get the new dual solution $(\check{y}, \check{r})$, and go to Step 2.

infeasible. This conflict set leads to the conflict constraint

$$\bigvee_{L_j^\mu \in \mathcal{B}_f} (x_j < \mu) \quad \vee \bigvee_{U_j^\mu \in \mathcal{B}_f} (x_j > \mu).$$

Bounds on continuous variables x_j, $j \in C = N \setminus I$, would remain strict inequalities which cannot be handled using floating point arithmetics and feasibility tolerances. Therefore, we have to relax the bounds on continuous variables by allowing equality in the conflict constraint. This leads to the conflict constraint

$$\bigvee_{\substack{L_j^\mu \in \mathcal{B}_f \\ j \in I}} (x_j \leq \mu - 1) \quad \vee \bigvee_{\substack{U_j^\mu \in \mathcal{B}_f \\ j \in I}} (x_j \geq \mu + 1) \quad \vee \bigvee_{\substack{L_j^\mu \in \mathcal{B}_f \\ j \in C}} (x_j \leq \mu) \quad \vee \bigvee_{\substack{U_j^\mu \in \mathcal{B}_f \\ j \in C}} (x_j \geq \mu). \tag{11.14}$$

As shown in the introduction of Section 11.2, this constraint can be linearized by the set covering constraint (11.1) if all conflict variables are binary. However, if a non-binary variable is involved in the conflict, we cannot use such a simple linearization. In this case, (11.14) can be modeled with the help of auxiliary variables $y_j^\mu, z_j^\mu \in \{0, 1\}$:

$$\begin{aligned}
\sum_{L_j^\mu \in \mathcal{B}_f} y_j^\mu + \sum_{U_j^\mu \in \mathcal{B}_f} z_j^\mu &\geq 1 \\
x_j - (\mu - 1)y_j^\mu &\leq 0 \qquad \text{for all } L_j^\mu \in \mathcal{B}_f, j \in I \\
x_j - (\mu + 1)z_j^\mu &\geq 0 \qquad \text{for all } U_j^\mu \in \mathcal{B}_f, j \in I \\
x_j - \mu y_j^\mu &\leq 0 \qquad \text{for all } L_j^\mu \in \mathcal{B}_f, j \notin I \\
x_j - \mu z_j^\mu &\geq 0 \qquad \text{for all } U_j^\mu \in \mathcal{B}_f, j \notin I
\end{aligned} \tag{11.15}$$

The question arises, whether the potential gain in the dual bound justifies the expenses in adding system (11.15) to the LP. Many fractional points violating conflict constraint (11.14) cannot even be separated by (11.15) if the integrality restrictions on the auxiliary variables are not enforced through other cutting planes or branching. This suggests that system (11.15) is probably very weak, although we did not verify this hypotheses by computational studies.

We have the following two possibilities to avoid adding system (11.15) to the LP: either we use conflict constraints involving non-binary variables only for domain propagation but not for cutting plane separation, or we prevent the generation of conflict constraints with non-binary variables. The former demands the possibility of including non-linear constraints into the underlying MIP framework. This is possible since SCIP provides support for arbitrary constraints. For the latter option we have to modify the cut selection rules in the conflict graph analysis such that the non-binary variables are not involved in the resulting conflict set. This can be achieved by moving the bound changes on non-binary variables from the reason side's frontier to the conflict side of the cut. The following example illustrates this idea.

Example 11.7 (continued from Examples 11.3 and 11.6). Figure 11.5 shows the conflict graph of Example 11.3 after branching on $x_1 = 1$, $x_3 = 1$, and $x_6 = 0$. The analysis of the LP relaxation identified $z_1 \leq 1$, $z_2 \geq 2$, and $z_3 \geq 2$ as sufficient to cause an infeasibility in the LP (see Example 11.6). The *1-FUIP* cut selection scheme leads to the cut labeled 'A' in the figure. The corresponding conflict constraint is

$$(z_2 \leq 1) \vee (z_3 \leq 1).$$

Because there are non-binary variables involved in the conflict constraint, it cannot be linearized by the set covering constraint (11.1). To avoid the introduction of the auxiliary variables of System (11.15), the bound changes $z_2 \geq 2$ and $z_3 \geq 2$ are put to the conflict side, resulting in cut 'B'. Thus, the conflict constraint that is added to the constraint database is

$$(x_2 = 1) \vee (x_4 = 0) \vee (x_7 = 0),$$

which can be written as

$$x_2 + (1 - x_4) + (1 - x_7) \geq 1$$

in terms of a set covering constraint.

Since branching vertices must be located on the reason side, the bound changes representing branching decisions on non-binary variables cannot be moved to the conflict side. In this case, we can just remove the bound change from the conflict set in order to obtain a set covering conflict constraint. However, we thereby destroy the global validity of the resulting conflict clause. The clause can therefore only be added to the local subtree which is rooted at the node where the bound change on the non-binary variable was applied.

11.3 Computational Results

In the following, we present the computational experiments that we conducted in order to assess the performance impact of conflict analysis on solving mixed integer programs. With the parameter settings we used for the experiments, we produce one *FUIP* conflict constraint for every depth level in the conflict graph, see Section 11.1.1. This includes the *1-FUIP* and *All-FUIP* schemes as extreme cases. In addition, we generate all reconvergence constraints, see Section 11.1.2, that are needed to link the *FUIPs* of the individual depth levels to the respective branching vertex. However, we

	test set	prop	prop/inflp	prop/inflp/age	prop/lp	all	full
time	MIPLIB	0	+6	0	+12	+14	+24
	CORAL	−10	−1	−7	+2	+2	+8
	MILP	−13	−26	−28	−31	−24	−18
	ENLIGHT	−11	−21	−49	−21	−26	−26
	ALU	−28	−35	−11	−39	−39	−38
	FCTP	+1	+2	+1	+23	+20	+41
	ACC	+19	+21	+17	−2	−19	+5
	FC	+1	+1	−1	+9	+9	+18
	ARCSET	+1	−6	−6	−4	−1	+1
	MIK	+7	+4	+7	−11	−15	−17
	total	**−7**	**−8**	**−12**	**−8**	**−6**	**+1**
nodes	MIPLIB	−6	−1	−9	−14	−10	−15
	CORAL	−20	−13	−29	−22	−27	−36
	MILP	−16	−29	−36	−48	−45	−47
	ENLIGHT	−29	−48	−79	−49	−49	−49
	ALU	−61	−74	−80	−75	−78	−77
	FCTP	+1	0	0	+2	−3	−8
	ACC	+51	+57	+52	+20	−21	+17
	FC	−1	−3	−3	+1	−15	−24
	ARCSET	−5	−16	−17	−29	−29	−34
	MIK	+1	−5	−12	−53	−55	−60
	total	**−14**	**−17**	**−28**	**−31**	**−32**	**−36**

Table 11.1. Performance effect of different variants of conflict analysis for solving MIP instances. The values denote the percental changes in the shifted geometric mean of the runtime (top) and number of branching nodes (bottom) compared to the default *hybrid reliability/inference branching* rule. Positive values represent a deterioration, negative values an improvement.

restrict the number of conflict constraints produced for a single infeasible subproblem to be no larger than 10.

Conflict sets involving non-binary variables, see Section 11.2.4, are treated by the non-linear bound disjunction constraints (11.14). We do not separate conflict constraints as cutting planes, neither by linearizing bound disjunction constraints into System (11.15), nor by adding set covering inequalities (11.1) for pure binary conflict constraints to the LP relaxation. Instead, conflict constraints are solely used for domain propagation.

If a conflict constraint implies a deduction at a search node that is an ancestor of the infeasible subproblem, a non-chronological backtracking is triggered: the ancestor and all of its offspring are marked to be repropagated, which means that they are again subject to domain propagation when they become a member of the active path.[3] It may happen that such a repropagation renders the node infeasible and thereby cuts off the whole underlying subtree.

In order to avoid a large increase in the total number of constraints during the solving process, we use an aging mechanism to delete conflict constraints that seem to be useless: every time a conflict constraint is considered for domain propagation and does not yield a deduction, its age counter is increased. If its age reaches a certain limit, the constraint is discarded.

We carried out benchmarks with various settings that differ in the effort that is spent on analyzing the infeasible subproblems. Table 11.1 shows a summary of the results. More details can be found in Tables B.191 to B.200 in Appendix B.

For column "prop", we only analyzed conflicts that have been generated by domain propagation, i.e., where the propagation of a constraint produced an empty

[3]The active path is the path in the search tree from the root node to the currently processed node, see Section 3.3.6.

domain for a variable. This already achieves a significant performance improvement for most of the test sets. Most notably, the average number of branch-and-bound nodes needed to prove the infeasibility of the ALU instances reduces by more than 60 %, which translates into a runtime reduction of almost 30 %.

Column "prop/inflp" shows the results for applying Algorithm 11.1 to analyze subproblems for which the LP relaxation was infeasible in addition to the propagation conflict analysis of column "prop". The results are quite similar to using only propagation conflict analysis: for some test sets, namely MILP, ENLIGHT, ALU, and ARCSET, the performance improves, but one can observe a considerable deterioration on the MIPLIB and CORAL instances. The differences in the average number of nodes for the "prop" and "prop/inflp" settings are strongly correlated to the differences in the solving time. This indicates that the computational costs for the additional analysis of infeasible LP relaxations are negligible.

The "prop/inflp/age" settings differ from the ones used in column "prop/inflp" in the aging rule that is used to discard seemingly useless conflict constraints. Here, conflict constraints are kept longer, which leads to a larger overhead in the constraint management but also to more deductions in domain propagation. It turns out that this less aggressive constraint removal policy does not only reduce the average number of nodes but also the average solving time for almost all test sets. The only counter-examples are the ALU and MIK instances. In particular for the ALU test set, the additional reduction in the number of nodes does not compensate the higher costs for constraint management and propagation: the average number of nodes is 25 % smaller than for the default aging policy, but the average runtime is 37 % larger.

In addition to conflicting propagations and infeasible LP relaxations, the "prop/lp" settings employ Algorithm 11.2 to analyze LPs that exceed the primal bound. For each conflict, the optional LP resolving in Step 5 is executed at most twice, each time performing at most 10 dual simplex iterations. Compared to the "prop/inflp" settings, analyzing bound-exceeding LPs yields an even larger reduction in the average number of nodes to solve the instances. Unfortunately, it does not improve the overall runtime performance.

The results on the ALU instances deserve an explanation. As noted earlier, these MIP instances are infeasible. Therefore, there will never be any incumbent solution during the solving process, and it seems strange that the results differ if bound exceeding LPs are analyzed. However, even though there is no feasible solution at hand, SCIP sets a cutoff bound for the LP solver, namely the trivial bound

$$c^\star \leq \max\{c^T x \mid l \leq x \leq u\} = \sum_{c_j<0} c_j l_j + \sum_{c_j>0} c_j u_j.$$

Therefore, it may happen that the dual simplex algorithm hits the cutoff bound before it detects the infeasibility of the LP relaxation.

The settings used in column "all" extend the conflict analysis of "prop/lp" to infeasible or bound-exceeding LP relaxations encountered during the strong branching evaluation of branching candidates, see Section 5.4. Compared to "prop/lp", a notable difference in the number of branching nodes can only be observed for the ACC and FC instances. The average runtime performance gets slightly worse by using strong branching LP conflict analysis, which is mostly due to the inferior result on the MILP test set.

Analyzing infeasible or bound-exceeding strong branching LPs involves a technical issue that is related to the API of CPLEX. Strong branching is performed

by calling the special purpose method `CPXstrongbranch()`. Unfortunately, this method does not return the necessary dual information that is needed to analyze an infeasible or bound-exceeding strong branching LP. Therefore, we have to solve these LPs again with the regular `CPXdualopt()` method before we can analyze the conflict. This imposes a small artificial runtime overhead, but it is unlikely that this overhead influences the benchmark results in a significant way.

Finally, the "full" conflict analysis settings achieve the largest reduction in the average number of branching nodes. In these settings, we produce conflict constraints for conflicting domain propagations, infeasible and bound-exceeding LP relaxations, and strong branching LPs. In Algorithm 11.2, which analyzes bound-exceeding LPs, we apply the LP resolving Step 5 as often as possible and do not impose any limit on the number of simplex iterations. Additionally, there is no limit on the number of constraints produced for each conflict.

Compared to the "all" settings, this very aggressive use of conflict analysis increases the average runtime on almost all test sets, with the most notable performance deterioration on the MIPLIB, FCTP, ACC, and FC instances. With the exception of ALU and ACC, however, the average number of nodes needed to solve the instances is reduced. This indicates that such an extensive use of conflict analysis is too time consuming and leads to an unreasonable overhead in the constraint management.

We conclude from our experiments that conflict analysis is indeed a useful tool to improve the performance of MIP solvers. The largest speedup can be achieved for infeasible MIPs like the ALU instances and models of combinatorial nature as the ones in the ENLIGHT test set. The details of the implementation, however, have to be considered carefully in order to avoid a large runtime overhead for the analysis and the conflict constraint management. In particular, the policy to discard seemingly useless conflict constraints seems to be an important factor.

Part III

Chip Design Verification

Chapter 12

Introduction

This part of the thesis addresses the *property checking problem* arising in the formal verification of integrated circuit designs. This problem was treated in work package 1 of the BMBF[1] project Valse-XT[2] in cooperation with Infineon[3], OneSpin Solutions[4], and TU Kaiserslautern[5]. The introduction is joint work with Raik Brinkmann and Markus Wedler.

A recent trend in the semiconductor industry is to produce so-called *Systems-on-Chips* (SoCs). These are circuits which integrate large parts of the functionality of complete electronic systems. They are employed in cell phones, car controls, digital televisions, network processors, video games, and many other devices (see, e.g., Jerraya and Wolf [123]).

Due to the complexity of SoCs, it is a very challenging task to ensure the correctness of the chip design. The following quotation is taken from ITRS 2005 [120], the International Technology Roadmap for Semiconductors:

> *Design verification is the task of establishing that a given design accurately implements the intended behavior. Today, the verification of modern computing systems has grown to dominate the cost of electronic system design, often with limited success, as designs continue to be released with latent bugs. In fact, in many application domains, the verification portion has become the predominant component of a project development, in terms of time, cost and human resources dedicated to it. In current projects verification engineers outnumber designers, with this ratio reaching two or three to one for the most complex designs. Design conception and implementation are becoming mere preludes to the main activity of verification.*

According to Infineon [207], 60 % to 80 % of the expenses in SoC chip design are spent on verification. The goal of the Valse-XT project was to improve the current state-of-the-art in chip design verification in order to cope with the ever increasing complexity of integrated circuits.

SoCs are composed of several smaller circuit modules. It is a natural idea to verify the correctness of the chip design in a hierarchical fashion. This imposes very strong quality restrictions on the individual modules. For example, if the SoC contains 100 modules with each of them behaving correctly on 99.9 % of the input patterns, the behavior of the SoC as a whole is only correct in about 90 % of the cases (at least if the modules fail on different inputs). Therefore, it is necessary to *prove* the correctness of the individual modules, thereby showing 100 % correctness of their input-output behavior.

[1] Bundesministerium für Bildung und Forschung, http://www.bmbf.de
[2] BMBF-Ekompass Project No. 01 M 3069 A, http://www.edacentrum.de/ekompass/
[3] http://www.infineon.com
[4] http://www.onespin-solutions.com
[5] http://www-eda.eit.uni-kl.de

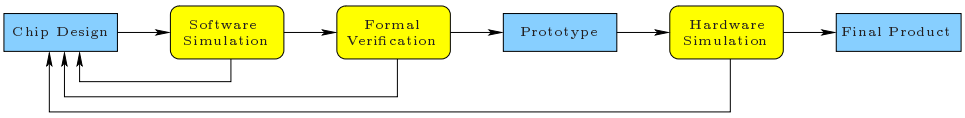

Figure 12.1. Chip manufacturing workflow.

Figure 12.1 sketches a typical work flow in the chip manufacturing process. The chip design is usually developed in a hardware design language like VERILOG, VHDL, SYSTEM-C, or SYSTEM VERILOG, which are very similar to ordinary imperative computer programming languages like C or FORTRAN. The design is tested by *software simulation*, which consists of applying many different input patterns to the input connectors of a virtual representation of the chip. If the computed output does not match the expected output, the design is flawed and has to be corrected.

Since there are 2^{nT} possible input patterns for a chip design with n input connectors running for T clock cycles, it is practically impossible to test all patterns by simulation in a reasonable amount of time. Therefore, an additional tool is needed that can actually *prove* the correctness of the chip. This task can be accomplished by *formal verification*, which is explained below. Again, if an erroneous situation is detected, the design has to be revised. Otherwise, a hardware prototype of the design is produced on which an additional round of input pattern simulation is applied. If the prototype passes this final test, the mass production of the circuit is initiated, and the chips are delivered to the customers.

In order to produce a hardware prototype, the initial circuit design has to undergo a chain of transformations via different representation levels. Figure 12.2 shows these levels, starting with the high-level chip design and ending with a detailed transistor level description, which can be used to produce a hardware prototype.

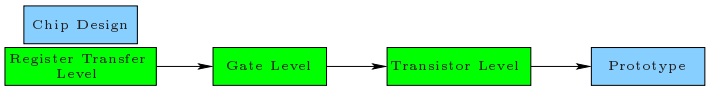

Figure 12.2. Chip design levels.

The chip design is implemented in a hardware programming language on the *register transfer level* (RT level). Here, the internal and external variables are represented as multi-bit registers. Their interrelations are expressed by arithmetic and logic operations. This representation is converted onto the *gate level* in which the registers are disaggregated into single bits, and the multi-bit operations are represented by networks of logical gates, e.g., AND, OR, XOR, and NOT gates. The *transistor level* implements the gate level by replacing the logical gates with appropriate transistors.

The idea of formal verification is that the verification engineer completely describes the expected behavior of the chip by a set of *properties*, which are formal relations between the inputs, outputs, and internal states of the circuit. Given these properties, the following two problems have to be solved:

- ▷ The *property checking problem* is to prove that the chip design satisfies a given property. This problem has to be solved for each individual property in the property set.

- ▷ The *equivalence checking problem* is to prove that the individual representations of the circuit are equivalent with respect to their input-output behavior.

The equivalence ensures that no errors are introduced during the transformation chain from chip design to transistor level.

If the chip design satisfies all of the given properties and all representations are equivalent, the chip design and the transistor model of the hardware circuit are provably correct. In the remainder of the thesis we focus on the property checking problem.

Constraint Integer Programming Approach

Properties can be defined in a language similar to the ones used to design the chip. Like the chip design, a property can be transformed to the different representation levels shown in Figure 12.2. Given that the equivalence checking has already been successfully accomplished, one can deal with the property checking problem at any suitable representation level.

Current state-of-the-art property checking algorithms operate on the *gate level* description. At this level, the design and the property can be converted into an instance of the satisfiability problem (see Section 1.2), which is then solved by a black-box SAT solver. In order to obtain a finite set of variables, one has to apply *bounded model checking* (Biere et al. [42]), which means to define a sufficiently large but finite time horizon T. Biere et al. fix the initial state at $t = 0$ to be the reset state of the circuit. In contrast, we leave the initial state of the circuit undefined, such that we can also identify errors that can only be reached after more than T time steps from the reset stage. The disadvantage is that we may report errors that result from an initial state $t = 0$, which is not reachable from the reset state and that can therefore never appear.

The reduction of the property checking problem to a SAT instance facilitates formal verification of industrial circuit designs far beyond the scope of classical model checking techniques like BDD[6] based approaches (see Biere et al. [43] or Bjesse et al. [47]). However, it is well known that SAT solvers have problems when dealing with instances derived from the verification of arithmetic circuits (as opposed to logic oriented circuits). Hence, although SAT based property checking can often be applied successfully to the control part of a design, it typically fails on data paths with large arithmetic blocks.

This motivated the development of word level solvers using integer programming (IP) [55, 83, 222] or constraint programming (CP) [221] that can be applied at the *register transfer level* where the structure of the arithmetic operations is still visible. Current IP and CP solvers, however, do not learn conflict clauses during the search like SAT solvers (see Chapter 11). Therefore, they usually perform poorly on the control part of a design. Moreover, in order to obtain highly optimized circuits designers often implement arithmetic functions at the bit level such that word level solvers are not adequate. Thus, a combination of word level and Boolean solvers has to be developed. Two promising ways of integrating IP and SAT have been proposed by Chai and Kuehlmann [59] and Audemard et al. [27]. Chai and Kuehlmann use pseudo-Boolean constraints (PBCs) as clauses in a branch-and-bound based solver, and Audemard et al. use linear equations as propositions. The reasoning of PBC solvers, however, is still limited to the bit level and the benefit of the stronger search space pruning due to learned PBCs usually does not justify the overhead for handling these more complex constraints. On the other hand, using IP techniques at

[6] binary decision diagram, see Akers [8], Bryant [57], or Madre and Billon [152]

the leaves of a decision tree without learning from infeasibilities sacrifices pruning potential in the logic part of the circuit.

Recently two alternative approaches called DPLL(T) [91] and HDPLL [184, 183] for integrating different theories into a unified DPLL[7]-style decision procedure have been proposed. DPLL(T) combines the theory of Boolean logic with the theory of uninterpreted functions with equality. In fact, there is no mechanism for learning across theories in DPLL(T). It can handle only comparisons with equality, which makes it currently unsuitable for RT level property checking. On the other hand, HDPLL combines the DPLL algorithm with techniques from CP and IP, namely domain propagation and Fourier-Motzkin elimination.

Like the mentioned word level solvers, we address the property checking problem at the register transfer level and try to exploit the structural information therein. Our approach differs from previous work in the techniques that are employed. We formulate the problem as a constraint integer program (CIP) and utilize a highly integrated combination of methods from CP, IP, and SAT solving, namely domain propagation (see Section 2.3), linear programming (LP) based branch-and-cut (see Sections 2.1 and 2.2), and generalized LP based conflict analysis (see Chapter 11). For each RT operation, a specific domain propagation algorithm is applied, using both bit and word level representations. We also provide *reverse* propagation algorithms to support conflict analysis at the RT level. In addition, we present linearizations for most of the RT operators in order to construct the LP relaxation.

In HDPLL, Fourier-Motzkin elimination is only used as a last resort to check the feasibility on the data path after all binary variables have been fixed. In contrast, we solve an LP relaxation at *every* subproblem in the search tree. Using the dual simplex algorithm the LPs can be resolved efficiently after applying changes to the variables' bounds. Due to the "global" view of the LP, infeasibilities of a subproblem can be detected much higher in the search tree than with CP or SAT techniques alone. In addition, a feasible LP solution can either yield a counter-example for the property, or can be used to control the next branching decision, thereby guiding the search (see Chapter 5).

The remaining part of the thesis is organized as follows. Chapter 13 gives a formal definition of the property checking problem as a CIP. Chapter 14 discusses the implementation of the different RT operators in detail. In particular, this includes descriptions of the operator specific LP relaxations, domain propagation procedures, and presolving algorithms. Chapter 15 explains additional global preprocessing techniques, while Chapter 16 describes the branching and node selection strategy that we employ. Finally, in Chapter 17 we give computational results on industrial benchmarks provided by ONESPIN SOLUTIONS and INFINEON, which demonstrate the effectiveness of our approach compared to state-of-the-art SAT based methods.

[7] Davis, Putnam, Logemann, and Loveland [77, 78]: branching algorithm for SAT, see Section 1.2.

Chapter 13

Formal Problem Definition

The property checking problem at register transfer level can be defined as follows:

Definition 13.1 (property checking problem). The *property checking problem* PCP is a triple $\text{PCP} = (\mathfrak{C}, P, \mathfrak{D})$ with $\mathfrak{D} = \mathcal{D}_1 \times \ldots \times \mathcal{D}_n$ representing the domains $\mathcal{D}_j = \{0, \ldots, 2^{\beta_j - 1}\}$ of register variables $\varrho_j \in \mathcal{D}_j$ with bit width $\beta_j \in \mathbb{N}$, $j = 1, \ldots, n$, $\mathfrak{C} = \{\mathcal{C}_1, \ldots, \mathcal{C}_m\}$ being a finite set of constraints $\mathcal{C}_i : \mathfrak{D} \to \{0, 1\}$, $i = 1, \ldots, m$, describing the behavior of the circuit, and $P : \mathfrak{D} \to \{0, 1\}$ being a constraint describing the property to be verified. The task is to decide whether

$$\forall \varrho \in \mathfrak{D} : \mathfrak{C}(\varrho) \to P(\varrho) \tag{13.1}$$

holds, i.e., to either find a counter-example ϱ satisfying $\mathfrak{C}(\varrho)$ but violating $P(\varrho)$ or to prove that no such counter-example exists.

In order to verify Condition (13.1) we search for a counter-example using the equivalence

$$\forall \varrho \in \mathfrak{D} : \mathfrak{C}(\varrho) \to P(\varrho) \quad \Leftrightarrow \quad \neg \left(\exists \varrho \in \mathfrak{D} : \mathfrak{C}(\varrho) \wedge \neg P(\varrho) \right). \tag{13.2}$$

The right hand side of (13.2) is a finite domain constraint satisfaction problem $\text{CSP} = (\mathfrak{C} \cup \{\neg P\}, \mathfrak{D})$, see Definition 1.1. Every feasible solution $\varrho^\star \in \mathfrak{D}$ of the CSP corresponds to a counter-example of the property. Therefore, the property is valid if and only if the CSP is infeasible.

13.1 Constraint Integer Programming Model

As shown in Proposition 1.7, any finite domain constraint satisfaction problem can be modeled as a constraint integer program (CIP). In the property checking CSP, the constraints $\mathcal{C}_i(r^i, x^i, y^i, z^i)$ resemble circuit operations $r^i = op^i(x^i, y^i, z^i)$ with up to three input registers x^i, y^i, z^i, and an output register r^i. We consider the circuit operations shown in Table 13.1 (see Brinkmann [54]). Their semantics is explained in detail in Chapter 14. Additionally, for each register variable ϱ_j, we introduce single bit variables $\varrho_{jb} \in \{0, 1\}$, $b = 0, \ldots, \beta_j - 1$, for which linking constraints

$$\varrho_j = \sum_{b=0}^{\beta_j - 1} 2^b \varrho_{jb} \tag{13.3}$$

Operation	Syntax	Signature	Semantics
Arithmetic Operations:			
Unary Minus	$r = \text{MINUS}(x)$	$[\beta] \leftarrow [\beta]$	$r = 2^\beta - x$
Addition	$r = \text{ADD}(x, y)$	$[\beta] \leftarrow [\beta] \times [\beta]$	$r = (x + y) \bmod 2^\beta$
Subtraction	$r = \text{SUB}(x, y)$	$[\beta] \leftarrow [\beta] \times [\beta]$	$r = (x - y) \bmod 2^\beta$
Multiplication	$r = \text{MULT}(x, y)$	$[\beta] \leftarrow [\beta] \times [\beta]$	$r = (x \cdot y) \bmod 2^\beta$
Bit Operations:			
Negation	$r = \text{NOT}(x)$	$[\beta] \leftarrow [\beta]$	$r_b = 1 - x_b$ for all b
Bitwise And	$r = \text{AND}(x, y)$	$[\beta] \leftarrow [\beta] \times [\beta]$	$r_b = x_b \wedge y_b$ for all b
Bitwise Or	$r = \text{OR}(x, y)$	$[\beta] \leftarrow [\beta] \times [\beta]$	$r_b = x_b \vee y_b$ for all b
Bitwise Xor	$r = \text{XOR}(x, y)$	$[\beta] \leftarrow [\beta] \times [\beta]$	$r_b = x_b \oplus y_b$ for all b
Data → Control Interface:			
Unary And	$r = \text{UAND}(x)$	$[1] \leftarrow [\beta]$	$r = x_0 \wedge \ldots \wedge x_{\beta-1}$
Unary Or	$r = \text{UOR}(x)$	$[1] \leftarrow [\beta]$	$r = x_0 \vee \ldots \vee x_{\beta-1}$
Unary Xor	$r = \text{UXOR}(x)$	$[1] \leftarrow [\beta]$	$r = x_0 \oplus \ldots \oplus x_{\beta-1}$
Equality	$r = \text{EQ}(x, y)$	$[1] \leftarrow [\beta] \times [\beta]$	$r = 1 \Leftrightarrow x = y$
Less-than	$r = \text{LT}(x, y)$	$[1] \leftarrow [\beta] \times [\beta]$	$r = 1 \Leftrightarrow x < y$
Control → Data Interface:			
If-then-else	$r = \text{ITE}(x, y, z)$	$[\beta] \leftarrow [1] \times [\beta] \times [\beta]$	$r = \begin{cases} y & \text{if } x = 1, \\ z & \text{if } x = 0 \end{cases}$
Word Extension:			
Zero Extension	$r = \text{ZEROEXT}(x)$	$[\beta] \leftarrow [\mu]$	$r_b = \begin{cases} x_b & \text{if } b < \mu, \\ 0 & \text{if } b \geq \mu \end{cases}$
Sign Extension	$r = \text{SIGNEXT}(x)$	$[\beta] \leftarrow [\mu]$	$r_b = \begin{cases} x_b & \text{if } b < \mu, \\ x_{\mu-1} & \text{if } b \geq \mu \end{cases}$
Concatenation	$r = \text{CONCAT}(x, y)$	$[\beta + \mu] \leftarrow [\beta] \times [\mu]$	$r_b = \begin{cases} y_b & \text{if } b < \mu, \\ x_{b-\mu} & \text{if } b \geq \mu \end{cases}$
Subword Access:			
Shift Left	$r = \text{SHL}(x, y)$	$[\beta] \leftarrow [\beta] \times [\mu]$	$r_b = \begin{cases} x_{b-y} & \text{if } b \geq y, \\ 0 & \text{if } b < y \end{cases}$
Shift Right	$r = \text{SHR}(x, y)$	$[\beta] \leftarrow [\beta] \times [\mu]$	$r_b = \begin{cases} x_{b+y} & \text{if } b + y < \beta, \\ 0 & \text{if } b + y \geq \beta \end{cases}$
Slicing	$r = \text{SLICE}(x, y)$	$[\beta] \leftarrow [\mu] \times [\nu]$	$r_b = \begin{cases} x_{b+y} & \text{if } b + y < \mu, \\ 0 & \text{if } b + y \geq \mu \end{cases}$
Multiplex Read	$r = \text{READ}(x, y)$	$[\beta] \leftarrow [\mu] \times [\nu]$	$r_b = \begin{cases} x_{b+y\cdot\beta} & \text{if } b + y \cdot \beta < \mu, \\ 0 & \text{if } b + y \cdot \beta \geq \mu \end{cases}$
Multiplex Write	$r = \text{WRITE}(x, y, z)$	$[\beta] \leftarrow [\beta] \times [\mu] \times [\nu]$	$r_b = \begin{cases} z_{b-y\cdot\nu} & \text{if } 0 \leq b - y \cdot \nu < \nu, \\ x_b & \text{otherwise} \end{cases}$

Table 13.1. Circuit operations. The domains in the signature are defined as $[\beta] = \{0, \ldots, \beta - 1\}$.

define their correlation to the register variable. Altogether, this yields the following constraint integer program:

$$
\begin{array}{lrll}
\min & c^T \varrho & & \\
\text{s.t.} & \mathcal{C}_i(\varrho) & & \text{for } i = 1, \ldots, m \\
& \neg P(\varrho) & & \\
& \varrho_j & = \sum_{b=0}^{\beta_j - 1} 2^b \varrho_{jb} & \text{for } j = 1, \ldots, n \\
0 \leq & \varrho_j & \leq 2^{\beta_j - 1} & \text{for } j = 1, \ldots, n \\
& \varrho_j & \in \mathbb{Z} & \text{for } j = 1, \ldots, n \\
& \varrho_{jb} & \in \{0, 1\} & \text{for } j = 1, \ldots, n \text{ and } b = 0, \ldots, \beta_j - 1
\end{array}
\tag{13.4}
$$

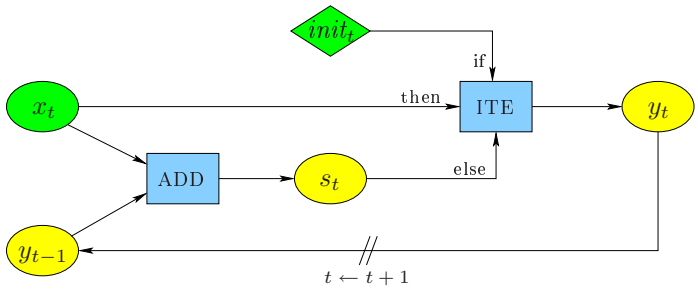

Figure 13.1. Adder which accumulates input over time.

Since the property checking problem is a pure feasibility problem, the objective function c is irrelevant and can be chosen arbitrarily. However, the choice of objective function influences the solving process. We experimented with three choices, namely $c = 0$, $c_{jb} = 1$ for all register bits ϱ_{jb}, and $c_{jb} = -1$ for all register bits. It turned out that this choice does not have a large impact on the solving performance. Therefore, we omit these benchmarks in the computational results of Chapter 17.

Example 13.2. Figure 13.1 shows a circuit $\mathcal{A}$ which adds up values that are given in the input register x_t at consecutive time steps t. The sum is accumulated in the internal register variable y_t, which is simultaneously used as the output of the circuit. If the single bit input $init_t$ is set to 1, the accumulator y_t is initialized with the value of x_t. Otherwise, x_t is added to the previous sum y_{t-1}.

We want to verify whether the addition performed by the circuit satisfies the commutative law. To formulate this property, we create a copy $\mathcal{A}'$ of the circuit with register variables $init'_t$, x'_t, and y'_t. In a first attempt we verify the property constraint

$$init_0 = init'_0 = 1 \ \wedge \ x_0 = x'_1 \ \wedge \ x_1 = x'_0 \quad \rightarrow \quad y_1 = y'_1. \tag{13.5}$$

To verify the property, we have to consider two time steps $t \in \{0, 1\}$. This gives the following constraint integer program, assuming that the widths of the registers are all equal to β:

$$\begin{array}{lllllllll}
\min & 0 & & & & & & & \\
\text{s.t.} & s_0 & = & \text{ADD}(x_0, y_{-1}) & s'_0 & = & \text{ADD}(x'_0, y'_{-1}) & init_0 & = & 1 \\
& y_0 & = & \text{ITE}(init_0, x_0, s_0) & y'_0 & = & \text{ITE}(init'_0, x'_0, s'_0) & init'_0 & = & 1 \\
& s_1 & = & \text{ADD}(x_1, y_0) & s'_1 & = & \text{ADD}(x'_1, y'_0) & x_0 & = & x'_1 \\
& y_1 & = & \text{ITE}(init_1, x_1, s_1) & y'_1 & = & \text{ITE}(init'_1, x'_1, s'_1) & x_1 & = & x'_0 \\
& & & & & & & y_1 & \neq & y'_1
\end{array} \tag{13.6}$$

$$\begin{array}{rcl}
x_0, x_1, y_{-1}, y_0, y_1, s_0, s_1 & \in & \{0, \ldots, 2^\beta - 1\} \\
init_0, init_1 & \in & \{0, 1\} \\
x'_0, x'_1, y'_{-1}, y'_0, y'_1, s'_0, s'_1 & \in & \{0, \ldots, 2^\beta - 1\} \\
init'_0, init'_1 & \in & \{0, 1\}
\end{array}$$

The left block of equations corresponds to the circuit operations, the middle block models the copy of the circuit, and the right block represents the negation of the property using $\neg(a \rightarrow b) \Leftrightarrow a \wedge \neg b$. Variables y_{-1} and y'_{-1} denote the value of the register in the time step prior to $t = 0$.

CIP (13.6) is feasible, for example with the input $init_0 = init'_0 = init_1 = init'_1 = 1$, $x_0 = x'_1 = 0$, and $x_1 = x'_0 = 1$, and the initial internal state $y_{-1} = y'_{-1} = 0$, which yields the values $s_0 = y_0 = 0$, $s_1 = y_1 = 1$, $s'_0 = y'_0 = s'_1 = 1$, and $y'_1 = 0$ for the constrained variables. This proves that the circuit violates Property (13.5). However, this does not necessarily mean that the circuit is flawed. In our case, the source of the error is the incorrect model of the property. To verify the commutativity law, we do not only have to fix $init_0 = init'_0 = 1$, but we also have to ensure that the *init* register is 0 in the following time step. This yields the revised property

$$init_0 = init'_0 = 1 \ \wedge \ init_1 = init'_1 = 0 \ \wedge \ x_0 = x'_1 \ \wedge \ x_1 = x'_0 \quad \rightarrow \quad y_1 = y'_1, \quad (13.7)$$

which introduces the two additional fixings $init_1 = 0$ and $init'_1 = 0$ to System (13.6). The modified CIP is infeasible, which proves the validity of Property (13.7).

13.2 FUNCTION GRAPH

The circuit operations link up to three input registers x, y, and z to an output register r. The output is uniquely defined by the input, since the circuit operators are well-defined mappings. In every circuit, each register can be the output of at most one operation. Registers that are not output of any operation are the input registers of the circuit, while the ones that are constrained by a circuit operation are internal or output registers. Therefore, the circuit as a whole is a well-defined mapping of input registers to internal and output registers.

This mapping can be represented as a *function graph*, which is a directed bipartite graph $G = (V_\varrho \cup V_\mathfrak{C}, A)$ with two different types of nodes, namely register nodes $V_\varrho = \{\varrho_1, \ldots, \varrho_n\}$ and constraint nodes $V_\mathfrak{C} = \{\mathcal{C}_1, \ldots, \mathcal{C}_m\}$. The arc set is defined as

$$\begin{aligned} A = \{&(\varrho_j, \mathcal{C}_i) \mid \text{register } \varrho_j \text{ is input of circuit operation } \mathcal{C}_i\} \\ \cup \{&(\mathcal{C}_i, \varrho_j) \mid \text{register } \varrho_j \text{ is output of circuit operation } \mathcal{C}_i\}. \end{aligned}$$

As the graph represents a chip design and therefore a well-defined mapping of inputs to outputs, it does not contain directed cycles.

If the input registers are fixed to certain values, the values of the internal and output registers can easily be calculated by forward propagation of the registers through the constraints, just like the chip would do in hardware. The property, however, may restrict the domains of internal and output registers, add other constraints, and can leave input registers unspecified. Therefore, an assignment of values to the unspecified input registers may result in conflicting assignments for the internal and output variables.

Example 13.3. Figure 13.2 shows the function graph for the adder of Example 13.2 including the invalid negated Property (13.5), which corresponds to the CIP (13.6). The left hand side part represents the time-expanded original circuit $\mathcal{A}$, while the right hand side part represents the copy $\mathcal{A}'$. The property is included by fixing the input $init_0$ in both circuits to $init_0 = init'_0 = 1$ and adding the constraints $1 = \text{EQ}(x_0, x'_1)$, $1 = \text{EQ}(x_1, x'_0)$, and $0 = \text{EQ}(y_1, y'_1)$.

The function graph contains the complete structure of the circuit and the property. We use this concept in the *irrelevance detection* described in Section 15.2 to

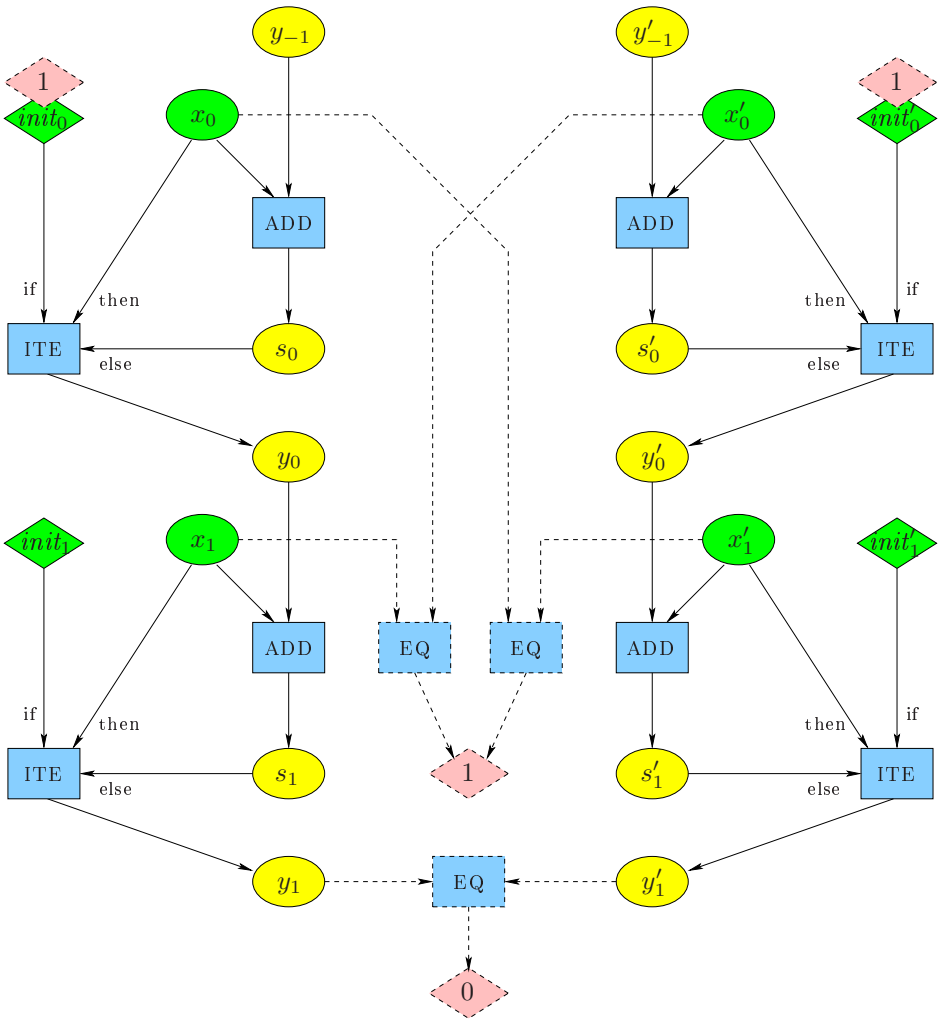

Figure 13.2. Function graph for adder of Example 13.2 expanded over two time steps $t = 0, 1$, including the invalid negated Property (13.5) denoted by dashed lines.

identify subgraphs with a single output register, which is neither an input of another operation nor constrained by the property restrictions. These subgraphs can be eliminated from the model, since the values of the involved registers can easily be calculated with forward propagation after a feasible solution for the remaining part of the circuit has been found.

Chapter 14

Operators in Detail

We solve the property checking CIP (13.4) with our branch-and-bound based constraint integer programming framework SCIP. The problem instance is successively divided into subproblems by splitting the domains of the register variables ϱ into two disjunctive parts: either by fixing a certain bit of a register to $\varrho_{jb} = 0$ and $\varrho_{jb} = 1$, respectively, or by introducing local upper bounds $\varrho_j \leq v$ and local lower bounds $\varrho_j \geq v + 1$ on the individual registers.

At each node in the resulting search tree, we apply methods of constraint programming (Section 1.1), SAT solving (Section 1.2), and integer programming (Section 1.3) to tighten the local subproblem and to prune the search tree. This includes domain propagation (Section 2.3), solving an LP relaxation with the optional possibility to add cutting planes (Section 2.2), and applying conflict analysis (Chapter 11) on infeasible subproblems.

The most important task in creating a SCIP application to solve a specific constraint integer programming model is to implement the *constraint handlers* (see Section 3.1.1) for the individual constraint classes that can appear in the model. In our case of property checking, these are the circuit operator constraints of Table 13.1 and the bit linking constraints (13.3). Each constraint handler defines the semantics, constructs the LP relaxation, and provides domain propagation and reverse propagation algorithms for the constraint class for which it is responsible. Additionally, presolving algorithms specialized to the individual constraint classes reduce the complexity of the problem instance and detect inherent relations, which can be exploited during the solving process.

In the following, we will take a detailed look at the different circuit operators and describe the algorithms dealing with the corresponding constraints. For each operator, we present the linear equations and inequalities that are used to model the operator within the linear programming relaxation, and we describe the domain propagation and presolving algorithms. The presentation is rather detailed and technical. It tries to convey the main algorithmic ideas to developers of chip design verification tools. For those who are not interested in the very details of the algorithms, it suffices to read the introduction paragraphs of the individual circuit operator sections in which we explain the main issues and difficulties associated with the respective constraint class.

Table 14.1 gives a summary of the linear relaxations used for the circuit operators. Very large coefficients like 2^{β_r} in the ADD linearization or in the bit linking constraints (13.3) can lead to numerical difficulties in the LP relaxation. Therefore, we split the register variables into words of $W = 16$ bits and apply the linearization to the individual words. The linkage between the words is established in a proper fashion. For example, the overflow bit of a word in an addition is added to the right hand side of the next word's linearization (see Section 14.3.1). The partitioning of registers into words and bits is explained in detail in Section 14.1. The relaxation of the MULT constraint involves additional variables $y^{\langle l \rangle}$ and $r^{\langle l \rangle}$ which are "nibbles"

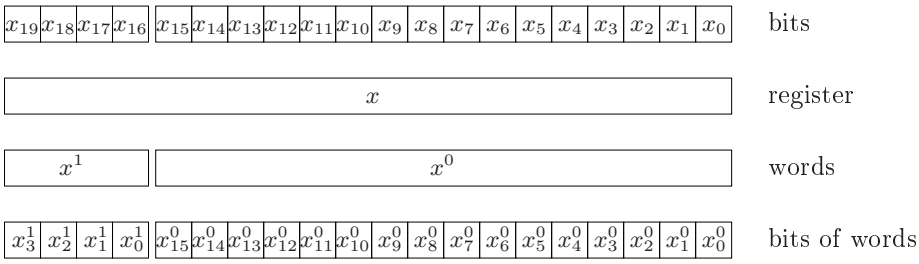

Figure 14.1. Partitioning of a register $x = \varrho_j$ into words and bits.

of y and r with $L \leq \frac{W}{2}$ bits. An elaborate presentation of the MULT linearization can be found in Section 14.5.

14.1 Bit and Word Partitioning

In order to support constraints that operate on bit level, we partition the registers ϱ_j into bits ϱ_{jb}, $b = 0, \ldots, \beta_j - 1$, with $\beta_j \in \mathbb{N}$ being the width of the register. In our property checking algorithm we employ "double modeling": both the integer valued register variables ϱ_j and the binary bit variables ϱ_{jb} are included in the model, such that the different components of the solver can access the registers on bit or word level as needed.

A register variable and its associated bit variables are linked by constraint (13.3). However, as already discussed above, large bit widths β_j would lead to huge coefficients in these constraints and also in the linearizations of some circuit operators. Therefore, the registers ϱ_j are split into $\omega_j = \lceil \beta_j / W \rceil$ words ϱ_j^w, $w = 0, \ldots, \omega_j - 1$, of $W := 16$ bits, and the bit linking constraints are defined on word level:

$$\varrho_j^w = \sum_{b=0}^{\gamma_j^w - 1} 2^b \varrho_{jb}^w. \tag{14.1}$$

Here, $\gamma_j^w = \min\{W, \beta_j - wW\}$ is the width of word w of register ϱ_j, and ϱ_{jb}^w is the b'th bit in word w of register ϱ_j, i.e.,

$$\varrho_{jb}^w = \varrho_{j,wW+b}.$$

For ease of notation, we define $\varrho_{jb} = 0$ for $b \geq \beta_j$, $\varrho_j^w = 0$ for $w \geq \omega_j$, and $\varrho_{jb}^w = 0$ for $w \geq \omega_j$ or $b \geq \gamma_j^w$. Figure 14.1 illustrates the partitioning of a register $x = \varrho_j$ into bits x_b and words x^w, and the partitioning of the individual words into bits x_b^w.

In the following, we also need to access subwords of a given register or other non-negative integer values or variables:

Definition 14.1 (subword). Given a non-negative integer $x \in \mathbb{Z}_{\geq 0}$ with binary representation $x = \sum_{b=0}^{\infty} 2^b x_b$, $x_b \in \{0, 1\}$ for all b, define

$$x[q, p] := \sum_{b=p}^{q} 2^{b-p} x_b$$

Operation	Linearization	
Arithmetic Operations:		
$r = \text{MINUS}(x)$	replaced by $0 = \text{ADD}(x, r)$	
$r = \text{ADD}(x, y)$	$r + 2^{\beta_r} o = x + y,$	$o \in \{0,1\}$
$r = \text{SUB}(x, y)$	replaced by $x = \text{ADD}(y, r)$	
$r = \text{MULT}(x, y)$	$p_b^{\langle l \rangle} \le u_{y^{\langle l \rangle}} x_b,\ p_b^{\langle l \rangle} \le y^{\langle l \rangle},$	$p_b^{\langle l \rangle} \in \mathbb{Z}_{\ge 0},$
	$p_b^{\langle l \rangle} \ge y^{\langle l \rangle} - u_{y^{\langle l \rangle}} (1 - x_b),$	
	$o^{\langle l \rangle} + \sum_{i+j=l} \sum_{b=0}^{L-1} 2^b p_{iL+b}^{\langle j \rangle} = 2^L o^{\langle l+1 \rangle} + r^{\langle l \rangle},$	$o^{\langle l \rangle} \in \mathbb{Z}_{\ge 0}$
Bit Operations:		
$r = \text{NOT}(x)$	removed in preprocessing	
$r = \text{AND}(x, y)$	$r_b \le x_b,\ r_b \le y_b,\ r_b \ge x_b + y_b - 1$	
$r = \text{OR}(x, y)$	$r_b \ge x_b,\ r_b \ge y_b,\ r_b \le x_b + y_b$	
$r = \text{XOR}(x, y)$	$x_b - y_b - r_b \le 0,\ -x_b + y_b - r_b \le 0,\ -x_b - y_b + r_b \le 0,\ x_b + y_b + r_b \le 2$	
Data → Control Interface:		
$r = \text{UAND}(x)$	$r \le x_b,\ r \ge \sum_{b=0}^{\beta_x - 1} x_b - \beta_x + 1$	
$r = \text{UOR}(x)$	$r \ge x_b,\ r \le \sum_{b=0}^{\beta_x - 1} x_b$	
$r = \text{UXOR}(x)$	$r + \sum_{b=0}^{\beta_x - 1} x_b = 2s,$	$s \in \mathbb{Z}_{\ge 0}$
$r = \text{EQ}(x, y)$	$x - y = s - t,$	$s, t \in \mathbb{Z}_{\ge 0},$
	$p \le s,\ s \le p(u_x - l_y),$	$p \in \{0,1\},$
	$q \le t,\ t \le q(u_y - l_x),$	$q \in \{0,1\},$
	$p + q + r = 1$	
$r = \text{LT}(x, y)$	$x - y = s - t,$	$s, t \in \mathbb{Z}_{\ge 0},$
	$p \le s,\ s \le p(u_x - l_y),$	$p \in \{0,1\},$
	$r \le t,\ t \le r(u_y - l_x),$	
	$p + r \le 1$	
Control → Data Interface:		
$r = \text{ITE}(x, y, z)$	$r - y \le (u_z - l_y)(1 - x),\ r - y \ge (l_z - u_y)(1 - x),$	
	$r - z \le (u_y - l_z)\, x,\ r - z \ge (l_y - u_z)\, x$	
Word Extension:		
$r = \text{ZEROEXT}(x)$ $r = \text{SIGNEXT}(x)$ $r = \text{CONCAT}(x, y)$	removed in preprocessing	
Subword Access:		
$r = \text{SHL}(x, y)$ $r = \text{SHR}(x, y)$ $r = \text{SLICE}(x, y)$	no linearization	
$r = \text{READ}(x, y)$	$\sum_{p=l_y}^{u_y} p \cdot \psi^p = y,\ \sum_{p=l_y}^{u_y} \psi^p = 1,$	$\psi^p \in \{0,1\},$
	$r_b - x_{b+p\cdot\beta_r} \le 1 - \psi^p,\ -r_b + x_{b+p\cdot\beta_r} \le 1 - \psi^p$	
$r = \text{WRITE}(x, y, z)$	$\sum_{p=l_y}^{u_y} p \cdot \psi^p = y,\ \sum_{p=l_y}^{u_y} \psi^p = 1,$	$\psi^p \in \{0,1\},$
	$r_{b+p\cdot\beta_z} - z_b \le 1 - \psi^p,\ -r_{b+p\cdot\beta z} + z_b \le 1 - \psi^p,$	
	$r_{b+p\cdot\beta_z} - x_{b+p\cdot\beta_z} \le \psi^p,\ -r_{b+p\cdot\beta_z} + x_{b+p\cdot\beta_z} \le \psi^p$	

Table 14.1. LP relaxation of the circuit operations. l_ϱ and u_ϱ denote the lower and upper bounds of a register variable ϱ.

Algorithm 14.1 Bit and Word Partitioning Domain Propagation

Input: Word $x = \varrho_j^w$ of width $\gamma = \gamma_j^w$ with current local bounds $[\tilde{l}_x, \tilde{u}_x]$, and contained bits $x_b = \varrho_{jb}^w$ with current local bounds $[\tilde{l}_{x_b}, \tilde{u}_{x_b}]$, $b = 0, \ldots, \gamma - 1$.

Output: Tightened local bounds $[\tilde{l}_x, \tilde{u}_x]$ and $[\tilde{l}_{x_b}, \tilde{u}_{x_b}]$.

1. Let $\tilde{l}_{\text{bits}} = \sum_{b=0}^{\gamma-1} 2^b \tilde{l}_{x_b}$ and $\tilde{u}_{\text{bits}} = \sum_{b=0}^{\gamma-1} 2^b \tilde{u}_{x_b}$ be the word's local bounds calculated from the bit fixings.
2. For $b = \gamma - 1, \ldots, 0$:
   (a) If $\tilde{l}_{\text{bits}}[b, 0] > \tilde{l}_x[b, 0]$, update $\tilde{l}_x[b, 0] := \tilde{l}_{\text{bits}}[b, 0]$.
   (b) If $\tilde{u}_{\text{bits}}[b, 0] < \tilde{l}_x[b, 0]$, update $\tilde{l}_x[b, 0] := \tilde{l}_{\text{bits}}[b, 0]$ and $\tilde{l}_x := \tilde{l}_x + 2^{b+1}$.
   (c) If $\tilde{u}_{\text{bits}}[b, 0] < \tilde{u}_x[b, 0]$, update $\tilde{u}_x[b, 0] := \tilde{u}_{\text{bits}}[b, 0]$.
   (d) If $\tilde{l}_{\text{bits}}[b, 0] > \tilde{u}_x[b, 0]$, update $\tilde{u}_x[b, 0] := \tilde{u}_{\text{bits}}[b, 0]$ and $\tilde{u}_x := \tilde{u}_x - 2^{b+1}$.
3. For $b = \gamma - 1, \ldots, 0$:
   (a) If $\tilde{u}_{x_b} = 1$ and $\tilde{u}_{\text{bits}} - 2^b < \tilde{l}_x$, fix $\tilde{l}_{x_b} := 1$.
   (b) If $\tilde{l}_{x_b} = 0$ and $\tilde{l}_{\text{bits}} + 2^b > \tilde{u}_x$, fix $\tilde{u}_{x_b} := 0$.
   (c) If still $\tilde{l}_{x_b} = 0$ and $\tilde{u}_{x_b} = 1$, stop.

to be the *subword* of x ranging from bits p to q, $p \leq q$. The *subword assignment* of a value $y \in \mathbb{Z}_{\geq 0}$, $y < 2^{p-q+1}$, to a subword $x[q, p]$ is defined as

$$x\langle [q, p] \leftarrow y \rangle := 2^{q+1} x[\infty, q+1] + 2^p y + x[p-1, 0].$$

If a subword is replaced by a different value in a procedural environment, we write

$$x[p, q] := y$$

as a short cut, which means that the new value of x is equal to $x\langle [q, p] \leftarrow y \rangle$. To access single bits we define $x[p] := x[p, p]$ and $x\langle [p] \leftarrow y \rangle := x\langle [p, p] \leftarrow y \rangle$.

14.1.1 LP Relaxation

The LP relaxation of the bit/word partitioning constraints is directly given by equation (14.1). These equations are included in the initial LP relaxation of the property checking CIP for all unfixed words ϱ_j^w, i.e., for all words with a global lower bound $l_{\varrho_j^w}$ smaller than the global upper bound $u_{\varrho_j^w}$.

14.1.2 Domain Propagation

Domain propagation is applied individually on each bit/word linking constraint (14.1). For each word $x = \varrho_j^w$, the current local bounds $[\tilde{l}_x, \tilde{u}_x]$ can lead to local fixings of the bits $x_b = \varrho_{jb}^w$, $b = 0, \ldots, \gamma_j^w - 1$, and vice versa.

Algorithm 14.1 shows the domain propagation procedure for a single word $x = \varrho_j^w$. Step 2 uses the local fixings of the bit variables to tighten the word's local bounds. The following lemma proves the validity of updates 2a and 2c:

Lemma 14.2. Let $x \in \mathbb{Z}_{\geq 0}$ be a non-negative integer with bit decomposition $x =$

$\sum_{b=0}^{\infty} 2^b x_b$. If $\tilde{l}_x \leq x \leq \tilde{u}_x$ and $\tilde{l}_{x_b} \leq x_b \leq \tilde{u}_{x_b}$ for all b, then

$$x \geq \tilde{l}_x\big\langle [b,0] \leftarrow \tilde{l}_{\text{bits}}[b,0]\big\rangle \quad \text{and} \quad x \leq \tilde{u}_x\big\langle [b,0] \leftarrow \tilde{u}_{\text{bits}}[b,0]\big\rangle \quad \text{for all } b \in \mathbb{Z}_{\geq 0}$$

with $\tilde{l}_{\text{bits}} = \sum_{b=0}^{\infty} 2^b \tilde{l}_{x_b}$ and $\tilde{u}_{\text{bits}} = \sum_{b=0}^{\infty} 2^b \tilde{u}_{x_b}$ being the integer values composed of the bounds for the bits of x.

Proof. We prove the case for the lower bound. Due to $x \geq \tilde{l}_x$ it follows $x[\infty, b+1] \geq \tilde{l}_x[\infty, b+1]$. This leads to

$$\begin{aligned} x &= 2^b x[\infty, b+1] + x[b,0] \\ &\geq 2^b \tilde{l}_x[\infty, b+1] + \tilde{l}_{\text{bits}}[b,0] \\ &= \tilde{l}_x\big\langle [b,0] \leftarrow \tilde{l}_{\text{bits}}[b,0]\big\rangle. \end{aligned}$$

The proof for the upper bound is analogous. □

The reasoning in updates 2b and 2d is slightly different since they also include the opposite bound:

Lemma 14.3. Let $x \in \mathbb{Z}_{\geq 0}$ be a non-negative integer with bit decomposition $x = \sum_{b=0}^{\infty} 2^b x_b$. If $\tilde{l}_x \leq x \leq \tilde{u}_x$ and $\tilde{l}_{x_b} \leq x_b \leq \tilde{u}_{x_b}$ for all b, then

$$\tilde{u}_{\text{bits}}[b,0] < \tilde{l}_x[b,0] \quad \Rightarrow \quad x \geq \tilde{l}_x\big\langle [b,0] \leftarrow \tilde{l}_{\text{bits}}[b,0]\big\rangle + 2^{b+1}$$

and

$$\tilde{l}_{\text{bits}}[b,0] > \tilde{u}_x[b,0] \quad \Rightarrow \quad x \leq \tilde{u}_x\big\langle [b,0] \leftarrow \tilde{u}_{\text{bits}}[b,0]\big\rangle - 2^{b+1}$$

for all $b \in \mathbb{Z}_{\geq 0}$.

Proof. We prove the first implication. Let $\tilde{u}_{\text{bits}}[b,0] < \tilde{l}_x[b,0]$ and assume $x[\infty, b+1] = \tilde{l}_x[\infty, b+1]$. From the bounds of the individual bits we know that $x[b,0] \leq \tilde{u}_{\text{bits}}[b,0]$. Hence, $x[b,0] < \tilde{l}_x[b,0]$ which means $x < \tilde{l}_x$, a contradiction. Therefore, $x[\infty, b+1] > \tilde{l}_x[\infty, b+1]$, which means

$$x \geq 2^{b+1}\big(\tilde{l}_x[\infty, b+1] + 1\big) = \tilde{l}_x\big\langle [b,0] \leftarrow 0\big\rangle + 2^{b+1}.$$

Lemma 14.2 yields

$$x \geq \big(\tilde{l}_x\big\langle [b,0] \leftarrow 0\big\rangle + 2^{b+1}\big)\big\langle [b,0] \leftarrow \tilde{l}_{\text{bits}}[b,0]\big\rangle$$

which is equivalent to

$$x \geq \tilde{l}_x\big\langle [b,0] \leftarrow \tilde{l}_{\text{bits}}[b,0]\big\rangle + 2^{b+1}$$

since the addition of 2^{b+1} does not influence the lower significant bits 0 to b. The proof of the second implication is analogous. □

The following lemma shows that changes made in iteration $b = b'$ of Step 2 do not enable further updates on the same or more significant bits $b \geq b'$. Therefore, the loop of Step 2 only needs to be executed once.

Lemma 14.4. After Step 2 of Algorithm 14.1 was performed for all bits $b > b'$, subsequent bound changes deduced in Step 2 for bit b' cannot enable additional deductions of the types of Step 2 for bits $b \geq b'$.

Proof. Let b' be the first (most significant) bit for which a deduction in Step 2 is performed, and let $\tilde{l}_x \leq x \leq \tilde{u}_x$ and $\tilde{l}_{x_b} \leq x_b \leq \tilde{u}_{x_b}$, $b = 0, \ldots, \gamma - 1$, be the bounds before iteration $b = b'$ of Step 2 is executed. Then, we have $\tilde{l}_{\text{bits}}[b, 0] \leq \tilde{l}_x[b, 0] \leq \tilde{u}_{\text{bits}}[b, 0]$ and $\tilde{l}_{\text{bits}}[b, 0] \leq \tilde{u}_x[b, 0] \leq \tilde{u}_{\text{bits}}[b, 0]$ for all $b > b'$, because no deductions were applied for $b > b'$. It follows that also for the subwords up to bit $b' + 1$ we have

$$\tilde{l}_{\text{bits}}[b, b' + 1] \leq \tilde{l}_x[b, b' + 1] \leq \tilde{u}_{\text{bits}}[b, b' + 1] \quad \text{for all } b > b' \tag{14.2}$$

and

$$\tilde{l}_{\text{bits}}[b, b' + 1] \leq \tilde{u}_x[b, b' + 1] \leq \tilde{u}_{\text{bits}}[b, b' + 1] \quad \text{for all } b > b'.$$

A deduction for $b = b'$ in Step 2a fixes $\tilde{l}_x[b', 0] := \tilde{l}_{\text{bits}}[b', 0]$. This update has no influence on deductions of type 2c and 2d, and due to (14.2), it also does not enable any further deductions of type 2a for bits $b \geq b'$. If no infeasibility is detected, after the update we have $\tilde{l}_x[b, 0] = \tilde{l}_{\text{bits}}[b, 0] \leq \tilde{u}_{\text{bits}}[b, 0]$ for all $b \leq b'$, and again due to (14.2) no further deductions of type 2b are possible for $b \geq b'$.

Now suppose that a deduction for $b = b'$ was applied in Step 2b. Again, the updates on $\tilde{l}_x$ do not influence Steps 2c and 2d. If Step 2b can be applied at the most significant bit $b' = \gamma - 1$, the constraint is proven to be infeasible. Therefore, it suffices to look at the case $b' < \gamma - 1$.

At first, assume $\tilde{u}_{\text{bits}}[b' + 1] \leq \tilde{l}_x[b' + 1]$. This implies that $\tilde{u}_{\text{bits}}[b' + 1, 0] < \tilde{l}_x[b' + 1, 0]$ would already have been satisfied in the previous iteration and Step 2b would have been applied for bit $b = b' + 1$. This contradicts the definition of b' being the first iteration for which a deduction is performed. Therefore, the assumption is wrong and instead $\tilde{u}_{\text{bits}}[b' + 1] > \tilde{l}_x[b' + 1]$ must hold which gives $\tilde{u}_{\text{bits}}[b' + 1] = 1$ and $\tilde{l}_x[b' + 1] = 0$.

It follows that even after adding $2^{b'+1}$ to $\tilde{l}_x$, inequality (14.2) stays valid. Like before, this shows that also the update of Step 2b cannot enable subsequent deductions for $b \geq b'$. With analogous reasoning it can be shown that also the updates of Steps 2c and 2d do not trigger additional deductions for $b \geq b'$. The case that b' is not the first iteration for which an update was applied follows by induction. □

Corollary 14.5. After Step 2 of Algorithm 14.1 was performed completely, a second run of Step 2 would not produce any additional deductions. In particular, the bounds satisfy $\tilde{l}_{\text{bits}}[b, 0] \leq \tilde{l}_x[b, 0] \leq \tilde{u}_{\text{bits}}[b, 0]$ and $\tilde{l}_{\text{bits}}[b, 0] \leq \tilde{u}_x[b, 0] \leq \tilde{u}_{\text{bits}}[b, 0]$ for all $b = 0, \ldots, \gamma - 1$.

In Step 3 of Algorithm 14.1 we check whether a fixing of a bit to a certain value would violate the word's bounds, and in this case fix the bit to the opposite value. If a bit remains unfixed, no more less significant bits can be fixed with the reasoning of Steps 3a and 3b:

Lemma 14.6. If an unfixed bit x_b was not fixed in Steps 3a or 3b of Algorithm 14.1, the conditions $\tilde{u}_{\text{bits}} - 2^{b'} < \tilde{l}_x$ and $\tilde{l}_{\text{bits}} + 2^{b'} > \tilde{u}_x$ do not hold for all less significant bits $x_{b'}$, $b' < b$.

Proof. First, suppose b' is the most significant unfixed bit smaller than b. Since x_b was not fixed in Loop 3, we have $\tilde{l}_x \leq \tilde{u}_{\text{bits}} - 2^b$ and $\tilde{u}_x \geq \tilde{l}_{\text{bits}} + 2^b$. Because no new fixings have been found since bit x_b was processed, $\tilde{l}_x$, $\tilde{u}_x$, $\tilde{l}_{\text{bits}}$, and $\tilde{u}_{\text{bits}}$ were not changed at the time $x_{b'}$ is considered. It follows

$$\tilde{u}_{\text{bits}} - 2^{b'} > \tilde{u}_{\text{bits}} - 2^b \geq \tilde{l}_x \quad \text{and} \quad \tilde{l}_{\text{bits}} + 2^{b'} < \tilde{l}_{\text{bits}} + 2^b \leq \tilde{u}_x.$$

The case of an arbitrary position $b' < b$ follows by induction. □

Therefore, Loop 3 is executed from most significant to least significant bit and aborted in Step 3c if a variable remained unfixed. We now show that Step 2 does not need to be executed again, even if fixings were applied in Step 3:

Lemma 14.7. After Step 2 of Algorithm 14.1 was performed, subsequent bound changes deduced in Step 3 cannot enable additional deductions of the types of Step 2.

Proof. Let $\tilde{l}_x \leq x \leq \tilde{u}_x$, and $\tilde{l}_{x_b} \leq x_b \leq \tilde{u}_{x_b}$, $b = 0, \ldots, \gamma - 1$, be the bounds after Step 2 was executed. Let b' be the first (most significant) bit for which a bound change was applied in Step 3. Due to Lemma 14.6 we know that all higher order bits $b > b'$ were already fixed, i.e., $\tilde{l}_{x_b} = \tilde{u}_{x_b}$ for all $b > b'$, which is equivalent to $\tilde{l}_{\text{bits}}[\gamma-1, b'+1] = \tilde{u}_{\text{bits}}[\gamma-1, b'+1]$. This implies $\tilde{l}_x[\gamma-1, b'+1] = \tilde{u}_x[\gamma-1, b'+1] = \tilde{u}_{\text{bits}}[\gamma-1, b'+1]$ due to Steps 2a and 2c.

Suppose we performed Step 3a to increase $\tilde{l}_{x_{b'}} = 0$ to 1. Further assume that $\tilde{l}_x[b'] = 0$. From the preconditions of Step 3a we know that $\tilde{u}_{x_{b'}} = 1$ and $\tilde{u}_{\text{bits}} - 2^{b'} < \tilde{l}_x$. Since $\tilde{u}_{\text{bits}}[\gamma-1, b'+1] = \tilde{l}_x[\gamma-1, b'+1]$ and $(\tilde{u}_{\text{bits}} - 2^{b'})[b'] = \tilde{l}_x[b'] = 0$ it follows that $\tilde{u}_{\text{bits}}[b'-1, 0] < \tilde{l}_x[b'-1, 0]$, which is a contradiction to Step 2b. Therefore, we have $\tilde{l}_x[b'] = 1$. If no infeasibility was detected, this means that also $\tilde{u}_x[b'] = 1$, and the subwords upto bit b' of the bounds are identical after applying the fixing $\tilde{l}_{x_{b'}} := 1$. It follows that after the fixing, deductions in Step 2 can be applied at bits $b \geq b'$ if and only if they can be applied at bit $b = b' - 1$. Because all deductions on lesser significant bits were already performed in Step 2, the deduction of Step 3a did not enable an additional deduction in Step 2. The proof for the deduction of Step 3b is analogous. □

Corollary 14.8. If Algorithm 14.1 did not detect infeasibility by producing an empty domain, the final bounds after applying the algorithm satisfy

$$\tilde{l}_{\text{bits}}[b, 0] \leq \tilde{l}_x[b, 0] \leq \tilde{u}_{\text{bits}}[b, 0] \quad \text{and} \quad \tilde{l}_{\text{bits}}[b, 0] \leq \tilde{u}_x[b, 0] \leq \tilde{u}_{\text{bits}}[b, 0]$$

for all $b = 0, \ldots, \gamma - 1$. In particular, this yields

$$\tilde{l}_{x_b} = \tilde{l}_{\text{bits}}[b] \leq \tilde{l}_x[b] \leq \tilde{u}_{\text{bits}}[b] = \tilde{u}_{x_b} \quad \text{and} \quad \tilde{l}_{x_b} = \tilde{l}_{\text{bits}}[b] \leq \tilde{u}_x[b] \leq \tilde{u}_{\text{bits}}[b] = \tilde{u}_{x_b}$$

for all individual bits $b = 0, \ldots, \gamma - 1$.

Proof. This result follows from Corollary 14.5 and Lemma 14.7. □

The following proposition shows that Algorithm 14.1 already achieves interval consistency (see Definition 2.6):

Proposition 14.9. After applying Algorithm 14.1 on constraint (14.1), the constraint becomes interval consistent or at least one domain becomes empty.

Proof. Let $x = \varrho_j^w$ be word w of register ϱ_j and let γ be the width of x. Suppose Algorithm 14.1 did not detect infeasibility by reducing a domain to the empty set, and let $\tilde{l}_x \leq \tilde{u}_x$ and $\tilde{l}_{x_b} \leq \tilde{u}_{x_b}$, $b = 0, \ldots, \gamma - 1$, be the final lower and upper bounds of the word and bit variables after applying the algorithm. In order to show interval consistency we have to construct for each of the bounds a solution $x^\star = \sum_{b=0}^{\gamma-1} 2^b x_b^\star$ with $\tilde{l}_x \leq x^\star \leq \tilde{u}_x$ and $\tilde{l}_{x_b} \leq x_b^\star \leq \tilde{u}_{x_b}$, $b = 0, \ldots, \gamma - 1$, which satisfies constraint (14.1) and with the corresponding variable being equal to the considered bound.

Algorithm 14.2 Bit and Word Partitioning Presolving

1. For all active bit/word linking constraints $x = \sum_{b=0}^{\gamma-1} x_b$:
   (a) Replace aggregated bit and word variables by their representative counterparts.
   (b) If $\gamma = 1$, aggregate $x :\overset{\star}{=} x_0$ and delete the constraint.
   (c) Apply domain propagation Algorithm 14.1 on the global bounds.
   (d) Add implications $x_b = 0 \to x \leq 2^\gamma - 1 - 2^b$ and $x_b = 1 \to x \geq 2^b$ for all $b = 0, \ldots, \gamma - 1$ to the implication graph of SCIP.
2. For all pairs of different active bit/word linking constraints $x = \sum_{b=0}^{\gamma_x-1} x_b$, $y = \sum_{b=0}^{\gamma_y-1} y_b$, with $\gamma_x \geq \gamma_y$:
   (a) If $x_b \overset{\star}{=} y_b$ for all $b = 0, \ldots, \gamma_y - 1$ and $x_b = 0$ for all $b = \gamma_y, \ldots, \gamma_x - 1$, aggregate $x :\overset{\star}{=} y$ and delete the constraint on y.
   (b) If $x \overset{\star}{=} y$, aggregate $x_b :\overset{\star}{=} y_b$ for all $b = 0, \ldots, \gamma_y - 1$, fix $x_b := 0$ for $b = \gamma_y, \ldots, \gamma_x - 1$, and delete the constraint on y.

First, we consider the lower bound $\tilde{l}_x$ of the word variable x. The value $x^\star = \tilde{l}_x$ with bit decomposition $x^\star = \sum_{b=0}^{\gamma-1} x_b^\star$ satisfies constraint (14.1) by definition of the bit decomposition. Inequality $\tilde{l}_{x_b} \leq x_b^\star = \tilde{l}_x[b] \leq \tilde{u}_{x_b}$ follows from Corollary 14.8. The same reasoning can be applied to $x^\star = \tilde{u}_x$.

Now consider a particular bit b' with bounds $\tilde{l}_{x_{b'}} \leq x_{b'} \leq \tilde{u}_{x_{b'}}$. We construct a solution $x^\star$ with $x_{b'}^\star = \tilde{l}_{x_{b'}}$. If $\tilde{l}_{x_{b'}} = \tilde{l}_x[b']$, the word's lower bound $x^\star = \tilde{l}_x$ already has the desired properties, as shown above. From Corollary 14.8 we know that $\tilde{l}_{x_{b'}} \leq \tilde{l}_x[b']$, such that only the case $\tilde{l}_{x_{b'}} = 0$ and $\tilde{l}_x[b'] = 1$ remains. Because $\tilde{l}_x[b'] \leq \tilde{u}_{x_{b'}}$ we also know that $\tilde{u}_{x_{b'}} = 1$, which means that $x_{b'}$ is unfixed.

Let $\bar{b}$ be the most significant unfixed bit, i.e., $\tilde{l}_{\text{bits}}[\gamma-1, \bar{b}+1] = \tilde{u}_{\text{bits}}[\gamma-1, \bar{b}+1]$, $\tilde{l}_{x_{\bar{b}}} = 0$, and $\tilde{u}_{x_{\bar{b}}} = 1$. Again from Corollary 14.8, it follows that also the bits of the word's bounds are fixed to the same value $\tilde{l}_x[\gamma-1, \bar{b}+1] = \tilde{u}_x[\gamma-1, \bar{b}+1] = \tilde{l}_{\text{bits}}[\gamma-1, \bar{b}+1] = \tilde{u}_{\text{bits}}[\gamma-1, \bar{b}+1]$. Additionally, $\tilde{l}_x[\bar{b}] = 0$ and $\tilde{u}_x[\bar{b}] = 1$ because otherwise $\tilde{l}_{x_{\bar{b}}}$ could have been increased to 1 in Step 3a. This implies $\bar{b} > b'$ due to $\tilde{l}_x[b'] = 1$.

Now we choose $x_b^\star = \tilde{u}_{x_b}$ for $b \geq \bar{b}$ and $x_b^\star = \tilde{l}_{x_b}$ for $b < \bar{b}$. In particular, this yields $x_{b'}^\star = \tilde{l}_{x_{b'}}$, and the bounds on the bits $\tilde{l}_{x_b} \leq x_b^\star \leq \tilde{u}_{x_b}$, $b = 0, \ldots, \gamma - 1$, are satisfied. Because $\tilde{l}_x[\gamma-1, \bar{b}+1] = \tilde{u}_{\text{bits}}[\gamma-1, \bar{b}+1] = x^\star[\gamma-1, \bar{b}+1]$, $\tilde{l}_x[\bar{b}] = 0$, and $x^\star[\bar{b}] = x_b^\star = \tilde{u}_{x_b} = 1$, the lower bound $\tilde{l}_x \leq x^\star$ is also valid. Finally, we have $x^\star[\gamma-1, \bar{b}] = \tilde{u}_{\text{bits}}[\gamma-1, \bar{b}] = \tilde{u}_x[\gamma-1, \bar{b}]$ and $x^\star[\bar{b}-1, 0] = \tilde{l}_{\text{bits}}[\bar{b}-1, 0] \leq \tilde{u}_x[\bar{b}-1, 0]$ from Corollary 14.8, which proves $x^\star \leq \tilde{u}_x$.

A solution $x^\star$ with $x_{b'}^\star = \tilde{u}_{x_{b'}}$ for a given bit b' can be constructed in the same fashion. □

14.1.3 PRESOLVING

In each round of the presolving step of SCIP (see Chapter 3.2.5), Algorithm 14.2 is applied as presolving method of the bit and word partitioning constraint handler.

In Step 1a the word variable and each bit variable is replaced by its equivalent representative variable, i.e., a selected representative of the variable's equivalence

class, see Section 3.3.4. Thereby, the constraint is normalized and two equivalent constraints would contain exactly the same variables. This property is needed for a fast equivalence detection in Step 2.

If the word consists of only one bit, the word variable x is equivalent to the bit variable x_0 and they are aggregated in Step 1b. Thereby, the equivalence classes of x and x_0 are implicitly united, see again Section 3.3.4. Since after the aggregation, the constraint $x = x_0$ is always valid, we can delete the constraint from the problem formulation.

Step 1c calls Algorithm 14.1 as a subroutine to perform domain propagation on the global bounds. Afterwards, we enrich the implication graph of SCIP (see Section 3.3.5) with the deductions on the word variable if a single bit is fixed to a certain value.

In Step 2 all pairs of bit/word linking constraints are processed. If all bit variables (counting non-existing ones as being fixed to zero) of the two constraints are equivalent, the word variables can be aggregated since they are also equivalent. If on the other hand the word variables are equivalent, the individual pairs of bit variables can be aggregated, and the excessive bit variables of the wider word can be fixed to zero. In both cases, one of the constraints can be deleted from the problem formulation.

14.1.4 Equivalence of Registers

Within the framework of constraint integer programming, the bit/word linking constraints (14.1) are just ordinary constraints without any special meaning. In the chip verification application, these constraints are tightly connected to the abstract objects of *registers*. Although being *constraints* in the CIP context, the registers appear as *variables* in the definitions of the circuit operations.

Like ordinary CIP variables, registers can be equivalent to each other, thereby forming equivalence classes with each of them being represented by a unique representative, see Section 3.3.4.

Definition 14.10 (equivalence of registers). We call two registers ϱ_i and ϱ_j *equivalent* if the binary variables in their respective bit decomposition are pairwise equivalent:

$$\varrho_i \overset{\star}{=} \varrho_j \quad \Leftrightarrow \quad \forall b \in \mathbb{Z}_{\geq 0} : \varrho_{ib} \overset{\star}{=} \varrho_{jb} \tag{14.3}$$

with $\varrho_{ib} = 0$ if $b \geq \beta_i$ and $\varrho_{jb} = 0$ if $b \geq \beta_j$. In an algorithmic environment, we denote by

$$\varrho_i :\overset{\star}{=} \varrho_j \quad \Leftrightarrow \quad \forall b \in \mathbb{Z}_{\geq 0} : \varrho_{ib} :\overset{\star}{=} \varrho_{jb}$$

the *aggregation* of register ϱ_i to ϱ_j, which means to aggregate the individual pairs of binary variables in the bit decompositions of the registers.

For example, if we already know that $y = 0$, the addition constraint $r = \text{ADD}(x, y)$ would detect that r and x are equivalent, which means that the individual word and bit variables of $r = \varrho_i$ are equivalent to their respective counterparts of $x = \varrho_j$. Besides aggregating $x :\overset{\star}{=} r$, we could in principle also identify the two register nodes x and r in the function graph, see Section 13.2, and replace all occurrences of x by r in the circuit constraints of the problem instance. Nevertheless, the identification of register nodes in the function graph can destroy the structure of the graph. For example, if two output registers are identified, the two operations producing these

outputs receive an artificial connection through the unified output register that did not exist in the original function graph. Consequently, loose ends of the graph, which are identified by the irrelevance detection of Section 15.2, may get lost, and the performance of the presolving deteriorates. For this reason, we do not replace registers with the representative of their equivalence class in the circuit constraints and in the function graph. Instead, equivalence of registers is always tested by checking the individual bits for pairwise equivalence.

Just like detecting equivalences between registers it might happen that the global *inequality* of two registers x and y is discovered. For example, if the constraint $r = \text{EQ}(x, y)$ is included in the problem and we know already that $r = 0$, we can conclude that x and y can never take the same value. We denote this fact by writing $x \stackrel{\star}{\neq} y$:

Definition 14.11 (inequality of registers). We call two registers ϱ_i and ϱ_j *unequal* if for each feasible solution of the property checking CIP (13.4) there is at least one pair of binary variables in the bit decomposition of the registers with opposite values:

$$\varrho_i \stackrel{\star}{\neq} \varrho_j \quad \Leftrightarrow \quad \forall \varrho^\star : \mathfrak{C}(\varrho^\star) \wedge \neg P(\varrho^\star) \;\; \exists b \in \mathbb{Z}_{\geq 0} : \varrho^\star_{ib} \neq \varrho^\star_{jb} \tag{14.4}$$

with $\varrho^\star_{ib} = 0$ if $b \geq \beta_i$ and $\varrho^\star_{jb} = 0$ if $b \geq \beta_j$.

For each register x we store the set of registers y with $x \stackrel{\star}{\neq} y$ as a sorted list on which a binary search can be performed. Note that for two registers x and y it may neither $x \stackrel{\star}{=} y$ nor $x \stackrel{\star}{\neq} y$, since $\stackrel{\star}{\neq}$ is not the negated relation of $\stackrel{\star}{=}$. However, $x \stackrel{\star}{=} y \;\wedge\; x \stackrel{\star}{\neq} y$ proves the infeasibility of the problem instance.

14.2 Unary Minus

The unary minus operator

$$\text{MINUS} : [\beta] \to [\beta], \quad x \mapsto r = \text{MINUS}(x)$$

calculates the two's complement of input register x, which is defined as

$$r = \text{MINUS}(x) \quad \Leftrightarrow \quad r = 2^\beta - x.$$

This is equivalent to $r + x = 2^\beta$, and due to the truncation of the overflow in the ADD operand, the constraint can be represented as $0 = \text{ADD}(x, r)$. This transformation is applied in the presolving stage of SCIP, such that we do not need to implement any specific algorithms for the MINUS operator.

14.3 Addition

The two's complement addition of two numbers $x, y \in \{0, \ldots, 2^\beta - 1\}$ is defined as

$$\text{ADD} : [\beta] \times [\beta] \to [\beta], \quad (x, y) \mapsto r = \text{ADD}(x, y)$$

with

$$r = \text{ADD}(x, y) \quad \Leftrightarrow \quad r = (x + y) \bmod 2^\beta.$$

To overcome the numerical difficulties caused by very large coefficients, the LP relaxation of addition constraints is defined on word level. Each word addition produces an overflow that is passed to the next word's calculations. The final overflow is ignored, which models the modulus operation in the above definition of the ADD operator.

The domain propagation is applied on bit level and exploits knowledge about fixings and equivalences of the involved register bits. The word level and bit level representations interact by means of the bit/word linking constraints (14.1). Presolving operates on both representations simultaneously. In addition to the domain propagation, it exploits global knowledge of equality or inequality of registers to tighten the bounds of the bit and word variables. It also compares pairs of ADD constraints to detect further problem simplifications.

14.3.1 LP Relaxation

Since for the input registers we have $x, y \leq 2^\beta - 1$, we know that the sum $x + y$ is bounded by $x + y \leq 2^\beta$. Therefore, the sum can be completely described by $\beta + 1$ bits, and the addition operand ADD just means to strip the most significant bit, called *overflow bit*, from the result. The LP relaxation of the constraint could be stated as

$$x + y = r + 2^\beta o \quad \text{with } o \in \{0, 1\}, \tag{14.5}$$

but for large bit widths β this would lead to numerical difficulties in the floating point calculations. The default parameter value for the *feasibility tolerance* for floating point operations in SCIP is $\hat{\delta} = 10^{-6}$. This value is also used for solving the LP relaxations. Equality of floating point numbers in this sense is defined as

$$x \doteq y \quad \Leftrightarrow \quad \frac{|x - y|}{\max\{|x|, |y|, 1\}} \leq \hat{\delta}. \tag{14.6}$$

This means, for example, that for $\beta = 20$, $x = y = 2^{19}$, $r = 1$, and $o = 1$ we have $x + y = 2^{20} \neq 2^{20} + 1 = r + 2^{20}o$, but $x + y \doteq r + 2^{20}o$ because

$$\frac{|2^{20} - (2^{20} + 1)|}{\max\{|2^{20}|, |2^{20} + 1|, 1\}} = \frac{1}{2^{20} + 1} \approx 9.5 \cdot 10^{-7} < \hat{\delta} = 10^{-6}.$$

Therefore, the invalid solution $1 = \text{ADD}(2^{19}, 2^{19})$ would be accepted as feasible by the LP solver.

In order to avoid these numerical difficulties, we restrict the coefficients in the linear relaxation (14.5) to be not larger than 2^{16}, which can be achieved by splitting the registers into words of $W = 16$ bits, as described in Section 14.1. For each word $w = 0, \ldots, \omega - 1$, $\omega = \lceil \beta / W \rceil$, of width $\gamma^w = \min\{W, \beta - wW\}$ we add the equation

$$x^w + y^w + o^w = r^w + 2^{\gamma^w} o^{w+1} \quad \text{with } o^w, o^{w+1} \in \{0, 1\} \tag{14.7}$$

to the LP relaxation. Here, the overflow for a word is passed to the left hand side of the next word's equation. Note that the least significant overflow is fixed to $o^w = 0$. The system of equations (14.7) for all words w together with the bit/word linking constraints (14.1) is equivalent to equation (14.5) but rules out invalid integer solutions even if the floating point equality (14.6) with feasibility tolerance $\hat{\delta} = 10^{-6}$ is applied.

Figure 14.2. Bit level addition scheme.

14.3.2 DOMAIN PROPAGATION

Domain propagation for the addition operand is applied on bit level using the usual bit level addition scheme depicted in Figure 14.2. For each bit $b = 0, \dots, \beta - 1$ we propagate the equation

$$x_b + y_b + o_b = r_b + 2o_{b+1}. \tag{14.8}$$

Note that $o_0 = 0$, since there is no overflow passed to the least significant column. Observe also that the overflow o^w of a word w in the LP relaxation (14.7) is always equal to the bit overflow o_{wW} at the word boundary because wordwise addition and bitwise addition are equivalent, which can be seen by adding up equations (14.8) for a word $w \in \{0, \dots, \omega - 1\}$ of width γ^w:

$$\begin{array}{llll}
x_{wW+0} & + y_{wW+0} & + 2^0 o_{wW+0} & = r_{wW+0} + 2o_{wW+1} \quad \big| \cdot 2^0 \\
 & & \dots & \\
x_{wW+\gamma^w-1} & + y_{wW+\gamma^w-1} & + o_{wW+\gamma^w-1} & = r_{wW+\gamma^w-1} + 2o_{wW+\gamma^w} \quad \big| \cdot 2^{\gamma^w-1} \\
\hline
x^w & + y^w & + \sum_{b=0}^{\gamma^w-1} 2^b o_{wW+b} & = r^w + \sum_{b=1}^{\gamma^w} 2^b o_{wW+b}
\end{array}$$

which is equivalent to

$$x^w + y^w + o_{wW} = r^w + 2^{\gamma^w} o_{wW+\gamma^w}.$$

Subtracting equation (14.7) yields

$$o_{wW} - o^w = 2^{\gamma^w}(o_{wW+\gamma^w} - o^{w+1}),$$

and since the variables are binary and $\gamma^w \geq 1$, it follows $o^w = o_{wW}$ and $o^{w+1} = o_{wW+\gamma^w}$. Thus, we can aggregate

$$o^w :\overset{\star}{=} o_{wW} \quad \text{for all words } w = 0, \dots, \omega - 1, \tag{14.9}$$

and

$$o^\omega :\overset{\star}{=} o_\beta \tag{14.10}$$

for the most significant overflow.

Algorithm 14.3 illustrates the domain propagation method applied to ADD constraints. In fact, the deductions performed in Step 1 are just the usual bound tightening operations for the linear constraint (14.8), and the deductions of Step 2 are equal to the ones for the linear constraint (14.7), see Section 7.1.

Algorithm 14.3 Addition Domain Propagation

Input: Addition constraint $r = \text{ADD}(x, y)$ on registers r, x, and y of width β with current local word bounds $\tilde{l}_{r^w} \leq r^w \leq \tilde{u}_{r^w}$, $\tilde{l}_{x^w} \leq x^w \leq \tilde{u}_{x^w}$, and $\tilde{l}_{y^w} \leq y^w \leq \tilde{u}_{y^w}$, and bit bounds $\tilde{l}_{r_b} \leq r_b \leq \tilde{u}_{r_b}$, $\tilde{l}_{x_b} \leq x_b \leq \tilde{u}_{x_b}$, and $\tilde{l}_{y_b} \leq y_b \leq \tilde{u}_{y_b}$.

Output: Tightened local bounds for words r^w, x^w, y^w, and bits r_b, x_b, y_b.

1. For $b = 0, \ldots, \beta - 1$:
   (a) If three out of the four variables x_b, y_b, o_b, and r_b in the addition column b are fixed, the fourth variable and the overflow o_{b+1} can be deduced by inspecting equation (14.8).
   (b) If $\tilde{u}_{x_b} + \tilde{u}_{y_b} + \tilde{u}_{o_b} - \tilde{l}_{r_b} \leq 1$, deduce $o_{b+1} = 0$.
   (c) If $\tilde{l}_{x_b} + \tilde{l}_{y_b} + \tilde{l}_{o_b} - \tilde{u}_{r_b} \geq 1$, deduce $o_{b+1} = 1$.
   (d) If $\tilde{u}_{o_{b+1}} = 0$ and $\tilde{u}_{r_b} = 0$, deduce $x_b = y_b = o_b = 0$.
   (e) If $\tilde{l}_{o_{b+1}} = 1$ and $\tilde{l}_{r_b} = 1$, deduce $x_b = y_b = o_b = 1$.
   (f) If $\tilde{u}_{o_{b+1}} = 0$ and one out of the three variables x_b, y_b, o_b is fixed to one, we can fix $r_b = 1$ and the other two variables can be deduced to zero.
   (g) If $\tilde{l}_{o_{b+1}} = 1$ and one out of the three variables x_b, y_b, o_b is fixed to zero, we can fix $r_b = 0$ and the other two variables can be deduced to one.
2. For $w = 0, \ldots, \omega - 1$: If o^{w+1} is fixed, deduce

$$\begin{aligned}
\tilde{l}_{r^w} - \tilde{u}_{y^w} - \tilde{u}_{o^w} + 2^{\gamma^w} o^{w+1} &\leq x^w \leq \tilde{u}_{r^w} - \tilde{l}_{y^w} - \tilde{l}_{o^w} + 2^{\gamma^w} o^{w+1} \\
\tilde{l}_{r^w} - \tilde{u}_{x^w} - \tilde{u}_{o^w} + 2^{\gamma^w} o^{w+1} &\leq y^w \leq \tilde{u}_{r^w} - \tilde{l}_{x^w} - \tilde{l}_{o^w} + 2^{\gamma^w} o^{w+1} \\
\tilde{l}_{x^w} + \tilde{l}_{y^w} + \tilde{l}_{o^w} - 2^{\gamma^w} o^{w+1} &\leq r^w \leq \tilde{u}_{x^w} + \tilde{u}_{y^w} + \tilde{u}_{o^w} - 2^{\gamma^w} o^{w+1}
\end{aligned}$$

Because the word overflow variables o^w are aggregated with the bit overflow variables at the word boundaries, see (14.9) and (14.10), the propagations on the two equations (14.8) and (14.7) can interact with each other. Additionally, the propagations on the overflow bits of Step 1 can influence the previous and next column in the addition scheme of Figure 14.2. Furthermore, fixings of the bits and tightenings of the words' bounds can lead to even more implications in the propagation Algorithm 14.1 for the bit/word linking constraints (14.1). SCIP exploits these interdependencies automatically by iteratively calling the individual domain propagation algorithms as long as additional deductions were found, see Section 3.1.4. Therefore, we can refrain from iterating Steps 1 and 2 inside Algorithm 14.3.

14.3.3 Presolving

In the presolving stage of SCIP, Algorithm 14.4 is executed to process the active addition constraints. Step 1a checks whether one of the two operands x and y is fixed to zero, which means that the resultant can be aggregated with the other operand. If on the other hand, one of the operands is proven to be non-zero, the resultant cannot be equal to the other operand and the sets of unequal registers for the resultant and the other operand can be extended in Step 1b. Consequently, we can fix an operand to zero in Step 1c whenever the resultant is detected to be always equal to the other

Algorithm 14.4 Addition Presolving

1. For all active addition constraints $r = \text{ADD}(x, y)$:
   (a) If $x = 0$, aggregate $r \stackrel{\star}{:=} y$ and delete the constraint.
   If $y = 0$, aggregate $r \stackrel{\star}{:=} x$ and delete the constraint.
   (b) If $l_{x^w} \geq 1$ for any word w, deduce $r \stackrel{\star}{\neq} y$.
   If $l_{y^w} \geq 1$ for any word w, deduce $r \stackrel{\star}{\neq} x$.
   (c) If $r \stackrel{\star}{=} x$, fix $y := 0$ and delete the constraint.
   If $r \stackrel{\star}{=} y$, fix $x := 0$ and delete the constraint.
   (d) If $r \stackrel{\star}{\neq} x$ and $\omega_y = 1$, deduce $y^0 \geq 1$.
   If $r \stackrel{\star}{\neq} y$ and $\omega_x = 1$, deduce $x^0 \geq 1$.
   (e) If $x \stackrel{\star}{=} y$, replace the constraint by $r = \text{SHL}(x, 1)$.
   (f) If $\beta = 1$, replace the constraint by $r = \text{XOR}(x, y)$.
   (g) Apply domain propagation Algorithm 14.3 on the global bounds.
   (h) If two of the binary variables in equation (14.8) are equivalent or negated equivalent, the others can be aggregated as shown in Algorithm 14.5.
   (i) Add implications on binary variables using Algorithm 14.6.
2. For all pairs of active addition constraints $r = \text{ADD}(x, y)$ and $r' = \text{ADD}(x', y')$ with $\beta_r \geq \beta_{r'}$:
   (a) For all $b = 0, \ldots, \beta_{r'}$:
   If (χ_b, ψ_b, ϕ_b) and $(\chi'_b, \psi'_b, \phi'_b)$ are permutations of (x_b, y_b, o_b) and (x'_b, y'_b, o'_b) such that $\chi_b \stackrel{\star}{=} \chi'_b$ and $\psi_b \stackrel{\star}{=} \psi'_b$, and
      i. if $\phi_b \stackrel{\star}{=} \phi'_b$, aggregate $r_b \stackrel{\star}{:=} r'_b$ and $o_{b+1} \stackrel{\star}{:=} o'_{b+1}$,
      ii. if $r_b \stackrel{\star}{=} r'_b$, aggregate $\phi_b \stackrel{\star}{:=} \phi'_b$ and $o_{b+1} \stackrel{\star}{:=} o'_{b+1}$,
      iii. if $r_b \stackrel{\star}{\neq} r'_b$ or $o_{b+1} \stackrel{\star}{\neq} o'_{b+1}$, aggregate $\phi_b \stackrel{\star}{:=} 1 - \phi'_b$,
      iv. if $\phi_b \stackrel{\star}{\neq} \phi'_b$ or $o_{b+1} \stackrel{\star}{\neq} o'_{b+1}$, aggregate $r_b \stackrel{\star}{:=} 1 - r'_b$.
   (b) If Steps 2(a)i or 2(a)ii were successfully applied to all bits $b = 0, \ldots, \beta_{r'}$, delete constraint $r' = \text{ADD}(x', y')$.
   (c) If $\beta_r = \beta_{r'}$, $x \stackrel{\star}{=} x'$, but $r \stackrel{\star}{\neq} r'$, deduce $y \stackrel{\star}{\neq} y'$.
   If $\beta_r = \beta_{r'}$, $y \stackrel{\star}{=} y'$, but $r \stackrel{\star}{\neq} r'$, deduce $x \stackrel{\star}{\neq} x'$.
   (d) If $\beta_r = \beta_{r'}$, $x \stackrel{\star}{=} y'$, but $r \stackrel{\star}{\neq} r'$, deduce $y \stackrel{\star}{\neq} x'$.
   If $\beta_r = \beta_{r'}$, $y \stackrel{\star}{=} x'$, but $r \stackrel{\star}{\neq} r'$, deduce $x \stackrel{\star}{\neq} y'$.

operand. Step 1d deals with the backward implication of Step 1a. If we know that the resultant is always unequal to one of the operands, we can conclude that the other operand cannot be zero. However, it is impossible to express an inequality like $y \geq 1$ for a register y in terms of bounds of CIP variables, since registers represent a *collection* of bit and word variables. We can only conclude that at least one of the words and one of the bits must be non-zero. Therefore, we are only able to deduce a bound change, if the register consists of a single word, i.e., $\omega = 1$.

In the case that the two operands are always equal, we can replace the ADD constraint in Step 1e by an SHL constraint, because $\text{ADD}(x, x) = \text{MULT}(x, 2) = \text{SHL}(x, 1)$ for all $x \in \mathbb{Z}_{\geq 0}$. Of course, the presolving algorithm for the SHL constraint with the second operand fixed to one aggregates the bits to $r_0 := 0$ and $r_b \stackrel{\star}{:=} x_{b-1}$ for $b = 1, \ldots, \beta_r - 1$, see Step 1e of Algorithm 14.28.

Algorithm 14.5 Addition Presolving – Bit Aggregation

Considering equation (14.8), we can apply the following aggregations:

1. If $x_b \overset{\star}{=} y_b$ aggregate $o_b :\overset{\star}{=} r_b$ and $o_{b+1} :\overset{\star}{=} x_b$.
2. If $x_b \overset{\star}{\neq} y_b$ aggregate $o_b :\overset{\star}{=} 1 - r_b$ and $o_{b+1} :\overset{\star}{=} o_b$.
3. If $x_b \overset{\star}{=} o_b$ aggregate $y_b :\overset{\star}{=} r_b$ and $o_{b+1} :\overset{\star}{=} x_b$.
4. If $x_b \overset{\star}{\neq} o_b$ aggregate $y_b :\overset{\star}{=} 1 - r_b$ and $o_{b+1} :\overset{\star}{=} y_b$.
5. If $x_b \overset{\star}{=} r_b$ aggregate $o_b :\overset{\star}{=} y_b$ and $o_{b+1} :\overset{\star}{=} y_b$.
6. If $x_b \overset{\star}{\neq} r_b$ aggregate $o_b :\overset{\star}{=} 1 - y_b$ and $o_{b+1} :\overset{\star}{=} x_b$.
7. If $y_b \overset{\star}{=} o_b$ aggregate $x_b :\overset{\star}{=} r_b$ and $o_{b+1} :\overset{\star}{=} y_b$.
8. If $y_b \overset{\star}{\neq} o_b$ aggregate $x_b :\overset{\star}{=} 1 - r_b$ and $o_{b+1} :\overset{\star}{=} x_b$.
9. If $y_b \overset{\star}{=} r_b$ aggregate $o_b :\overset{\star}{=} x_b$ and $o_{b+1} :\overset{\star}{=} x_b$.
10. If $y_b \overset{\star}{\neq} r_b$ aggregate $o_b :\overset{\star}{=} 1 - x_b$ and $o_{b+1} :\overset{\star}{=} y_b$.
11. If $o_b \overset{\star}{=} r_b$ aggregate $x_b :\overset{\star}{=} y_b$ and $o_{b+1} :\overset{\star}{=} y_b$.
12. If $o_b \overset{\star}{\neq} r_b$ aggregate $x_b :\overset{\star}{=} 1 - y_b$ and $o_{b+1} :\overset{\star}{=} o_b$.
13. If $x_b \overset{\star}{\neq} o_{b+1}$ aggregate $y_b :\overset{\star}{=} o_{b+1}$, $o_b :\overset{\star}{=} o_{b+1}$, and $r_b :\overset{\star}{=} x_b$.
14. If $y_b \overset{\star}{\neq} o_{b+1}$ aggregate $x_b :\overset{\star}{=} o_{b+1}$, $o_b :\overset{\star}{=} o_{b+1}$, and $r_b :\overset{\star}{=} y_b$.
15. If $o_b \overset{\star}{\neq} o_{b+1}$ aggregate $x_b :\overset{\star}{=} o_{b+1}$, $y_b :\overset{\star}{=} o_{b+1}$, and $r_b :\overset{\star}{=} o_b$.
16. If $r_b \overset{\star}{=} o_{b+1}$ aggregate $x_b :\overset{\star}{=} r_b$, $y_b :\overset{\star}{=} r_b$, and $o_b :\overset{\star}{=} r_b$.

In the very special case that the registers are single bits, i.e., $\beta = 1$, truncated addition is equivalent to exclusive or, such that the constraint $r = \text{ADD}(x, y)$ can be replaced by $r = \text{XOR}(x, y)$ in Step 1f. The bitwise exclusive or constraint described in Section 14.9 comes with a stronger LP relaxation than equation (14.5) because the introduction of the auxiliary binary overflow variable o can be avoided.

It may happen that two of the bit variables in equation (14.8) are equivalent or negated equivalent. The latter means that the two bits always take opposite values. For each bit b, there are $\binom{5}{2} = 10$ different pairs out of the five variables involved in equation (14.8), which gives 20 combinations of potential equivalent or negated equivalent binary variables. We can draw useful conclusions from 16 of these relationships, as shown in Algorithm 14.5. For the remaining four, namely $x_b \overset{\star}{=} o_{b+1}$, $y_b \overset{\star}{=} o_{b+1}$, $o_b \overset{\star}{=} o_{b+1}$, and $r_b \overset{\star}{\neq} o_{b+1}$, we cannot deduce any further equivalence or negated equivalence relations between two variables. For example, if $x_b \overset{\star}{=} o_{b+1}$, there are still the six solutions

$$(x_b, y_b, o_b, r_b, o_{b+1}) \in \big\{(0,0,0,0,0), (0,1,0,1,0), (0,0,1,1,0), \\ (1,1,0,0,1), (1,0,1,0,1), (1,1,1,1,1)\big\}$$

possible, and none of $y_b \overset{\star}{=} o_b$, $y_b \overset{\star}{=} r_b$, $o_b \overset{\star}{=} r_b$, $y_b \overset{\star}{\neq} o_b$, $y_b \overset{\star}{\neq} r_b$, and $o_b \overset{\star}{\neq} r_b$ holds. Note that Algorithm 14.5 would also find the fixings and aggregations of Steps 1a and 1c of Algorithm 14.4. Thus, the only additional value of those steps is the deletion of the constraint.

The final task for each bit b in Loop 1 of Algorithm 14.4 is to add implications to the implication graph of SCIP, see Section 3.3.5. Algorithm 14.6 subsumes the

Algorithm 14.6 Addition Presolving – Implications

Considering equation (14.8), we can add the following implications to the implication graph of SCIP:

1. If $x_b = 0$, add $y_b = 0 \rightarrow o_{b+1} = 0$ and $o_b = 0 \rightarrow o_{b+1} = 0$.
2. If $x_b = 1$, add $y_b = 1 \rightarrow o_{b+1} = 1$ and $o_b = 1 \rightarrow o_{b+1} = 1$.
3. If $y_b = 0$, add $x_b = 0 \rightarrow o_{b+1} = 0$ and $o_b = 0 \rightarrow o_{b+1} = 0$.
4. If $y_b = 1$, add $x_b = 1 \rightarrow o_{b+1} = 1$ and $o_b = 1 \rightarrow o_{b+1} = 1$.
5. If $o_b = 0$, add $x_b = 0 \rightarrow o_{b+1} = 0$ and $y_b = 0 \rightarrow o_{b+1} = 0$.
6. If $o_b = 1$, add $x_b = 1 \rightarrow o_{b+1} = 1$ and $y_b = 1 \rightarrow o_{b+1} = 1$.
7. If $o_{b+1} = 0$, add $x_b = 1 \rightarrow y_b = 0$, $x_b = 1 \rightarrow o_b = 0$, and $y_b = 1 \rightarrow o_b = 0$.
8. If $o_{b+1} = 1$, add $x_b = 0 \rightarrow y_b = 1$ and $x_b = 0 \rightarrow o_b = 1$, and $y_b = 0 \rightarrow o_b = 1$.

implications that we can generate for each addition column, given that a certain bit in the column is already fixed.

Step 2 of Algorithm 14.4 compares all pairs of active addition constraints $r = \text{ADD}(x, y)$ and $r' = \text{ADD}(x', y')$. Suppose two of the variables in $\{x_b, y_b, o_b\}$ are pairwise equivalent to two of $\{x'_b, y'_b, o'_b\}$. We call the remaining variables ϕ_b and ϕ'_b, respectively. Then, subtracting the bit equations (14.8) from each other yields

$$\phi_b - \phi'_b = r_b - r'_b + 2(o_{b+1} - o'_{b+1}).$$

Since all involved variables are binary, we can deduce the aggregations of Steps 2(a)i to 2(a)iv. Note that 2(a)iii and 2(a)iv are just the inverse implications of 2(a)i and 2(a)ii. If the equivalence of the bit equations (14.8) is detected for all bits $b = 0, \ldots, \beta_{r'}$ by successful applications of Rules 2(a)i or 2(a)ii, the constraint on the smaller or equally wide registers can be deleted in Step 2b, since its semantics is already captured by the other constraint.

The trivial implications

$$\beta_r \geq \beta_{r'} \wedge (x \overset{\star}{=} x') \wedge (y \overset{\star}{=} y') \;\rightarrow\; r[\beta_{r'} - 1, 0] \overset{\star}{=} r' \quad \text{and}$$
$$\beta_r \geq \beta_{r'} \wedge (x \overset{\star}{=} y') \wedge (y \overset{\star}{=} x') \;\rightarrow\; r[\beta_{r'} - 1, 0] \overset{\star}{=} r'$$

are already covered by Step 2a. Steps 2c and 2d apply these implications in the opposite direction to deduce inequalities of operands. Note that we can only conclude that the operands are unequal if the resultants are of equal width. Otherwise, the inequality of the resultants may result from the different truncation of the sum. An alternative approach would be to check whether $r[\beta_{r'} - 1, 0] \overset{\star}{\neq} r'$, but we do not store inequalities between *subwords* of registers in our data structures.

14.4 SUBTRACTION

The subtraction operator

$$\text{SUB} : [\beta] \times [\beta] \rightarrow [\beta], \quad (x, y) \mapsto r = \text{SUB}(x, y)$$

yields the difference of the two input registers x and y in the two's complement representation. It is defined as

$$r = \text{SUB}(x, y) \quad \Leftrightarrow \quad r = (x - y) \bmod 2^\beta.$$

Because the integers modulo 2^β together with the truncated addition and multiplication form a ring, we have $x = (r + y) \bmod 2^\beta$. Therefore, we can replace each subtraction constraint by an equivalent addition constraint, which is performed in the presolving stage of SCIP.

14.5 Multiplication

The multiplication operator

$$\text{MULT} : [\beta] \times [\beta] \to [\beta], \quad (x, y) \mapsto r = \text{MULT}(x, y)$$

with

$$r = \text{MULT}(x, y) \quad \Leftrightarrow \quad r = (x \cdot y) \bmod 2^\beta$$

is the most complex operator in terms of the algorithms involved to process the constraints. Our implementation contains more than 7000 lines of C source code, while the implementation of the algorithms for addition, which is the second largest code, consists of only around 3500 lines. This corresponds to the observation, that SAT and BDD based techniques often fail on circuits that involve lots of multiplications, which can also be seen in the computational results of Chapter 17.

The LP relaxation of multiplication constraints is quite involved. Like in written multiplication learned in school, we multiply each digit of one operand by the digits of the other operand and align the results with respect to the significance of the involved digits. Afterwards, these *partial products* are added up to yield the resultant of the multiplication constraint. Since a multiplication of two variables is a non-linear operation, we cannot use it directly inside a linear relaxation. However, a product of a *binary* variable with another variable *can* be linearized by three inequalities. Therefore, the "digits" of one of the operands are given by its bit decomposition, while for the "digits" of the other operand we use half-words, so called *nibbles*. This asymmetric treatment of the two operands in a commutative operation raises the question which operand should be split into bits and which should be split into nibbles. This question is addressed in the presolving algorithm explained in Section 14.5.3.

We employ three different types of domain propagation algorithms for MULT constraints. The first operates on the multiplication table of the LP relaxation and the involved auxiliary variables, namely the partial products and overflows. The second is applied on the bit level, again on a table similar to written multiplication. It involves a second set of auxiliary variables and propagates fixings of register bits and auxiliary variables. The third domain propagation algorithm constructs the symbolic terms for the intermediate bits of the binary multiplication table and simplifies them, exploiting fixings and equivalences of the register bits. This may yield additional deductions that cannot be found by the standard bit level domain propagation alone.

The presolving on MULT constraints also uses both representations, the bit/nibble and the bit/bit multiplication table. In addition to the domain propagation, it performs aggregations and detects implications. In particular, the terms constructed in the symbolic propagation are compared with each other, and the corresponding variables are aggregated if the terms are equal.

14.5.1 LP RELAXATION

In order to express the multiplication $r = \text{MULT}(x, y)$ of two registers x and y with linear constraints, we decompose x into the bits x_b, $b = 0, \ldots, \beta - 1$, and y and r into half-words, or *nibbles*, of $L = W/2$ bits:

Definition 14.12 (nibble). A register $\varrho_j = \sum_{b=0}^{\beta_j - 1} 2^b \varrho_{jb}$ is subdivided into $\eta_j = \lceil \beta_j / L \rceil$ *nibbles*

$$\varrho_j^{\langle l \rangle} = \varrho_j \left[\delta_j^{\langle l \rangle} - 1, lL \right] = \sum_{b=0}^{\delta_j^{\langle l \rangle} - 1} 2^b \varrho_{j,lL+b}, \quad l = 0, \ldots, \eta_j - 1, \tag{14.11}$$

of L bits with $\delta_j^{\langle l \rangle} = \min\{L, \beta_j - lL\}$ being the width of nibble l.

We do not include the nibbles as additional problem variables into the CIP model. Instead, we use $\varrho_j^{\langle l \rangle}$ only as a shortcut for the subword expressed as a sum of bit variables as in equation (14.11). Therefore, the bounds of the nibble $\varrho_j^{\langle l \rangle}$ are also only shortcuts for

$$l_{\varrho_j^{\langle l \rangle}} = \sum_{b=0}^{\delta_j^{\langle l \rangle} - 1} 2^b l_{\varrho_{j,lL+b}} \quad \text{and} \quad u_{\varrho_j^{\langle l \rangle}} = \sum_{b=0}^{\delta_j^{\langle l \rangle} - 1} 2^b u_{\varrho_{j,lL+b}}. \tag{14.12}$$

We model the relevant *partial products* $p_b^{\langle l \rangle} = x_b \cdot y^{\langle l \rangle}$ of bits x_b and nibbles $y^{\langle l \rangle}$ with the following system of linear inequalities:

$$p_b^{\langle l \rangle} \leq u_{y^{\langle l \rangle}} \cdot x_b \tag{14.13}$$

$$p_b^{\langle l \rangle} \leq y^{\langle l \rangle} \tag{14.14}$$

$$p_b^{\langle l \rangle} \geq y^{\langle l \rangle} - u_{y^{\langle l \rangle}} \cdot (1 - x_b) \tag{14.15}$$

Equation (14.13) enforces the implication $x_b = 0 \rightarrow p_b^{\langle l \rangle} = 0$, while Equations (14.14) and (14.15) are redundant for $x_b = 0$, but ensure $p_b^{\langle l \rangle} = y^{\langle l \rangle}$ for $x_b = 1$. For every integral solution (x, y) the above system of equations ensures $p_b^{\langle l \rangle} \in \mathbb{Z}_{\geq 0}$ for all b and l. Therefore, we do not need to include $p_b^{\langle l \rangle} \in \mathbb{Z}_{\geq 0}$ as an additional integrality restriction. Instead, we mark the partial product variables to be *implicitly integral*. Such variables are not used as branching variables, but their integrality can be exploited in presolving, domain propagation, and cutting plane separation.

In order to calculate the resultant r of the multiplication, we have to add up the partial products as shown in Figure 14.3. A nibble $r^{\langle l \rangle}$ of the resultant is equal to the sum of all partial products $p_b^{\langle j \rangle}$ with $\lfloor b/L \rfloor + j = l$ and the overflow nibble $o^{\langle l \rangle}$ of the previous addition column. In the example of Figure 14.3, we have to add up the dark shaded partial products in their respective significance and the overflow $o^{\langle 2 \rangle}$ to calculate $r^{\langle 2 \rangle}$ and the corresponding overflow $o^{\langle 3 \rangle}$. The summations are described by the equation

$$o^{\langle l \rangle} + \sum_{i+j=l} \sum_{b=0}^{\delta_x^{\langle i \rangle} - 1} 2^b p_{iL+b}^{\langle j \rangle} = 2^L o^{\langle l+1 \rangle} + r^{\langle l \rangle} \tag{14.16}$$

for $l = 0, \ldots, \eta_r - 1$. The upper bound of the overflow variables can be calculated recursively as follows:

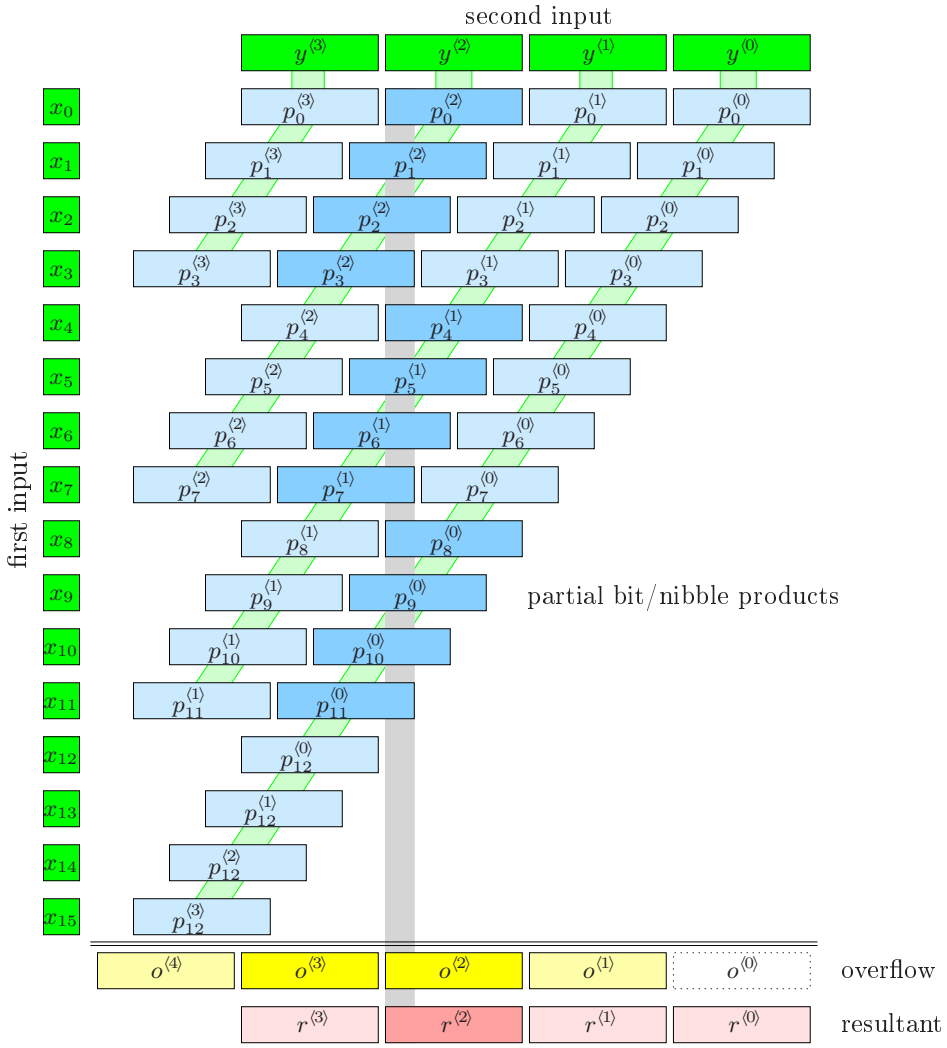

Figure 14.3. Multiplication table used in LP relaxation of $r = \text{MULT}(x, y)$. The first input register x is split into bits x_b, and the second input register y and the output register r are split into nibbles $y^{\langle l \rangle}$ and $r^{\langle l \rangle}$.

Proposition 14.13. For the overflow variable $o^{\langle l+1 \rangle}$ in equation (14.16) the recursively defined value

$$u_{o^{\langle l+1 \rangle}} = \left\lfloor \frac{u_{o^{\langle l \rangle}} + (l+1)(2^L - 1)^2}{2^L} \right\rfloor \quad \text{with } u_{o^{\langle 0 \rangle}} = 0$$

is a valid upper bound.

Proof. From equation (14.16) it follows that

$$\begin{aligned} 2^L o^{\langle l+1 \rangle} &= o^{\langle l \rangle} + \sum_{i+j=l} \sum_{b=0}^{\delta_x^i - 1} 2^b p_{iL+b}^{\langle j \rangle} - r^{\langle l \rangle} \le o^{\langle l \rangle} + \sum_{i+j=l} \sum_{b=0}^{L-1} 2^b (2^L - 1) \\ &= o^{\langle l \rangle} + \sum_{i+j=l} (2^L - 1)^2 = o^{\langle l \rangle} + (l+1)(2^L - 1)^2, \end{aligned}$$

which, together with the integrality of $o^{\langle l+1\rangle}$, proves the claim. □

Note. The upper bounds on the first seven overflow variables are $u^{\langle 0\rangle} = 0$, $u^{\langle 1\rangle} = 254$, $u^{\langle 2\rangle} = 509$, $u^{\langle 3\rangle} = 764$, $u^{\langle 4\rangle} = 1019$, $u^{\langle 5\rangle} = 1274$, and $u^{\langle 6\rangle} = 1529$.

As a summary, the LP relaxation of a multiplication constraint $r = \text{MULT}(x, y)$ is given by equations (14.13), (14.14), and (14.15) for all b and l with $\lfloor b/L \rfloor + l < \eta_r$, and equation (14.16) for $l = 0, \ldots, \eta_r - 1$. We have to introduce auxiliary variables $p_b^{\langle l\rangle} \in \mathbb{R}_{\geq 0}$ for $\lfloor b/L \rfloor + l < \eta_r$ with bounds $0 \leq p_b^{\langle l\rangle} \leq u_{y^{\langle l\rangle}}$, and $o^{\langle l\rangle} \in \mathbb{Z}_{\geq 0}$ for $l = 0, \ldots, \eta_r$ with bounds $0 \leq o^{\langle l\rangle} \leq u_{o^{\langle l\rangle}}$ as defined in Proposition 14.13.

14.5.2 Domain Propagation

The propagation of multiplication constraints $r = \text{MULT}(x, y)$ is performed in three stages. The first stage applies domain propagation on the LP formulation stated in Section 14.5.1 and its auxiliary variables $p_b^{\langle l\rangle}$ and $o^{\langle l\rangle}$. In the second stage, we perform domain propagation on a multiplication table consisting of binary partial products $p_{ij} = x_i \cdot y_j$, i.e., on a formulation that can be expressed with binary variables only. The third stage applies a symbolic propagation on this binary multiplication table by using a term rewriting system.

Propagation on the LP Relaxation

In this stage of the propagation, we try to deduce tighter bounds for the bits x_b, the nibbles $y^{\langle l\rangle}$ and $r^{\langle l\rangle}$, and the auxiliary variables $p_b^{\langle l\rangle}$ and $o^{\langle l\rangle}$, which are used in the LP relaxation presented in Section 14.5.1. Since it can yield tighter bounds for the propagation, we also look at the current bounds of the partial bit product variables $p_{ij} = x_i \cdot y_j \in \{0, 1\}$ that are used in the next stage of the domain propagation.

As already noted in Section 14.5.1, the bounds of the nibbles $y^{\langle l\rangle}$ and $r^{\langle l\rangle}$ are only shortcuts for the sums of the corresponding bit bounds, see equation (14.12). These bounds get strengthened in presolving and domain propagation whenever one of the involved bits is fixed. Conversely, a strengthening of a nibble bound corresponds to fixings of the involved bits. However, for given deduced nibble bounds $l \leq y^{\langle l\rangle} \leq u$, we can only fix the most significant bits y_{lL+b}, namely $y_{lL+b} = 1$ for bits b with $\sum_{i=0}^{\delta-1} 2^i u_{y_{lL+i}} - 2^b u_{y_{lL+b}} < l$, and $y_{lL+b} = 0$ for bits b with $\sum_{i=0}^{\delta-1} 2^i l_{y_{lL+i}} - 2^b l_{y_{lL+b}} + 2^b > u$. If we deduce $l \leq y^{\langle l\rangle} \leq u$ in an algorithm, we mean to fix as many bit variables as possible in this fashion. The same holds for deduced nibble bounds on $r^{\langle l\rangle}$. Note that variables $o^{\langle l\rangle}$ are actual problem variables for which the bounds can be deduced normally.

Algorithm 14.7 shows how the deductions on the registers x, y, and r, and the auxiliary variables of the LP relaxation are performed. Step 1 processes the partial product equation $p_b^{\langle l\rangle} = x_b \cdot y^{\langle l\rangle}$. If $x_b = 0$ we can conclude $p_b^{\langle l\rangle} = 0$ in Step 1a. If on the other hand $x_b = 1$, we can tighten the bounds of $p_b^{\langle l\rangle}$ and $y^{\langle l\rangle}$ using $p_b^{\langle l\rangle} = y^{\langle l\rangle}$ in Step 1b. Independent from the value of x_b, the partial product $p_b^{\langle l\rangle}$ cannot be larger than $y^{\langle l\rangle}$, and the upper bound of $p_b^{\langle l\rangle}$ and the lower bound of $y^{\langle l\rangle}$ can be tightened correspondingly in Step 1c.

Step 1d provides the link to the auxiliary partial bit product variables $p_{ij} = x_i \cdot y_j$ which are used in the second stage of the domain propagation, see Algorithm 14.8.

Algorithm 14.7 Multiplication Domain Propagation – LP Propagation

Input: Multiplication constraint $r = \text{MULT}(x, y)$ on registers r, x, and y of width β with current local word bounds $\tilde{l}_{r^w} \leq r^w \leq \tilde{u}_{r^w}$, $\tilde{l}_{x^w} \leq x^w \leq \tilde{u}_{x^w}$, and $\tilde{l}_{y^w} \leq y^w \leq \tilde{u}_{y^w}$, and bit bounds $\tilde{l}_{r_b} \leq r_b \leq \tilde{u}_{r_b}$, $\tilde{l}_{x_b} \leq x_b \leq \tilde{u}_{x_b}$, and $\tilde{l}_{y_b} \leq y_b \leq \tilde{u}_{y_b}$; current local bounds on auxiliary variables $\tilde{l}_{p_b^{\langle l \rangle}} \leq p_b^{\langle l \rangle} \leq \tilde{u}_{p_b^{\langle l \rangle}}$, $\tilde{l}_{p_{ij}} \leq p_{ij} \leq \tilde{u}_{p_{ij}}$, and $\tilde{l}_{o^{\langle l \rangle}} \leq o^{\langle l \rangle} \leq \tilde{u}_{o^{\langle l \rangle}}$.

Output: Tightened local bounds for words r^w, x^w, y^w, bits r_b, x_b, y_b, and auxiliary variables $p_b^{\langle l \rangle}$ and $o^{\langle l \rangle}$.

1. For all $b = 0, \ldots, \beta - 1$ and all $l = 0, \ldots, \eta - 1$:
   (a) If $\tilde{u}_{x_b} = 0$, deduce $p_b^{\langle l \rangle} = 0$.
   (b) If $\tilde{l}_{x_b} = 1$, deduce $p_b^{\langle l \rangle} \geq \tilde{l}_{y^{\langle l \rangle}}$ and $y^{\langle l \rangle} \leq \tilde{u}_{p_b^{\langle l \rangle}}$.
   (c) Deduce $p_b^{\langle l \rangle} \leq \tilde{u}_{y^{\langle l \rangle}}$ and $y^{\langle l \rangle} \geq \tilde{l}_{p_b^{\langle l \rangle}}$.
   (d) Deduce $p_b^{\langle l \rangle} \leq \sum_{i=0}^{L-1} 2^i \tilde{u}_{p_{b,lL+i}}$.
   (e) If $\tilde{l}_{p_b^{\langle l \rangle}} \geq 1$, deduce $x_b = 1$.
   (f) If $\tilde{u}_{p_b^{\langle l \rangle}} < \tilde{l}_{y^{\langle l \rangle}}$, deduce $x_b = 0$.
2. For $l = 0, \ldots, \eta - 1$, tighten the bounds using equation (14.16):

$$\tilde{l}_{r^{\langle l \rangle}} - \sum_{i+j=l} \sum_{b=0}^{\delta_x^i - 1} 2^b \tilde{u}_{p_{iL+b}^{\langle j \rangle}} + 2^L \tilde{l}_{o^{\langle l+1 \rangle}} \leq \quad o^{\langle l \rangle} \quad \leq \tilde{u}_{r^{\langle l \rangle}} - \sum_{i+j=l} \sum_{b=0}^{\delta_x^i - 1} 2^b \tilde{l}_{p_{iL+b}^{\langle j \rangle}} + 2^L \tilde{u}_{o^{\langle l+1 \rangle}}$$

$$\tilde{l}_{o^{\langle l \rangle}} + \sum_{i+j=l} \sum_{b=0}^{\delta_x^i - 1} 2^b \tilde{l}_{p_{iL+b}^{\langle j \rangle}} - 2^L \tilde{u}_{o^{\langle l+1 \rangle}} \leq \quad r^{\langle l \rangle} \quad \leq \tilde{u}_{o^{\langle l \rangle}} + \sum_{i+j=l} \sum_{b=0}^{\delta_x^i - 1} 2^b \tilde{u}_{p_{iL+b}^{\langle j \rangle}} - 2^L \tilde{l}_{o^{\langle l+1 \rangle}}$$

$$\left\lceil \frac{\tilde{l}_{o^{\langle l \rangle}} + \sum_{i+j=l} \sum_{b=0}^{\delta_x^i - 1} 2^b \tilde{l}_{p_{iL+b}^{\langle j \rangle}} - \tilde{u}_{r^{\langle l \rangle}}}{2^L} \right\rceil \leq o^{\langle l+1 \rangle} \leq \left\lfloor \frac{\tilde{u}_{o^{\langle l \rangle}} + \sum_{i+j=l} \sum_{b=0}^{\delta_x^i - 1} 2^b \tilde{u}_{p_{iL+b}^{\langle j \rangle}} - \tilde{l}_{r^{\langle l \rangle}}}{2^L} \right\rfloor$$

By the definition of the partial products, we know that $p_b^{\langle l \rangle} = \sum_{i=0}^{L-1} 2^i p_{b,lL+i}$. Therefore, we can deduce a corresponding upper bound for $p_b^{\langle l \rangle}$ in Step 1d. The lower bound of $p_b^{\langle l \rangle}$ does not need to be tightened, because if $\sum_{i=0}^{L-1} 2^i \tilde{l}_{p_{b,lL+i}} > 0$ we can deduce $x_b = 1$ in Algorithm 14.8, and the lower bound of $p_b^{\langle l \rangle}$ is already tightened in a subsequent Step 1b of Algorithm 14.7. Note that after Algorithm 14.8 was performed, we have $\sum_{i=0}^{L-1} 2^i \tilde{u}_{p_{b,lL+i}} \leq \tilde{u}_{y^{\langle l \rangle}}$, which means that Step 1c is redundant in subsequent iterations.

Steps 1e and 1f are the inverse implications of Steps 1a and 1b, respectively. If $p_b^{\langle l \rangle} > 0$, the bit x_b must be one, and if $p_b^{\langle l \rangle} < y^{\langle l \rangle}$, we know that $x_b = 0$. Additional conclusions like $p_b^{\langle l \rangle} = 0$ if $p_b^{\langle l \rangle} < y^{\langle l \rangle}$ are automatically drawn in Steps 1a and 1b of the subsequent iteration.

Propagation on Binary Multiplication Table

The second stage in the domain propagation of the multiplication constraint $r = \text{MULT}(x, y)$ of width β consists of the calculation and propagation of a binary multiplication table. We introduce new auxiliary variables

$$p_{ij} = x_i \cdot y_j \in \{0, 1\} \quad \text{for} \quad i + j < \beta, \tag{14.17}$$

which are the *partial products* of individual bits in x and y. These partial products are added up columnwise to calculate the resultant bits r_b. Of course, the sum of a column can produce an overflow that has to be passed to the more significant columns.

Figure 14.4 illustrates the multiplication table for an 8 bit multiplication. The partial products p_{ij} are assigned as addends to a column that corresponds to its significance, i.e., p_{ij} is assigned to column $i + j$. We sum up the addends of the columns in blocks of three. For each block and for each column we compute an intermediate sum out of the at most three addends. This sum is in the range of 0 to 3 and can therefore be represented as a 2 bit value. We call the low significant bit of block k and column j the *subtotal* $s_j^k \in \{0, 1\}$, and the high significant bit the *overflow* $o_j^k \in \{0, 1\}$. Therefore, for addends $a_j^{k0}, a_j^{k1}, a_j^{k2} \in \{0, 1\}$ of block k and column j the equation

$$a_j^{k0} + a_j^{k1} + a_j^{k2} = 2o_j^k + s_j^k \tag{14.18}$$

defines the values of the subtotal and overflow. The subtotal variable s_j^k is used as the first addend $a_j^{k+1,0}$ in the next block of the same column. The overflow variable o_j^k is passed as additional addend to the next column $j+1$. The dark shaded variables in Figure 14.4 are the variables that are involved in the subtotal calculation (14.18) of block 1 and column 4. The last subtotals of each column define the resultant bits:

$$r_b = s_b^{b-1}, \quad b = 0, \ldots, \beta - 1, \quad \text{with } s_0^{-1} = p_{00}.$$

Note that one can add up the partial products, the subtotals, and the overflows in any order. This order may affect the propagation, since for a different order the subtotals and overflows are defined differently. We choose the static order that is indicated in Figure 14.4: the subtotals s_j^k are always the first addend of the next block $k + 1$, and the overflows are the last addends in their respective column with overflows of smaller block numbers preceeding the ones of larger block numbers. The partial products p_{ij} fill the empty slots in the order of increasing index i. This yields the following assignment to the addends $q \in \{0, 1, 2\}$ of each block k and column j:

$$\begin{aligned}
\text{block } 0: \quad a_j^{0q} &= \begin{cases} p_{q,j-q} & \text{if } j \geq q \\ 0 & \text{if } j < q \end{cases} \\
\text{block } k \geq 1: \quad a_j^{kq} &= \begin{cases} s_j^{k+1} & \text{if } q = 0 \\ p_{2k+q,j-(2k+q)} & \text{if } q \geq 1 \text{ and } 2k+1 \leq j \\ o_{j-1}^{2k+q-1-j} & \text{if } q \geq 1 \text{ and } 2k+1 > j \end{cases}
\end{aligned} \tag{14.19}$$

We use the overflows as late as possible because we also employ the same multiplication table with the same variables in the symbolic propagation with term rewriting, see below. The symbolic terms for the overflow variables are more complex than the ones for the partial products and the subtotals. This complexity is introduced to the subtotal terms of a column as soon as an overflow variable is

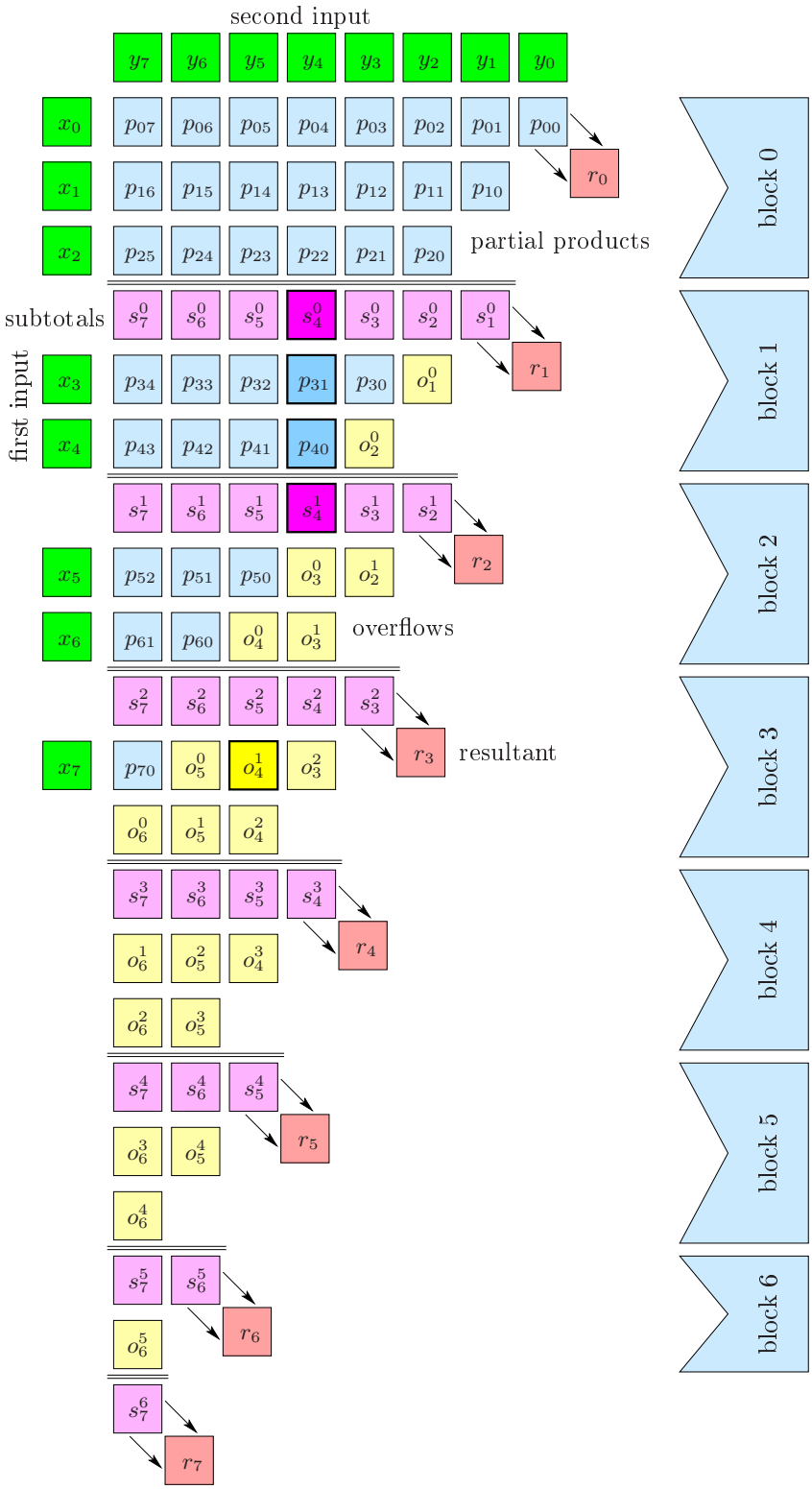

Figure 14.4. Binary multiplication table used in domain propagation of $r = \text{MULT}(x, y)$.

Algorithm 14.8 Multiplication Domain Propagation – Binary Propagation

Input: Multiplication constraint $r = \text{MULT}(x, y)$ on registers r, x, and y of width β with current local bit bounds $\tilde{l}_{r_b} \leq r_b \leq \tilde{u}_{r_b}$, $\tilde{l}_{x_b} \leq x_b \leq \tilde{u}_{x_b}$, and $\tilde{l}_{y_b} \leq y_b \leq \tilde{u}_{y_b}$; current local bounds on auxiliary variables $\tilde{l}_{p_{ij}} \leq p_{ij} \leq \tilde{u}_{p_{ij}}$, $\tilde{l}_{s_j^k} \leq s_j^k \leq \tilde{u}_{s_j^k}$, and $\tilde{l}_{o_j^k} \leq o_j^k \leq \tilde{u}_{o_j^k}$.

Output: Tightened local bounds for bits r_b, x_b, y_b, and auxiliary variables p_{ij}, s_j^k, and o_j^k.

1. For all $i + j < \beta$, propagate partial product (14.17) as $p_{ij} = \text{AND}(x_i, y_j)$ with the corresponding domain propagation Algorithm 14.13 for bitwise and constraints.
2. For all columns $j = 1, \ldots, \beta - 1$ and all of its blocks $k = 0, \ldots, j - 1$, propagate equation (14.18) as in Step 1 of the domain propagation Algorithm 14.3 of the addition constraint. Use the substitutions $x_b \to a_j^{k0}$, $y_b \to a_j^{k1}$, $o_b \to a_j^{k2}$, $r_b \to s_j^k$, and $o_{b+1} \to o_j^k$, with a_j^{kq} being defined by equation (14.19).

used as addend. Since it is more likely to identify equal terms on terms with low complexity, we try to postpone the addition of overflow terms as long as possible.

Wedler, Stoffel, and Kunz [209] use a dynamic approach for ordering the addends of a column. They add up those variables first that are fixed in the local subproblem. The goal is to produce as many fixings as possible in the overflow variables added to the next column. They proved that this technique leads to the maximal number of forward propagations of the "external" variables x_b and y_b to the subtotals and overflows. The disadvantage of this approach within our framework is that without static definitions of the internal variables, they cannot be used in conflict analysis and cutting plane separation. Additionally, the statically defined internal variables can be used as "memory" for already discovered deductions. If dynamic addends are used, we would have to propagate the full multiplication table from scratch at every iteration.

Algorithm 14.8 describes the domain propagation that is applied on the binary multiplication table. There are only two types of constraints involved, namely the partial product constraints (14.17) and the addition constraints (14.18). A multiplication of two bits $p_{ij} = x_i \cdot y_j$ is equivalent to the AND concatenation of the bits. Therefore, we can just call the domain propagation algorithm of the bitwise addition as a subroutine in Step 1. The bit addition constraint (14.18) has the same structure as equation (14.8) which appears in the propagation of the addition constraint. Hence, we just call the appropriate part of Algorithm 14.3 in Step 2 to propagate these equations.

Symbolic Propagation with Term Rewriting

In the previous section about propagation on the binary multiplication table, we investigated how fixings of binary variables can deduce further fixings of other binary variables. In this section, we also try to keep track of the unfixed variables, since it may happen that their contribution to a later subtotal or overflow is canceled out, regardless of their actual values. In order to do this, we perform symbolic calculations on terms in which the unfixed variables appear as symbols.

The variables in the binary multiplication table are calculated with equations (14.17)

and (14.18), i.e.,

$$p_{ij} = x_i \cdot y_j \quad \text{and} \quad 2o_j^k + s_j^k = a_j^{k0} + a_j^{k1} + a_j^{k2},$$

with a_j^{kq} being either a partial product p, a subtotal s, or an overflow o. Now we express the equations to calculate p, s, and o as logical terms over x_i and y_j:

$$p_{ij} = x_i \wedge y_j \tag{14.20}$$

$$s_j^k = a_j^{k0} \oplus a_j^{k1} \oplus a_j^{k2} \tag{14.21}$$

$$\begin{aligned} o_j^k &= (a_j^{k0} \wedge a_j^{k1}) \vee (a_j^{k0} \wedge a_j^{k2}) \vee (a_j^{k1} \wedge a_j^{k2}) \\ &= (a_j^{k0} \wedge a_j^{k1}) \oplus (a_j^{k0} \wedge a_j^{k2}) \oplus (a_j^{k1} \wedge a_j^{k2}). \end{aligned} \tag{14.22}$$

We can replace the disjunction $\vee$ by the exclusive or $\oplus$ in the equation of o_j^k, because it cannot happen that exactly two of the three subterms are true, and in the other three cases (0, 1, or 3 terms are true), both operators yield the same result:

Observation 14.14. For all $x, y, z \in \{0, 1\}$, the equivalence

$$(x \vee y \vee z \;\neq\; x \oplus y \oplus z) \;\Leftrightarrow\; (x + y + z = 2)$$

holds.

Proof. We only have to evaluate the four cases $x + y + z \in \{0, 1, 2, 3\}$, since both operators $\vee$ and $\oplus$ are associative and commutative. □

Due to Observation 14.14, we can express all emerging expressions for internal variables in the binary multiplication table as terms over $\wedge$ and $\oplus$. We call the resulting algebra the *binary multiplication term algebra*:

Definition 14.15 (binary multiplication signature). The algebraic signature $\Sigma = (B, O)$ with the sort B and operations $O = O_0 \cup O_2$ with 0-ary symbols $O_0 = \{0, 1, x_0, \ldots, x_{\beta-1}, y_0, \ldots, y_{\beta-1}\}$ and binary operators $O_2 = \{\wedge, \oplus\}$,

$$0 :\rightarrow B,\;\; 1 :\rightarrow B,\;\; x_b :\rightarrow B,\;\; y_b :\rightarrow B,\;\; \wedge :\; B \times B \rightarrow B,\;\; \oplus :\; B \times B \rightarrow B$$

with $b = 0, \ldots, \beta - 1$ is called *binary multiplication signature*. We call $\mathcal{T}_\Sigma$ the *term algebra* of Σ, which consists of all terms that can be generated from the symbols in Σ and which fit to the arity of the operators.

By applying the distributivity law

$$(s \oplus t) \wedge u = (s \wedge u) \oplus (t \wedge u),\;\; s \wedge (t \oplus u) = (s \wedge t) \oplus (s \wedge u)$$

for terms $s, t, u \in \mathcal{T}_\Sigma$ we can rewrite any term $t \in \mathcal{T}_\Sigma$ as an equivalent term in *disjunctive normal form*

$$t \equiv \pi_n \oplus \ldots \oplus \pi_1,\;\; \pi_i = z_{im_i} \wedge \ldots \wedge z_{i1},\;\; z_{ij} \in O_0$$

for $i = 1, \ldots, n$ and $j = 1, \ldots, m_i$. Due to commutativity and associativity of $\wedge$, the individual conjunctions can be reordered in a second step such that

$$\pi_i = z_{im_i} \wedge \ldots \wedge z_{i1} \text{ with } z_{im_i} \succeq \ldots \succeq z_{i1}$$

holds for the precedence relation $\succ$ on the symbols O of Σ, which is defined as the transitive closure of

$$\wedge \succ \oplus \succ y_{\beta-1} \succ \ldots \succ y_0 \succ x_{\beta-1} \succ \ldots \succ x_0 \succ 1 \succ 0. \tag{14.23}$$

Afterwards, we apply the equations

$$z \wedge 0 = 0, \quad z \wedge 1 = z, \quad z \wedge z = z$$

on the conjunctions which yields equivalent conjunctions in normal form. The addends π_i of a term $t \in \mathcal{T}_\Sigma$ in disjunctive normal form with normalized conjunctions are then reordered using commutativity and associativity of $\oplus$ such that

$$t \equiv \pi_n \oplus \ldots \oplus \pi_1, \quad \pi_i = z_{im_i} \wedge \ldots \wedge z_{i1}, \quad z_{ij} \in O_0$$

and $(z_{im_i}, \ldots, z_{i1}) \succeq_{\text{lex}} (z_{jm_j}, \ldots, z_{j1})$ for $i > j$. The relation $\succ_{\text{lex}}$ is the lexicographic (right-left) ordering with respect to $\succ$:

Definition 14.16 (lexicographic ordering). Given an ordering $\succ \subseteq O \times O$ on a set O, the relation $\succ_{\text{lex}} \subseteq O^\star \times O^\star$ on the strings $O^\star$ with

$$\begin{aligned}(t_n, \ldots, t_1) \succ_{\text{lex}} (s_m, \ldots, s_1) \\ \Leftrightarrow \quad \text{(i)} \quad & n > m, \text{ or} \\ \text{(ii)} \quad & n = m \text{ and } t_1 \succ s_1, \text{ or} \\ \text{(iii)} \quad & n = m \text{ and } t_1 = s_1 \text{ and } (t_n, \ldots, t_2) \succ_{\text{lex}} (s_m, \ldots, s_2)\end{aligned}$$

is called *lexicographic (right-left) ordering* with respect to $\succ$ on strings $O^\star$.

Note. The literature distinguishes between two versions of the lexicographic ordering: the left-right ordering $\succ_{\text{lexlr}}$ and the right-left ordering $\succ_{\text{lexrl}}$. Since we only need the right-left variant, we abbreviate $\succ_{\text{lexrl}}$ by $\succ_{\text{lex}}$ and call this version the *lexicographic ordering* as a shortcut.

After having reordered the addends of the disjunction, we simplify the term by applying the equations

$$\pi \oplus 0 = \pi \quad \text{and} \quad \pi \oplus \pi = 0.$$

We call the resulting term t' the *normal form* of t if $t \equiv t'$.

Algorithm 14.9 subsumes this procedure in a more formal fashion. In Step 1 we apply the distributivity law to achieve disjunctive normal form. Step 2 reorders the symbols in the conjunctions π_i, and Step 3 applies the simplifications for the $\wedge$ operator. Note that these simplifications do not destroy the ordering property of the conjunctions. In Step 4 we reorder the resulting addends of the XOR expression, and finally, in Step 5 the simplifications on the $\oplus$ operator are applied. The replacement 5b replaces the innermost two addends of the term by the symbol "0", which has to be moved to the right by applying Step 4 again to restore the lexicographic ordering.

In the definition of the commutativity rewriting Rules 4b and 4c we used the concept of the *lexicographic recursive path ordering* (see Kamin and Levy [128] and Dershowitz [79]):

Algorithm 14.9 Multiplication Domain Propagation – Term Normalization

Input: Term $t \in \mathcal{T}_\Sigma$
Output: Term $t' \equiv t$ in normal form

1. Until no more replacements are possible:
   (a) Replace subterms $(t_3 \oplus t_2) \wedge t_1 \to (t_3 \wedge t_1) \oplus (t_2 \wedge t_1)$.
   (b) Replace subterms $t_3 \wedge (t_2 \oplus t_1) \to (t_3 \wedge t_2) \oplus (t_3 \wedge t_1)$.
2. Until no more replacements are possible:
   (a) Replace subterms $\pi_3 \wedge (\pi_2 \wedge \pi_1) \to (\pi_3 \wedge \pi_2) \wedge \pi_1$.
   (b) Replace subterms $z_1 \wedge z_2 \to z_2 \wedge z_1$ if $z_1, z_2 \in O_0$ and $z_2 \succ z_1$.
   (c) Replace subterms $(\pi \wedge z_1) \wedge z_2 \to (\pi \wedge z_2) \wedge z_1$ if $z_1, z_2 \in O_0$ and $z_2 \succ z_1$.
3. Until no more replacements are possible:
   (a) Replace subterms $\pi \wedge 0 \to 0$.
   (b) Replace subterms $\pi \wedge 1 \to \pi$.
   (c) Replace subterms $z \wedge z \to z$ with $z \in O_0$.
   (d) Replace subterms $(\pi \wedge z) \wedge z \to \pi \wedge z$ with $z \in O_0$.
4. Until no more replacements are possible:
   (a) Replace subterms $\pi_3 \oplus (\pi_2 \oplus \pi_1) \to (\pi_3 \oplus \pi_2) \oplus \pi_1$.
   (b) Replace subterms $\pi_1 \oplus \pi_2 \to \pi_2 \oplus \pi_1$ if $\pi_2 \succ_{\text{lrpo}} \pi_1$.
   (c) Replace subterms $(\pi_3 \oplus \pi_1) \oplus \pi_2 \to (\pi_3 \oplus \pi_2) \oplus \pi_1$ if $\pi_2 \succ_{\text{lrpo}} \pi_1$.
5. Until no more replacements are possible:
   (a) Replace subterms $\pi \oplus 0 \to \pi$.
   (b) Replace subterms $\pi \oplus \pi \to 0$; if applied, goto Step 4.
   (c) Replace subterms $(\pi_2 \oplus \pi_1) \oplus \pi_1 \to \pi_2$.

Definition 14.17 (lexicographic recursive path ordering). Let $\Sigma = (S, O)$ be an algebraic signature and $\succ \subseteq O \times O$ be a partial ordering on the operator symbols of Σ. Then $\succ_{\text{lrpo}} \subseteq \mathcal{T}_\Sigma \times \mathcal{T}_\Sigma$ with

$$
\begin{aligned}
& g(t_n, \ldots, t_1) \succ_{\text{lrpo}} f(s_m, \ldots, s_1) \\
& \qquad \Leftrightarrow \quad \text{(i)} \;\; t_j \succeq_{\text{lrpo}} f(s_m, \ldots, s_1) \text{ for some } j \in \{1, \ldots, n\}, \text{ or} \\
& \qquad\qquad \text{(ii)} \;\; g \succ f \text{ and } g(t_n, \ldots, t_1) \succ_{\text{lrpo}} s_i \text{ for all } i \in \{1, \ldots, m\}, \text{ or} \\
& \qquad\qquad \text{(iii)} \;\; g = f,\ g(t_n, \ldots, t_1) \succ_{\text{lrpo}} s_i \text{ for all } i \in \{1, \ldots, m\}, \\
& \qquad\qquad\qquad \text{and } (t_n, \ldots, t_1)\ (\succ_{\text{lrpo}})_{\text{lex}}\ (s_m, \ldots, s_1)
\end{aligned}
$$

is called the *lexicographic recursive path ordering* of Σ with respect to $\succ$.

Note again, that we employ the right-left variant of the lexicographic ordering in our definition of $\succ_{\text{lrpo}}$.

The lexicographic recursive path ordering is a very useful tool for proving the termination of term rewriting systems. In the following, we employ this concept to prove the termination of Algorithm 14.9.

Lemma 14.18. Let $\Sigma = (S, O)$ be an algebraic signature and $\succ \subseteq O \times O$ a partial ordering on the operator symbols. If $s \in \mathcal{T}_\Sigma$ is a proper subterm of $t \in \mathcal{T}_\Sigma$ then $t \succ_{\text{lrpo}} s$.

Proof. We prove the claim by induction on the depth $d \in \mathbb{Z}_{>0}$ of s in the tree representation of $t = g(t_n, \ldots, t_1)$. If $s = t_j$ for any $j \in \{1, \ldots, n\}$, i.e., $d = 1$, we have $t \succ_{\text{lrpo}} s$ by Condition (i) of Definition 14.17. Otherwise, let t_j be the subterm of t that contains s. Then, s is in depth $d-1$ of the tree representation of t_j, and we have $t_j \succ_{\text{lrpo}} s$ by induction. Again by Condition (i), it follows $t \succ_{\text{lrpo}} s$. □

Definition 14.19 (well-founded ordering). A partial ordering $\succ \subseteq S \times S$ on a set S is called *well-founded* if there does not exist an infinite descending chain $s_1 \succ s_2 \succ \ldots$ of elements $s_i \in S$.

Definition 14.20 (monotonic ordering). Given an algebraic signature $\Sigma = (S, O)$ with term algebra $\mathcal{T}_\Sigma$, a partial ordering $\succ \subseteq \mathcal{T}_\Sigma \times \mathcal{T}_\Sigma$ on the terms is called *monotonic*, if

$$s \succ s' \quad \Rightarrow \quad g(t_n, \ldots, s, \ldots, t_1) \succ g(t_n, \ldots, s', \ldots, t_1)$$

for all $n \in \mathbb{Z}_{\geq 0}$, n-ary symbols $g \in O$, and terms $s, s', t_1, \ldots, t_n \in \mathcal{T}_\Sigma$.

Theorem 14.21 (Dershowitz [79]). If $\succ \subseteq O \times O$ is a well-founded partial ordering on the operations of a finite algebraic signature $\Sigma = (S, O)$, then the lexicographic recursive path ordering $\succ_{\text{lrpo}} \subseteq \mathcal{T}_\Sigma \times \mathcal{T}_\Sigma$ is a well-founded monotonic partial ordering on the terms $\mathcal{T}_\Sigma$.

For our precedence relation $\succ$ defined by (14.23), the lexicographic recursive path ordering $\succ_{\text{lrpo}} \subseteq \mathcal{T}_\Sigma \times \mathcal{T}_\Sigma$ of Σ with respect to $\succ$ is given by

$$z_2 \succ_{\text{lrpo}} z_1 \qquad \text{if } z_2 \succ z_1 \tag{a}$$
$$t_2 \oplus t_1 \succ_{\text{lrpo}} z \tag{b}$$
$$t_2 \wedge t_1 \succ_{\text{lrpo}} z \tag{c}$$
$$t_2 \oplus t_1 \succ_{\text{lrpo}} s_2 \oplus s_1 \quad \text{if } t_1 \succeq_{\text{lrpo}} s_2 \oplus s_1 \text{ or } t_2 \succeq_{\text{lrpo}} s_2 \oplus s_1 \tag{d}$$
$$t_2 \oplus t_1 \succ_{\text{lrpo}} s_2 \wedge s_1 \quad \text{if } t_1 \succeq_{\text{lrpo}} s_2 \wedge s_1 \text{ or } t_2 \succeq_{\text{lrpo}} s_2 \wedge s_1 \tag{e}$$
$$t_2 \wedge t_1 \succ_{\text{lrpo}} s_2 \oplus s_1 \quad \text{if } t_1 \succeq_{\text{lrpo}} s_2 \oplus s_1 \text{ or } t_2 \succeq_{\text{lrpo}} s_2 \oplus s_1 \tag{f}$$
$$t_2 \wedge t_1 \succ_{\text{lrpo}} s_2 \wedge s_1 \quad \text{if } t_1 \succeq_{\text{lrpo}} s_2 \wedge s_1 \text{ or } t_2 \succeq_{\text{lrpo}} s_2 \wedge s_1 \tag{g}$$
$$t_2 \wedge t_1 \succ_{\text{lrpo}} s_2 \oplus s_1 \quad \text{if } t_2 \wedge t_1 \succ_{\text{lrpo}} s_i \text{ for } i = 1, 2 \tag{h}$$
$$t_2 \oplus t_1 \succ_{\text{lrpo}} s_2 \oplus s_1 \quad \text{if } t_2 \oplus t_1 \succ_{\text{lrpo}} s_i \text{ for } i = 1, 2, \text{ and } t_1 \succ_{\text{lrpo}} s_1 \tag{i}$$
$$t_2 \oplus t_1 \succ_{\text{lrpo}} s_2 \oplus s_1 \quad \text{if } t_2 \oplus t_1 \succ_{\text{lrpo}} s_i \text{ for } i = 1, 2, \text{ and } t_1 = s_1, t_2 \succ_{\text{lrpo}} s_2 \tag{j}$$
$$t_2 \wedge t_1 \succ_{\text{lrpo}} s_2 \wedge s_1 \quad \text{if } t_2 \wedge t_1 \succ_{\text{lrpo}} s_i \text{ for } i = 1, 2, \text{ and } t_1 \succ_{\text{lrpo}} s_1 \tag{k}$$
$$t_2 \wedge t_1 \succ_{\text{lrpo}} s_2 \wedge s_1 \quad \text{if } t_2 \wedge t_1 \succ_{\text{lrpo}} s_i \text{ for } i = 1, 2, \text{ and } t_1 = s_1, t_2 \succ_{\text{lrpo}} s_2 \tag{l}$$

for all $z, z_1, z_2 \in O_0$, $s_1, s_2, t_1, t_2 \in \mathcal{T}_\Sigma$.

Proposition 14.22. Algorithm 14.9 terminates.

Proof. The relation $\succ$ is obviously a well-founded partial ordering (in fact, even a total ordering) on O. By Theorem 14.21 $\succ_{\text{lrpo}}$ is a well-founded monotonic partial

ordering on $\mathcal{T}_\Sigma$. Therefore, to show the termination of Algorithm 14.9 it suffices to prove that each replacement rule reduces the order of the term t, i.e., if t is replaced by t', then $t \succ_{\text{lrpo}} t'$. Due to the monotonicity, it even suffices to show this implication for the subterms on which the replacements are applied.

Step 1a reduces the order due to condition (h) since $(t_3 \oplus t_2) \wedge t_1 \succ_{\text{lrpo}} t_3 \wedge t_1$ and $(t_3 \oplus t_2) \wedge t_1 \succ_{\text{lrpo}} t_2 \wedge t_1$ by condition (l) and Lemma 14.18. The same is true for Step 1b by condition (h), condition (k), and Lemma 14.18. Step 2a reduces the order due to (k) and Lemma 14.18. Step 2b is due to (k), (a), and Lemma 14.18, and the order reduction of Step 2c follows from (k), (a), (l), and Lemma 14.18.

The replacements in Step 3 reduce the order of the term due to Lemma 14.18. The proof for the replacements in Step 4 is analogous to the one for Step 2. The order reduction of Step 5a and 5c is again due to Lemma 14.18 and follows for Step 5b by condition (b). □

Corollary 14.23. Algorithm 14.9 would also terminate if the term rewriting rules were applied in an arbitrary order.

Proof. Since each individual rewriting rule reduces the order of the term, termination is not dependent on the order in which they are applied. □

After having defined our term rewriting system in Algorithm 14.9 to normalize arbitrary terms $t \in \mathcal{T}_\Sigma$, we are ready to present the term algebra domain propagation algorithm for MULT constraints, which is illustrated in Algorithm 14.10. The algorithm consists of a loop over the columns and addition blocks of the binary multiplication table of Figure 14.4, i.e., over the individual equations (14.18). The terms are calculated and processed in Steps 2 to 6. If deductions or substitutions in the terms have been found, the loop counters are reset in Step 8 to reevaluate the affected additions.

In Step 2 we identify the addends that are used in the sum calculated in the current part of the addition table. If an addend is a partial product, we construct the corresponding term, normalize it, and propagate the variable-term equation in Step 3. If an addend is a subtotal or an overflow, we already constructed and processed the term in a previous iteration of the loop. Steps 4, 5, and 6 perform this term construction, normalization, and processing for the current subtotal and overflow.

If in the local subproblem only a few variables are fixed such that only a few propagations and substitutions can be performed, the terms for the subtotals and overflows can grow very quickly. This results in a large consumption of memory and processing time. In order to avoid such a large resource consumption, we abort the algorithm if the number of addends in a normalized subtotal term exceeds a certain value. In our implementation, we use the limit $\mathsf{maxaddends} = 20$.

The propagation of a variable-term equation $\xi = t(z_1, \ldots, z_m)$, $t \in \mathcal{T}_\Sigma$, which is performed in Steps 3c and 6 of Algorithm 14.10 is depicted in Algorithm 14.11. If the term is equal to $t = 0$ or $t = 1$, the variable ξ can be fixed to the corresponding value in Step 1. If the term consists of only one variable symbol z_1 and the variable ξ is fixed, we can also fix z_1 to the same value in Step 2a. If $\xi = 1$, we can even fix all term variables to $z_i = 1$ in Step 2b if the term has only one addend. If the term only consists of one addend π_1 but no fixings could be deduced, we can at least replace all occurrences of π_1 in the other terms produced by Algorithm 14.10 with ξ. This substitution is performed in Step 2c. Since $1 \oplus \pi = \neg\,\pi$, we can also apply

Algorithm 14.10 Multiplication Domain Propagation – Symbolic Propagation

Input: Multiplication constraint $r = \text{MULT}(x, y)$ on registers r, x, and y of width β with current local bit bounds $\tilde{l}_{r_b} \leq r_b \leq \tilde{u}_{r_b}$, $\tilde{l}_{x_b} \leq x_b \leq \tilde{u}_{x_b}$, and $\tilde{l}_{y_b} \leq y_b \leq \tilde{u}_{y_b}$; current local bounds on auxiliary variables $\tilde{l}_{p_{ij}} \leq p_{ij} \leq \tilde{u}_{p_{ij}}$, $\tilde{l}_{s_j^k} \leq s_j^k \leq \tilde{u}_{s_j^k}$, and $\tilde{l}_{o_j^k} \leq o_j^k \leq \tilde{u}_{o_j^k}$; Parameter $\mathsf{maxaddends} \in \mathbb{N}$.

Output: Tightened local bounds for bits r_b, x_b, y_b, and auxiliary variables p_{ij}, s_j^k, and o_j^k.

1. Set $j := 0$ and $k := 0$.
2. Identify the addends $a_j^{k0}, a_j^{k1}, a_j^{k2}$ of column j, block k by equation (14.19).
3. For all q with $a_j^{kq} = p_{bb'}$ for bits b and b':
   (a) Assign the term $t[p_{bb'}] := x_b \wedge y_{b'}$ as in (14.20).
   (b) Normalize $t[p_{bb'}]$ by calling Algorithm 14.9.
   (c) Propagate $p_{bb'} = t[p_{bb'}]$ with Algorithm 14.11.
4. Assign the terms
$$\begin{aligned} & t[s_j^k] := t[a_j^{k0}] \oplus t[a_j^{k1}] \oplus t[a_j^{k2}] \\ \text{and} \quad & t[o_j^k] := (t[a_j^{k0}] \wedge t[a_j^{k1}]) \oplus (t[a_j^{k0}] \wedge t[a_j^{k2}]) \oplus (t[a_j^{k1}] \wedge t[a_j^{k2}]) \end{aligned}$$
   as in (14.21) and (14.22).
5. Normalize $t[s_j^k]$ and $o[s_j^k]$ by calling Algorithm 14.9.
6. Propagate $s_j^k = t[s_j^k]$ and $o_j^k = t[o_j^k]$ with Algorithm 14.11.
7. If $t[s_j^k] = \pi_1 \oplus \ldots \oplus \pi_n$ and $n >$ maxaddends, stop.
8. If at least one of the calls of Algorithm 14.11 produced a fixing of a variable or a substitution in a term, set j to be the minimal column and set k to be the minimal block number in this column for which a participating term was affected.
   Otherwise, set $k := k + 1$. If $k \geq j$, set $j := j + 1$ and $k := 0$.
9. If $j < \beta$, goto Step 2.

the reasoning of Step 2 to $t = 1 \oplus \pi_2$, which is done in Step 3. However, we have to consider ξ in its negated version since

$$\xi = 1 \oplus \pi_2(z_1, \ldots, z_m) \;\Leftrightarrow\; \neg\xi = \pi_2(z_1, \ldots, z_m).$$

If $t = \pi_1 \oplus \pi_2$ consists of exactly two addends and ξ is fixed, we can draw the following conclusions in Step 4. If $\xi = 0$, it follows $\pi_1 = \pi_2$ which allows propagation in Step 4a or substitution in Step 4c, depending on whether π_1 is a single variable symbol or not. If $\xi = 1$, analogous reasoning can be applied in Steps 4b and 4d. Again, a term $t = 1 \oplus \pi_2 \oplus \pi_3$ can be processed in the same fashion by negating the value of ξ, which is performed in the final Step 5.

Algorithm 14.11 Multiplication Domain Propagation – Variable-Term Equation

Input: Equation $\xi = t(z_1, \ldots, z_m)$ with ξ being a binary CIP variable and $t \in \mathcal{T}_\Sigma$ being a normalized term $t = \pi_1 \oplus \ldots \oplus \pi_n$ with 0-ary symbols $z_1, \ldots, z_m \in O_0$; current local bit bounds $\tilde{l}_\xi \leq \xi \leq \tilde{u}_\xi$ and $\tilde{l}_{z_j} \leq z_j \leq \tilde{u}_{z_j}$ for the CIP variables corresponding to the involved 0-ary symbols.

Output: Tightened local bounds for bits $\xi, z_1, \ldots, z_m$; Modifications of the terms $s \in T$ produced by Algorithm 14.10.

1. If $t = 0$, deduce $\xi = 0$.
   If $t = 1$, deduce $\xi = 1$.
2. If $t = \pi_1$:
   (a) If $\pi_1 = z_1$ and $\tilde{u}_\xi = 0$, deduce $z_1 = 0$.
   (b) If $\pi_1 = z_1 \wedge \ldots \wedge z_m$ and $\tilde{l}_\xi = 1$, deduce $z_j = 1$ for all $j = 1, \ldots, m$.
   (c) If neither 2a nor 2b was applied and $\xi \in O_0$, substitute $t \to \xi$ in all terms $s \in T$.
3. If $t = 1 \oplus \pi_2$:
   (a) If $\pi_2 = z_1$ and $\tilde{l}_\xi = 1$, deduce $z_1 = 0$.
   (b) If $\pi_2 = z_1 \wedge \ldots \wedge z_m$ and $\tilde{u}_\xi = 0$, deduce $z_j = 1$ for all $j = 1, \ldots, m$.
   (c) If neither 3a nor 3b was applied and $\xi \in O_0$, substitute $t \to \xi$ in all terms $s \in T$.
4. If $t = \pi_1 \oplus \pi_2$:
   (a) If $\pi_1 = z_j$ and $\tilde{u}_\xi = 0$, propagate $z_j = \pi_2$ as in Step 2.
   (b) If $\pi_1 = z_j$ and $\tilde{l}_\xi = 1$, propagate $z_j = 1 \oplus \pi_2$ as in Step 3.
   (c) If $\pi_1 \notin O_0$ and $\tilde{u}_\xi = 0$, substitute $\pi_2 \to \pi_1$ in all terms $s \in T$.
   (d) If $\pi_1 \notin O_0$ and $\tilde{l}_\xi = 1$, substitute $\pi_2 \to 1 \oplus \pi_1$ in all terms $s \in T$.
5. If $t = 1 \oplus \pi_2 \oplus \pi_3$:
   (a) If $\pi_2 = z_j$ and $\tilde{l}_\xi = 1$, propagate $z_j = \pi_3$ as in Step 2.
   (b) If $\pi_2 = z_j$ and $\tilde{u}_\xi = 0$, propagate $z_j = 1 \oplus \pi_3$ as in Step 3.
   (c) If $\pi_2 \notin O_0$ and $\tilde{l}_\xi = 1$, substitute $\pi_3 \to \pi_2$ in all terms $s \in T$.
   (d) If $\pi_2 \notin O_0$ and $\tilde{u}_\xi = 0$, substitute $\pi_3 \to 1 \oplus \pi_2$ in all terms $s \in T$.

14.5.3 Presolving

The presolving of multiplication constraints is performed as illustrated in Algorithm 14.12. Similar to the presolving of addition constraints, we first check for very easy situations. If one of the operands is fixed to zero, the resultant can also be fixed to zero in Step 1a, and the multiplication constraint can be deleted. If an operand is fixed to one, the resultant must always be equal to the other operand. They are aggregated in Step 1b, which also leads to the deletion of the constraint. If the resultant is zero, we could conclude that one of the operands must be zero if the constraint would be an ordinary multiplication $r = x \cdot y$. However, this is not necessarily true in the truncated multiplication. We can only fix an operand to zero in Step 1c if the least significant bit of the other operand is one:

Algorithm 14.12 Multiplication Presolving

1. For all active multiplication constraints $r = \text{MULT}(x, y)$:
   (a) If $x = 0$, fix $r := 0$ and delete the constraint.
   If $y = 0$, fix $r := 0$ and delete the constraint.
   (b) If $x = 1$, aggregate $r \overset{\star}{:=} y$ and delete the constraint.
   If $y = 1$, aggregate $r \overset{\star}{:=} x$ and delete the constraint.
   (c) If $r = 0$ and $x_0 = 1$, fix $y := 0$ and delete the constraint.
   If $r = 0$ and $y_0 = 1$, fix $x := 0$ and delete the constraint.
   (d) If $r \overset{\star}{=} x$ and $x_0 = 1$, fix $y := 1$ and delete the constraint.
   If $r \overset{\star}{=} y$ and $y_0 = 1$, fix $x := 1$ and delete the constraint.
   (e) If $\beta = 1$, replace the constraint by $r = \text{AND}(x, y)$.
   (f) If maximal non-zero bit of y is less significant than the one of x, i.e., $\max\{b \mid u_{y_b} = 1\} < \max\{b \mid u_{x_b} = 1\}$, replace the constraint by $r = \text{MULT}(y, x)$.
   (g) Apply LP domain propagation Algorithm 14.7 on the global bounds.
   (h) Apply binary domain propagation Algorithm 14.8 on the global bounds, but additionally:
      i. For all partial products $p_{ij} = x_i \cdot y_j$, $i + j < \beta$, apply presolving Algorithm 14.14 for AND constraints.
      ii. For all columns j and blocks k in the binary multiplication table of Figure 14.4, apply bit aggregation Algorithm 14.5 of ADD constraints.
   (i) Apply symbolic domain propagation Algorithm 14.10 on the global bounds. Whenever a substitution $t \to u$ in Algorithm 14.11 is applied:
      i. If $t = \xi_1$ and $u = \xi_2$, or $t = 1 \oplus \xi_1$ and $u = 1 \oplus \xi_2$ with CIP variables ξ_1, ξ_2, aggregate $\xi_1 \overset{\star}{:=} \xi_2$.
      ii. If $t = \xi_1$ and $u = 1 \oplus \xi_2$, or $t = 1 \oplus \xi_1$ and $u = \xi_2$ with CIP variables ξ_1, ξ_2, aggregate $\xi_1 \overset{\star}{:=} 1 - \xi_2$.

2. For all pairs of active multiplication constraints $r = \text{MULT}(x, y)$ and $r' = \text{MULT}(x', y')$ with $\beta_r \geq \beta_{r'}$:
   (a) For all $b = 0, \ldots, \beta_{r'} - 1$:
      i. If $x[b, 0] \overset{\star}{=} x'[b, 0]$ and $y[b, 0] \overset{\star}{=} y'[b, 0]$, aggregate $r[b, 0] \overset{\star}{:=} r'[b, 0]$.
      ii. If $x[b, 0] \overset{\star}{=} y'[b, 0]$ and $y[b, 0] \overset{\star}{=} x'[b, 0]$, aggregate $r[b, 0] \overset{\star}{:=} r'[b, 0]$.
   (b) If Step 2(a)i or 2(a)ii was successfully applied for $b = \beta_{r'} - 1$, delete the constraint $r' = \text{MULT}(x', y')$.
   (c) If $\beta_r = \beta_{r'}$, $x \overset{\star}{=} x'$, but $r \overset{\star}{\neq} r'$, deduce $y \overset{\star}{\neq} y'$.
   If $\beta_r = \beta_{r'}$, $y \overset{\star}{=} y'$, but $r \overset{\star}{\neq} r'$, deduce $x \overset{\star}{\neq} x'$.
   (d) If $\beta_r = \beta_{r'}$, $x \overset{\star}{=} y'$, but $r \overset{\star}{\neq} r'$, deduce $y \overset{\star}{\neq} x'$.
   If $\beta_r = \beta_{r'}$, $y \overset{\star}{=} x'$, but $r \overset{\star}{\neq} r'$, deduce $x \overset{\star}{\neq} y'$.

Proposition 14.24. Let $\beta \in \mathbb{Z}_{>0}$ and $p, q \in \mathbb{Z}_{\geq 0}$ be two integers with p being odd, $q < 2^\beta$, and $0 = (p \cdot q) \bmod 2^\beta$. Then, it follows $q = 0$.

Proof. By definition of the modulus operator we have

$$0 = (p \cdot q) \bmod 2^\beta \quad \Leftrightarrow \quad \exists k \in \mathbb{Z} : 2^\beta k = p \cdot q.$$

Since p is odd and $2^\beta k$ is even for any $k \in \mathbb{Z}$, q must be even. For $\beta = 1$ this means $q = 0$. For larger β we divide both sides of the equation by 2, which yields

$$\exists k \in \mathbb{Z} : 2^{\beta-1} k = p \cdot \frac{q}{2} \quad \Leftrightarrow \quad 0 = (p \cdot \frac{q}{2}) \bmod 2^{\beta-1}.$$

Since $\frac{q}{2} \in \mathbb{Z}_{\geq 0}$ and $\frac{q}{2} < 2^{\beta-1}$, it follows $\frac{q}{2} = 0$ by induction and therefore $q = 0$. □

If $r = 0$ and $b \geq 1$ is the least significant bit for which $x_b = 1$, we can only conclude that $y[\beta - 1 - b, 0] = 1$, since the more significant bits in y do not affect the resultant if $x_{b'} = 0$ for all $b' < b$. Such deductions are already detected in Step 1h. Therefore, we only consider the cases $x_0 = 1$ and $y_0 = 1$ in Step 1c which in addition to the fixings allow the deletion of the multiplication constraint.

The equivalences $r \overset{\star}{=} x$ and $r \overset{\star}{=} y$ are treated in Step 1d. For $r \overset{\star}{=} x$ we can conclude

$$x = \text{MULT}(x, y) \quad \Leftrightarrow \quad x = (x \cdot y) \bmod 2^\beta \quad \Leftrightarrow \quad 0 = \big(x \cdot (y - 1)\big) \bmod 2^\beta,$$

and by Proposition 14.24 it follows $y = 1$ if the least significant bit of x is $x_0 = 1$. Again, the deductions for $x_b = 1$ with $b \geq 1$ are performed in Step 1h.

The special situation of a single-bit multiplication is addressed in Step 1e. For $\beta = 1$ we have $\text{MULT}(x, y) = \text{AND}(x, y)$. Since the AND constraint comes with an LP relaxation without auxiliary variables (see Section 14.7.1) and features a much less cumbersome propagation algorithm, we replace single-bit MULT constraints by AND constraints.

In the LP relaxation of the multiplication constraint $r = \text{MULT}(x, y)$, the second operand y is split into nibbles while the first operand x is used in its bit representation. Thus, the interchange of the operands leads to a different LP relaxation. Different behavior can also appear in the domain propagation algorithms, since for $r = \text{MULT}(y, x)$ the multiplication tables of Figures 14.3 and 14.4 would change as well as the definition of the auxiliary variables. We experimented with different heuristic strategies for switching operands. The results indicated that the best strategy is to select the first operand x to be the one that has more high-significant bits fixed to zero. Therefore, the constraint $r = \text{MULT}(x, y)$ is replaced by $r = \text{MULT}(y, x)$ in Step 1f if the position of the highest bit not fixed to zero is smaller in y than in x.

Step 1g applies domain propagation on the multiplication table of the LP relaxation. We do not perform additional aggregations on the auxiliary variables of the LP relaxation for the following reason. The nibbles are not represented as CIP variables. Instead, they are only shortcuts for a subword of the register's bit string. Therefore, most of the possible aggregations would be *multi-aggregations* which are aggregations with more than one variable on the right hand side. In the current version of SCIP it is not possible to apply local bound changes to multi-aggregated variables, since this can only be realized by changing the left and right hand sides of the corresponding inequality, which is not supported by the data structures of SCIP. Take the aggregation $p_b^{\langle l \rangle} \overset{\star}{=} y^{\langle l \rangle}$ as an example, which could be applied if

$x_b = 1$. In fact, this means to multi-aggregate $p_b^{\langle l \rangle} \stackrel{\star}{=} \sum_{i=0}^{\delta-1} 2^i y_{lL+i}$, which produces the additional constraint

$$l_{p_b^{\langle l \rangle}} \leq \sum_{i=0}^{\delta-1} 2^i y_{lL+i} \leq u_{p_b^{\langle l \rangle}} \tag{14.24}$$

to reflect the bounds of $p_b^{\langle l \rangle}$. Changing the local bounds of $p_b^{\langle l \rangle}$ means to modify the left and right hand side of inequality (14.24).

In Step 1h of Algorithm 14.12 we perform the domain propagation on the binary multiplication table. Additionally, aggregations are possible for the partial product equations (14.17) and the subtotal equations (14.18). In the symbolic domain propagation of Step 1i, we can also perform aggregations whenever a substitution was found that identifies two terms containing only a single register bit or auxiliary binary variable each.

Pairs of MULT constraints are compared in Step 2. We can aggregate the subwords of the resultants $r[b,0] :\stackrel{\star}{=} r'[b,0]$ in Step 2a if the corresponding subwords of the operands in the two constraints are pairwise equivalent in any order. If this aggregation was successfully performed on the full width of r', we can delete the second constraint in Step 2b. Steps 2c and 2d deduce the inequality of the operands by applying the implications of 2(a)i and 2(a)ii in the opposite direction. This is only possible if the resultants are of equal width. Otherwise, the inequality of the resultants may result from the different truncation of the product.

14.6 Bitwise Negation

The bitwise negation operation

$$\text{NOT} : [\beta] \to [\beta], \quad x \mapsto r = \text{NOT}(x)$$

negates each individual bit such that

$$r = \text{NOT}(x) \quad \Leftrightarrow \quad \forall b \in \{0, \ldots, \beta - 1\} : r_b = 1 - x_b.$$

In order to implement such a constraint, we just have to aggregate the variables $r_b :\stackrel{\star}{=} 1 - x_b$, $b = 0, \ldots, \beta - 1$, in the presolving stage of SCIP. Afterwards, the constraint can be deleted from the problem formulation.

14.7 Bitwise And

The bitwise AND combination of two bit vectors $x, y \in \{0, \ldots, 2^{\beta-1}\}$ with $x = \sum_{b=0}^{\beta-1} 2^b x_b$ and $y = \sum_{b=0}^{\beta-1} 2^b y_b$, $x_b, y_b \in \{0, 1\}$ for all b, is defined as

$$\text{AND} : [\beta] \times [\beta] \to [\beta], \quad (x, y) \mapsto r = \text{AND}(x, y)$$

with

$$r = \text{AND}(x, y) \quad \Leftrightarrow \quad \forall \in \{0, \ldots, \beta - 1\} : r_b = x_b \wedge y_b.$$

Note that the individual bits of an AND constraint are completely independent. This is reflected in the LP relaxation and the domain propagation and presolving algorithms presented in this section. In fact, in our implementation bitwise AND constraints are actually disaggregated into separate single-bit AND constraints on binary variables, which are supported by SCIP. Nevertheless, we will treat them here in their aggregated form in favor of a unified presentation of the operators.

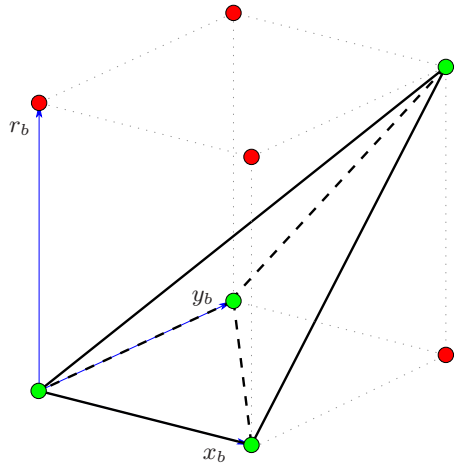

Figure 14.5. LP relaxation of $r_b = x_b \wedge y_b$.

14.7.1 LP Relaxation

We use the following well-known LP relaxation of bitwise AND constraints $r = \text{AND}(x, y)$, see, e.g., Brinkmann and Drechsler [55]:

$$\begin{aligned} r_b - x_b &\leq 0 \quad \text{for all } b = 0, \ldots, \beta - 1 \\ r_b - y_b &\leq 0 \quad \text{for all } b = 0, \ldots, \beta - 1 \\ -r_b + x_b + y_b &\leq 1 \quad \text{for all } b = 0, \ldots, \beta - 1. \end{aligned} \tag{14.25}$$

The first inequality models the implication $x_b = 0 \rightarrow r_b = 0$, the second one models $y_b = 0 \rightarrow r_b = 0$, and the third inequality models $x_b = 1 \wedge y_b = 1 \rightarrow r_b = 1$.

Figure 14.5 shows the convex hull of the integer feasible point for a single bit equation $r_b = x_b \wedge y_b$. One can see that the facets of the polyhedron are given by inequalities (14.25) and the lower bound $r_b \geq 0$ of the resultant bit. In this sense, the LP relaxation (14.25) is "optimal", since it completely describes the non-trivial facets of the substructure represented by the constraint.

14.7.2 Domain Propagation

The domain propagation of bitwise AND constraints is straightforward. For each bit b we propagate the equation

$$r_b = x_b \wedge y_b. \tag{14.26}$$

This is illustrated in Algorithm 14.13. If one of the operand bits is zero, the resultant bit can be also fixed to zero in Step 1a. If both operands are one, the resultant must also be one, see Step 1b. Conversely, if the resultant bit is fixed to one, both operand bits can be fixed to one in Step 1c. Finally, if the resultant bit is zero and one of the operands is one, then the other operand must be zero, see Step 1d.

14.7.3 Presolving

Like the domain propagation and the LP relaxation, the presolving algorithm is applied independently on each bit of the registers involved in the bitwise AND constraints. Algorithm 14.14 illustrates the procedure. As usual in the presolving stage,

Algorithm 14.13 Bitwise And Domain Propagation

Input: Bitwise AND constraint $r = \text{AND}(x, y)$ on registers r, x, and y of width β with current local bit bounds $\tilde{l}_{r_b} \leq r_b \leq \tilde{u}_{r_b}$, $\tilde{l}_{x_b} \leq x_b \leq \tilde{u}_{x_b}$, and $\tilde{l}_{y_b} \leq y_b \leq \tilde{u}_{y_b}$.

Output: Tightened local bounds for bits r_b, x_b, y_b.

1. For all $b = 0, \ldots, \beta - 1$:
   (a) If $\tilde{u}_{x_b} = 0$ or $\tilde{u}_{y_b} = 0$, deduce $r_b = 0$.
   (b) If $\tilde{l}_{x_b} = 1$ and $\tilde{l}_{y_b} = 1$, deduce $r_b = 1$.
   (c) If $\tilde{l}_{r_b} = 1$, deduce $x_b = 1$ and $y_b = 1$.
   (d) If $\tilde{u}_{r_b} = 0$ and $\tilde{l}_{x_b} = 1$, deduce $y_b = 0$.
   If $\tilde{u}_{r_b} = 0$ and $\tilde{l}_{y_b} = 1$, deduce $x_b = 0$.

we apply domain propagation for the global bounds, which is performed in Step 1. If one of the operand bits is fixed to one, the resultant is always equal to the other operand and can be aggregated in Step 2a. If the operand bits are equivalent, the resultant must also take the same value, and the constraint can be deleted in Step 2b. Conversely, if the operand bits are negated equivalent, the resultant is always zero and can be fixed in Step 2c. Steps 2d and 2e add the derivable implications to the implication graph of SCIP, see Section 3.3.5.

Step 3 compares all pairs of existing single-bit equations (14.26). If the operand bits turn out to be pairwise equivalent in any order, the resultant bits can be aggregated. Note that there seem to be additional presolving possibilities in comparing two single-bit equations that have equal or negated resultants. For example, from

Algorithm 14.14 Bitwise And Presolving

1. Apply domain propagation Algorithm 14.13 on the global bounds.
2. For all active bitwise AND constraints $r = \text{AND}(x, y)$ and all involved bits $b = 0, \ldots, \beta_r - 1$:
   (a) If $x_b = 1$, aggregate $r_b :\overset{\star}{=} y_b$.
   If $y_b = 1$, aggregate $r_b :\overset{\star}{=} x_b$.
   (b) If $x_b \overset{\star}{=} y_b$, aggregate $r_b :\overset{\star}{=} x_b$ and delete the constraint.
   (c) If $x_b \overset{\star}{\neq} y_b$, fix $r_b := 0$ and delete the constraint.
   (d) If $r_b = 0$, add implication $x_b = 1 \rightarrow y_b = 0$ to the implication graph of SCIP.
   (e) Add implications $r_b = 1 \rightarrow x_b = 1$ and $r_b = 1 \rightarrow y_b = 1$ to the implication graph of SCIP.
3. For all pairs of active bitwise AND constraints $r = \text{AND}(x, y)$ and $r' = \text{AND}(x', y')$, including pairs with equal constraints, and all $b = 0, \ldots, \beta_r - 1$ and $b' = 0, \ldots, \beta_{r'} - 1$:
   (a) If $x_b \overset{\star}{=} x'_{b'}$ and $y_b \overset{\star}{=} y'_{b'}$, aggregate $r_b :\overset{\star}{=} r'_{b'}$.
   If $x_b \overset{\star}{=} y'_{b'}$ and $y_b \overset{\star}{=} x'_{b'}$, aggregate $r_b :\overset{\star}{=} r'_{b'}$.

the equations

$$r_b = x_b \wedge y_b \quad \text{and} \quad \neg\, r_b = x_b \wedge y_b'$$

we can conclude $x_b = 1$. However, such deductions are automatically detected by the implication graph analysis of SCIP after the implications $r_b = 1 \rightarrow x_b = 1$ and $r_b = 0 \rightarrow x_b = 1$ have been added to the implication graph in Step 2e, see Section 10.7. The same holds in situations where an operator bit is negated in one of the two constraints which are equal in all other respects, e.g.,

$$r_b = x_b \wedge y_b \quad \text{and} \quad r_b = x_b \wedge \neg\, y_b.$$

In this case, we can conclude $r_b = 0$ and $x_b = 0$. Again, the implication graph analysis automatically detects $r_b = 0$ due to the implications $r_b = 1 \rightarrow y_b = 1$ and $r_b = 1 \rightarrow y_b = 0$. Then, the new implications $x_b = 1 \rightarrow y_b = 0$ and $x_b = 1 \rightarrow y_b = 1$ are added in Step 2d of Algorithm 14.14, such that the implication graph analysis can conclude $x_b = 0$.

14.8 Bitwise Or

The bitwise OR combination of two bit vectors $x, y \in \{0, \dots, 2^{\beta-1}\}$ with $x = \sum_{b=0}^{\beta-1} 2^b x_b$ and $y = \sum_{b=0}^{\beta-1} 2^b y_b$, $x_b, y_b \in \{0, 1\}$ for all b, is defined as

$$\text{OR} : [\beta] \times [\beta] \rightarrow [\beta], \quad (x, y) \mapsto r = \text{OR}(x, y)$$

with

$$r = \text{OR}(x, y) \quad \Leftrightarrow \quad \forall b \in \{0, \dots, \beta - 1\} : r_b = x_b \vee y_b.$$

For each individual bit we can transform the constraint to

$$r_b = x_b \vee y_b \quad \Leftrightarrow \quad \neg\, r_b = \neg\, x_b \wedge \neg\, y_b. \tag{14.27}$$

In terms of circuit operators, we can rewrite

$$r = \text{OR}(x, y) \quad \Leftrightarrow \quad \text{NOT}(r) = \text{AND}(\text{NOT}(x), \text{NOT}(y)).$$

This symmetry is also reflected in the convex hull of the feasible points of the single-bit equations (14.27). Comparing Figures 14.5 and 14.6, one can see that they are point-symmetric to each other with $(\frac{1}{2}, \frac{1}{2}, \frac{1}{2})$ being the reflection center.

Like bitwise AND constraints, bitwise OR constraints are disaggregated into separate single-bit OR constraints on binary variables which are supported by SCIP. Afterwards, SCIP automatically rewrites them as bitwise AND constraints on the negated variables. This has the advantage, that both AND and OR constraints take part in the pairwise presolving Step 3 of the AND presolving Algorithm 14.14.

14.9 Bitwise Xor

Analogous to the AND and OR operators, the bitwise XOR combination of two bit vectors $x, y \in \{0, \dots, 2^{\beta-1}\}$ with $x = \sum_{b=0}^{\beta-1} 2^b x_b$ and $y = \sum_{b=0}^{\beta-1} 2^b y_b$, $x_b, y_b \in \{0, 1\}$ for all b, is defined as

$$\text{XOR} : [\beta] \times [\beta] \rightarrow [\beta], \quad (x, y) \mapsto r = \text{XOR}(x, y)$$

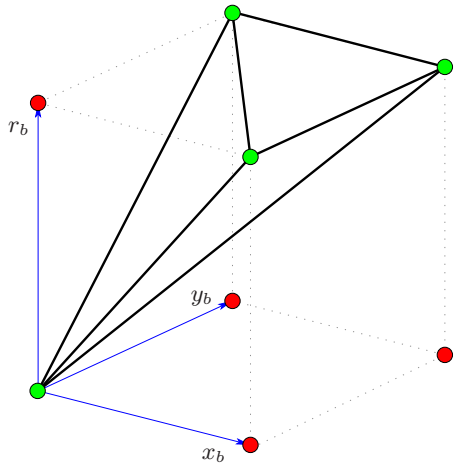

Figure 14.6. LP relaxation of $r_b = x_b \vee y_b$.

with

$$r = \text{XOR}(x, y) \quad \Leftrightarrow \quad \forall b \in \{0, \ldots, \beta - 1\} : r_b = x_b \oplus y_b.$$

We will see, however, that these constraints behave quite different compared to the previously described bitwise combination operands AND and OR. In particular, there is no situation where a single fixing of one of the three variables can be used in domain propagation to deduce fixings on the other variables. This can be seen in Figure 14.7: each facet of the cube $[0, 1]^3$ contains exactly two feasible solutions of the XOR constraint, and these are at diagonally opposite vertices. Thus, if one variable is fixed to any value, there are still the two possibilities $(0, 1)$ and $(1, 0)$, or $(0, 0)$ and $(1, 1)$ left for the other variables.

14.9.1 LP RELAXATION

Figure 14.7 shows the convex hull of the feasible solutions for a single-bit XOR constraint

$$r_b = x_b \oplus y_b. \tag{14.28}$$

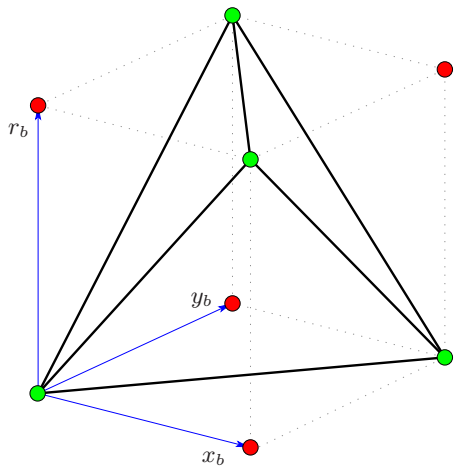

Figure 14.7. LP relaxation of $r_b = x_b \oplus y_b$.

Algorithm 14.15 Bitwise Xor Domain Propagation

Input: Bitwise XOR constraint $r = \text{XOR}(x, y)$ on registers r, x, and y of width β with current local bit bounds $\tilde{l}_{r_b} \leq r_b \leq \tilde{u}_{r_b}$, $\tilde{l}_{x_b} \leq x_b \leq \tilde{u}_{x_b}$, and $\tilde{l}_{y_b} \leq y_b \leq \tilde{u}_{y_b}$.

Output: Tightened local bounds for bits r_b, x_b, y_b.

1. For all $b = 0, \ldots, \beta - 1$:
   (a) If two of the three bits in equation (14.28) are fixed, deduce the corresponding value for the remaining variable.

All of the four facets are non-trivial, i.e., they are not orthogonal to a unit vector. Therefore, we need four "real" inequalities (as opposed to the bounds of the variables) to describe the LP relaxation of the constraint:

$$\begin{aligned} r_b - x_b - y_b &\leq 0 \\ -r_b + x_b - y_b &\leq 0 \\ -r_b - x_b + y_b &\leq 0 \\ r_b + x_b + y_b &\leq 2. \end{aligned} \tag{14.29}$$

Each inequality cuts off one infeasible vertex of the unit cube. The first inequality cuts off $(1, 0, 0)$, the second $(0, 1, 0)$, the third $(0, 0, 1)$, and the fourth inequality cuts off $(1, 1, 1)$. Again, the LP relaxation (14.29) is "optimal" in the sense that it describes all facets of the convex hull of feasible solutions for the XOR constraint.

14.9.2 Domain Propagation

As already mentioned, the domain propagation of the XOR constraint is rather weak. The value of a variable cannot be decided until all other variables are fixed. Therefore, the domain propagation consists of only one step, as depicted in Algorithm 14.15.

14.9.3 Presolving

The presolving of bitwise XOR constraints is illustrated in Algorithm 14.16. Since $r_b = x_b \oplus y_b \Leftrightarrow 0 = r_b \oplus x_b \oplus y_b$, the resultant bit r_b does not play a special role as in the AND constraint. If any pair of variables is equivalent, the third variable can be fixed to zero in Step 1a. On the other hand, if any pair of variables is negated equivalent, the third variable can be fixed to one in Step 1b. If any of the bits is fixed to zero, the other two have to be equal and can be aggregated in Step 1c. Conversely, if any of the bits is fixed to one, the other two must be opposite and can be aggregated accordingly in Step 1d. The domain propagation Algorithm 14.15 does not need to be called since the deductions applied therein are already covered by Steps 1c and 1d.

The presolving in Step 2 for pairs of constraints is slightly more involved than the one for bitwise AND constraints. As before, we treat the constraints for each bit b in the form $0 = r_b \oplus x_b \oplus y_b$ and compare all pairs of those equations. If there are two pairs of equivalent or negated equivalent binary variables, the remaining pair of variables can be aggregated.

Algorithm 14.16 Bitwise Xor Presolving

1. For all active bitwise XOR constraints $r = \text{XOR}(x, y)$:
   (a) If $r_b \overset{\star}{=} x_b$, fix $y_b := 0$.
   If $r_b \overset{\star}{=} y_b$, fix $x_b := 0$.
   If $x_b \overset{\star}{=} y_b$, fix $r_b := 0$.
   (b) If $r_b \overset{\star}{\neq} x_b$, fix $y_b := 1$.
   If $r_b \overset{\star}{\neq} y_b$, fix $x_b := 1$.
   If $x_b \overset{\star}{\neq} y_b$, fix $r_b := 1$.
   (c) If $r_b = 0$, aggregate $x_b :\overset{\star}{=} y_b$.
   If $x_b = 0$, aggregate $r_b :\overset{\star}{=} y_b$.
   If $y_b = 0$, aggregate $r_b :\overset{\star}{=} x_b$.
   (d) If $r_b = 1$, aggregate $x_b :\overset{\star}{=} 1 - y_b$.
   If $x_b = 1$, aggregate $r_b :\overset{\star}{=} 1 - y_b$.
   If $y_b = 1$, aggregate $r_b :\overset{\star}{=} 1 - x_b$.
2. For all pairs of active bitwise XOR constraints $r = \text{XOR}(x, y)$ and $r' = \text{XOR}(x', y')$, including pairs with equal constraints, and all $b = 0, \ldots, \beta_r - 1$ and $b' = 0, \ldots, \beta_{r'} - 1$:
   If (ξ, ψ, φ) is any permutation of (r_b, x_b, y_b), (ξ', ψ', φ') is any permutation of (r'_b, x'_b, y'_b), and
   (a) if $\xi \overset{\star}{=} \xi'$ and $\psi \overset{\star}{=} \psi'$, aggregate $\varphi :\overset{\star}{=} \varphi'$,
   (b) if $\xi \overset{\star}{=} \xi'$ and $\psi \overset{\star}{\neq} \psi'$, aggregate $\varphi :\overset{\star}{=} 1 - \varphi'$,
   (c) if $\xi \overset{\star}{\neq} \xi'$ and $\psi \overset{\star}{\neq} \psi'$, aggregate $\varphi :\overset{\star}{=} \varphi'$.

14.10 Unary And

The unary logic operators UAND, UOR, and UXOR combine all bits of one register ϱ to calculate a single resultant bit. Thus, they are the first operators described here that provide a link from multi-bit registers of the data path to single-bit registers of the control logic of a circuit. The UAND constraint

$$\text{UAND} : [\beta] \to [1], \quad x \mapsto r = \text{UAND}(x)$$

is defined by

$$r = \text{UAND}(x) \quad \Leftrightarrow \quad r = x_0 \wedge \ldots \wedge x_{\beta-1}.$$

As mentioned in Section 14.7, the binary bitwise AND constraints are disaggregated into β single-bit constraints $r_b = x_b \wedge y_b$. In this regard, the unary UAND constraint is just a generalization of this single-bit equation to arbitrary many variables in the conjunction. In fact, in SCIP we only treat the general case

$$r = x_0 \wedge \ldots \wedge x_{k-1}$$

of conjunctions of binary variables. The equations for the individual bits in a binary bitwise AND constraint are just conjunctions with $k = 2$, while UAND constraints are conjunctions with $k = \beta$. Therefore, the LP relaxation and the domain propagation and presolving algorithms of Section 14.7 are just special incarnations of the LP relaxation and algorithms described below.

Algorithm 14.17 Unary And Domain Propagation

Input: Unary AND constraint $r = \text{UAND}(x)$ on single-bit register r and multi-bit register x of width β with current local bit bounds $\tilde{l}_r \leq r \leq \tilde{u}_r$ and $\tilde{l}_{x_b} \leq x_b \leq \tilde{u}_{x_b}$, $b = 0, \ldots, \beta - 1$.

Output: Tightened local bounds for bits r and x_b.

1. If $\tilde{u}_{x_b} = 0$ for any $b \in \{0, \ldots, \beta - 1\}$, deduce $r = 0$.
2. If $\tilde{l}_{x_b} = 1$ for all $b = 0, \ldots, \beta - 1$, deduce $r = 1$.
3. If $\tilde{l}_r = 1$, deduce $x_b = 1$ for all $b = 0, \ldots, \beta - 1$.
4. If $\tilde{u}_r = 0$ and $\tilde{l}_{x_b} = 1$ for all $b \neq k$, $k \in \{0, \ldots, \beta - 1\}$, deduce $x_k = 0$.

14.10.1 LP Relaxation

The LP relaxation of a UAND constraint $r = \text{UAND}(x)$ can be stated as

$$r - x_b \leq 0 \qquad \text{for all } b = 0, \ldots, \beta - 1 \tag{14.30}$$

$$-r + \sum_{b=0}^{\beta-1} x_b \leq \beta - 1. \tag{14.31}$$

The first inequality enforces the implication $x_b = 0 \rightarrow r = 0$ for all bits b, while the second inequality represents the implication $(\forall b : x_b = 1) \rightarrow r = 1$.

Lemma 14.25. The convex hull $P_{\text{UAND}} = \text{conv}\{(r, x) \mid r \in \{0,1\}, x \in \{0,1\}^\beta, r = \text{UAND}(x)\}$ of the feasible solutions for the UAND constraint is full-dimensional if $\beta \geq 2$.

Proof. The feasible solutions $(r = 0, x = 0)$, $(r = 0, x_b = 1, x_i = 0 \text{ for } i \neq b)$, $b = 0, \ldots, \beta - 1$, and $(r = 1, x = 1)$ define $\beta + 2$ affinely independent vectors in $\mathbb{R}^{\beta+1}$. Thus, the dimension of P_{UAND} is $\dim(P_{\text{UAND}}) = \beta + 1$. □

Proposition 14.26. Inequalities (14.30) and (14.31) and the lower bound $r \geq 0$ define facets of P_{UAND} if $\beta \geq 2$.

Proof. P_{UAND} is full-dimensional as shown in Lemma 14.25. Thus, it suffices for each inequality to provide $\beta + 1$ affinely independent feasible solution vectors that fulfill the inequality with equality. The solutions $(r = 0, x = 0)$, $(r = 0, x_i = 1, x_j = 0 \text{ for } j \neq i)$, $i \in \{0, \ldots, \beta - 1\} \setminus \{b\}$, and $(r = 1, x = 1)$ are affinely independent and fulfill inequality (14.30) for bit b with equality. The solutions $(r = 0, x_b = 0, x_i = 1 \text{ for } i \neq b)$, $b = 0, \ldots, \beta - 1$, and $(r = 1, x = 1)$ are also affinely independent and fulfill inequality (14.31) with equality. The solutions $(r = 0, x = 0)$ and $(r = 0, x_b = 1, x_i = 0 \text{ for } i \neq b)$, $b = 0, \ldots, \beta - 1$, are affinely independent and fulfill $r \geq 0$ with equality. □

14.10.2 Domain Propagation

The domain propagation Algorithm 14.17 is the canonical generalization of Algorithm 14.13. If a bit in the operand is zero, the resultant can be fixed to zero in Step 1. If all bits of the operand are one, the resultant must also be one, see Step 2. Conversely, if the resultant is fixed to one, all bits of the operand can be fixed to

Algorithm 14.18 Unary And Presolving

1. For all active unary AND constraints $r = \text{UAND}(x)$ represented as $r = x_0 \wedge \ldots \wedge x_{\beta-1}$:
   (a) Apply domain propagation Algorithm 14.17 on the global bounds. If any deduction was applied, delete the constraint.
   (b) Simplify the constraint as much as possible ($i, j \in \{0, \ldots, \beta - 1\}$):
      i. If $x_i = 1$, remove the variable from the constraint.
      ii. If $x_i \overset{\star}{=} x_j$ for $i \neq j$, remove one of the variables from the constraint.
      iii. If $x_i \overset{\star}{\neq} x_j$ for $i \neq j$, fix $r := 0$ and delete the constraint.

      Note that this step may reduce the number of involved operand bits β.
   (c) If $\beta = 1$, aggregate $r :\overset{\star}{=} x_0$ and delete the constraint.
   (d) If $\beta = 2$ and $r_b = 0$, add implication $x_0 = 1 \rightarrow x_1 = 0$ to the implication graph of SCIP.
   (e) Add implications $r = 1 \rightarrow x_b = 1$ for all $b = 0, \ldots, \beta - 1$ to the implication graph of SCIP.
2. For all pairs of active unary AND constraints $r = \text{UAND}(x)$ and $r' = \text{UAND}(x')$, let $X \subseteq \{x_0, \ldots, x_{\beta_x - 1}\}$ and $X' \subseteq \{x'_0, \ldots, x'_{\beta_{x'} - 1}\}$ be the sets of remaining operand bits after applying the simplifications of Step 1b.
   (a) If for each $x_b \in X$ there exists an $x'_{b'} \in X'$ with $x_b \overset{\star}{=} x'_{b'}$, add implication $r' = 1 \rightarrow r = 1$ to the implication graph of SCIP.
   (b) If for each $x'_{b'} \in X'$ there exists an $x_b \in X$ with $x'_{b'} \overset{\star}{=} x_b$, add implication $r = 1 \rightarrow r' = 1$ to the implication graph of SCIP.
   (c) If both implications $r = 1 \rightarrow r' = 1$ and $r' = 1 \rightarrow r = 1$ were added in Steps 2a and 2b, aggregate $r :\overset{\star}{=} r'$ and delete $r' = \text{UAND}(x')$.

one in Step 3. Finally, if the resultant bit is zero and all but one of the operand bits are one, the remaining bit of the operand must be zero, see Step 4.

14.10.3 PRESOLVING

As the domain propagation algorithm, the presolving Algorithm 14.18 is a straightforward generalization of the presolving Algorithm 14.14 for bitwise AND constraints. It only differs slightly in the pairwise comparison of constraints. We apply domain propagation for the global bounds in Step 1a. Afterwards, the constraint is simplified by exploiting the properties

$$a \wedge 1 = a, \quad a \wedge a = a, \quad \text{and} \quad a \wedge \neg a = 0$$

of the $\wedge$ operator, which reduces the number of involved bits β. Note that at least one operand bit remains in the constraint, since otherwise, the domain propagation in Step 1a would already have fixed $r := 1$ and deleted the constraint. If only one unfixed operand bit remained in the simplified constraint, the resultant can be aggregated to this remaining bit in Step 1c. Steps 1d and 1e add the derivable implications to the implication graph of SCIP, see Section 3.3.5.

In the pairwise comparison of Step 2, we look at the sets of operand bits which remained after the simplifications of Step 1b. If—up to equivalence—one is a subset of the other, the larger set defines a stronger restriction on the resultant. Thus, if

the resultant for the larger set is one, then the other resultant must also be one. If both sets are equal up to equivalence, the resultants can be aggregated and one of the constraints can be deleted.

14.11 Unary Or

The unary OR constraint is defined similar to the unary AND constraint as

$$\text{UOR} : [\beta] \to [1], \quad x \mapsto r = \text{UOR}(x)$$

with

$$r = \text{UOR}(x) \quad \Leftrightarrow \quad r = x_0 \vee \ldots \vee x_{\beta-1}.$$

By using the equivalence

$$r = x_0 \vee \ldots \vee x_{\beta-1} \quad \Leftrightarrow \quad \neg\, r = \neg\, x_0 \wedge \ldots \wedge \neg\, x_{\beta-1}$$

we can transform any UOR constraint into an equivalent UAND constraint. This transformation is applied in the presolving stage of SCIP. As already mentioned in Section 14.8, it is advantageous for the pairwise constraint comparison in the presolving algorithm to represent all AND, OR, UAND, and UOR constraints as constraints of the same type.

14.12 Unary Xor

The unary exclusive OR operator

$$\text{UXOR} : [\beta] \to [1], \quad x \mapsto r = \text{UXOR}(x)$$

with

$$r = \text{UXOR}(x) \quad \Leftrightarrow \quad r = x_0 \oplus \ldots \oplus x_{\beta-1}$$

is the generalization of equation (14.28) for the individual bits in a bitwise XOR constraint as discussed in Section 14.9. It inherits the poor domain propagation potential of equation (14.28): a variable involved in a UXOR constraint can only be fixed to a certain value after *all* other variables have been fixed. In contrast to the unary AND and unary OR constraints, the canonical LP relaxation of UXOR constraints consists of exponentially many inequalities. To avoid a blow-up of the LP relaxation, we add an integer auxiliary variable, such that a single equation suffices to model the UXOR constraint. Unfortunately, this relaxation is almost useless if the integrality of the auxiliary variable is not exploited by cutting planes, branching, or conflict analysis.

14.12.1 LP Relaxation

The number of inequalities in the LP relaxation of UAND and UOR constraints is linear in the number of operand bits β. Unfortunately, this is not true for UXOR constraints. Using the equivalence

$$r = x_0 \oplus \ldots \oplus x_{\beta-1} \quad \Leftrightarrow \quad 0 = x_0 \oplus \ldots \oplus x_{\beta-1} \oplus r$$

and identifying $x_\beta \stackrel{\star}{:=} r$, the LP relaxation of a UXOR constraint can be stated as

$$\sum_{b\in S} x_b - \sum_{b\in X\setminus S} x_b \;\leq\; 2k \;\text{ for all } k = 0,\ldots,\lfloor\frac{\beta}{2}\rfloor, S\subseteq X \text{ with } |S| = 2k+1 \quad (14.32)$$

with $X = \{0,\ldots,\beta\}$. Here, each inequality cuts off only one of the infeasible $\{0,1\}$-vectors, such that we need exponentially many inequalities. We cannot do better with inequalities defined in the space $\mathbb{R}^X$ of problem variables, since all inequalities of type (14.32) represent facets of the convex hull

$$P_{\text{UXOR}} = \text{conv}\left\{(r,x)\in\{0,1\}^X \mid r = \text{UXOR}(x)\right\}$$

of integer points that satisfy the constraint:

Lemma 14.27. The convex hull P_{UXOR} of the feasible solutions for the UXOR constraint is full-dimensional if $\beta \geq 2$.

Proof. The feasible solutions $(r = 0, x = 0)$, $(r = 1, x_b = 1, x_i = 0 \text{ for } i \neq b)$, $b = 0,\ldots,\beta-1$, and $(r = 0, x_0 = 1, x_1 = 1, x_b = 0 \text{ for } b = 2,\ldots,\beta-1)$ define $\beta+2$ affinely independent vectors in $\mathbb{R}^{\beta+1}$. Thus, the dimension of P_{UXOR} is $\dim(P_{\text{UXOR}}) = \beta+1$. □

Proposition 14.28. Each inequality (14.32) defines a facet of P_{UXOR} if $\beta \geq 2$. The bounds $x_b \geq 0$ and $x_b \leq 1$ define facets of P_{UXOR} for all $b \in X$ if $\beta \geq 3$.

Proof. P_{UXOR} is full-dimensional as shown in Lemma 14.27. Thus, it suffices for each inequality to provide $\beta+1$ affinely independent solution vectors that fulfill the inequality with equality.

Let $k \in \{0,\ldots,\lfloor\frac{\beta}{2}\rfloor\}$ and $S \subseteq X$ with $|S| = 2k+1$. For $\beta \geq 2$, the vectors

$$x^b = (x^b_i)_{i=0,\ldots,\beta} \quad \text{with} \quad x^b_i = \begin{cases} 1 & \text{if } i \in S\setminus\{b\} \\ 0 & \text{otherwise,} \end{cases}$$

$b = 0,\ldots,\beta$, are $\beta+1$ affinely independent solutions of $0 = x_0 \oplus \ldots \oplus x_\beta$ that fulfill inequality (14.32) for the given k and S with equality. For $\beta \geq 3$, the vector $(0,\ldots,0)$, the vectors

$$x^b = (x^b_i)_{i=0,\ldots,\beta} \quad \text{with} \quad x^b_i = \begin{cases} 1 & \text{if } i \in \{1,b\} \\ 0 & \text{otherwise,} \end{cases}$$

$b = 2,\ldots,\beta$, and the vector $(0,\ldots,0,1,1)$ are $\beta+1$ affinely independent solutions of $0 = x_0 \oplus \ldots \oplus x_\beta$ that fulfill the lower bound $x_0 \geq 0$ with equality. For $\beta \geq 3$, the upper bound $x_0 \leq 1$ is fulfilled with equality by the $\beta+1$ affine independent solution vectors

$$x^b = (x^b_i)_{i=0,\ldots,\beta} \quad \text{with} \quad x^b_i = \begin{cases} 1 & \text{if } i \in \{0,b\} \\ 0 & \text{otherwise,} \end{cases}$$

$b = 1,\ldots,\beta$ and $(1,1,1,1,0,\ldots,0)$. If $\beta \geq 3$, the other bounds are facets for analogous reasons. □

Corollary 14.29. The number of facets of the convex hull P_{UXOR} of feasible solutions of the UXOR constraint is exponential in the number of variables.

Algorithm 14.19 Unary Xor Domain Propagation

Input: Unary XOR constraint $r = \text{UXOR}(x)$ on single-bit register r and multi-bit register x of width β with current local bit bounds $\tilde{l}_r \le r \le \tilde{u}_r$ and $\tilde{l}_{x_b} \le x_b \le \tilde{u}_{x_b}$, $b = 0, \dots, \beta - 1$.

Output: Tightened local bounds for bits r and x_b.

1. If all variables except one are fixed, deduce the corresponding value for the remaining variable.

In order to avoid the exponential blow-up of the LP relaxation, we use system (14.32) only if $\beta = 2$. Otherwise, we introduce an auxiliary integer variable $q \in \{0, \dots, \lceil \frac{\beta}{2} \rceil\}$ and state by the single equation

$$r + x_0 + \ldots + x_{\beta-1} = 2q \tag{14.33}$$

that the number of variables set to one in an UXOR constraint must be even. The disadvantage of equation (14.33) and the introduction of the auxiliary variable q is that this LP relaxation is much weaker than the exponentially large system (14.32). For example, in the case $\beta = 2$, the polyhedron defined by equation (14.33) and the bounds of the variables has the vertices

$$(r, x_0, x_1, q) \in \{(0,0,0,0), (0,1,1,1), (1,0,1,1), (1,1,0,1), (0,0,1,0.5), (0,1,0,0.5), (1,0,0,0.5)\}$$

with the last three being invalid solutions of the constraint. Without enforcing the integrality of q, the LP relaxation 14.33 is almost useless. However, it may become useful during the course of the solution process, if q starts to appear in cutting planes or conflict constraints or if it is selected as branching variable.

14.12.2 Domain Propagation

As already mentioned above, we only apply a single type of propagation to UXOR constraints: if all but one of the variables are fixed, we can deduce the corresponding value for the remaining variable. This procedure is depicted in Algorithm 14.19, which is a generalization of the propagation Step 1a of Algorithm 14.15.

14.12.3 Presolving

The presolving of unary XOR constraints is illustrated in Algorithm 14.20. It is a generalization of Algorithm 14.16 to exclusive disjunctions with arbitrary many addends. In Step 1a we simplify the constraint by removing variables fixed to zero and pairs of equivalent or negated equivalent variables, exploiting the properties

$$a \oplus 0 = a, \quad a \oplus a = 0, \quad \text{and} \quad a \oplus \neg a = 1$$

of the $\oplus$ operator. The "$\oplus 1$" addends appended by rule 1(a)iii are treated like variables which are fixed to one. Thus, if there exists another variable fixed to one in the constraint or if rule 1(a)iii is applied multiple times, the "$\oplus 1$" addends cancel each other by rule 1(a)ii. Thus, at most one fixed variable remains in the

Algorithm 14.20 Unary Xor Presolving

1. For all active unary XOR constraints $r = \text{UXOR}(x)$ represented as $0 = x_0 \oplus \ldots \oplus x_\beta$ with $x_\beta \overset{\star}{=} r$:
   (a) Simplify the constraint as much as possible ($i, j \in \{0, \ldots, \beta\}$):
      i. If $x_i = 0$, remove the variable from the constraint.
      ii. If $x_i \overset{\star}{=} x_j$ for $i \neq j$, remove both variables from the constraint.
      iii. If $x_i \overset{\star}{\neq} x_j$ for $i \neq j$, remove both variables from the constraint and append the addend "$\oplus 1$".

      Note that this step may reduce the number of involved bits $\beta + 1$.
   (b) If the simplified constraint has the form $0 = x_0 \oplus x_1$, aggregate $x_0 :\overset{\star}{=} x_1$.
   (c) If the simplified constraint has the form $0 = x_0 \oplus x_1 \oplus 1$, aggregate $x_0 :\overset{\star}{=} 1 - x_1$.

2. For all pairs of active unary XOR constraints $r = \text{UXOR}(x)$ and $r' = \text{UXOR}(x')$ with $\beta_x \geq \beta_{x'}$:

   Consider the sum of both constraints

   $$0 = x_0 \oplus \ldots \oplus x_{\beta_x} \oplus x'_0 \oplus \ldots \oplus x'_{\beta_{x'}} = \xi_0 \oplus \ldots \oplus \xi_{\beta_x + \beta_{x'} + 1} \tag{14.34}$$

   and perform the simplifications of Step 1a. Let $0 = \psi_0 \oplus \ldots \oplus \psi_{k-1} \oplus c$ with $c \in \{0, 1\}$ be the simplified equation. Apply the following presolving operations:
   (a) If $k = 0$ and $c = 0$, delete one of the constraints.
   (b) If $k = 0$ and $c = 1$, report the infeasibility of the problem.
   (c) If $k = 1$, fix $\psi_0 := c$.
   (d) If $k = 2$ and $c = 0$, aggregate $\psi_0 :\overset{\star}{=} \psi_1$.
   (e) If $k = 2$ and $c = 1$, aggregate $\psi_0 :\overset{\star}{=} 1 - \psi_1$.
   (f) If $k \leq \beta_x$, replace the longer UXOR constraint $0 = x_0 \oplus \ldots \oplus x_{\beta_x}$ with $0 = \psi_0 \oplus \ldots \oplus \psi_{k-1} \oplus c$.

constraint. Steps 1b and 1c check whether there are only two unfixed variables left in the constraint and aggregate them accordingly.

Step 2 compares all pairs of UXOR constraints. We add up the two equations $0 = x_0 \oplus \ldots \oplus x_{\beta_x - 1} \oplus r$ and $0 = x'_0 \oplus \ldots \oplus x'_{\beta_{x'} - 1} \oplus r'$ and simplify the result as in Step 1a. If at most two variables and a constant addend are left in the simplified equation, we can perform the presolving operations depicted in Steps 2a to 2e. If the simplified sum of the constraints has fewer variables than one of the two constraints, the sum can replace the longer constraint in Step 2f. This may trigger additional presolving reductions in subsequent presolving rounds. The replacement of Step 2f is valid due to the following observation:

Observation 14.30. Given two XOR terms $s(x) = x_{s_1} \oplus \ldots \oplus x_{s_m}$ and $t(x) = x_{t_1} \oplus \ldots \oplus x_{t_m}$, it follows

$$s(x) = 0 \wedge t(x) = 0 \quad \Leftrightarrow \quad s(x) \oplus t(x) = 0 \wedge t(x) = 0.$$

Proof. If $s(x) = 0$ and $t(x) = 0$ it follows $s(x) \oplus t(x) = 0 \oplus 0 = 0$. On the other hand, if $s(x) \oplus t(x) = 0$ and $t(x) = 0$ if follows $s(x) = s(x) \oplus 0 = s(x) \oplus t(x) = 0$. □

14.13 Equality

The equality operator

$$\text{EQ} : [\beta] \times [\beta] \to [1], \quad (x, y) \mapsto r = \text{EQ}(x, y)$$

with

$$r = \text{EQ}(x, y) \quad \Leftrightarrow \quad r \leftrightarrow (x = y)$$

provides a very important link from the data path to the control logic of the circuit. Often, x and y are results of some arithmetic computations, and the behavior of the control logic depends on whether the two results are equal or not.

Many property checking problems are modeled like Example 13.2, where the output of a copy of the circuit with different input assignments is compared to the output of the original circuit by an equality constraint. Other common properties compare the circuit with a reference implementation and check whether the two outputs are always equal. In both cases, the property states that equality of the outputs must hold. This means, that in the corresponding CIP model which includes the *negated* property, the resultant of the equality constraint is fixed to zero, and we search for a counter-example where the outputs are *unequal*. Unfortunately, the domain propagation behavior of EQ constraints is much worse for $r = 0$ than for $r = 1$, see Section 14.13.2. The same holds for presolving as discussed in Section 14.13.3. If the resultant is fixed to $r = 1$, we can apply pairwise aggregation of the bits in the two operands and delete the constraint from the model. For $r = 0$, we can only conclude that $x \stackrel{\star}{\neq} y$, which is a very weak information.

The LP relaxation is defined on word level and contains a number of auxiliary variables. Additionally, two "big-M" coefficients are included with values up to 2^W, with $W = 16$ being the width of the words.

14.13.1 LP Relaxation

We use an LP relaxation of the constraint $r = \text{EQ}(x, y)$ on the word level which uses four auxiliary variables $s^w, t^w \in \mathbb{Z}_{\geq 0}$, and $p^w, q^w \in \{0, 1\}$ per word:

$$x^w - y^w = s^w - t^w \qquad \text{for all } w = 0, \ldots, \omega - 1 \tag{14.35}$$

$$s^w \leq (u_{x^w} - l_{y^w}) \cdot p^w \qquad \text{for all } w = 0, \ldots, \omega - 1 \tag{14.36}$$

$$t^w \leq (u_{y^w} - l_{x^w}) \cdot q^w \qquad \text{for all } w = 0, \ldots, \omega - 1 \tag{14.37}$$

$$p^w - s^w \leq 0 \qquad \text{for all } w = 0, \ldots, \omega - 1 \tag{14.38}$$

$$q^w - t^w \leq 0 \qquad \text{for all } w = 0, \ldots, \omega - 1 \tag{14.39}$$

$$r + p^w + q^w \leq 1 \qquad \text{for all } w = 0, \ldots, \omega - 1 \tag{14.40}$$

$$r + \sum_{w=0}^{\omega-1} (p^w + q^w) \geq 1 \tag{14.41}$$

If $\omega = 1$, we can replace (14.40) and (14.41) by the equation $r + p^0 + q^0 = 1$.

Equation (14.35) splits the difference $x^w - y^w$ into the positive part s^w and the negative part t^w. The binary variable p^w should be one if and only if $s^w > 0$ which means $x^w > y^w$. The binary variable q^w should be one if and only if $t^w > 0$ which means $x^w < y^w$. Thus, the variables p^w and q^w reflect the sign of the difference $x^w - y^w$. These relations are ensured by inequalities (14.36) to (14.39). Inequality (14.36) models the implication $p^w = 0 \to s^w = 0$. Inequality (14.37)

ensures $q^w = 0 \rightarrow t^w = 0$. Both inequalities are redundant for $p^w = 1$ or $q^w = 1$, respectively. Inequalities (14.38) and (14.39) model $p^w = 1 \rightarrow s^w \geq 1$ and $q^w = 1 \rightarrow t^w \geq 1$. The last inequality (14.40) introduced for all words w ensures that at most one of s^w and t^w can be positive, and if $r = 1$ both have to be zero, which means that $x^w = y^w$. Finally, the single inequality (14.41) reflects the opposite implication $(\forall w : x^w = y^w) \rightarrow r = 1$.

14.13.2 DOMAIN PROPAGATION

As already noted in the introduction of this section, the domain propagation of the equality constraint is very unbalanced comparing the cases that r is fixed to zero or one. In the case $r = 1$, we can immediately conclude that each pair of bits x_b and y_b in the operators must have the same value, i.e., $x_b = y_b$. If one of the bits is fixed, we can deduce the corresponding value for the other bit. In the case $r = 0$, however, we can conclude almost nothing. Only if for all pairs except one the bits have the same value and one variable of the remaining pair is already fixed, we can conclude that the other variable must take the opposite value in order to enforce the inequality of the two registers.

In the other direction, we can conclude $r = 0$ immediately after a bit pair with opposite values is detected. The deduction of $r = 1$ can only be carried out if all operand bits are fixed to pairwise equal values.

Algorithm 14.21 illustrates this procedure. Step 1a implements the implication $r = 1 \rightarrow \forall b : x_b = y_b$, while Step 1b propagates the implication in the opposite direction $(\exists b : x_b \neq y_b) \rightarrow r = 0$. Steps 2 and 3 apply the very weak deduction rules $(\forall b : x_b = y_b) \rightarrow r = 1$ and $r = 0 \rightarrow \exists b : x_b \neq y_b$, respectively.

The auxiliary variables are propagated in Steps 4 to 6. If the resultant is fixed to $r = 1$, we can fix all sign variables p^w and q^w to zero in Step 4a. If one of the sign variables is fixed for an individual word w, we can apply the corresponding deductions in Step 4b. Propagating the bounds on $t^w = y^w - x^w$ means to iteratively tighten the bounds of the variables by

$$\begin{aligned} \tilde{l}_{y^w} - \tilde{u}_{x^w} \leq\ & t^w \leq \tilde{u}_{y^w} - \tilde{l}_{x^w} \\ \tilde{l}_{y^w} - \tilde{u}_{t^w} \leq\ & x^w \leq \tilde{u}_{y^w} - \tilde{l}_{t^w} \\ \tilde{l}_{t^w} + \tilde{l}_{x^w} \leq\ & y^w \leq \tilde{u}_{t^w} + \tilde{u}_{x^w} \end{aligned}$$

as long as no more bounds can be tightened or an infeasibility is detected. If we know that one of the variables s^w or t^w is either zero or non-zero, we can draw the appropriate conclusions on the sign variables in Step 4c. Note that afterwards additional deductions are performed in the next iteration in Step 4b. From the lower and upper bounds of the word variables x^w and y^w, we can also deduce fixings of p^w and q^w as shown in Step 4d. We could already tighten the lower bounds of s^w and t^w, but this will automatically happen in Step 4b of the next iteration of the domain propagation loop. The upper bounds of s^w and t^w, however, are tightened in Step 4e.

If the sign variables p^w and q^w are zero for all words, we can deduce $r = 1$. Note that this deduction does not automatically follow from rules 4b and 2, because although we already know that $s^w = t^w = 0$ in Step 4b, we usually cannot fix the bounds of x^w and y^w to a specific value which is necessary to deduce fixings of the bit variables in the domain propagation Algorithm 14.1 of the bit/word partitioning constraints. Finally in Step 6, if the sign variables p^w and q^w are zero for all words

Algorithm 14.21 Equality Domain Propagation

Input: Equality constraint $r = \text{EQ}(x, y)$ on single-bit register r and multi-bit registers x and y of width β with current local bit bounds $\tilde{l}_r \le r \le \tilde{u}_r$, $\tilde{l}_{x_b} \le x_b \le \tilde{u}_{x_b}$, and $\tilde{l}_{y_b} \le y_b \le \tilde{u}_{y_b}$, $b = 0, \ldots, \beta - 1$; current local bounds on auxiliary variables $\tilde{l}_{s^w} \le s^w \le \tilde{u}_{s^w}$, $\tilde{l}_{t^w} \le t^w \le \tilde{u}_{t^w}$, $\tilde{l}_{p^w} \le p^w \le \tilde{u}_{p^w}$, and $\tilde{l}_{q^w} \le q^w \le \tilde{u}_{q^w}$, $w = 0, \ldots, \omega - 1$.

Output: Tightened local bounds for bits r, x_b, y_b, and auxiliary variables s^w, t^w, p^w, and q^w.

1. For all $b = 0, \ldots, \beta - 1$:
   (a) If $\tilde{l}_r = 1$ and $\tilde{u}_{x_b} = 0$, deduce $y = 0$.
   If $\tilde{l}_r = 1$ and $\tilde{l}_{x_b} = 1$, deduce $y = 1$.
   If $\tilde{l}_r = 1$ and $\tilde{u}_{y_b} = 0$, deduce $x = 0$.
   If $\tilde{l}_r = 1$ and $\tilde{l}_{y_b} = 1$, deduce $x = 1$.
   (b) If $\tilde{u}_{x_b} = 0$ and $\tilde{l}_{y_b} = 1$, deduce $r = 0$.
   If $\tilde{l}_{x_b} = 1$ and $\tilde{u}_{y_b} = 0$, deduce $r = 0$.
2. If for all bits $b \in \{0, \ldots, \beta - 1\}$ we have $\tilde{u}_{x_b} = \tilde{u}_{y_b} = 0$ or $\tilde{l}_{x_b} = \tilde{l}_{y_b} = 1$, deduce $r = 1$.
3. If for all bits $b \in \{0, \ldots, \beta - 1\} \setminus \{k\}$ we have $\tilde{u}_{x_b} = \tilde{u}_{y_b} = 0$ or $\tilde{l}_{x_b} = \tilde{l}_{y_b} = 1$, and if $\tilde{u}_r = 0$, and
   (a) if $\tilde{l}_{x_k} = 1$, deduce $y_k = 0$,
   (b) if $\tilde{u}_{x_k} = 0$, deduce $y_k = 1$,
   (c) if $\tilde{l}_{y_k} = 1$, deduce $x_k = 0$,
   (d) if $\tilde{u}_{y_k} = 0$, deduce $x_k = 1$.
4. For all $w = 0, \ldots, \omega - 1$:
   (a) If $\tilde{l}_r = 1$, deduce $p^w = 0$ and $q^w = 0$.
   (b) If $\tilde{l}_{p^w} = 1$, deduce $r = 0$, $q^w = 0$, and $s^w \ge 1$.
   If $\tilde{l}_{q^w} = 1$, deduce $r = 0$, $p^w = 0$, and $t^w \ge 1$.
   If $\tilde{u}_{p^w} = 0$, deduce $s^w = 0$ and propagate bounds on $t^w = y^w - x^w$.
   If $\tilde{u}_{q^w} = 0$, deduce $t^w = 0$ and propagate bounds on $s^w = x^w - y^w$.
   (c) If $\tilde{l}_{s^w} \ge 1$, deduce $p^w = 1$.
   If $\tilde{l}_{t^w} \ge 1$, deduce $q^w = 1$.
   If $\tilde{u}_{s^w} = 0$, deduce $p^w = 0$.
   If $\tilde{u}_{t^w} = 0$, deduce $q^w = 0$.
   (d) If $\tilde{l}_{x^w} > \tilde{u}_{y^w}$, deduce $p^w = 1$.
   If $\tilde{l}_{x^w} \ge \tilde{u}_{y^w}$, deduce $q^w = 0$.
   If $\tilde{u}_{x^w} < \tilde{l}_{y^w}$, deduce $q^w = 1$.
   If $\tilde{u}_{x^w} \le \tilde{l}_{y^w}$, deduce $p^w = 0$.
   (e) Tighten bounds $s^w \le \tilde{u}_{x^w} - \tilde{l}_{y^w}$ and $t^w \le \tilde{u}_{y^w} - \tilde{l}_{x^w}$.
5. If $\tilde{u}_{p^w} = \tilde{u}_{q^w} = 0$ for all $w \in \{0, \ldots, \omega - 1\}$, deduce $r = 1$.
6. If $\tilde{u}_{p^w} = \tilde{u}_{q^w} = 0$ for all $w \in \{0, \ldots, \omega - 1\} \setminus \{k\}$ and $\tilde{u}_r = 0$, and
   (a) if $\tilde{u}_{p^k} = 0$, deduce $q^k = 1$,
   (b) if $\tilde{u}_{q^k} = 0$, deduce $p^k = 1$.

except word k, but the resultant is fixed to $r = 0$, we can propagate the equation $p^k + q^k = 1$, since one of the two variables must be non-zero.

14.13.3 PRESOLVING

The presolving of equality constraints suffers from the same limitations as the domain propagation. If the resultant r is fixed to one, we can aggregate all pairs of bits, but if the resultant is fixed to zero, we usually cannot apply any problem reductions.

Algorithm 14.22 shows the presolving steps that are performed on equality constraints. If the resultant bit is one, we can aggregate the bits of the operands accordingly in Step 1b. Conversely, if the operands are equivalent, the resultant is fixed to one in Step 1c. In both cases the equality constraint becomes redundant and can be deleted.

In the opposite situation of $r = 0$, we can conclude that the operands cannot be equal. Therefore, we add in Step 1d each operand to the list of unequal registers of the other operand, see Section 14.1.4. On the other hand, if we know that the operand registers cannot be equal because they appear in each other's list of unequal registers, we can fix $r = 0$ in Step 1e. Note that in both cases the constraint cannot be deleted, since the inequality $x \stackrel{\star}{\neq} y$ of *registers* is only an information stored in the register data structures. The inequality is not enforced automatically during the solving process. In particular, the constraint itself may have detected the inequality and has to be kept active in order to enforce the inequality. In contrast, the equivalence of registers means that the bits are pairwise aggregated in the variable aggregation graph of SCIP, see Section 3.3.4. Thus, they are represented in all other constraints by a unique representative, and the deletion of the constraint in Steps 1b and 1c is valid, since the whole information of the constraint is captured by the aggregations.

Step 1f is a special case of Step 1e where the inequality of the operands is proven by a pair of bits that are negated equivalent, i.e., for which the equivalence $x_b \stackrel{\star}{=} 1 - y_b$ is stored in the variable aggregation tree. In this case, we can delete the constraint in addition to fix $r := 0$.

Step 1g is similar to Step 3 of the domain propagation Algorithm 14.21. If there is only one bit pair k for which equivalence was not detected, and if one of the remaining three variables r, x_k, and y_k is fixed, we can aggregate the other two variables accordingly. Of course, we do not need to treat the fixing $r = 1$, because this is already captured by Step 1b.

Step 1h corresponds to Step 6 of the domain propagation Algorithm 14.21. If there is only one word k for which the sign variables p^w and q^w are not fixed to zero, two of the three variables r, p^k, and q^k can be aggregated, if the third variable is fixed to zero. Note that we cannot delete the constraint, since the link of the auxiliary variables p^k and q^k to the external variables x^w and y^w has still to be enforced.

For each fixed operand bit, we can add an implication to the implication graph of SCIP in Step 1i which states that if the other operand's bit takes the opposite value, the resultant must be zero.

Pairs of equality constraints are processed in Step 2. If each pair of bits $\{x'_{b'}, y'_{b'}\}$ of constraint $r' = \text{EQ}(x', y')$ appears also as bit pair or as negated bit pair in constraint $r = \text{EQ}(x, y)$, the equality of x and y implies the equality of x' and y'. Therefore, we can in this case derive the implication $r = 1 \rightarrow r' = 1$ in Step 2a. If this is detected with interchanged roles of the constraints, the corresponding im-

Algorithm 14.22 Equality Presolving

1. For all active equality constraints $r = \text{EQ}(x, y)$:
   (a) Apply domain propagation Algorithm 14.21 on the global bounds.
   (b) If $r = 1$, aggregate $x \stackrel{\star}{=} y$ and delete the constraint.
   (c) If $x \stackrel{\star}{=} y$, fix $r := 1$ and delete the constraint.
   (d) If $r = 0$, deduce $x \stackrel{\star}{\neq} y$.
   (e) If $x \stackrel{\star}{\neq} y$, fix $r := 0$.
   (f) If for any bit $b \in \{0, \dots, \beta - 1\}$ we have $x_b \stackrel{\star}{\neq} y_b$, fix $r := 0$ and delete the constraint.
   (g) If for all bits $b \in \{0, \dots, \beta - 1\} \setminus \{k\}$ we have $x_b \stackrel{\star}{=} y_b$, and
      i. if $r = 0$, aggregate $x_k \stackrel{\star}{=} 1 - y_k$ and delete the constraint,
      ii. if $x_k = 0$, aggregate $r \stackrel{\star}{=} 1 - y_k$ and delete the constraint,
      iii. if $x_k = 1$, aggregate $r \stackrel{\star}{=} y_k$ and delete the constraint,
      iv. if $y_k = 0$, aggregate $r \stackrel{\star}{=} 1 - x_k$ and delete the constraint,
      v. if $y_k = 1$, aggregate $r \stackrel{\star}{=} x_k$ and delete the constraint.
   (h) If for all words $w \in \{0, \dots, \omega - 1\} \setminus \{k\}$ we have $p^w = q^w = 0$, and
      i. if $r = 0$, aggregate $p^k \stackrel{\star}{=} 1 - q^k$,
      ii. if $p^k = 0$, aggregate $r \stackrel{\star}{=} 1 - q^k$,
      iii. if $q^k = 0$, aggregate $r \stackrel{\star}{=} 1 - p^k$.
   (i) For all $b = 0, \dots, \beta - 1$:
      i. If $x_b = 0$, add implication $y_b = 1 \to r = 0$ to the implication graph.
      ii. If $x_b = 1$, add implication $y_b = 0 \to r = 0$ to the implication graph.
      iii. If $y_b = 0$, add implication $x_b = 1 \to r = 0$ to the implication graph.
      iv. If $y_b = 1$, add implication $x_b = 0 \to r = 0$ to the implication graph.

2. For all pairs of active equality constraints $r = \text{EQ}(x, y)$ and $r' = \text{EQ}(x', y')$ with $\beta_x \geq \beta_{x'}$:
   (a) If for all $b' \in \{0, \dots, \beta_{x'} - 1\}$ there exists $b \in \{0, \dots, \beta_x - 1\}$ such that
      i. $x_b \stackrel{\star}{=} x'_{b'}$ and $y_b \stackrel{\star}{=} y'_{b'}$, or
      ii. $x_b \stackrel{\star}{=} y'_{b'}$ and $y_b \stackrel{\star}{=} x'_{b'}$, or
      iii. $x_b \stackrel{\star}{\neq} x'_{b'}$ and $y_b \stackrel{\star}{\neq} y'_{b'}$, or
      iv. $x_b \stackrel{\star}{\neq} y'_{b'}$ and $y_b \stackrel{\star}{\neq} x'_{b'}$,

      add the implication $r = 1 \to r' = 1$ to the implication graph of SCIP.
   (b) If for all $b \in \{0, \dots, \beta_x - 1\}$ there exists $b' \in \{0, \dots, \beta_{x'} - 1\}$ such that at least one of 2(a)i to 2(a)iv holds, add the implication $r' = 1 \to r = 1$ to the implication graph of SCIP.
   (c) If both Steps 2a and 2b were successfully applied, aggregate $r \stackrel{\star}{=} r'$ and delete the constraint $r = \text{EQ}(x, y)$.
   (d) If $x \stackrel{\star}{=} x'$ and $r \stackrel{\star}{\neq} r'$, deduce $y \stackrel{\star}{\neq} y'$.
   If $x \stackrel{\star}{=} y'$ and $r \stackrel{\star}{\neq} r'$, deduce $y \stackrel{\star}{\neq} x'$.
   If $y \stackrel{\star}{=} x'$ and $r \stackrel{\star}{\neq} r'$, deduce $x \stackrel{\star}{\neq} y'$.
   If $y \stackrel{\star}{=} y'$ and $r \stackrel{\star}{\neq} r'$, deduce $x \stackrel{\star}{\neq} x'$.
   (e) If $x \stackrel{\star}{=} x'$ and $y \stackrel{\star}{\neq} y'$, add $r = 1 \to r' = 0$ to the implication graph.
   If $x \stackrel{\star}{=} y'$ and $x \stackrel{\star}{\neq} y'$, add $r = 1 \to r' = 0$ to the implication graph.
   If $x \stackrel{\star}{\neq} x'$ and $y \stackrel{\star}{=} y'$, add $r = 1 \to r' = 0$ to the implication graph.
   If $x \stackrel{\star}{\neq} y'$ and $y \stackrel{\star}{=} x'$, add $r = 1 \to r' = 0$ to the implication graph.

plication is added to the implication graph of SCIP in Step 2b. Having detected both implications for a pair of constraints means that the resultants are equivalent. Consequently, one of the constraints can be deleted in Step 2c.

Steps 2d and 2e exploit our knowledge about the inequality of registers. If one of the operand in the first constraint is equivalent to an operand of the second constraint, but the resultants are negated equivalent, the remaining operand pair must be unequal, which is detected in Step 2d. Conversely, if one pair of operands is equivalent but the other pair of operands is unequal, we can conclude in Step 2e that at least one of the resultants must be zero.

14.14 LESS-THAN

Like the equality constraint, the less-than operator

$$\text{LT} : [\beta] \times [\beta] \to [1], \quad (x, y) \mapsto r = \text{LT}(x, y)$$

defined by

$$r = \text{LT}(x, y) \quad \Leftrightarrow \quad r \leftrightarrow (x < y)$$

provides a link from the data path to the control logic of the circuit. By negating the resultant and exchanging the operands, one can also model the other three inequality operands:

$$\begin{aligned} \text{NOT}(r) = \text{LT}(y, x) &\Leftrightarrow r \leftrightarrow (x \leq y) \\ r = \text{LT}(y, x) &\Leftrightarrow r \leftrightarrow (x > y) \\ \text{NOT}(r) = \text{LT}(x, y) &\Leftrightarrow r \leftrightarrow (x \geq y) \end{aligned}$$

Although the LP relaxation of LT constraints is very similar to the relaxation of EQ constraints, the domain propagation behavior of the less-than operator is even worse than the one of the equality operator. The propagation performance of a fixed resultant value is symmetric in LT constraints: in both cases, we usually can not deduce anything. The same holds for presolving. Due to the transitivity of the $<$ operator, however, we have additional possibilities to discover inequality relations between registers in the pairwise comparison of LT constraints.

14.14.1 LP RELAXATION

The LP relaxation for less-than constraints $r = \text{LT}(x, y)$ differs only slightly from the relaxation of EQ constraints. It is also defined on the word level and uses four

auxiliary variables $s^w, t^w \in \mathbb{Z}_{\geq 0}$, and $p^w, q^w \in \{0, 1\}$ per word:

$$x^w - y^w = s^w - t^w \qquad \text{for all } w = 0, \ldots, \omega - 1 \tag{14.42}$$

$$s^w \leq (u_{x^w} - l_{y^w}) \cdot p^w \qquad \text{for all } w = 0, \ldots, \omega - 1 \tag{14.43}$$

$$t^w \leq (u_{y^w} - l_{x^w}) \cdot q^w \qquad \text{for all } w = 0, \ldots, \omega - 1 \tag{14.44}$$

$$p^w - s^w \leq 0 \qquad \text{for all } w = 1, \ldots, \omega - 1 \tag{14.45}$$

$$q^w - t^w \leq 0 \qquad \text{for all } w = 0, \ldots, \omega - 1 \tag{14.46}$$

$$p^w + q^w \leq 1 \qquad \text{for all } w = 1, \ldots, \omega - 1 \tag{14.47}$$

$$p^0 + q^0 = 1 \tag{14.48}$$

$$r + p^w - \sum_{k=w+1}^{\omega-1} q^k \leq 1 \qquad \text{for all } w = 0, \ldots, \omega - 1 \tag{14.49}$$

$$-r + q^w - \sum_{k=w+1}^{\omega-1} p^k \leq 0 \qquad \text{for all } w = 0, \ldots, \omega - 1 \tag{14.50}$$

As before, constraints (14.42) to (14.47) split the difference $x^w - y^w$ into the positive part s^w and the negative part t^w with $p^w = 1$ if and only if $x^w > y^w$ and $q^w = 1$ if and only if $x^w < y^w$ (with a slight deviation for the least significant word $w = 0$, see below). As opposed to the equality constraint, the position of the operand words play an important role: a less significant word can only influence the resultant, if all more significant words of x and y are pairwise equal. Therefore, the resultant r is not necessarily affected by all words and does not appear in inequality (14.47). Instead, the significance of the words and their relation to the resultant is captured by inequalities (14.49) and (14.50). For the most significant word $w = \omega - 1$, the sums are empty and the inequalities are reduced to

$$r + p^{\omega-1} \leq 1 \quad \text{and} \quad -r + q^{\omega-1} \leq 0,$$

which model the valid implications $p^{\omega-1} = 1 \rightarrow r = 0$ and $q^{\omega-1} = 1 \rightarrow r = 1$, respectively. Only if $p^{\omega-1} = q^{\omega-1} = 0$, i.e., $x^{\omega-1} = y^{\omega-1}$, the value of the resultant is determined by the lower significant words. Thus, we had to include the sums into inequalities (14.49) and (14.50) to render them redundant whenever the resultant is already determined on higher significant words. Altogether, inequality (14.49) represents the implication

$$(\forall k > w : q^k = 0) \wedge p^w = 1 \rightarrow r = 0,$$

and inequality (14.50) models

$$(\forall k > w : p^k = 0) \wedge q^w = 1 \rightarrow r = 1$$

for words $w = 0, \ldots, \omega - 1$.

The least significant word $w = 0$ plays a slightly different role than the other words. The equality $x^0 = y^0$ is, with respect to the value of the resultant, equivalent to $x^0 > y^0$: in both cases the resultant is zero if all more significant words are pairwise equal. Therefore, we define p^0 to be one if and only if $x^0 \geq y^0$, in contrast to $x^w > y^w$ for the other words. This turns inequality (14.47) into the equation (14.48) and makes inequality (14.45) invalid for word $w = 0$. Of course, we can immediately aggregate $p^0 :\stackrel{\star}{=} 1 - q^0$ and replace the occurrences of p^0 accordingly.

14.14.2 DOMAIN PROPAGATION

The domain propagation for less-than constraints is depicted in Algorithm 14.23. Steps 1 and 2 apply domain propagation on the bits of the operand registers x and y. We iterate over the bits from most significant to least significant bit, since less significant bits might be irrelevant if the value of the resultant is already determined by the more significant bits.

If we know that in the current local bounds $x_b < y_b$ holds for a bit b and if $x_j \leq y_j$ for all higher significant bits $j > b$, the fixings at bit b prove that $x < y$, and we can deduce $r = 1$ in Step 1a. Conversely, if for a bit b we have $x_b > y_b$ and $x_j \geq y_j$ for all $j > b$, it clearly follows $x > y$ and thus $r = 0$, which is deduced in Step 1b.

Steps 1c and 1d apply this reasoning in the other way. If we already know that $r = 1$ but there is a bit b with $x_b > y_b$ and we have $x_j \geq y_j$ for all more significant bits $j > b$ except for a single bit k, we can conclude in Step 1c that the inequality $x < y$ must hold due to bit k. Thus, it follows $x_k < y_k$ which means $x_k = 0$ and $y_k = 1$. On the other hand, if we know that $r = 0$ but $x_b < y_b$ for a bit b with $x_j \leq y_j$ for all $j > b$ except for bit k, we can deduce $x_k > y_k$, i.e., $x_k = 1$ and $y_k = 0$ in Step 1d.

If $r = 1$ but for the highest significant bits $j \geq b$ we have $x_j \geq y_j$, it must be $x_j = y_j$ for all $j \geq b$. This deduction is iteratively applied at Step 1e while the bits are scanned from $b = \beta - 1$ down to $b = 0$, such that we only need to fix the current bits x_b and y_b. Conversely, if $r = 0$ but $x_j \leq y_j$ for all $j \geq b$, we can conclude $x_j = y_j$ for all $j \geq b$ in Step 1f.

If we detected during the loop of Step 1 that $x_b \geq y_b$ for all bits, we have $x \geq y$ and can conclude $r = 0$ in Step 2.

Steps 3 and 4 propagate the auxiliary variables for the operand words. Again, we iterate from most significant to least significant word. Steps 3a to 3f are very similar to the bit level propagations of Step 1. If for a word w we have $q^w = 1$ and $p^j = 0$ for all $j > w$, we know that $x^w < y^w$ and $x^j \leq y^j$ for all $j > w$. It follows $x < y$ and thus $r = 1$, which is deduced in Step 3a. Conversely, if $p^w = 1$ and $q^j = 0$ for all $j > w$, we have $x > y$ (or $x \geq y$ if $w = 0$) and we can conclude $r = 0$ in Step 3b. If we know that $r = 1$ but $p^w = 1$ and $q^j = 0$ for all words $j > w$ except word k, it must be $q^k = 1$, which is deduced in Step 3c. On the other hand, if $r = 0$ but $q^w = 1$ and $p^j = 0$ for all words $j > w$ except word k, we can conclude $p^k = 1$ in Step 3d. If $r = 1$ and $q^j = 0$ for all $j \geq w$, it must also be $p^j = 0$ for all $j \geq w$. This deduction is performed successively in Step 3e. Step 3f applies the opposite propagation: if $r = 0$ and $p^j = 0$ for all $j \geq w$, it follows $q^j = 0$ for all $j \geq w$.

Steps 3g to 3j apply the same reasoning as Steps 4b to 4e of the domain propagation Algorithm 14.21 for equality constraints, except that the different meaning of p^0 must be regarded. Finally, if in the loop of Step 3 we detected $q^w = 0$ for all words w, we can conclude $r = 0$ in Step 4.

14.14.3 PRESOLVING

The presolving Algorithm 14.24 for less-than constraints does not consist of much more than to call the domain propagation on the global bounds in Step 1a. However, if a propagation on the external variables was applied that completely determined the status of the constraint, we can delete the constraint from the model. If the operands are equivalent, we can clearly deduce $r = 0$ in Step 1b and delete the constraint. If on the other hand, the resultant is fixed to $r = 1$ in Step 1c, the

Algorithm 14.23 Less-Than Domain Propagation

Input: Less-than constraint $r = \text{LT}(x, y)$ on single-bit register r and multi-bit registers x and y of width β with current local bit bounds $\tilde{l}_r \le r \le \tilde{u}_r$, $\tilde{l}_{x_b} \le x_b \le \tilde{u}_{x_b}$, and $\tilde{l}_{y_b} \le y_b \le \tilde{u}_{y_b}$, $b = 0, \ldots, \beta - 1$; current local bounds on auxiliary variables $\tilde{l}_{s^w} \le s^w \le \tilde{u}_{s^w}$, $\tilde{l}_{t^w} \le t^w \le \tilde{u}_{t^w}$, $\tilde{l}_{p^w} \le p^w \le \tilde{u}_{p^w}$, and $\tilde{l}_{q^w} \le q^w \le \tilde{u}_{q^w}$, $w = 0, \ldots, \omega - 1$.

Output: Tightened local bounds for bits r, x_b, y_b, and auxiliary variables s^w, t^w, p^w, and q^w.

1. For all $b = \beta - 1, \ldots, 0$:
   (a) If $\tilde{u}_{x_b} = 0$, $\tilde{l}_{y_b} = 1$, and $\tilde{u}_{x_j} \le \tilde{l}_{y_j}$ for all $j > b$, deduce $r = 1$.
   (b) If $\tilde{l}_{x_b} = 1$, $\tilde{u}_{y_b} = 0$, and $\tilde{l}_{x_j} \ge \tilde{u}_{y_j}$ for all $j > b$, deduce $r = 0$.
   (c) If $\tilde{l}_r = 1$, $\tilde{l}_{x_b} = 1$, $\tilde{u}_{y_b} = 0$, and $\tilde{l}_{x_j} \ge \tilde{u}_{y_j}$ for all $j \in \{b+1, \ldots, \beta-1\} \setminus \{k\}$, deduce $x_k = 0$ and $y_k = 1$.
   (d) If $\tilde{u}_r = 0$, $\tilde{u}_{x_b} = 0$, $\tilde{l}_{y_b} = 1$, and $\tilde{u}_{x_j} \le \tilde{l}_{y_j}$ for all $j \in \{b+1, \ldots, \beta-1\} \setminus \{k\}$, deduce $x_k = 1$ and $y_k = 0$.
   (e) If $\tilde{l}_r = 1$, and $\tilde{l}_{x_j} \ge \tilde{u}_{y_j}$ for all $j \ge b$, deduce $x_b \le \tilde{u}_{y_b}$ and $y_b \ge \tilde{l}_{x_b}$.
   (f) If $\tilde{u}_r = 0$, and $\tilde{u}_{x_j} \le \tilde{l}_{y_j}$ for all $j \ge b$, deduce $x_b \ge \tilde{l}_{y_b}$ and $y_b \le \tilde{u}_{x_b}$.
2. If $\tilde{l}_{x_b} \ge \tilde{u}_{y_b}$ for all $b = 0, \ldots, \beta - 1$, deduce $r = 0$.
3. For all $w = \omega - 1, \ldots, 0$:
   (a) If $\tilde{l}_{q^w} = 1$ and $\tilde{u}_{p^j} = 0$ for all $j > w$, deduce $r = 1$.
   (b) If $\tilde{l}_{p^w} = 1$ and $\tilde{u}_{q^j} = 0$ for all $j > w$, deduce $r = 0$.
   (c) If $\tilde{l}_r = 1$, $\tilde{l}_{p^w} = 1$, and $\tilde{u}_{q^j} = 0$ for all $j \in \{w+1, \ldots, \omega-1\} \setminus \{k\}$, deduce $q^k = 1$.
   (d) If $\tilde{l}_r = 0$, $\tilde{l}_{q^w} = 1$, and $\tilde{u}_{p^j} = 0$ for all $j \in \{w+1, \ldots, \omega-1\} \setminus \{k\}$, deduce $p^k = 1$.
   (e) If $\tilde{l}_r = 1$, and $\tilde{u}_{q^j} = 0$ for all $j \ge w$, deduce $p^w = 0$.
   (f) If $\tilde{l}_r = 0$, and $\tilde{u}_{p^j} = 0$ for all $j \ge w$, deduce $q^w = 0$.
   (g) If $\tilde{l}_{p^w} = 1$, deduce $q^w = 0$, and if additionally $w \ge 1$, deduce $s^w \ge 1$.
   If $\tilde{l}_{q^w} = 1$, deduce $p^w = 0$ and $t^w \ge 1$.
   If $\tilde{u}_{p^w} = 0$, deduce $s^w = 0$ and propagate bounds on $t^w = y^w - x^w$.
   If $\tilde{u}_{q^w} = 0$, deduce $t^w = 0$ and propagate bounds on $s^w = x^w - y^w$.
   (h) If $\tilde{l}_{s^w} \ge 1$, deduce $p^w = 1$.
   If $\tilde{l}_{t^w} \ge 1$, deduce $q^w = 1$.
   If $\tilde{u}_{s^w} = 0$ and $w \ge 1$, deduce $p^w = 0$.
   If $\tilde{u}_{t^w} = 0$, deduce $q^w = 0$.
   (i) If $\tilde{l}_{x^w} > \tilde{u}_{y^w}$, deduce $p^w = 1$.
   If $\tilde{l}_{x^w} \ge \tilde{u}_{y^w}$, deduce $q^w = 0$.
   If $\tilde{u}_{x^w} < \tilde{l}_{y^w}$, deduce $q^w = 1$.
   If $\tilde{u}_{x^w} \le \tilde{l}_{y^w}$ and $w \ge 1$, deduce $p^w = 0$.
   (j) Tighten bounds $s^w \le \tilde{u}_{x^w} - \tilde{l}_{y^w}$ and $t^w \le \tilde{u}_{y^w} - \tilde{l}_{x^w}$.
4. If $\tilde{u}_{q^w} = 0$ for all $w = 0, \ldots, \omega - 1$, deduce $r = 0$.

Algorithm 14.24 Less-Than Presolving

1. For all active less-than constraints $r = \text{LT}(x, y)$:
   (a) Apply domain propagation Algorithm 14.23 on the global bounds. If any of rules 1a to 1d or rule 2 was applied, delete the constraint.
   (b) If $x \stackrel{\star}{=} y$, fix $r := 0$ and delete the constraint.
   (c) If $r = 1$, deduce $x \stackrel{\star}{\neq} y$.
   (d) For all $b = \beta - 1, \ldots, 0$:
   If $x_j \stackrel{\star}{=} y_j$ for all $j \in \{b+1, \ldots, \beta-1\} \setminus \{k\}$, and
   i. if $x_b = 0$, $y_b = 1$, and $x_k = 1$, aggregate $r :\stackrel{\star}{=} y_k$,
   ii. if $x_b = 0$, $y_b = 1$, and $y_k = 0$, aggregate $r :\stackrel{\star}{=} 1 - x_k$.
   iii. if $x_b = 1$, $y_b = 0$, and $x_k = 0$, aggregate $r :\stackrel{\star}{=} y_k$,
   iv. if $x_b = 1$, $y_b = 0$, and $y_k = 1$, aggregate $r :\stackrel{\star}{=} 1 - x_k$.
   If any of the aggregations was performed, delete the constraint.
2. For all pairs of active less-than constraints $r = \text{LT}(x, y)$ and $r' = \text{LT}(x', y')$:
   (a) If $x \stackrel{\star}{=} x'$ and $y \stackrel{\star}{=} y'$, aggregate $r :\stackrel{\star}{=} r'$ and delete constraint $r = \text{LT}(x, y)$.
   (b) If $x \stackrel{\star}{=} y'$ and $y \stackrel{\star}{=} x'$, add $r = 1 \rightarrow r' = 0$ to the implication graph of SCIP.
   (c) If $x \stackrel{\star}{=} x'$ and $r \stackrel{\star}{\neq} r'$, deduce $y \stackrel{\star}{\neq} y'$.
   If $y \stackrel{\star}{=} y'$ and $r \stackrel{\star}{\neq} r'$, deduce $x \stackrel{\star}{\neq} x'$.
   (d) If $r = 1$, $r' = 1$, and $y \stackrel{\star}{=} x'$, deduce $x \stackrel{\star}{\neq} y'$.
   If $r = 1$, $r' = 1$, and $x \stackrel{\star}{=} y'$, deduce $y \stackrel{\star}{\neq} x'$.
   If $r = 1$, $r' = 0$, and $y \stackrel{\star}{=} y'$, deduce $x \stackrel{\star}{\neq} x'$.
   If $r = 1$, $r' = 0$, and $x \stackrel{\star}{=} x'$, deduce $y \stackrel{\star}{\neq} y'$.
   If $r = 0$, $r' = 1$, and $x \stackrel{\star}{=} x'$, deduce $y \stackrel{\star}{\neq} y'$.
   If $r = 0$, $r' = 1$, and $y \stackrel{\star}{=} y'$, deduce $x \stackrel{\star}{\neq} x'$.

operands cannot be equal and we can remember the inequality $x \stackrel{\star}{\neq} y$ in our register data structures, see Section 14.1.4. Note that in this case the constraint must not be deleted, since the inequality of the operands has still to be enforced. Finally, if in the most significant part of the operands there are only two bit pairs $k > b$ of non-equivalent bits, and if the lesser significant bits x_b and y_b are fixed to opposite values, the value of the resultant would be determined if the more significant bits x_k and y_k were equal. Thus, if one of the bits x_k or y_k is fixed, the resultant can be aggregated to the other operand's bit variable in Step 1d. Afterwards, the constraint can be deleted.

The presolving procedure for pairs of less-than constraints in Step 2 is also rather weak. We can aggregate the two resultants in Step 2a if the operands are pairwise equivalent in the same order. If they are pairwise equivalent but in opposite order, i.e., $x \stackrel{\star}{=} y'$ and $y \stackrel{\star}{=} x'$, we can only conclude that at most one of the resultants can be one. Thus, in Step 2b we add the corresponding implication to the implication graph of SCIP. If only one of the operand pairs is equivalent but the resultant is negated equivalent, the other pair of operands is detected to be unequal in Step 2c.

Step 2d applies the transitivity law of the $<$ and $\leq$ operators:

$$\begin{aligned}
x < y \overset{\star}{=} x' < y' \quad &\Rightarrow \quad x < y' \quad \Rightarrow \quad x \overset{\star}{\neq} y' \\
x' < y' \overset{\star}{=} x < y \quad &\Rightarrow \quad x' < y \quad \Rightarrow \quad y \overset{\star}{\neq} x' \\
x < y \overset{\star}{=} y' \leq x' \quad &\Rightarrow \quad x < x' \quad \Rightarrow \quad x \overset{\star}{\neq} x' \\
y' \leq x' \overset{\star}{=} x < y \quad &\Rightarrow \quad y' < y \quad \Rightarrow \quad y \overset{\star}{\neq} y' \\
y \leq x \overset{\star}{=} x' < y' \quad &\Rightarrow \quad y < y' \quad \Rightarrow \quad y \overset{\star}{\neq} y' \\
x' < y' \overset{\star}{=} y \leq x \quad &\Rightarrow \quad x' < x \quad \Rightarrow \quad x \overset{\star}{\neq} x'
\end{aligned}$$

Note that we cannot deduce any inequality relations if $r = 0$ and $r' = 0$, since in this case it is possible that all operands are equal.

14.15 IF-THEN-ELSE

The unary logic operators UAND, UOR, and UXOR, and the comparison operators EQ and LT provide links from the data path to the control logic of the circuit. The if-then-else operator

$$\text{ITE} : [1] \times [\beta] \times [\beta] \rightarrow [\beta], \quad (x, y, z) \mapsto r = \text{ITE}(x, y, z)$$

with

$$r = \text{ITE}(x, y, z) \quad \Leftrightarrow \quad r = \begin{cases} y & \text{if } x = 1 \\ z & \text{if } x = 0 \end{cases}$$

addresses the reverse direction: the control logic bit x influences which part of the data path (y or z) should be passed to the resultant r. One example for the application of the ITE operator is the "opcode" selection in an arithmetical logic unit (ALU). For each supported operation, an internal variable stores the result of the corresponding combination of the input registers. Afterwards, a case split consisting of equality and if-then-else constraints selects the variable that should be passed to the output register of the circuit. Thus, a very simple ALU which supports the operations "+", "−", and "×" can look as illustrated in Figure 14.8.

If-then-else constraints have interesting domain propagation and presolving possibilities. They share an aspect with the equality constraints, namely that we can exploit the implications $x = 1 \rightarrow \forall b : r_b = y_b$ and $x = 0 \rightarrow \forall b : r_b = z_b$. On the other hand, if we know $y_b = z_b$ we can already conclude $r_b = y_b$ without knowing the value of x. As explained in Chapter 15.2, if-then-else constraints are the main reason for the detection of irrelevant parts of the circuit.

14.15.1 LP RELAXATION

The LP relaxation of if-then-else constraints is defined on word level. It consists of four inequalities per word:

$$r^w - y^w \leq (u_{z^w} - l_{y^w}) \cdot (1 - x) \quad \text{for all } w = 0, \ldots, \omega - 1 \tag{14.51}$$

$$r^w - y^w \geq (l_{z^w} - u_{y^w}) \cdot (1 - x) \quad \text{for all } w = 0, \ldots, \omega - 1 \tag{14.52}$$

$$r^w - z^w \leq (u_{y^w} - l_{z^w}) \cdot x \quad \text{for all } w = 0, \ldots, \omega - 1 \tag{14.53}$$

$$r^w - z^w \geq (l_{y^w} - u_{z^w}) \cdot x \quad \text{for all } w = 0, \ldots, \omega - 1 \tag{14.54}$$

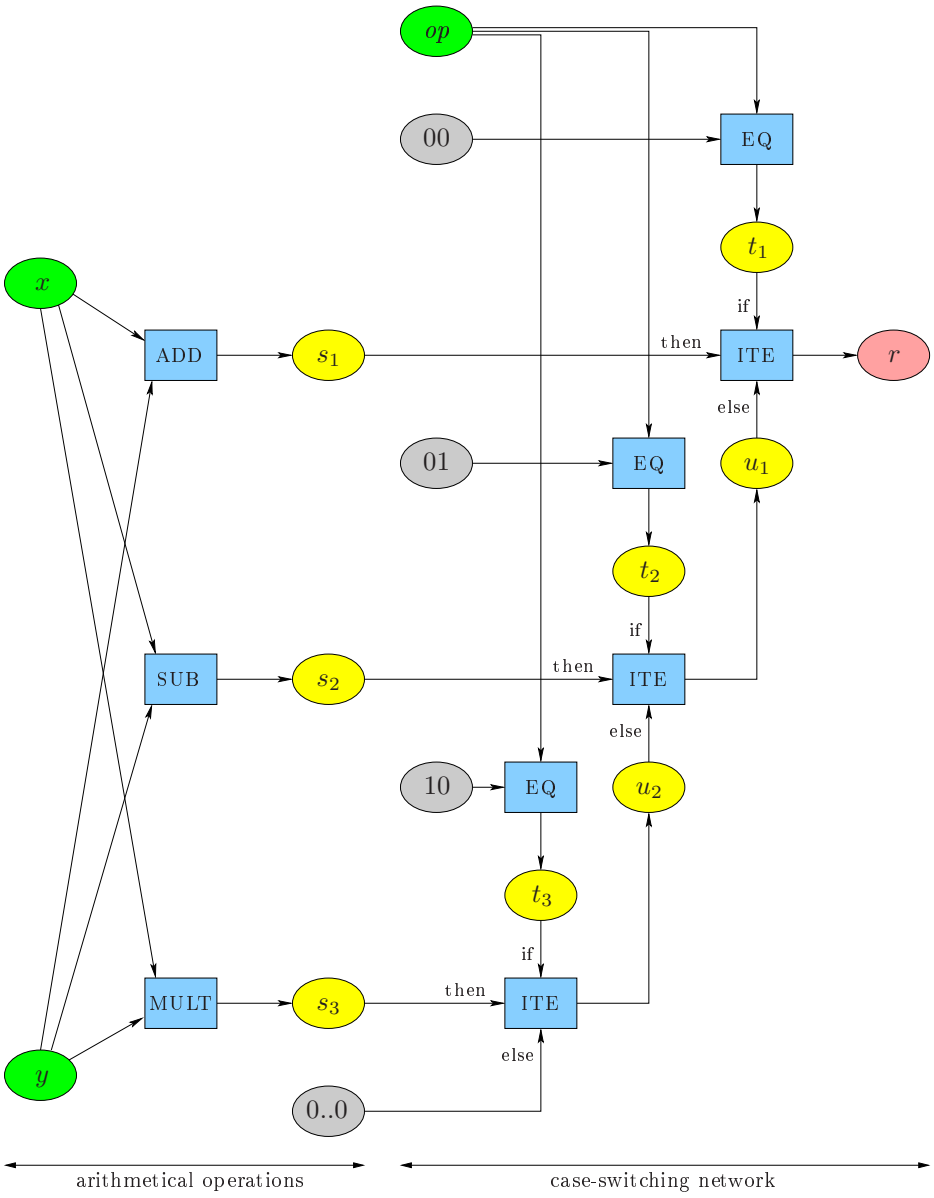

Figure 14.8. Simple arithmetical logic unit. Different arithmetical operations combine the input registers x and y to produce intermediate results s_i. One of these operations is selected by the input register op. A case-switching network consisting of EQ and ITE constraints passes the result of the selected operation to the output register r.

In the case $x = 1$, inequalities (14.51) and (14.52) imply $r^w = y^w$, and inequalities (14.53) and (14.54) are redundant. Conversely, in the case $x = 0$, inequalities (14.53) and (14.54) force $r^w = z^w$ and inequalities (14.51) and (14.52) are redundant.

Lemma 14.31. The convex hull $P_{\text{ITE}} = \text{conv}\{(r, x, y, z) \mid x \in \{0, 1\}, r, y, z \in \mathbb{Z}, l_y \leq y \leq u_y, l_z \leq z \leq u_z, r = \text{ITE}(x, y, z)\}$ of the feasible solutions for an ITE constraint is full-dimensional if $l_y < u_y$, $l_z < u_z$, $l_y < u_z$, and $l_z < u_y$.

Proof. The feasible solutions

$$(r, x, y, z) \in \{(l_z, 0, l_y, l_z), (l_z, 0, u_y, l_z), (u_z, 0, l_y, u_z), (l_y, 1, l_y, l_z), (u_y, 1, u_y, l_z)\}$$

define five affinely independent vectors in $\mathbb{R}^4$. Thus, the dimension of P_{ITE} is $\dim(P_{\text{ITE}}) = 4$. □

Proposition 14.32. Inequalities (14.51) to (14.54) define facets of P_{ITE} if $l_y < u_y$, $l_z < u_z$, $l_y < u_z$, and $l_z < u_y$.

Proof. P_{ITE} is full-dimensional as shown in Lemma 14.31. Thus, it suffices for each inequality to provide four affinely independent solution vectors that fulfill the inequality with equality. For inequality (14.51), the solutions

$$(r, x, y, z) \in \{(l_y, 1, l_y, l_z), (l_y, 1, l_y, u_z), (u_y, 1, u_y, l_z), (u_z, 0, l_y, u_z)\}$$

meet this criterion. The affinely independent solution vectors

$$(r, x, y, z) \in \{(l_y, 1, l_y, l_z), (l_y, 1, l_y, u_z), (u_y, 1, u_y, l_z), (l_z, 0, u_y, l_z)\}$$

fulfill inequality (14.52) with equality. Corresponding solution vectors for inequalities (14.53) and (14.54) can be constructed analogously. □

14.15.2 Domain Propagation

The domain propagation for if-then-else constraint is depicted in Algorithm 14.25. Step 1 compares the bits of the resultant and the operands y and z. If $y_b = z_b$, we can deduce $r_b = y_b$ independently from the value of x in Step 1a. If the selector bit is fixed to $x = 1$, Step 1b propagates $r_b = y_b$. In the other case of $x = 0$, we can propagate the equation $r_b = z_b$ in Step 1c. Steps 1d and 1e apply the inverse reasoning of Steps 1b and 1c: if for a bit b we find $r_b \neq y_b$, we can conclude $x = 0$, and if $r_b \neq z_b$, we can deduce $x = 1$.

Step 2 applies exactly the same rules as Step 1 on word level. Independent from x, the bounds of r^w can be tightened in Step 2a. If x is fixed, we can propagate $r^w = y^w$ or $r^w = z^w$ in Steps 1b and 1c, respectively. If the bounds of the words make a certain selector value impossible, the selector bit x is fixed to the opposite value in Steps 2d and 2e.

14.15.3 Presolving

Presolving for if-then-else constraints applies the same rules as domain propagation, but can also find aggregations of variables. Additionally, we exploit our knowledge about equality or inequality of registers. Like for other constraints, we compare pairs of ITE constraints to detect further simplifications.

Algorithm 14.25 If-Then-Else Domain Propagation

Input: If-then-else constraint $r = \text{ITE}(x, y, z)$ on registers single-bit register x and multi-bit registers r, y, and z of width β with current local bounds $\tilde{l}_x \leq x \leq \tilde{u}_x$ and local bit bounds $\tilde{l}_{r_b} \leq r_b \leq \tilde{u}_{r_b}$, $\tilde{l}_{y_b} \leq y_b \leq \tilde{u}_{y_b}$, and $\tilde{l}_{z_b} \leq z_b \leq \tilde{u}_{z_b}$.

Output: Tightened local bounds for bits x, r_b, y_b, z_b.

1. For all $b = 0, \ldots, \beta - 1$:
   (a) If $\tilde{l}_{y_b} = 1$ and $\tilde{l}_{z_b} = 1$, deduce $r_b = 1$.
   If $\tilde{u}_{y_b} = 0$ and $\tilde{u}_{z_b} = 0$, deduce $r_b = 0$.
   (b) If $\tilde{l}_x = 1$ and $\tilde{u}_{y_b} = 0$, deduce $r_b = 0$.
   If $\tilde{l}_x = 1$ and $\tilde{l}_{y_b} = 1$, deduce $r_b = 1$.
   If $\tilde{l}_x = 1$ and $\tilde{u}_{r_b} = 0$, deduce $y_b = 0$.
   If $\tilde{l}_x = 1$ and $\tilde{l}_{r_b} = 1$, deduce $y_b = 1$.
   (c) If $\tilde{u}_x = 0$ and $\tilde{u}_{z_b} = 0$, deduce $r_b = 0$.
   If $\tilde{u}_x = 0$ and $\tilde{l}_{z_b} = 1$, deduce $r_b = 1$.
   If $\tilde{u}_x = 0$ and $\tilde{u}_{r_b} = 0$, deduce $z_b = 0$.
   If $\tilde{u}_x = 0$ and $\tilde{l}_{r_b} = 1$, deduce $z_b = 1$.
   (d) If $\tilde{l}_{r_b} = 1$ and $\tilde{u}_{y_b} = 0$, or if $\tilde{u}_{r_b} = 0$ and $\tilde{l}_{y_b} = 1$, deduce $x = 0$.
   (e) If $\tilde{l}_{r_b} = 1$ and $\tilde{u}_{z_b} = 0$, or if $\tilde{u}_{r_b} = 0$ and $\tilde{l}_{z_b} = 1$, deduce $x = 1$.
2. For all $w = 0, \ldots, \omega - 1$:
   (a) Tighten bounds of r^w: $\min\{\tilde{l}_{y^w}, \tilde{l}_{z^w}\} \leq r^w \leq \max\{\tilde{u}_{y^w}, \tilde{u}_{z^w}\}$.
   (b) If $\tilde{l}_x = 1$, deduce $\tilde{l}_{y^w} \leq r^w \leq \tilde{u}_{y^w}$ and $\tilde{l}_{r^w} \leq y^w \leq \tilde{u}_{r^w}$.
   (c) If $\tilde{u}_x = 0$, deduce $\tilde{l}_{z^w} \leq r^w \leq \tilde{u}_{z^w}$ and $\tilde{l}_{r^w} \leq z^w \leq \tilde{u}_{r^w}$.
   (d) If $\tilde{l}_{r^w} > \tilde{u}_{y^w}$, or if $\tilde{u}_{r^w} < \tilde{l}_{y^w}$, deduce $x = 0$.
   (e) If $\tilde{l}_{r^w} > \tilde{u}_{z^w}$, or if $\tilde{u}_{r^w} < \tilde{l}_{z^w}$, deduce $x = 1$.

Algorithm 14.26 shows the presolving rules applied to if-then-else constraints. If the if-case y and the else-case z are equivalent, the selector bit x does not play a role and we can aggregate $r :\overset{\star}{=} y$ in Step 1a. Afterwards, the constraint is redundant and can be deleted. If we know that the resultant is unequal to one of the operands, we can fix the selection bit accordingly in Step 1b. After the domain propagation on the global bounds was applied in Step 1c, we compare pairs of bits (i, j) in the resultant and the operands. If y_i and y_j are (negated) equivalent and z_i and z_j are (negated) equivalent, the resultant bits at these positions must also be (negated) equivalent, which is discovered in Step 1(d)i. If on the other hand the equivalences of the operand bits and the resultant bits do not match, the selector variable can be fixed to the opposite decision value in Steps 1(d)ii and 1(d)iii.

If the selector variable x is fixed after applying the previous presolving rules, we can aggregate the resultant with the respective operand and delete the ITE constraint in Steps 1e and 1f. Step 1g introduces the derivable implications to the implication graph of SCIP, see Section 3.3.5. If a bit in the resultant r is fixed, we can add the corresponding implications for the values of x in Step 1(g)i. Note that the valid implications $y_b = 1 \rightarrow z_b = 0$ for $r_b = 0$ and $y_b = 0 \rightarrow z_b = 1$ for $r_b = 1$ are automatically added through the transitive closure of the implication graph. If

Algorithm 14.26 If-Then-Else Presolving

1. For all active if-then-else constraints $r = \text{ITE}(x, y, z)$:
   (a) If $y \stackrel{\star}{=} z$, aggregate $r \stackrel{\star}{:=} y$ and delete the constraint.
   (b) If $r \stackrel{\star}{\neq} z$, fix $x := 1$.
   If $r \stackrel{\star}{\neq} y$, fix $x := 0$.
   (c) Apply domain propagation Algorithm 14.25 on the global bounds.
   (d) For all $i, j \in \{0, \ldots, \beta - 1\}$ with $i < j$:
      i. If $y_i \stackrel{\star}{=} y_j$ and $z_i \stackrel{\star}{=} z_j$, aggregate $r_i \stackrel{\star}{:=} r_j$.
      If $y_i \stackrel{\star}{\neq} y_j$ and $z_i \stackrel{\star}{\neq} z_j$, aggregate $r_i \stackrel{\star}{:=} 1 - r_j$.
      ii. If $y_i \stackrel{\star}{=} y_j$ and $r_i \stackrel{\star}{\neq} r_j$, fix $x := 0$.
      If $y_i \stackrel{\star}{\neq} y_j$ and $r_i \stackrel{\star}{=} r_j$, fix $x := 0$.
      iii. If $z_i \stackrel{\star}{=} z_j$ and $r_i \stackrel{\star}{\neq} r_j$, fix $x := 1$.
      If $z_i \stackrel{\star}{\neq} z_j$ and $r_i \stackrel{\star}{=} r_j$, fix $x := 1$.
   (e) If $x = 1$, aggregate $r \stackrel{\star}{:=} y$ and delete the constraint.
   (f) If $x = 0$, aggregate $r \stackrel{\star}{:=} z$ and delete the constraint.
   (g) For all $b = 0, \ldots, \beta - 1$, add the following implications to the implication graph of SCIP:
      i. If $r_b = 0$, add implications $x = 1 \to y_b = 0$ and $x = 0 \to z_b = 0$.
      If $r_b = 1$, add implications $x = 1 \to y_b = 1$ and $x = 0 \to z_b = 1$.
      ii. If $y_b = 0$, add implications $x = 1 \to r_b = 0$ and $z_b = 0 \to r_b = 0$.
      If $y_b = 1$, add implications $x = 1 \to r_b = 1$ and $z_b = 1 \to r_b = 1$.
      iii. If $z_b = 0$, add implications $x = 0 \to r_b = 0$ and $y_b = 0 \to r_b = 0$.
      If $z_b = 1$, add implications $x = 0 \to r_b = 1$ and $y_b = 1 \to r_b = 1$.

2. For all pairs of active if-then-else constraints $r = \text{ITE}(x, y, z)$ and $r' = \text{EQ}(x', y', z')$ with $\beta_r \geq \beta_{r'}$:
   (a) For all $b = 0, \ldots, \beta_r - 1$:
      i. If $x \stackrel{\star}{=} x'$, $y_b \stackrel{\star}{=} y'_b$, and $z_b \stackrel{\star}{=} z'_b$, aggregate $r_b \stackrel{\star}{:=} r'_b$.
      If $x \stackrel{\star}{=} x'$, $y_b \stackrel{\star}{\neq} y'_b$, and $z_b \stackrel{\star}{\neq} z'_b$, aggregate $r_b \stackrel{\star}{:=} 1 - r'_b$.
      ii. If $x \stackrel{\star}{\neq} x'$, $y_b \stackrel{\star}{=} z'_b$, and $z_b \stackrel{\star}{=} y'_b$, aggregate $r_b \stackrel{\star}{:=} r'_b$.
      If $x \stackrel{\star}{\neq} x'$, $y_b \stackrel{\star}{\neq} z'_b$, and $z_b \stackrel{\star}{\neq} y'_b$, aggregate $r_b \stackrel{\star}{:=} 1 - r'_b$.
   If all of the resultant bits were aggregated, delete $r' = \text{EQ}(x', y', z')$.
   (b) If $\beta_r = \beta_{r'}$ and $r \stackrel{\star}{\neq} r'$, add the following implications to the implication graph of SCIP:
      i. If $y \stackrel{\star}{=} y'$, add $x = 1 \to x' = 0$.
      ii. If $z \stackrel{\star}{=} z'$, add $x = 0 \to x' = 1$.
      iii. If $y \stackrel{\star}{=} z'$, add $x = 1 \to x' = 1$.
      iv. If $z \stackrel{\star}{=} y'$, add $x = 0 \to x' = 0$.
   (c) If $y \stackrel{\star}{=} r'$ and $x = 1 \to x' = 1$ holds, replace y by y'.
   If $y \stackrel{\star}{=} r'$ and $x = 1 \to x' = 0$ holds, replace y by z'.
   (d) If $z \stackrel{\star}{=} r'$ and $x = 0 \to x' = 1$ holds, replace z by y'.
   If $z \stackrel{\star}{=} r'$ and $x = 0 \to x' = 0$ holds, replace z by z'.
   (e) If $y' \stackrel{\star}{=} r$ and $x' = 1 \to x = 1$ holds, replace y' by y.
   If $y' \stackrel{\star}{=} r$ and $x' = 1 \to x = 0$ holds, replace y' by z.
   (f) If $z' \stackrel{\star}{=} r$ and $x' = 0 \to x = 1$ holds, replace z' by y.
   If $z' \stackrel{\star}{=} r$ and $x' = 0 \to x = 0$ holds, replace z' by z.

one of the operand bits is fixed, similar implications can be added in Steps 1(g)ii and 1(g)iii.

In the pairwise presolving Step 2 we can also look at the individual bits of the two constraints in Step 2a. As usual, we define $r'_b = y'_b = z'_b = 0$ if $b \geq \beta_{r'}$. Now if the two selector variables x and x' are equivalent, the bits of the if-cases y_b and y'_b are equivalent, and the bits of the else-cases z_b and z'_b are also equivalent, the resultant bits r_b and r'_b must also be equivalent. If the operand bits are pairwise negated equivalent, the resultant bits must also be negated equivalent. Conversely, if the selector variables are negated equivalent and the operand bits are (negated) equivalent with interchanged roles, the resultant bits must also be (negated) equivalent.

If we know that the resultants have the same width but are unequal, the implications of Step 2b can be deduced: if an operand from the first constraint is equivalent to an operand of the second constraint, the selector bits x and x' cannot both select this operand, since otherwise, the resultants would also be equivalent. Note that the implications of Steps 2(b)i and 2(b)ii immediately fix the selector bits if $x \stackrel{\star}{=} x'$, and Steps 2(b)iii and 2(b)iv fix them if $x \stackrel{\star}{\neq} x'$. Due to Steps 1e and 1f, this would also lead to the aggregation of the resultants and the deletion of the two constraints.

Steps 2c to 2f check for chainings of ITE constraints. If the resultant r' of one constraint $r' = \text{ITE}(x', y', z')$ appears as operand in another constraint $r = \text{ITE}(x, y, z)$, and if there is a matching implication between the selection variables x and x', we can substitute the resultant r' by the corresponding operand y' or z' in the second constraint. Thereby, we avoid the unnecessary detour via r' and the first ITE constraint. For example, if there are two constraints

$$r = \text{ITE}(x, r', z) \quad \text{and} \quad r' = \text{ITE}(x, y', z'),$$

we can replace r' by y' in the first constraint which gives

$$r = \text{ITE}(x, y', z) \quad \text{and} \quad r' = \text{ITE}(x, y', z').$$

This substitution simplifies the structure of the function graph, see Section 13.2, and offers additional potential for the irrelevance detection of Section 15.2. For example, suppose r' is not used in other constraints. Then we can delete the second constraint from the problem because it is not relevant for the validity of the property. If after the deletion of the constraint z' does only appear as the resultant of a single constraint, it can also be deleted from the problem instance for the same reason. This can trigger a chain of problem reductions with the possibility to delete a whole subtree from the function graph.

14.16 Zero Extension

With the zero extension operator

$$\text{ZEROEXT} : [\mu] \to [\beta], \quad x \mapsto r = \text{ZEROEXT}(x)$$

a register x is copied into a (usually wider) register r with the excessive bits of r fixed to zero. Although usually $\beta > \mu$ we define the constraint for general register widths:

$$r = \text{ZEROEXT}(x) \quad \Leftrightarrow \quad \forall b \in \{0, \ldots, \beta - 1\} : r_b = \begin{cases} x_b & \text{if } b < \mu, \\ 0 & \text{if } b \geq \mu. \end{cases}$$

Thereby it is possible with $\beta < \mu$ to extract a low-significant subword of a register. Nevertheless, the zero extension operator can easily be implemented by performing the corresponding aggregations $r_b :\overset{\star}{=} x_b$ for $b < \mu$ and fixings $r_b := 0$ for $b \geq \mu$ in the presolving stage of SCIP. Afterwards, the constraint can be deleted from the problem formulation.

14.17 Sign Extension

Like the zero extension, the sign extension operator

$$\text{SIGNEXT} : [\mu] \to [\beta], \quad x \mapsto r = \text{SIGNEXT}(x)$$

copies a register x into a register r. But in this case, the excessive bits of r are all equal to the most significant bit of x, thereby preserving the signed value in the two's complement representation. The constraint is defined for general register widths μ and β as

$$r = \text{SIGNEXT}(x) \quad \Leftrightarrow \quad \forall b \in \{0, \ldots, \beta - 1\} : r_b = \begin{cases} x_b & \text{if } b < \mu, \\ x_{\mu-1} & \text{if } b \geq \mu. \end{cases}$$

Again, we can implement the operator by performing the corresponding aggregations $r_b :\overset{\star}{=} x_b$ for $b < \mu$ and $r_b :\overset{\star}{=} x_{\mu-1}$ for $b \geq \mu$ in the presolving stage of SCIP and deleting the constraint afterwards.

14.18 Concatenation

Concatenation with the operator

$$\text{CONCAT} : [\beta] \times [\mu] \to [\beta + \mu], \quad (x, y) \mapsto r = \text{CONCAT}(x, y)$$

means to form a single register bit string out of the two input registers by chaining up their individual bits. The bits of the first input register x form the high-significant part of the result, while the bits of the second input register y are used in the low-significant part:

$$r = \text{CONCAT}(x, y) \quad \Leftrightarrow \quad \forall b \in \{0, \ldots, \beta + \mu - 1\} : r_b = \begin{cases} y_b & \text{if } b < \mu, \\ x_{b-\mu} & \text{if } b \geq \mu. \end{cases}$$

Like the other word extension operators, concatenation can be implemented by performing the necessary presolving aggregations $r_b :\overset{\star}{=} y_b$ for $b < \mu$ and $r_b :\overset{\star}{=} x_{b-\mu}$ for $b \geq \mu$ and deleting the constraint afterwards.

14.19 Shift Left

The shift left operation

$$\text{SHL} : [\beta] \times [\mu] \to [\beta], \quad (x, y) \mapsto r = \text{SHL}(x, y)$$

```
param B  := 64;
set Bits := { 0 .. B-1 };
set DomY := { 0 .. B };

var x[Bits] binary;
var y       integer >= 0 <= B;
var r[Bits] binary;

minimize obj:  0*x[0];

subto    shl:  forall <p,b> in DomY*Bits with b-p >= 0:
                 vif y == p then r[b] == x[b-p] end;
subto    shl0: forall <p,b> in DomY*Bits with b-p <  0:
                 vif y == p then r[b] == 0      end;
```

Figure 14.9. ZIMPL model of the SHL constraint.

performs a shifting of the bits of x by y positions to the left, which corresponds to multiplying x by 2^y. The resultant is defined as

$$r = \text{SHL}(x, y) \quad \Leftrightarrow \quad \forall b \in \{0, \ldots, \beta - 1\} : r_b = \begin{cases} x_{b-y} & \text{if } b \geq y, \\ 0 & \text{if } b < y. \end{cases}$$

As y appears in the *subscript* of the variable x in the definition of the shift left constraint, the SHL operator is related to the ELEMENT constraint of constraint programming (see, e.g., van Hentenryck [115], or Marriott and Stuckey [158]). However, domain propagation can be applied in a much more sophisticated way if the SHL constraint is treated as a whole instead of propagating each ELEMENT constraint $r_b = x_{b-y}$ individually.

The role of y in the constraint is highly non-linear. Therefore, it is very hard to come up with a reasonably small LP relaxation of the shift left constraint. The canonical way of combining the relaxations for each of the involved ELEMENT constraints yields around $\frac{1}{2}\beta^2$ auxiliary variables and $\frac{3}{2}\beta^2$ inequalities. Such a large relaxation would noticeably increase the time to solve the LP relaxations at the nodes of the branch-and-bound tree. Thus, we refrain from including a linear relaxation of the SHL constraint at all.

14.19.1 LP RELAXATION

In a first attempt to provide an LP relaxation for the shift left operand we model the constraint in ZIMPL, see Koch [133, 134]. Figure 14.9 shows a ZIMPL implementation of the SHL constraint $r = \text{SHL}(x, y)$ with $\beta_r = \mathsf{B}$ bits in the registers r and x. To keep things simple, we assume that y is bounded by $u_y = \beta_r$. For all $y \geq \beta_r$ we have $r = 0$ anyway.

The two "subto" statements in the ZIMPL model produce the linear inequalities which define the LP relaxation of the SHL constraint. Note that we use the "vif" command to model a case distinction on the variable y. ZIMPL automatically translates the "vif" statement into a system of inequalities and auxiliary variables.

Table 14.2 shows the number of variables, inequalities, and non-zero coefficients in the linear relaxation generated by ZIMPL 2.04 with simple preprocessing being activated through the "-O" option. It is obvious that this automatically generated

width β	1	2	4	8	16	32	64	128	256
variables	13	35	109	377	1 393	5 345	20 929	82 817	329 473
inequalities	11	37	134	508	1 976	7 792	30 944	123 328	492 416
non-zeros	24	84	312	1 200	4 704	18 624	74 112	295 680	1 181 184

Table 14.2. Size of LP relaxation resulting from Zimpl model of Figure 14.9.

relaxation is way too large for being useful in our context. It has approximately $5\beta^2$ variables, $7\beta^2$ inequalities, and $18\beta^2$ non-zero coefficients.

Our second approach tries to exploit the special structure of the SHL constraint and uses a relaxation similar to the one of the ELEMENT constraint proposed in Milano et al. [164]. Again, we assume $u_y = \beta$ for the sake of simplicity.

$$\sum_{p=0}^{\beta} p \cdot \psi^p = y \tag{14.55}$$

$$\sum_{p=0}^{\beta} \psi^p = 1 \tag{14.56}$$

$$\sum_{p=0}^{b} \pi_{b-p}^p = r_b \quad \text{for all } b = 0, \ldots, \beta - 1 \tag{14.57}$$

$$\pi_b^p - x_b \leq 0 \quad \text{for all } b, p = 0, \ldots, \beta - 1 \text{ with } b + p < \beta \tag{14.58}$$

$$\pi_b^p - \psi^p \leq 0 \quad \text{for all } b, p = 0, \ldots, \beta - 1 \text{ with } b + p < \beta \tag{14.59}$$

$$-\pi_b^p + x_b + \psi^p \leq 1 \quad \text{for all } b, p = 0, \ldots, \beta - 1 \text{ with } b + p < \beta \tag{14.60}$$

Equations (14.55) and (14.56) splits the shift selection variable y into binary variables $\psi^p \in \{0, 1\}$ such that $\psi^y = 1$ and $\psi^p = 0$ for $p \neq y$. Equation (14.57) states that the resultant bit r_b is equal to

$$r_b = x_b \cdot \psi^0 + \ldots + x_0 \cdot \psi^b, \tag{14.61}$$

with inequalities (14.58) to (14.60) providing the linear relaxation for the products $\pi_b^p = x_b \cdot \psi^p$ with $\pi_b^p \in \{0, 1\}$ as already shown in Section 14.7.1. Because $\psi^p = 1$ if and only if $y = p$ due to (14.55), at most one of the products appearing in equation (14.61) can be non-zero. The equation reduces to $r_b = x_{b-y}$ if $b \geq y$ and $r_b = 0$ if $b < y$, which corresponds to the definition of the SHL constraint.

Although the above relaxation is more compact than the automatically generated relaxation of Zimpl, it still includes $\frac{1}{2}\beta^2 + \mathcal{O}(\beta)$ auxiliary variables, $\frac{3}{2}\beta^2 + \mathcal{O}(\beta)$ inequalities and equations, and $4\beta^2 + \mathcal{O}(\beta)$ non-zero coefficients. These are only approximately 10% of the number of variables and 20% of the number of constraints and non-zeros of the Zimpl relaxation, but the size of the relaxation remains quite large, as depicted in Table 14.3. Note that this relaxation is considerably larger than the already large relaxation of the multiplication constraint, see Section 14.5.1. For MULT constraints, we can use nibbles of width $L = 8$ instead of single bits to define the partial products. This reduces both the number of variables and the number of constraints in the LP relaxation by a factor of 8 compared to the above relaxation of the SHL constraint.

Although we did not support this by computational studies, we suppose that the benefit of adding system (14.55) to (14.60) does not justify the large increase in

width β	1	2	4	8	16	32	64	128	256
variables	3	6	15	45	153	561	2 145	8 385	33 153
inequalities	6	13	36	118	426	1 618	6 306	24 898	98 946
non-zeros	13	32	94	314	1 138	4 322	16 834	66 434	263 938

Table 14.3. Size of LP relaxation given by constraints (14.55) to (14.60).

the size of the LP relaxation and the corresponding deterioration of the LP solving process. The relaxation of the SHL constraint becomes even more complicated if the shift selection variably y is not bounded by $u_y = \beta$ but may take larger values. Therefore, we refrain from including a relaxation of shift left constraints into the LP and rely solely on constraint programming techniques, i.e., domain propagation.

14.19.2 DOMAIN PROPAGATION

The domain propagation Algorithm 14.27 for shift left constraints performs a pattern matching to check which values are potentially feasible for the shifting variable y. Step 1 checks whether $x = 0$. In this case, we can also fix $r = 0$, independent from the value of y, and do not need to apply further propagation. Otherwise, we proceed with Step 2 which calculates the current local bounds of the register y by summing up the bounds of the words y^w, $w = 0, \ldots, \omega_y - 1$. Note that in our implementation we demand $\beta < 2^{31}$, which means that we can store all meaningful bound values of y in a 32-bit integer.

The bounds of y can be tightened in Step 3 by inspecting the bits of x and r, see Figure 14.10. If the resultant has a bit which is fixed to one at position p, and in the subword of x up to position p the most significant bit which is not fixed to zero is at position i, x must be shifted at least by $p - i$ bits to the left to yield r. On the other hand, if $r_q = 1$ and j is the least significant position of x with a bit not fixed to zero, x must be shifted at most by $q - j$ bits to the left.

This simple bound tightening helps to shorten the loop of Step 5 where the actual pattern matching takes place. First, we initialize the sets $\mathcal{D}$ of potential values for the involved variables in Step 4 to be empty. Then we check for each value p in the current domain of y whether it would lead to an inconsistency with the current bounds of the bit variables. Condition 5a verifies whether the value is representable with the current fixings of the bits y_b. For example, if $y_0 = 0$ in the current subproblem, only even values p are possible. Condition 5b checks whether the shifting would move a fixed bit $x_{b-p} = v$ to the slot of a resultant bit which is fixed to the opposite value $r_b = 1 - v$. Such a situation rules out the shifting value

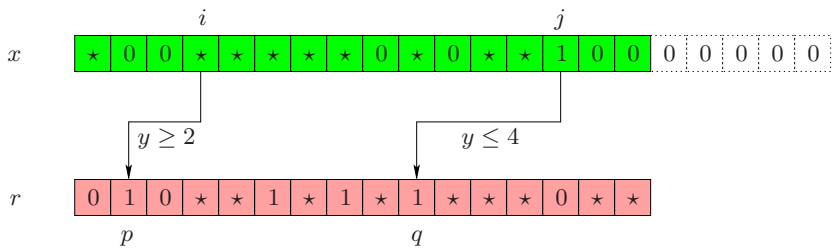

Figure 14.10. Resultant bits fixed to one imply bounds for the shifting variable y.

Algorithm 14.27 Shift Left Domain Propagation

Input: Shift left constraint $r = \text{SHL}(x, y)$ on registers r and x of width $\beta_r = \beta_x = \beta$ with current local bit bounds $\tilde{l}_{r_b} \le r_b \le \tilde{u}_{r_b}$ and $\tilde{l}_{x_b} \le x_b \le \tilde{u}_{x_b}$, and y of width $\beta_y = \mu$ with current local bit bounds $\tilde{l}_{y_b} \le y_b \le \tilde{u}_{y_b}$ and word bounds $\tilde{l}_{y^w} \le y^w \le \tilde{u}_{y^w}$.

Output: Tightened local bounds for bits r_b, x_b, and y_b, and words y^w.

1. If $\tilde{u}_{x_b} = 0$ for all $b = 0, \dots, \beta-1$, deduce $r_b = 0$ for all b and abort propagation.
2. Calculate bounds for the register y from the words' bounds:
$$\tilde{l}_y := \sum_{w=0}^{\omega_y-1} 2^{wW} \tilde{l}_{y^w} \quad \text{and} \quad \tilde{u}_y := \sum_{w=0}^{\omega_y-1} 2^{wW} \tilde{u}_{y^w}.$$
3. Let $b^{r=1}_{\min} := \min\{b \mid \tilde{l}_{r_b} = 1\}$ and $b^{r=1}_{\max} := \max\{b \mid \tilde{l}_{r_b} = 1\}$.
Let $b^{x\neq 0}_{\min} := \min\{b \mid \tilde{u}_{x_b} = 1\}$ and $b^{x\neq 0}_{\max} := \max\{b \mid \tilde{u}_{x_b} = 1 \text{ and } b \le b^{r=1}_{\max}\}$.
Tighten $\tilde{l}_y := \max\left\{\tilde{l}_y, b^{r=1}_{\max} - b^{x\neq 0}_{\max}\right\}$ and $\tilde{u}_y := \min\left\{\tilde{u}_y, b^{r=1}_{\min} - b^{x\neq 0}_{\min}\right\}$.
4. Initialize $\mathcal{D}_{x_b} := \emptyset$ and $\mathcal{D}_{r_b} := \emptyset$ for all $b = 0, \dots, \beta-1$, $\mathcal{D}_{y_b} := \emptyset$ for all $b = 0, \dots, \mu-1$, and $\mathcal{D}_y := \emptyset$.
5. For all $p = \tilde{l}_y, \dots, \tilde{u}_y$:
Let $p = \sum_{b=0}^{\mu-1} 2^b p_b$ be the bit decomposition of p. If the following holds:
   (a) $p_b \in \{\tilde{l}_{y_b}, \tilde{u}_{yb}\}$ for all $b = 0, \dots, \mu-1$,
   (b) $\mathcal{D}_b := \{\tilde{l}_{r_b}, \tilde{u}_{r_b}\} \cap \{\tilde{l}_{x_{b-p}}, \tilde{u}_{x_{b-p}}\} \neq \emptyset$ for all $b = 0, \dots, \beta-1$, and
   (c) there is no bit $b \in \{0, \dots, \beta-1\}$ with $r_b \overset{\star}{\neq} x_{b-p}$,

   the shifting value $y = p$ is valid. In this case, update
   (a) $\mathcal{D}_y := \mathcal{D}_y \cup \{p\}$,
   (b) $\mathcal{D}_{y_b} := \mathcal{D}_{y_b} \cup \{p_b\}$ for all $b = 0, \dots, \mu-1$,
   (c) $\mathcal{D}_{r_b} := \mathcal{D}_{r_b} \cup \mathcal{D}_b$ for all $b = 0, \dots, \beta-1$,
   (d) $\mathcal{D}_{x_b} := \mathcal{D}_{x_b} \cup \mathcal{D}_{b+p}$ for all $b = 0, \dots, \beta-1-p$, and
   (e) $\mathcal{D}_{x_b} := \{\tilde{l}_{x_b}, \tilde{u}_{x_b}\}$ for all $b = \beta-p, \dots, \beta-1$.
6. Tighten word bounds of y^w, $w = 0, \dots, \omega_y - 1$, corresponding to the register bounds $\min\{\mathcal{D}_y\} \le y \le \max\{\mathcal{D}_y\}$.
7. For all $b = 0, \dots, \mu-1$: Tighten $\min\{\mathcal{D}_{y_b}\} \le y_b \le \max\{\mathcal{D}_{y_b}\}$.
For all $b = 0, \dots, \beta-1$: Tighten $\min\{\mathcal{D}_{r_b}\} \le r_b \le \max\{\mathcal{D}_{r_b}\}$.
For all $b = 0, \dots, \beta-1$: Tighten $\min\{\mathcal{D}_{x_b}\} \le x_b \le \max\{\mathcal{D}_{x_b}\}$.

p, see Figure 14.11. Note that we define $x_i = 0$ for all $i < 0$, since the SHL operator moves in zeros to the least significant bits of r. The value $y = p$ is also invalid by Condition 5c if there exists a pair $r_b \overset{\star}{\neq} x_{b-p}$ of negated equivalent bits. If all three conditions are satisfied, the shifting value p is potentially feasible, and we extend the sets of potential values for the involved variables accordingly. Bits $x_{\beta-p}, \dots, x_{\beta-1}$ do not need to be considered for the pattern matching, because they are "shifted out" of the region covered by r. Therefore, if p is a valid match, these bits of x are not affected by the bits of r, and their sets of potential values have to be set to their current domains $\mathcal{D}_{x_b} := \{\tilde{l}_{x_b}, \tilde{u}_{x_b}\}$.

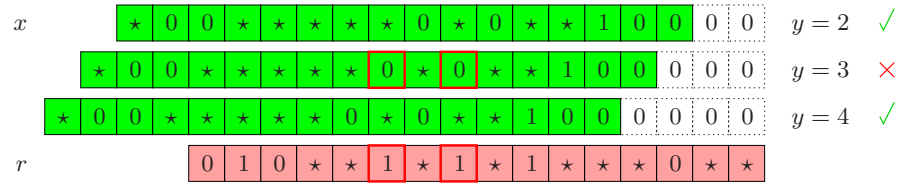

Figure 14.11. Pattern matching to identify potential values of y.

After the pattern matching was performed, we inspect the resulting sets $\mathcal{D}$ of potential values to tighten the bounds of the variables. In Step 6, we tighten the bounds of y to the minimal and maximal feasible values for p, respectively. Since y itself is not a CIP variable, we have to apply the bound changes to the words y^w. Note, however, that usually only the most significant word y^{ω_y-1} can be tightened. Deductions on the bounds of a word $w \in \{0, \ldots, \omega_y - 2\}$ can only be performed, if all more significant words $w+1, \ldots, \omega_y - 1$ are fixed. Fortunately, in most applications y consists of only one word, since with the word size $W = 16$ a single word of y suffices to treat registers x and r of up to $2^{16} - 1$ bits. Finally in Step 7, we fix those bits y_b, r_b, x_b, which only have a single element in their corresponding sets of potential values, see Figure 14.12.

14.19.3 PRESOLVING

The presolving for shift left constraints is depicted in Algorithm 14.28. Step 1a detects the inequality of x and r, if the shifting operand is non-zero. Note that even with $y > 0$, r could be equal to x if $r = x = 0$. Therefore, we have to explicitly exclude this case. The same reasoning is applied the other way around in Step 1b. If r and x are equivalent, the shifting operand y must be zero. This, however, is only true if $r = x \neq 0$. If we know that r and x are unequal, i.e., $r \overset{\star}{\neq} x$, we can

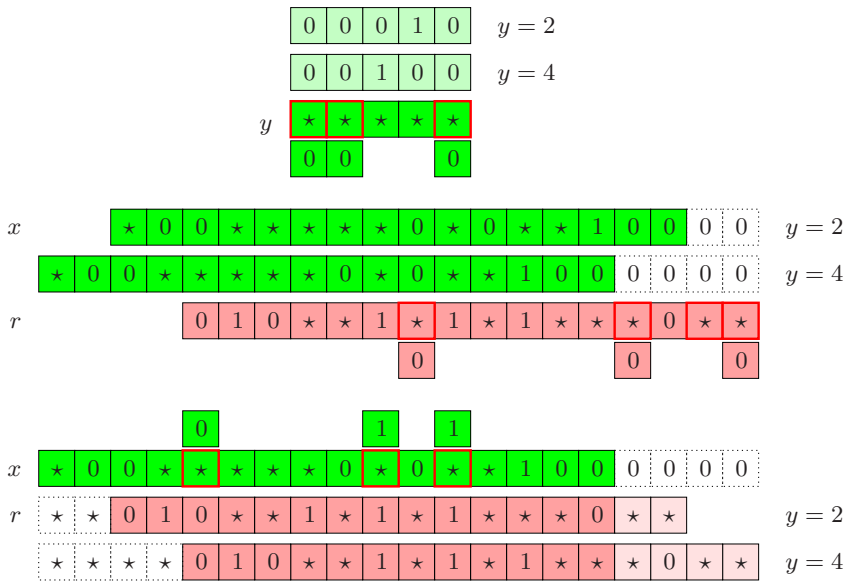

Figure 14.12. Fixing of shifting operand bits y_b (top), resultant bits r_b (middle), and first operand bits x_b (bottom).

Algorithm 14.28 Shift Left Presolving

1. For all active shift left constraints $r = \text{SHL}(x, y)$:
   (a) If $y^w > 0$ for any word $w \in \{0, \dots, \omega_y - 1\}$, and $x^w > 0$ or $r^w > 0$ for any $w \in \{0, \dots, \omega_x - 1\}$, deduce $r \stackrel{\star}{\neq} x$.
   (b) If $r \stackrel{\star}{=} x$ and $x^w > 0$ for any $w \in \{0, \dots, \omega_x - 1\}$, fix $y := 0$.
   (c) If $r \stackrel{\star}{\neq} x$ and $\omega_y = 1$, deduce $y^0 \geq 1$.
   (d) Apply domain propagation Algorithm 14.27 on the global bounds.
   (e) If $l_y = u_y$, aggregate $r_b :\stackrel{\star}{=} x_{b-l_y}$ for all $b = 0, \dots, \beta_r - 1$ using $x_i = 0$ for $i < 0$, and delete the constraint.
   (f) For all $b = 0, \dots, \beta_r - 1$:
      i. Let $p^b_{\min} = \min\{p \mid l_y \leq p \leq u_y \text{ and } u_{x_{b-p}} = 1\}$ be the minimal and $p^b_{\max} = \max\{p \mid l_y \leq p \leq u_y \text{ and } u_{x_{b-p}} = 1\}$ be the maximal shifting value y for $r_b = 1$.
      ii. For all $w = 0, \dots, \omega_y - 1$: Add implications

$$r_b = 1 \rightarrow y^w \geq \left\lceil \frac{p^b_{\min} - \sum_{i \neq w} 2^{iW} u_{y^i}}{2^{wW}} \right\rceil \text{ and}$$

$$r_b = 1 \rightarrow y^w \leq \left\lfloor \frac{p^b_{\max} - \sum_{i \neq w} 2^{iW} l_{y^i}}{2^{wW}} \right\rfloor$$

      to the implication graph of SCIP.
      iii. If $x_i = 0$ for all $i < b$, add implication $x_b = 0 \rightarrow r_{b+l_y} = 0$ to the implication graph of SCIP.

2. For all pairs of active shift left constraints $r = \text{SHL}(x, y)$ and $r' = \text{SHL}(x', y')$ with $\beta_r \geq \beta_{r'}$:
   (a) If $x \stackrel{\star}{=} x'$ and $y \stackrel{\star}{=} y'$, aggregate $r[\beta_{r'} - 1, 0] :\stackrel{\star}{=} r'$ and delete constraint $r' = \text{SHL}(x', y')$.
   (b) If $\beta_r = \beta_{r'}$, $x \stackrel{\star}{=} x'$, and $r \stackrel{\star}{\neq} r'$, deduce $y \stackrel{\star}{\neq} y'$.
   If $\beta_r = \beta_{r'}$, $y \stackrel{\star}{=} y'$, and $r \stackrel{\star}{\neq} r'$, deduce $x \stackrel{\star}{\neq} x'$.
   (c) If $r \stackrel{\star}{=} r'$ and $x \stackrel{\star}{=} x'$, and if $r^w \geq 1$ for any word $w \in \{0, \dots, \omega_r - 1\}$, aggregate $y :\stackrel{\star}{=} y'$ and delete constraint $r' = \text{SHL}(x', y')$.

deduce $y \geq 1$ in Step 1c. Since y itself is not a CIP variable, we have to apply the deduction on the words of y. This is only possible if y consists of only a single word.

Step 1d applies the domain propagation on the global bounds. If it turns out that the shifting variable y is fixed, we can aggregate the resultant r and the first operand x accordingly in Step 1e and delete the constraint afterwards.

Step 1f adds implications to the implication graph of SCIP which can be deduced from the SHL constraint. We regard the bounds $l_y \leq y \leq u_y$ of the shifting register y as calculated in Step 2 of Algorithm 14.27. If a resultant bit r_b is set to one, the value of the shifting variable y must be in the range $p^b_{\min} \leq y \leq p^b_{\max}$, which is computed in Step 1(f)i by similar reasoning as in Step 3 of Algorithm 14.27. Step 1(f)ii translates these bounds on y into bounds on the individual words y^w. In order to do this, we have to assume that all other words $i \neq w$ are set to their upper or lower bounds, respectively, and divide the leftover value by the significance of the word.

The domain propagation already fixed $r_0 = \ldots = r_{l_y-1} = 0$. The next bit r_{l_y} is either equal to x_0 or zero, depending on whether $y = l_y$ or $y > l_y$. Thus, the implication $x_0 = 0 \rightarrow r_{l_y} = 0$ is valid. If all of the lower significant bits up to bit b are zero in register x, we can extend this argument to the more significant bits x_b and r_{b+l_y}.

Step 2 compares pairs of SHL constraints. Clearly, if the operands are equivalent as required for Step 2a, we can aggregate the resultants on their overlapping part and delete the constraint operating on the shorter registers. Applying this implication in the inverse direction in Step 2b, we can conclude from the inequality of two equally wide resultant registers and the equivalence of one pair of operands, that the other operand pair must be unequal. If the resultants are equivalent, and if the first operands are equivalent, we can aggregate $y :\overset{\star}{=} y'$ in Step 2c. However, we have to exclude $r = r' = 0$. In this case, the shifting operands y and y' could take any values that "shift out" all bits of $x \overset{\star}{=} x'$ which are set to one.

14.20 SHIFT RIGHT

The shift right operation

$$\text{SHR} : [\beta] \times [\mu] \rightarrow [\beta], \quad (x, y) \mapsto r = \text{SHR}(x, y)$$

performs a shifting of the bits of x by y positions to the right. The resultant is defined as

$$r = \text{SHR}(x, y) \quad \Leftrightarrow \quad \forall b \in \{0, \ldots, \beta - 1\} : r_b = \begin{cases} x_{b+y} & \text{if } b + y < \beta, \\ 0 & \text{if } b + y \geq \beta. \end{cases}$$

The shift right operator is a special case of the more general SLICE operator, which is covered in the next section. Therefore, it suffices to replace a constraint $r = \text{SHR}(x, y)$ by $r = \text{SLICE}(x, y)$ in the presolving stage of SCIP.

14.21 SLICING

The slice operator

$$\text{SLICE} : [\mu] \times [\nu] \rightarrow [\beta], \quad (x, y) \mapsto r = \text{SLICE}(x, y)$$

allows to access subwords of the operand x. The resultant is defined as

$$r = \text{SLICE}(x, y) \quad \Leftrightarrow \quad \forall b \in \{0, \ldots, \beta - 1\} : r_b = \begin{cases} x_{b+y} & \text{if } b + y < \mu, \\ 0 & \text{if } b + y \geq \mu, \end{cases}$$

which means that $r = x[y + \beta_r - 1, y]$. For $\mu = \beta$, the slice operator is equivalent to the shift right operator, i.e., $\text{SLICE}(x, y) = \text{SHR}(x, y)$. Therefore, SLICE can be seen as generalization of SHR.

Consequently, slice constraints are very similar to shift left constraints. The LP relaxation would be as involved as the one of the shift left operator. Therefore, we also refrain from including a linear relaxation of SLICE in the LP. Like for the SHL constraint, the domain propagation of the slice constraint applies a pattern matching algorithm to check which values for the slice start operand y are possible.

```
param Bx  := 64;
param Br  := 16;
set BitsX := { 0 .. Bx-1 };
set BitsR := { 0 .. Br-1 };
set DomY  := { 0 .. Bx };

var x[BitsX] binary;
var y        integer >= 0 <= Bx;
var r[BitsR] binary;

minimize obj:  0*x[0];

subto    slice:  forall <p,b> in DomY*BitsR with b+p <  Bx:
                   vif y == p then r[b] == x[b+p] end;
subto    slice0: forall <p,b> in DomY*BitsR with b+p >= Bx:
                   vif y == p then r[b] == 0      end;
```

Figure 14.13. Zimpl model of the slice constraint.

14.21.1 LP Relaxation

Like for the shl constraint, we tried to generate an LP relaxation for the slice operand by modeling the constraint $r = \text{slice}(x, y)$ in Zimpl as shown in Figure 14.13 with $\beta_r = \text{Br}$ and $\beta_x = \text{Bx}$. For $\beta_x = \beta_r$ (i.e., for shr constraints) this yields LP relaxations of the same size as the ones of the shl constraint, see Table 14.2. We could also use a more compact model similar to the one of shl constraints given by equations (14.55) to (14.60). But again, this would still be a very large relaxation, such that we decided to not include it in the LP relaxation of the property checking CIP.

14.21.2 Domain Propagation

Algorithm 14.29 depicts the domain propagation for slice constraints. It performs a pattern matching to check which values are potentially feasible for the slice start variable y. Besides the potentially different bit widths β_r and β_x, it only differs from the domain propagation Algorithm 14.27 in Steps 3 and 5. Figure 14.14 illustrates Step 3. If $r_p = 1$ and the maximal position with a bit x_b not fixed to zero is i, the slice start variable y can be at most $i - p$, i.e., $y \leq i - p$. If $r_q = 1$ and the minimal position not smaller than q with a bit x_b not fixed to zero is j, we can conclude $y \geq j - p$. In Step 5 we define $x_b = 0$ for $b \geq \beta_x$ as usual. One update of the sets $\mathcal{D}$ of potential values for the involved variables is illustrated in Figure 14.15.

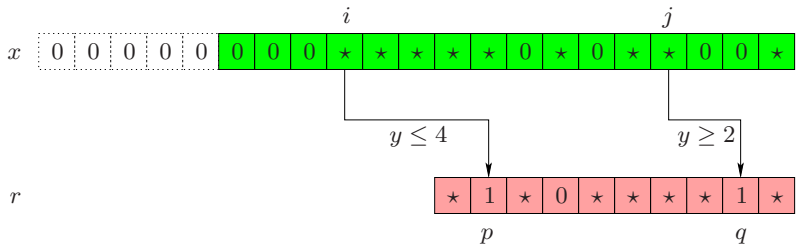

Figure 14.14. Resultant bits fixed to one imply bounds for the slice start variable y.

Algorithm 14.29 Slice Domain Propagation

Input: Slice constraint $r = \text{SLICE}(x, y)$ on registers r, x, and y with current local bit bounds $\tilde{l}_{r_b} \le r_b \le \tilde{u}_{r_b}$, $\tilde{l}_{x_b} \le x_b \le \tilde{u}_{x_b}$, and $\tilde{l}_{y_b} \le y_b \le \tilde{u}_{y_b}$, and current local word bounds $\tilde{l}_{y^w} \le y^w \le \tilde{u}_{y^w}$.

Output: Tightened local bounds for bits r_b, x_b, and y_b, and words y^w.

1. If $\tilde{u}_{x_b} = 0$ for all $b = 0, \dots, \beta_x - 1$, deduce $r_b = 0$ for all $b = 0, \dots, \beta_r - 1$ and abort the propagation.
2. Calculate bounds for the register y from the words' bounds:
$$\tilde{l}_y := \sum_{w=0}^{\omega_y - 1} 2^{wW} \tilde{l}_{y^w} \quad \text{and} \quad \tilde{u}_y := \sum_{w=0}^{\omega_y - 1} 2^{wW} \tilde{u}_{y^w}.$$
3. Let $b^{r=1}_{\min} := \min\{b \mid \tilde{l}_{r_b} = 1\}$ and $b^{r=1}_{\max} := \max\{b \mid \tilde{l}_{r_b} = 1\}$.
Let $b^{x\neq 0}_{\min} := \min\{b \mid \tilde{u}_{x_b} = 1 \text{ and } b \ge b^{r=1}_{\min}\}$ and $b^{x\neq 0}_{\max} := \max\{b \mid \tilde{u}_{x_b} = 1\}$.
Tighten $\tilde{l}_y := \max\left\{\tilde{l}_y, b^{x\neq 0}_{\min} - b^{r=1}_{\min}\right\}$ and $\tilde{u}_y := \min\left\{\tilde{u}_y, b^{x\neq 0}_{\max} - b^{r=1}_{\max}\right\}$.
4. Initialize $\mathcal{D}_{x_b} := \emptyset$ for all $b = 0, \dots, \beta_x - 1$, $\mathcal{D}_{r_b} := \emptyset$ for all $b = 0, \dots, \beta_r - 1$, $\mathcal{D}_{y_b} := \emptyset$ for all $b = 0, \dots, \beta_y - 1$, and $\mathcal{D}_y := \emptyset$.
5. For all $p = \tilde{l}_y, \dots, \tilde{u}_y$:
Let $p = \sum_{b=0}^{\beta_y - 1} 2^b p_b$ be the bit decomposition of p. If the following holds:
   (a) $p_b \in \{\tilde{l}_{y_b}, \tilde{u}_{yb}\}$ for all $b = 0, \dots, \beta_y - 1$,
   (b) $\mathcal{D}_b := \{\tilde{l}_{r_b}, \tilde{u}_{r_b}\} \cap \{\tilde{l}_{x_{b+p}}, \tilde{u}_{x_{b+p}}\} \neq \emptyset$ for all $b = 0, \dots, \beta_r - 1$, and
   (c) there is no bit $b \in \{0, \dots, \beta_r - 1\}$ with $r_b \overset{\star}{\neq} x_{b+p}$,

   the slice start value $y = p$ is valid. In this case, update
   (a) $\mathcal{D}_y := \mathcal{D}_y \cup \{p\}$,
   (b) $\mathcal{D}_{y_b} := \mathcal{D}_{y_b} \cup \{p_b\}$ for all $b = 0, \dots, \beta_y - 1$,
   (c) $\mathcal{D}_{r_b} := \mathcal{D}_{r_b} \cup \mathcal{D}_b$ for all $b = 0, \dots, \beta_r - 1$,
   (d) $\mathcal{D}_{x_b} := \mathcal{D}_{x_b} \cup \mathcal{D}_{b-p}$ for all $b = p, \dots, n_p - 1$, and
   (e) $\mathcal{D}_{x_b} := \{\tilde{l}_{x_b}, \tilde{u}_{x_b}\}$ for all $b = 0, \dots, p - 1$ and all $b = n_p, \dots, \beta_x - 1$,

   with $n_p = \min\{\beta_r + p, \beta_x\}$.
6. Tighten word bounds of y^w, $w = 0, \dots, \omega_y - 1$, corresponding to the register bounds $\min\{\mathcal{D}_y\} \le y \le \max\{\mathcal{D}_y\}$.
7. For all $b = 0, \dots, \beta_y - 1$: Tighten $\min\{\mathcal{D}_{y_b}\} \le y_b \le \max\{\mathcal{D}_{y_b}\}$.
For all $b = 0, \dots, \beta_r - 1$: Tighten $\min\{\mathcal{D}_{r_b}\} \le r_b \le \max\{\mathcal{D}_{r_b}\}$.
For all $b = 0, \dots, \beta_x - 1$: Tighten $\min\{\mathcal{D}_{x_b}\} \le x_b \le \max\{\mathcal{D}_{x_b}\}$.

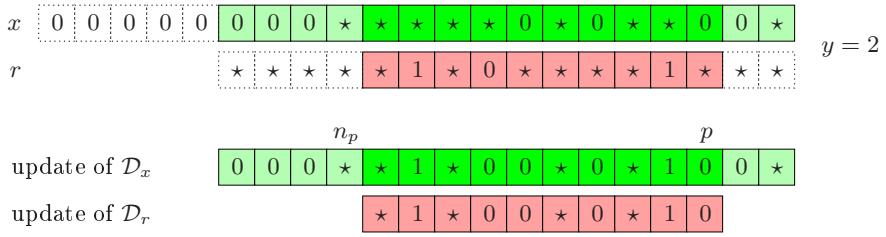

Figure 14.15. Update of the potential values sets $\mathcal{D}_x$ and $\mathcal{D}_r$ for a pattern match in $y = p$. Each set $\mathcal{D}$ is extended by the set represented by the corresponding symbol "0", "1", or "⋆" in the update row, which denote the sets $\{0\}$, $\{1\}$, and $\{0, 1\}$, respectively.

The resultant sets $\mathcal{D}_{r_b}$ and the sets $\mathcal{D}_{x_{b+p}}$ for those operand bits x_{b+p} which overlap with the resultant are extended by the corresponding intersection $\mathcal{D}_b$ of their current domains. The remaining bits of x are not affected by the bits of r, and their sets of potential values have to be set to their current domains $\mathcal{D}_{x_b} := \{\tilde{l}_{x_b}, \tilde{u}_{x_b}\}$.

14.21.3 Presolving

Algorithm 14.30 illustrates the presolving for slice constraints. It differs marginally from the presolving Algorithm 14.28 for shift left constraints. In Step 1c we can only deduce $y \geq 1$ if $\beta_r \geq \beta_x$. For example if $(x_b)_b = (1, 1)$ and $(r_b)_b = (1)$ it clearly follows $r \overset{\star}{\neq} x$, but $y = 0$ is still a valid slice start value. In contrast to shift left constraints, the implications added in Step 1(f)iii operate on the *most* significant bits of r and x. If $x_{\beta_x - 1} = \ldots = x_b = 0$, then it must also be $r_i = 0$ for all $i \geq b - l_y$. Note that we define $r_i = 0$ for $i \geq \beta_r$, which means that implications with consequents $x_i = 0$, $i \geq \beta_r$, are redundant and can be ignored.

Steps 2a and 2b are equal to the corresponding steps in Algorithm 14.28. In contrast to the shift left operator, we cannot aggregate $y :\overset{\star}{=} y'$ if $r \overset{\star}{=} r'$ and $x \overset{\star}{=} x'$ in Step 2c for any register widths. For example, the constraint $r = \text{SLICE}(x, y)$ with bit strings $(x_b)_b = (0, 1, 0, 1, 0, 1, 0, 1)$ and $(r_b)_b = (0, 1, 0, 1)$ has three different solutions for y, namely $y = 0$, $y = 2$, and $y = 4$. The aggregation $y :\overset{\star}{=} y'$ can only be performed, if the widths of the operands do not exceed the widths of the resultants.

14.22 Multiplex Read

A multiplexer is a module in a digital circuit that allows to select an output from a set of input signals by a control signal. One can think of the multiplex read operator

$$\text{READ} : [\mu] \times [\nu] \to [\beta], \quad (x, y) \mapsto r = \text{READ}(x, y)$$

with

$$r = \text{READ}(x, y) \quad \Leftrightarrow \quad \forall b \in \{0, \ldots, \beta - 1\} : r_b = \begin{cases} x_{b+y \cdot \beta} & \text{if } b + y \cdot \beta < \mu, \\ 0 & \text{if } b + y \cdot \beta \geq \mu, \end{cases}$$

as a read access $r = x[y]$ to an array x with elements of β bits, as illustrated in Figure 14.16. In a typical READ constraint, the array register x can be quite large. For example, x can represent internal memory like the level-1 cache of a CPU. Another application of a multiplexer is the serialization of data into a stream of

Algorithm 14.30 Slice Presolving

1. For all active slice constraints $r = \text{SLICE}(x, y)$:
   (a) If $y^w > 0$ for any word $w \in \{0, \dots, \omega_y - 1\}$, and $x^w > 0$ or $r^w > 0$ for any $w \in \{0, \dots, \omega_x - 1\}$, deduce $r \stackrel{\star}{\neq} x$.
   (b) If $r \stackrel{\star}{=} x$ and $x^w > 0$ for any $w \in \{0, \dots, \omega_x - 1\}$, fix $y := 0$.
   (c) If $r \stackrel{\star}{\neq} x$, $\beta_r \geq \beta_x$, and $\omega_y = 1$, deduce $y^0 \geq 1$.
   (d) Apply domain propagation Algorithm 14.29 on the global bounds.
   (e) If $l_y = u_y$, aggregate $r_b :\stackrel{\star}{=} x_{b+l_y}$ for all $b = 0, \dots, \beta_r - 1$ using $x_i = 0$ for $i \geq \beta_x$, and delete the constraint.
   (f) For all $b = 0, \dots, \beta_r - 1$:
      i. Let $p^b_{\min} = \min\{p \mid l_y \leq p \leq u_y \text{ and } u_{x_{b+p}} = 1\}$ be the minimal and $p^b_{\max} = \max\{p \mid l_y \leq p \leq u_y \text{ and } u_{x_{b+p}} = 1\}$ be the maximal slice start value y for $r_b = 1$.
      ii. For all $w = 0, \dots, \omega_y - 1$: Add implications

$$r_b = 1 \to y^w \geq \left\lceil \frac{p^b_{\min} - \sum_{i \neq w} 2^{iW} u_{y^i}}{2^{wW}} \right\rceil \text{ and}$$

$$r_b = 1 \to y^w \leq \left\lfloor \frac{p^b_{\max} - \sum_{i \neq w} 2^{iW} l_{y^i}}{2^{wW}} \right\rfloor$$

      to the implication graph of SCIP.
      iii. If $x_i = 0$ for all $i > b$, add implication $x_b = 0 \to r_{b-l_y} = 0$ to the implication graph of SCIP.

2. For all pairs of active slice constraints $r = \text{SLICE}(x, y)$ and $r' = \text{SLICE}(x', y')$ with $\beta_r \geq \beta_{r'}$:
   (a) If $x \stackrel{\star}{=} x'$ and $y \stackrel{\star}{=} y'$, aggregate $r[\beta_{r'} - 1, 0] :\stackrel{\star}{=} r'$ and delete constraint $r' = \text{SHL}(x', y')$.
   (b) If $\beta_r = \beta_{r'}$, $x \stackrel{\star}{=} x'$, and $r \stackrel{\star}{\neq} r'$, deduce $y \stackrel{\star}{\neq} y'$.
   If $\beta_r = \beta_{r'}$, $y \stackrel{\star}{=} y'$, and $r \stackrel{\star}{\neq} r'$, deduce $x \stackrel{\star}{\neq} x'$.
   (c) If $\beta_r \geq \beta_x$, $\beta_{r'} \geq \beta_{x'}$, $r \stackrel{\star}{=} r'$, and $x \stackrel{\star}{=} x'$, and if $r^w \geq 1$ for any word $w \in \{0, \dots, \omega_r - 1\}$, aggregate $y :\stackrel{\star}{=} y'$ and delete constraint $r' = \text{SHL}(x', y')$.

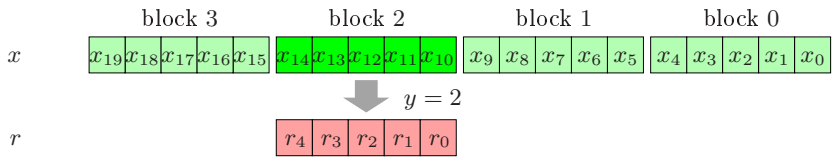

Figure 14.16. Multiplex read with array elements of $\beta = 5$ bits.

```
param Bx   := 1024;
param Br   := 8;
param nBlk:= ceil(Bx/Br);
set BitsX  := { 0 .. Bx-1 };
set BitsR  := { 0 .. Br-1 };
set DomY   := { 0 .. nBlk };
set Blk    := { 0 .. nBlk-1 };

var x[BitsX] binary;
var y        integer >= 0 <= nBlk;
var r[BitsR] binary;

minimize obj:       0*x[0];

subto    muxread:   forall <p,b> in Blk*BitsR with b + p*Br < Bx:
                      vif y == p then r[b] == x[b+p*Br] end;
subto    muxread0:  forall <p,b> in Blk*BitsR with b + p*Br >= Bx:
                      vif y == p then r[b] == 0         end;
```

Figure 14.17. Zimpl model of the multiplex read constraint.

bits or words. In this case, the block selection variable y would be incremented at each time step such that the data words are consecutively passed to the data bus represented by the resultant register r.

The multiplex read operator is very similar to the slice operator presented in the previous section. The only difference in the bitwise definition is the multiplication of the block selection variable y by the resultant's width β. Therefore, the algorithms for the two constraint classes share most of the ideas, in particular the pattern matching in the domain propagation. In contrast to the shift left and slice operators, the LP relaxation of the multiplex read constraint is much smaller with respect to the size of the input registers. It only needs a number of inequalities and auxiliary variables which grows linearly with the width of x.

14.22.1 LP Relaxation

The difference of the multiplex read operator to the slice operator is that each bit x_b is only copied to the resultant for exactly one value of the block selection operand y. Furthermore, if its block is selected, the bit x_b is copied to a predefined location in the resultant register r. This leads to a linear relaxation which grows linearly in β_x, in contrast to the quadratically growing relaxation of the slice constraint.

Figure 14.17 shows a Zimpl model of the constraint $r = \text{READ}(x, y)$. Like before, we assume that y can only hit one "non-existing" block in x, i.e., the upper bound of y is $y \leq \lceil \frac{\beta_x}{\beta_r} \rceil$. Although the Zimpl model looks pretty similar to the one in Figure 14.13 for slice constraints, one has to observe that the "forall" loop in the constraints only ranges over $\lceil \frac{\beta_x}{\beta_r} \rceil \cdot \beta_r \approx \beta_x$ elements, while the loops in the SLICE model range over $(\beta_x + 1) \cdot \beta_r$ elements. This difference is also highlighted by the relaxation sizes depicted in Table 14.4, which were calculated using $\beta_r = 8$. Note that these numbers change only marginally for different values of β_r.

We improve the automatically Zimpl-generated relaxation by constructing a cus-

width β_x	8	16	32	64	128	256	512	1 024	2 048
variables	57	105	201	393	777	1 545	3 081	6 153	12 297
inequalities	48	112	240	496	1 008	2 032	4 080	8 176	16 368
non-zeros	112	272	592	1 232	2 512	5 072	10 192	20 432	40 912

Table 14.4. Size of LP relaxation resulting from ZIMPL model of Figure 14.17 for fixed $\beta_r = 8$.

tomized compact LP relaxation as follows:

$$\sum_{p=l_y}^{u_y} p \cdot \psi^p = y \tag{14.62}$$

$$\sum_{p=l_y}^{u_y} \psi^p = 1 \tag{14.63}$$

$$r_b - x_{b+p\cdot\beta_r} \leq 1 - \psi^p \text{ for all } b = 0, \ldots, \beta_r - 1 \text{ and } p = l_y, \ldots, u_y \tag{14.64}$$

$$-r_b + x_{b+p\cdot\beta_r} \leq 1 - \psi^p \text{ for all } b = 0, \ldots, \beta_r - 1 \text{ and } p = l_y, \ldots, u_y \tag{14.65}$$

This relaxation is only applicable if the block selection operand consists of only a single word $y = y^0$, which is, however, a reasonable assumption. We do not linearize multiplex read constraints with $\omega_y \geq 2$.

The relaxation contains $u_y - l_y + 1$ auxiliary binary variables $\psi^p \in \{0, 1\}$ and $2\beta_r(u_y - l_y + 1) + 2$ constraints. As in the proposed relaxation of the SHL constraint, equations (14.62) and (14.63) disaggregate the block selection operand y into a series of binary variables ψ^p, each of which corresponding to a single value $p \in \{l_y, \ldots, u_y\}$ in the domain of y. Inequalities (14.64) and (14.65) model the implication

$$y = p \rightarrow r_b = x_{b+p\cdot\beta_r}.$$

Here we define again $x_i = 0$ for $i \geq \beta_x$ such that this implication conforms to the definition of the READ operator. Since exactly one of ψ^p is equal to one, the resultant bits r_b are uniquely and well-defined.

14.22.2 DOMAIN PROPAGATION

The domain propagation Algorithm 14.31 is very similar to the previous Algorithms 14.27 and 14.29 for SHL and SLICE constraints, respectively. Since the pattern matching Step 3 for READ constraints runs in linear time in β_x, we do not need to pre-calculate tighter bounds for y as in Step 3 of Algorithm 14.29. We can also dispense with the check whether x is equal to zero, since this is automatically performed with equal effort in the pattern matching of Step 3. The pattern matching itself compares—corresponding to the definition of the READ operand—the resultant bits r_b with the proper operand bits $x_{b+p\cdot\beta_r}$ in block p of x. If a potential match is identified, the sets of potential values are updated as usual. It does not make sense to keep track of the potential values for x_b, since they will be unrestricted as soon as there are two different valid blocks p. The special case of only a single valid block $\mathcal{D}_y = \{p\}$ is dealt with in Step 4, where we propagate the equation $r_b = x_{b+p\cdot\beta_r}$ for all $b = 0, \ldots, \beta_r - 1$. Finally, as for the SHL and SLICE constraints, the calculated sets of potential values are evaluated in Steps 5 and 6 to tighten the bounds of the variables y^w, y_b, and r_b.

Algorithm 14.31 Multiplex Read Domain Propagation

Input: Multiplex read constraint $r = \text{READ}(x, y)$ on registers r, x, and y with current local bit bounds $\tilde{l}_{r_b} \leq r_b \leq \tilde{u}_{r_b}$, $\tilde{l}_{x_b} \leq x_b \leq \tilde{u}_{x_b}$, and $\tilde{l}_{y_b} \leq y_b \leq \tilde{u}_{y_b}$, and current local word bounds $\tilde{l}_{y^w} \leq y^w \leq \tilde{u}_{y^w}$.

Output: Tightened local bounds for bits r_b, x_b, and y_b, and words y^w.

1. Calculate bounds for the register y from the words' bounds:
$$\tilde{l}_y := \sum_{w=0}^{\omega_y-1} 2^{wW} \tilde{l}_{y^w} \quad \text{and} \quad \tilde{u}_y := \sum_{w=0}^{\omega_y-1} 2^{wW} \tilde{u}_{y^w}.$$
2. Initialize $\mathcal{D}_{r_b} := \emptyset$ for all $b = 0, \ldots, \beta_r - 1$, $\mathcal{D}_{y_b} := \emptyset$ for all $b = 0, \ldots, \beta_y - 1$, and $\mathcal{D}_y := \emptyset$.
3. For all $p = \tilde{l}_y, \ldots, \tilde{u}_y$:
   Let $p = \sum_{b=0}^{\beta_y-1} 2^b p_b$ be the bit decomposition of p. If the following holds:
   (a) $p_b \in \{\tilde{l}_{y_b}, \tilde{u}_{yb}\}$ for all $b = 0, \ldots, \beta_y - 1$,
   (b) $\mathcal{D}_b := \{\tilde{l}_{r_b}, \tilde{u}_{r_b}\} \cap \{\tilde{l}_{x_{b+p\cdot\beta_r}}, \tilde{u}_{x_{b+p\cdot\beta_r}}\} \neq \emptyset$ for all $b = 0, \ldots, \beta_r - 1$, and
   (c) there is no bit $b \in \{0, \ldots, \beta_r - 1\}$ with $r_b \overset{\star}{\neq} x_{b+p\cdot\beta_r}$,

   the block selection value $y = p$ is valid. In this case, update
   (a) $\mathcal{D}_y := \mathcal{D}_y \cup \{p\}$,
   (b) $\mathcal{D}_{y_b} := \mathcal{D}_{y_b} \cup \{p_b\}$ for all $b = 0, \ldots, \beta_y - 1$, and
   (c) $\mathcal{D}_{r_b} := \mathcal{D}_{r_b} \cup \mathcal{D}_b$ for all $b = 0, \ldots, \beta_r - 1$.
4. If $\mathcal{D}_y = \{p\}$, then for all $b = 0, \ldots, \beta_r - 1$:
   (a) If $\tilde{l}_{r_b} = \tilde{u}_{r_b}$, deduce $x_{b+p\cdot\beta_r} = \tilde{l}_{r_b}$.
   (b) If $\tilde{l}_{x_{b+p\cdot\beta_r}} = \tilde{u}_{x_{b+p\cdot\beta_r}}$, deduce $r_b = \tilde{l}_{x_{b+p\cdot\beta_r}}$.
5. Tighten word bounds of y^w, $w = 0, \ldots, \omega_y - 1$, corresponding to the register bounds $\min\{\mathcal{D}_y\} \leq y \leq \max\{\mathcal{D}_y\}$.
6. For all $b = 0, \ldots, \beta_y - 1$: Tighten $\min\{\mathcal{D}_{y_b}\} \leq y_b \leq \max\{\mathcal{D}_{y_b}\}$.
   For all $b = 0, \ldots, \beta_r - 1$: Tighten $\min\{\mathcal{D}_{r_b}\} \leq r_b \leq \max\{\mathcal{D}_{r_b}\}$.

14.22.3 Presolving

Like the domain propagation, the presolving for multiplex read constraints does not differ much from the presolving of slice constraints. Algorithm 14.32 illustrates the procedure. We do not apply Step 1c of Algorithm 14.30. Although the reasoning is valid for READ constraints as well, the condition $\beta_r \geq \beta_x$ will never be satisfied, since in every reasonable multiplexer the array x has more bits than the resultant r. Of course, the aggregations in Step 1d and the calculations in Step 1(e)i are adapted to the definition of the READ operator. Due to the blockwise selection of the input bits x_b by the operand y we cannot generate implications like the ones of Step 1(f)iii of Algorithm 14.30.

Pairs of READ constraints are only compared in Step 2 if their resultants have the same width. Otherwise, the partitioning of x into blocks would be different and the values of the block selection operands y and y' would have different meaning.

Algorithm 14.32 Multiplex Read Presolving

1. For all active multiplex read constraints $r = \text{READ}(x, y)$:
   (a) If $y^w > 0$ for any word $w \in \{0, \dots, \omega_y - 1\}$, and $x^w > 0$ or $r^w > 0$ for any $w \in \{0, \dots, \omega_x - 1\}$, deduce $r \stackrel{\star}{\neq} x$.
   (b) If $r \stackrel{\star}{=} x$ and $x^w > 0$ for any $w \in \{0, \dots, \omega_x - 1\}$, fix $y := 0$.
   (c) Apply domain propagation Algorithm 14.31 on the global bounds.
   (d) If $l_y = u_y$, aggregate $r_b \stackrel{\star}{:=} x_{b+l_y \cdot \beta_r}$ for all $b = 0, \dots, \beta_r - 1$ using $x_i = 0$ for $i \geq \beta_x$, and delete the constraint.
   (e) For all $b = 0, \dots, \beta_r - 1$:
      i. Let $p^b_{\min} = \min\{p \mid l_y \leq p \leq u_y \text{ and } u_{x_{b+p \cdot \beta_x}} = 1\}$ be the minimal and $p^b_{\max} = \max\{p \mid l_y \leq p \leq u_y \text{ and } u_{x_{b+p \cdot \beta_x}} = 1\}$ be the maximal block y that can be selected for $r_b = 1$.
      ii. For all $w = 0, \dots, \omega_y - 1$: Add implications

$$r_b = 1 \rightarrow y^w \geq \left\lceil \frac{p^b_{\min} - \sum_{i \neq w} 2^{iW} u_{y^i}}{2^{wW}} \right\rceil \text{ and}$$

$$r_b = 1 \rightarrow y^w \leq \left\lfloor \frac{p^b_{\max} - \sum_{i \neq w} 2^{iW} l_{y^i}}{2^{wW}} \right\rfloor$$

      to the implication graph of SCIP.

2. For all pairs of active multiplex read constraints $r = \text{READ}(x, y)$ and $r' = \text{READ}(x', y')$ with $\beta_r = \beta_{r'}$:
   (a) If $x \stackrel{\star}{=} x'$ and $y \stackrel{\star}{=} y'$, aggregate $r \stackrel{\star}{:=} r'$ and delete constraint $r' = \text{READ}(x', y')$.
   (b) If $x \stackrel{\star}{=} x'$ but $r \stackrel{\star}{\neq} r'$, deduce $y \stackrel{\star}{\neq} y'$.
   If $y \stackrel{\star}{=} y'$ but $r \stackrel{\star}{\neq} r'$, deduce $x \stackrel{\star}{\neq} x'$.

We can only deduce trivial aggregations for constraint pairs. If the operands are pairwise equivalent, the resultants are detected to be equivalent as well in Step 2a. Step 2b states the negated versions of this implication. The aggregation of the block selection operands y and y' as in Step 2c of Algorithm 14.30 is not possible, since again, the condition $\beta_r \geq \beta_x$ will never be satisfied.

14.23 Multiplex Write

The multiplex write operator

$$\text{WRITE} : [\beta] \times [\mu] \times [\nu] \rightarrow [\beta], \quad (x, y, z) \mapsto r = \text{WRITE}(x, y, z)$$

with

$$r = \text{WRITE}(x, y, z) \quad \Leftrightarrow \quad \forall b \in \{0, \dots, \beta - 1\} : r_b = \begin{cases} z_{b - y \cdot \nu} & \text{if } 0 \leq b - y \cdot \nu < \nu, \\ x_b & \text{otherwise,} \end{cases}$$

is the counterpart to the multiplex read operator. As shown in Figure 14.18, it stores a given value z at position y in a data array x, thus implementing the assignment

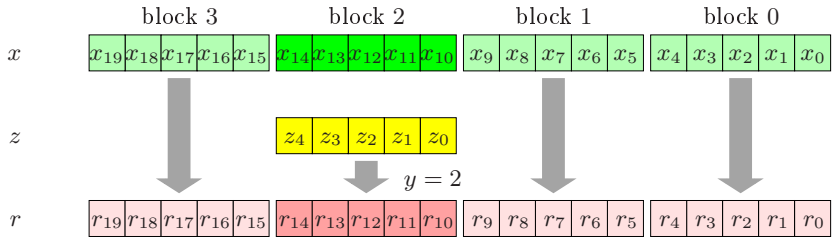

Figure 14.18. Multiplex write with array elements of $\beta = 5$ bits.

$x[y] := z$. The resultant register r denotes the state of the array after the assignment was executed.

The WRITE operator can also be used to demultiplex a serialized data stream as illustrated in Figure 14.19. At each time step t, the block selector operand y_t of the multiplexer $z_t = \text{READ}(x, y_t)$ and the demultiplexer $r_t = \text{WRITE}(r_{t-1}, y_t, z_t)$ is incremented, such that the content of x is copied into the array r within a full cycle of y through the arrays.

The multiplex write operator resembles the previous operators SHL, SLICE, and READ. Like for the READ operator, the LP relaxation is reasonably small because each resultant bit r_b can only receive the value of exactly two variables: the bit x_b of the first operand, or the bit $z_{b-y \cdot \nu}$ of the replacement operand z. As before, the domain propagation is performed with a pattern matching algorithm.

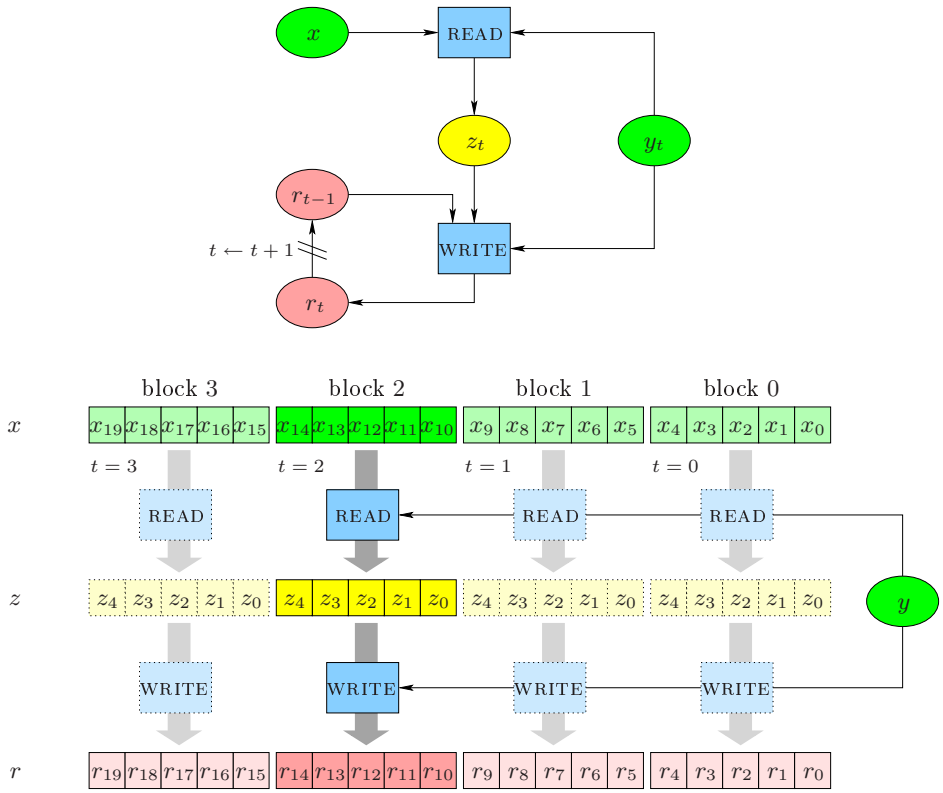

Figure 14.19. Multiplexer and demultiplexer to copy a data array x as serialized data stream into the target array r via a data bus z over multiple time steps.

14.23.1 LP RELAXATION

Analogously to the READ operator relaxation, we define the relaxation of WRITE constraints as follows:

$$\sum_{p=l_y}^{u_y} p \cdot \psi^p = y \tag{14.66}$$

$$\sum_{p=l_y}^{u_y} \psi^p = 1 \tag{14.67}$$

$$r_{b+p\cdot\beta_z} - z_b \leq 1 - \psi^p \quad \text{for all } b = 0,\dots,\beta_z - 1 \text{ and } p = l_y,\dots,u_y \tag{14.68}$$

$$-r_{b+p\cdot\beta z} + z_b \leq 1 - \psi^p \quad \text{for all } b = 0,\dots,\beta_z - 1 \text{ and } p = l_y,\dots,u_y \tag{14.69}$$

$$r_{b+p\cdot\beta_z} - x_{b+p\cdot\beta_z} \leq \psi^p \quad \text{for all } b = 0,\dots,\beta_z - 1 \text{ and } p = l_y,\dots,u_y \tag{14.70}$$

$$-r_{b+p\cdot\beta_z} + x_{b+p\cdot\beta_z} \leq \psi^p \quad \text{for all } b = 0,\dots,\beta_z - 1 \text{ and } p = l_y,\dots,u_y \tag{14.71}$$

As before, by equations (14.66) and (14.67) we split the block selection operand y into a series of binary variables $\psi^p \in \{0,1\}$, each of which represents one value in the domain $y \in \{l_y,\dots,u_y\}$. For this being valid, we assume that $y = y^0$ consists of only one word, i.e., $\omega_y = 1$. We do not linearize multiplex write constraints with $\omega_y \geq 2$.

Inequalities (14.68) to (14.71) model the constraint $r_{b+p\cdot\beta_z} = \text{ITE}(\psi^p, z_b, x_{b+p\cdot\beta_z})$, compare the LP relaxation of if-then-else constraints in Section 14.15.1. The first two inequalities ensure

$$\psi^p = 1 \rightarrow r_{b+p\cdot\beta_z} = z_b,$$

while the latter two force

$$\psi^p = 0 \rightarrow r_{b+p\cdot\beta_z} = x_{b+p\cdot\beta_z}.$$

Note that the bits in the blocks $p < l_y$ or $p > u_y$ that cannot be selected by variable y are aggregated as $r_{b+p\cdot\beta_z} :\overset{\star}{=} x_{b+p\cdot\beta_z}$ for $b = 0,\dots,\beta_z - 1$ in Step 1(b)iii of the presolving Algorithm 14.34. Thus, we do not need additional inequalities of types (14.70) and (14.71) for $p \notin \{l_y,\dots,u_y\}$.

14.23.2 DOMAIN PROPAGATION

The domain propagation Algorithm 14.33 for multiplex write constraints is somewhat different than the one for the READ operator. After the bounds for the block selection operand y are calculated as usual in Step 1, we perform a first check in Step 2 whether there is an inconsistency of bits $r_b \neq x_b$ in block $p = \left\lfloor \frac{b}{\beta_z} \right\rfloor$. In this situation r_b must have been overwritten by z, and y can therefore be fixed to p.

The pattern matching of Step 4 runs as usual with the additional update 4d: if the overlapping bits z_b and x_i, $i = b + p\cdot\beta_z$, for block p are fixed to the same value, the corresponding resultant bit r_i must also take this value. In the special case that only one possible matching $\mathcal{D}_y = \{p\}$ was found, we can propagate $r_{b+p\cdot\beta_z} = z_b$ in Step 5 for all $b = 0,\dots,\beta_z - 1$. Step 6 processes all blocks where z does not match to the corresponding part of r. For the bits b in these blocks it must be $r_b = x_b$. Finally, Steps 7 and 8 apply our knowledge about the potential values of y, y_b, and z_b to tighten the corresponding word and bit variables.

Algorithm 14.33 Multiplex Write Domain Propagation

Input: Multiplex write constraint $r = \text{WRITE}(x, y, z)$ on registers r, x, y, and z with current local bit bounds $\tilde{l}_{r_b} \le r_b \le \tilde{u}_{r_b}$, $\tilde{l}_{x_b} \le x_b \le \tilde{u}_{x_b}$, and $\tilde{l}_{y_b} \le y_b \le \tilde{u}_{y_b}$, and $\tilde{l}_{z_b} \le z_b \le \tilde{u}_{z_b}$, and current local word bounds $\tilde{l}_{y^w} \le y^w \le \tilde{u}_{y^w}$.

Output: Tightened local bounds for bits r_b, x_b, y_b, and z_b, and words y^w.

1. Calculate bounds for the register y from the words' bounds:
$$\tilde{l}_y := \sum_{w=0}^{\omega_y - 1} 2^{wW} \tilde{l}_{y^w} \quad \text{and} \quad \tilde{u}_y := \sum_{w=0}^{\omega_y - 1} 2^{wW} \tilde{u}_{y^w}.$$
2. For all $b = 0, \ldots, \beta_r - 1$:
If $\tilde{l}_{r_b} > \tilde{u}_{x_b}$, $\tilde{u}_{r_b} < \tilde{l}_{x_b}$, or $r_b \overset{\star}{\neq} x_b$, set $\tilde{l}_y := \tilde{u}_y := \lfloor \frac{b}{\beta_z} \rfloor$ and abort the loop.
3. Initialize $\mathcal{D}_{z_b} := \emptyset$ for all $b = 0, \ldots, \beta_z - 1$, $\mathcal{D}_{y_b} := \emptyset$ for all $b = 0, \ldots, \beta_y - 1$, and $\mathcal{D}_y := \emptyset$.
4. For all $p = \tilde{l}_y, \ldots, \tilde{u}_y$:
Let $p = \sum_{b=0}^{\beta_y - 1} 2^b p_b$ be the bit decomposition of p. If the following holds:
   (a) $p_b \in \{\tilde{l}_{y_b}, \tilde{u}_{yb}\}$ for all $b = 0, \ldots, \beta_y - 1$,
   (b) $\mathcal{D}_b := \{\tilde{l}_{r_{b+p\cdot\beta_z}}, \tilde{u}_{r_{b+p\cdot\beta_z}}\} \cap \{\tilde{l}_{z_b}, \tilde{u}_{z_b}\} \neq \emptyset$ for all $b = 0, \ldots, \beta_z - 1$, and
   (c) there is no bit $b \in \{0, \ldots, \beta_z - 1\}$ with $r_{b+p\cdot\beta_z} \overset{\star}{\neq} z_b$,

   the block selection value $y = p$ is valid. In this case, update
   (a) $\mathcal{D}_y := \mathcal{D}_y \cup \{p\}$,
   (b) $\mathcal{D}_{y_b} := \mathcal{D}_{y_b} \cup \{p_b\}$ for all $b = 0, \ldots, \beta_y - 1$,
   (c) $\mathcal{D}_{z_b} := \mathcal{D}_{z_b} \cup \mathcal{D}_b$ for all $b = 0, \ldots, \beta_z - 1$, and
   (d) if $\tilde{l}_{z_b} = \tilde{u}_{z_b} = \tilde{l}_{x_{b+p\cdot\beta_z}} = \tilde{u}_{x_{b+p\cdot\beta_z}}$, deduce $r_{b+p\cdot\beta_z} = \tilde{l}_{z_b}$.
5. If $\mathcal{D}_y = \{p\}$, then for all $b = 0, \ldots, \beta_z - 1$:
   (a) If $\tilde{l}_{z_b} = \tilde{u}_{z_b}$, deduce $r_{b+p\cdot\beta_z} = \tilde{l}_{z_b}$.
   (b) If $\tilde{l}_{r_{b+p\cdot\beta_z}} = \tilde{u}_{r_{b+p\cdot\beta_z}}$, deduce $z_b = \tilde{l}_{r_{b+p\cdot\beta_z}}$.
6. For all $b = 0, \ldots, \beta_r - 1$ with $\lfloor \frac{b}{\beta_z} \rfloor \notin \mathcal{D}_y$:
   (a) If $\tilde{l}_{x_b} = \tilde{u}_{x_b}$, deduce $r_b = \tilde{l}_{x_b}$.
   (b) If $\tilde{l}_{r_b} = \tilde{u}_{r_b}$, deduce $x_b = \tilde{l}_{r_b}$.
7. Tighten word bounds of y^w, $w = 0, \ldots, \omega_y - 1$, corresponding to the register bounds $\min\{\mathcal{D}_y\} \le y \le \max\{\mathcal{D}_y\}$.
8. For all $b = 0, \ldots, \beta_y - 1$: Tighten $\min\{\mathcal{D}_{y_b}\} \le y_b \le \max\{\mathcal{D}_{y_b}\}$.
For all $b = 0, \ldots, \beta_z - 1$: Tighten $\min\{\mathcal{D}_{z_b}\} \le z_b \le \max\{\mathcal{D}_{z_b}\}$.

Algorithm 14.34 Multiplex Write Presolving

1. For all active multiplex write constraints $r = \text{WRITE}(x, y)$:
   (a) If $r \overset{\star}{=} z$, $z^w > 0$ for any $w \in \{0, \dots, \omega_z - 1\}$, and $u_y < \left\lceil \frac{\beta_r}{\beta_z} \right\rceil$, fix $y := 0$.
   (b) Apply domain propagation Algorithm 14.33 on the global bounds with the following modifications:
      i. Instead of Update 4d, if $x_{b+p\cdot\beta_z} \overset{\star}{=} z_b$, aggregate $r_{b+p\cdot\beta_z} :\overset{\star}{=} z_b$.
      ii. In Step 5, aggregate $r_{b+p\cdot\beta_z} :\overset{\star}{=} z_b$ instead of only deducing bounds.
      iii. In Step 6, aggregate $r_b :\overset{\star}{=} x_b$ instead of only deducing bounds.
   (c) For all $p = l_y, \dots, u_y$ and all $b = 0, \dots, \beta_z - 1$, define $j = b + p\cdot\beta_z$ to be the corresponding bit position in x and r, and add the following implications to the implication graph of SCIP:
      i. If $x_j = 0$, add implications $r_j = 1 \rightarrow z_b = 1$ and $r_j = 1 \rightarrow y = p$.
         If $x_j = 1$, add implications $r_j = 0 \rightarrow z_b = 0$ and $r_j = 0 \rightarrow y = p$.
      ii. If $r_j = 0$, add implications $x_j = 1 \rightarrow z_b = 0$ and $x_j = 1 \rightarrow y = p$.
         If $r_j = 1$, add implications $x_j = 0 \rightarrow z_b = 1$ and $x_j = 0 \rightarrow y = p$.
      iii. If $z_b = 0$, add implication $x_j = 0 \rightarrow r_j = 0$.
         If $z_b = 1$, add implication $x_j = 1 \rightarrow r_j = 1$.
      iv. If $r_j = 0$, $p = l_y$, and $\omega_y = 1$, add implication $z_b = 1 \rightarrow y \geq l_y + 1$.
         If $r_j = 1$, $p = l_y$, and $\omega_y = 1$, add implication $z_b = 0 \rightarrow y \geq l_y + 1$.
         If $r_j = 0$, $p = u_y$, and $\omega_y = 1$, add implication $z_b = 1 \rightarrow y \leq u_y - 1$.
         If $r_j = 1$, $p = u_y$, and $\omega_y = 1$, add implication $z_b = 0 \rightarrow y \leq u_y - 1$.
      v. If $z_b = 0$, $p = l_y$, and $\omega_y = 1$, add implication $r_j = 1 \rightarrow y \geq l_y + 1$.
         If $z_b = 1$, $p = l_y$, and $\omega_y = 1$, add implication $r_j = 0 \rightarrow y \geq l_y + 1$.
         If $z_b = 0$, $p = u_y$, and $\omega_y = 1$, add implication $r_j = 1 \rightarrow y \leq u_y - 1$.
         If $z_b = 1$, $p = u_y$, and $\omega_y = 1$, add implication $r_j = 0 \rightarrow y \leq u_y - 1$.

2. For all pairs of active multiplex write constraints $r = \text{WRITE}(x, y, z)$ and $r' = \text{WRITE}(x', y', z')$ with $\beta_z = \beta_{z'}$ and $\beta_r \geq \beta_{r'}$:
   (a) If $x \overset{\star}{=} x'$, $y \overset{\star}{=} y'$, and $z \overset{\star}{=} z'$, aggregate $r[\beta_{r'} - 1, 0] :\overset{\star}{=} r'$ and delete the constraint $r' = \text{WRITE}(x', y', z')$.
   (b) If $\beta_r = \beta_{r'}$, $x \overset{\star}{=} x'$, $y \overset{\star}{=} y'$, but $r \overset{\star}{\neq} r'$, deduce $z \overset{\star}{\neq} z'$.
      If $\beta_r = \beta_{r'}$, $x \overset{\star}{=} x'$, $z \overset{\star}{=} z'$, but $r \overset{\star}{\neq} r'$, deduce $y \overset{\star}{\neq} y'$.
      If $\beta_r = \beta_{r'}$, $y \overset{\star}{=} y'$, $z \overset{\star}{=} z'$, but $r \overset{\star}{\neq} r'$, deduce $x \overset{\star}{\neq} x'$.
   (c) If $r \overset{\star}{=} r'$, $y \overset{\star}{=} y'$, and $u_y < \left\lceil \frac{\min\{\beta_r, \beta'_r\}}{\beta_z} \right\rceil$, aggregate $z :\overset{\star}{=} z'$.
   (d) If $z \overset{\star}{\neq} z'$, $y \overset{\star}{=} y'$, and $u_y < \left\lceil \frac{\min\{\beta_r, \beta'_r\}}{\beta_z} \right\rceil$, deduce $r \overset{\star}{\neq} r'$.

14.23.3 PRESOLVING

The presolving of multiplex write constraints as depicted in Algorithm 14.34 starts by checking in Step 1a whether the resultant is equivalent to the write operand z. If the latter is non-zero, and if the upper bound of the block selection operand y ensures that the complete bit string of z is written into r, z must be written into block 0 of r, and we can fix $y := 0$. The main part of the algorithm consists of calling the domain propagation Algorithm 14.33 on the global bounds in Step 1b. However, we replace the steps that propagate the equality of two registers by a corresponding aggregation. Step 1c considers all potential values of the block selection operand y and identifies implications that can be added to the implication graph of SCIP. If

x_j, z_b, and r_j are the bits that belong together in the current block p, Steps 1(c)i to 1(c)iii apply the implication

$$x_j \neq r_j \rightarrow r_j = z_b \wedge y = p \tag{14.72}$$

for all fixings of a single variable. In particular, the implication $x_j = z_b \rightarrow r_j = x_j$ used in Step 1(c)iii follows from (14.72), since

$$(x_j = z_b) \wedge (x_j \neq r_j \rightarrow r_j = z_b) \;\Rightarrow\; (x_j \neq r_j \rightarrow r_j = x_j) \;\Rightarrow\; r_j = x_j.$$

Steps 1(c)iv and 1(c)v apply the same implication reorganized to the form

$$r_j \neq z_b \rightarrow y \neq p.$$

Such an inequality $y \neq p$ (which is a disjunction $y < p \vee y > p$) in the conclusion of an implication cannot be stored in the implication graph of SCIP. Therefore, we can only exploit this situation in the blocks p corresponding to the lower and upper bounds of the block selection variable y, where the inequality becomes $y \leq p-1$ or $y \geq p+1$, respectively. Additionally, we can only use these implications if the block selection operand consists of only a single word. Otherwise, the conclusion would be a disjunction on the different word variables y^w.

The pairwise presolving of Step 2 can only be applied for WRITE constraints of equal β_z. Otherwise, the partitioning of x and r into array elements would be different. Step 2a applies the trivial aggregation which states that two constraints with equivalent operands must have equivalent resultant bits. Step 2b executes this rule in the inverse direction. Step 2c is a bit more interesting: if the resultants and the block selection operands are pairwise equivalent, we can deduce that the written value z must also be equivalent. This is, however, only true if z would be absorbed completely by r, i.e., if the upper bound of the block selection operand y is small enough. The inverse implication of this rule is stated in Step 2d: if unequal values are written to the same position, and if all of these bits will affect the resultants, the resultants must be unequal.

Chapter 15

Presolving

In the previous chapter, we presented the algorithms used in the constraint handlers of our property checking tool to process the different circuit operators. One important ingredient of a constraint handler is the presolving component. These constraint based presolving algorithms try to simplify the problem instance by considering the constraints of a single constraint class independently from all other constraints. Even more, most of the reductions are deduced from a single constraint and the bounds of the involved variables. The only "global" information exploited is our knowledge about the equality or inequality of variables and—to a smaller extent—the discovered implications between the variables which are stored in the implication graph, see Section 3.3.5.

In addition to the presolving of linear constraints and its specializations, Chapter 10 introduced a few general purpose presolving techniques that can be applied independent of an individual constraint class. They can be used not only for mixed integer programs but for any constraint integer program. These concepts, in particular *probing* and *implication graph analysis*, are also employed here. In the following, we present two additional presolving techniques that are specifically tailored to the property checking problem. The *term algebra preprocessing* features a term replacement system to exploit the commutativity, associativity, and distributivity of the arithmetic constraints ADD and MULT. The *irrelevance detection* discards parts of the circuit that are not relevant for the property at hand. This does not help to tighten the domains of the variables but reduces the effort spent on the individual subproblems during the solving process. Furthermore, removing irrelevant constraints and variables can help the branching strategy to concentrate on the relevant part on the circuit where the crucial decisions have to be taken, see Section 16.1.

As described in Section 3.2.5, the preprocessing algorithms of the constraints and the global presolving algorithms are applied periodically until no more problem reductions can be found. Whenever a new fixing, aggregation, domain reduction, or implication was found by one of the preprocessing algorithms, presolving is applied again to the affected constraints. More expensive presolving operations like probing or term algebra preprocessing are delayed until the other algorithms failed to produce further reductions.

15.1 Term Algebra Preprocessing

Analog to the symbolic propagation with term rewriting applied in the domain propagation of multiplication constraints, see Section 14.5.2, we employ a term rewriting system as a global presolving engine. Recall that the symbolic propagation for multiplication constraints was defined on a simple signature consisting of binary variable symbols B and the operations $\wedge : B \times B \to B$ and $\oplus : B \times B \to B$. In contrast, the term algebra preprocessing operates on a more complex signature of low significant register subwords, truncated addition, and truncated multiplication operators. The

goal is to exploit the commutativity, associativity, and distributivity of the two operators in order to identify equivalent subword-defining terms, thereby concluding that the subwords themselves must be equivalent.

We need the following preliminaries to justify the term rewriting rules that we want to employ. A comprehensive introduction on group and ring theory can be found in Allenby [9].

Definition 15.1 (ring homomorphism). If $(R, +_R, \cdot_R)$ and $(S, +_S, \cdot_S)$ are rings, a *ring homomorphism* is a mapping $f : R \to S$ such that

1. $f(a +_R b) = f(a) +_S f(b)$ for all $a, b \in R$, and
2. $f(a \cdot_R b) = f(a) \cdot_S f(b)$ for all $a, b \in R$.

A ring homomorphism $f : R \to S$ on unitary rings $(R, +_R, \cdot_R, 1_R)$ and $(S, +_S, \cdot_S, 1_S)$ which satisfies

3. $f(1_R) = 1_S$

is called *unitary ring homomorphism*.

Proposition 15.2. For all $n, m \in \mathbb{Z}_{>0}$ with $\frac{n}{m} \in \mathbb{Z}$, the modulus operation

$$\bmod m : \mathbb{Z}_n \to \mathbb{Z}_m, \quad a \mapsto a \bmod m$$

is a unitary ring homomorphism from the unitary ring $(\mathbb{Z}_n, +_n, \cdot_n, 1)$ to the unitary ring $(\mathbb{Z}_m, +_m, \cdot_m, 1)$ with $\mathbb{Z}_k = \{0, \ldots, k-1\}$, $a +_k b = (a + b) \bmod k$, and $a \cdot_k b = (a \cdot b) \bmod k$.

Proof. To show Conditions 1 and 2 of Definition 15.1, let $\circ \in \{+, \cdot\}$ be either multiplication or addition. Let $a, b \in \mathbb{Z}_n$ and $c = (a \circ_n b) = (a \circ b) \bmod n$. Define $k \in \mathbb{Z}$ to be the unique value with $a \circ b = c + kn$. Let $a' = a \bmod m$, $b' = b \bmod m$, and $c' = c \bmod m$. We have to show $c' = a' \circ_m b'$. Let $p, q, r \in \mathbb{Z}$ be the unique values such that $a = a' + pm$, $b = b' + qm$, and $c = c' + rm$. Then, for $\circ = +$ we have

$$c' = c - rm = a + b - kn - rm = a' + b' + \big(p + q - k\frac{n}{m} - r\big)m = a' +_m b',$$

and for $\circ = \cdot$ it follows

$$c' = c - rm = a \cdot b - kn - rm = a' \cdot b' + \big(pq + aq + bp - k\frac{n}{m} - r\big)m = a' \cdot_m b',$$

since $\frac{n}{m} \in \mathbb{Z}$ by assumption and $c' \in \mathbb{Z}_m$. The validity of Condition 3 is obvious. □

Corollary 15.3. Truncated addition and truncated multiplication respect the subword property, i.e.,

$$\begin{aligned} r = \textsc{add}(x, y) &\;\Rightarrow\; r[b, 0] = \textsc{add}(x[b, 0], y[b, 0]) \\ r = \textsc{mult}(x, y) &\;\Rightarrow\; r[b, 0] = \textsc{mult}(x[b, 0], y[b, 0]) \end{aligned}$$

for all $b = 0, \ldots, \beta_r - 1$.

Our term rewriting system operates on terms composed of the constants 0 and 1, the register bits ϱ_{jb} and their negations $\bar{\varrho}_{jb}$, the concatenations of these constant

symbols into bit strings, the truncation of bit strings to shorter strings, and the ADD and MULT operators. In order to obtain a finite signature, we impose a maximum $\beta_{\max} \in \mathbb{Z}_{>0}$ on the width of the bit strings on which the operators are defined. One can think of $\beta_{\max}$ as the maximal width of the registers contained in the property checking instance at hand.

Definition 15.4 (bit string arithmetics signature). Let $\beta_{\max} \in \mathbb{Z}_{>0}$ and $\mathcal{B} = \{1, \ldots, \beta_{\max}\}$. The algebraic signature $\Sigma = (S, O)$ with the sorts $S = \{[\beta] \mid \beta \in \mathcal{B}\}$ and the following operators $O = O_0 \cup O_1 \cup O_2$:

$$
\begin{array}{lll}
O_0\colon & 0 : \rightarrow [1] & \\
& 1 : \rightarrow [1] & \\
& \varrho_{jb} : \rightarrow [1] & \text{with } j = 1, \ldots, n \text{ and } b = 0, \ldots, \beta_{\varrho_j} - 1 \\
& \bar{\varrho}_{jb} : \rightarrow [1] & \text{with } j = 1, \ldots, n \text{ and } b = 0, \ldots, \beta_{\varrho_j} - 1 \\
O_1\colon & -_\beta : [\beta] \rightarrow [\beta] & \\
& |_{\beta\,\mu} : [\mu] \rightarrow [\beta] & \text{with } \beta, \mu \in \mathcal{B} \text{ and } \mu \geq \beta \\
O_2\colon & \otimes_{\beta\mu} : [\beta - \mu] \times [\mu] \rightarrow [\beta] & \text{with } \beta, \mu \in \mathcal{B} \text{ and } \beta > \mu \\
& \cdot_\beta : [\beta] \times [\beta] \rightarrow [\beta] & \text{with } \beta \in \mathcal{B} \\
& +_\beta : [\beta] \times [\beta] \rightarrow [\beta] & \text{with } \beta \in \mathcal{B}
\end{array}
$$

is called *bit string arithmetics signature*. We call $\mathcal{T}_\Sigma$ the *term algebra* of Σ, which consists of all terms that can be generated from the symbols in Σ and which fit to the arity of the operators. The terms of sort $[\beta]$ are denoted by $\mathcal{T}_{[\beta]}$. Thus, $\mathcal{T}_{[\beta]}$ contains all terms $t \in \mathcal{T}_\Sigma$ whose outermost operator has codomain $[\beta]$.

Whenever it is non-ambiguous, we will omit the domain subscripts of the operators. We want to interpret the bit string arithmetics signature to be in line with the definition of the circuit operators. In particular, the signature operators $-$, $|_\beta$, $\otimes$, $\cdot$, and $+$ should reflect the properties of the MINUS, SLICE$(\cdot, 0)$, CONCAT, MULT, and ADD operators, respectively. Formally, we have to introduce equations E to the signature Σ in order to describe the relevant properties of the circuit operators. We start with the associative law, which is valid for the three binary operators:

$$
\begin{aligned}
t^\nu \otimes (t^\mu \otimes t^\beta) &= (t^\nu \otimes t^\mu) \otimes t^\beta \\
t_3^\beta \cdot (t_2^\beta \cdot t_1^\beta) &= (t_3^\beta \cdot t_2^\beta) \cdot t_1^\beta \\
t_3^\beta + (t_2^\beta + t_1^\beta) &= (t_3^\beta + t_2^\beta) + t_1^\beta
\end{aligned}
\tag{15.1}
$$

Here we have $t^\beta, t_i^\beta \in \mathcal{T}_{[\beta]}$, $t^\mu \in \mathcal{T}_{[\mu]}$, and $t^\nu \in \mathcal{T}_{[\nu]}$. These equations allow to simplify the notation by removing the brackets with the implicit meaning that the operators from left to right represent innermost to outermost operations. Now we extend

system (15.1) by the following equations to yield the final set of equations E:

$$
\begin{aligned}
-(-t^\beta) &= t^\beta & -(0\otimes\cdots\otimes 0) &= 0\otimes\cdots\otimes 0\\
(c_{\mu-1}\otimes\cdots\otimes c_0)|_\beta &= c_{\beta-1}\otimes\cdots\otimes c_0 & (-t^\beta)\otimes 0 &= -(t^\beta\otimes 0)\\
t^\beta\cdot(0\otimes\cdots\otimes 0) &= 0\otimes\cdots\otimes 0 & t^\beta+(0\otimes\cdots\otimes 0) &= t^\beta\\
t^\beta\cdot(0\otimes\cdots\otimes 0\otimes 1) &= t^\beta & (t_1^\beta+t_2^\beta)\otimes 0 &= (t_1^\beta\otimes 0)+(t_2^\beta\otimes 0)\\
t_1^\beta\cdot(t_2^{\beta-1}\otimes 0) &= (t_1^\beta|_{\beta-1}\cdot t_2^{\beta-1})\otimes 0 & t_1^\beta+t_2^\beta &= t_2^\beta+t_1^\beta\\
t_1^\beta\cdot t_2^\beta &= t_2^\beta\cdot t_1^\beta & (t_3^\beta+t_1^\beta)+t_2^\beta &= (t_3^\beta+t_2^\beta)+t_1^\beta\\
(t_3^\beta\cdot t_1^\beta)\cdot t_2^\beta &= (t_3^\beta\cdot t_2^\beta)\cdot t_1^\beta & (t_2^\mu+t_1^\mu)|_\beta &= (t_2^\mu|_\beta)+(t_1^\mu|_\beta)\\
(t_2^\mu\cdot t_1^\mu)|_\beta &= (t_2^\mu|_\beta)\cdot(t_1^\mu|_\beta) & t^\beta+(-t^\beta) &= 0\otimes\cdots\otimes 0\\
t_2^\beta\cdot(-t_1^\beta) &= -(t_2^\beta\cdot t_1^\beta) & t_3^\beta\cdot(t_2^\beta+t_1^\beta) &= (t_3^\beta\cdot t_2^\beta)+(t_3^\beta\cdot t_1^\beta)\\
(-t_2^\beta)\cdot t_1^\beta &= -(t_2^\beta\cdot t_1^\beta) & (t_3^\beta+t_2^\beta)\cdot t_1^\beta &= (t_3^\beta\cdot t_1^\beta)+(t_2^\beta\cdot t_1^\beta)
\end{aligned}
$$

with $c_i\in O_0$ being constant symbols. These equations define an equivalence relation $\equiv_E\subseteq\mathcal{T}_\Sigma\times\mathcal{T}_\Sigma$ which identifies terms $s\equiv_E t$, $s,t\in\mathcal{T}_\Sigma$, that would always yield the same results when evaluated with circuit operators. We can apply these equations to terms $t\in\mathcal{T}_\Sigma$ in order to transform them into *normal form*.

Definition 15.5 (normal form). We call a term $t\in\mathcal{T}_{[\beta]}$ to be in *normal form* if it has the form

$$
\begin{aligned}
& t = (a_m+\ldots+a_1)\\
\text{with}\quad & a_i = [-]\big((f_{im_i}\cdot\ldots\cdot f_{i1})\underbrace{\otimes 0\otimes\cdots\otimes 0}_{s_i\text{ times}}\big) && \text{for } i=1,\ldots,m,\\
\text{and}\quad & f_{ij} = c_{ij,\beta-s_i-1}\otimes\cdots\otimes c_{ij,0} && \text{for } i=1,\ldots,m,\ \text{and } j=1,\ldots,m_i
\end{aligned}
$$

with $s_i\in\{0,\ldots,\beta-1\}$, $c_{ij,k}\in O_0$, $c_{ij,0}\neq 0$, $i=1,\ldots,m$, $j=1,\ldots,m_i$, $k=0,\ldots,\beta-s_i-1$. The minus symbol enclosed in brackets $[-]$ is optional.

Note. Observe that the number s_i of appended "$\otimes 0$"s may be zero, which means that no "$\otimes$" operator is contained in a_i.

Note. We cannot transform *all* terms $t\in\mathcal{T}_\Sigma$ into normal form using equations E. For example, no equation can be applied on the term $t=(1+1)\otimes 1$, although it is not in normal form. In our presolving algorithm, however, we will only produce terms that can be transformed into normal form.

The normalization of a term $t\in\mathcal{T}_\Sigma$ is accomplished by executing term rewriting rules $R=\{l_e\to r_e\mid e\in E\}$ in an arbitrary order until no more rules are applicable, see Algorithm 15.1. In this definition, l_e and r_e are the left and right hand sides of the equations $e\in E$. In order to ensure the termination of this rewriting process, we have to restrict the use of the commutativity Rules 2e, 2f, 3d, and 3e. Let $\succ$ be

Algorithm 15.1 Term Algebra Presolving – Term Normalization

Input: Term $t \in \mathcal{T}_\Sigma$

Output: Term $t' \equiv_E t$ in normal form

Apply the following subterm rewriting rules in any order until no more rules are applicable, with $t^k, t_i^k \in \mathcal{T}_{[\beta]}$, and $c_i \in O_0$ being subterms of t:

1. Minus, Truncation, and Concatenation:
   (a) $-\big(-\,t^\beta\big) \;\to\; t^\beta$
   (b) $-\,(0 \otimes \cdots \otimes 0) \;\to\; 0 \otimes \cdots \otimes 0$
   (c) $\big(-\,t^\beta\big) \otimes 0 \;\to\; -\,\big(t^\beta \otimes 0\big)$
   (d) $\big(c_{\mu-1} \otimes \cdots \otimes c_0\big)|_\beta \;\to\; c_{\beta-1} \otimes \cdots \otimes c_0$
   (e) $t^\nu \otimes \big(t^\mu \otimes t^\beta\big) \;\to\; \big(t^\nu \otimes t^\mu\big) \otimes t^\beta$
2. Multiplication:
   (a) $t_3^\beta \cdot \big(t_2^\beta \cdot t_1^\beta\big) \;\to\; \big(t_3^\beta \cdot t_2^\beta\big) \cdot t_1^\beta$
   (b) $t^\beta \cdot \big(0 \otimes \cdots \otimes 0\big) \;\to\; 0 \otimes \cdots \otimes 0$
   (c) $t^\beta \cdot \big(0 \otimes \cdots \otimes 0 \otimes 1\big) \;\to\; t^\beta$
   (d) $t_1^\beta \cdot \big(t_2^{\beta-1} \otimes 0\big) \;\to\; \big(t_1^\beta|_{\beta-1} \cdot t_2^{\beta-1}\big) \otimes 0$
   (e) $t_1^\beta \cdot t_2^\beta \;\to\; t_2^\beta \cdot t_1^\beta$ $\quad\big(\text{if } t_2^\beta \succ_{\text{lrpo}} t_1^\beta\big)$
   (f) $\big(t_3^\beta \cdot t_1^\beta\big) \cdot t_2^\beta \;\to\; \big(t_3^\beta \cdot t_2^\beta\big) \cdot t_1^\beta$ $\quad\big(\text{if } t_2^\beta \succ_{\text{lrpo}} t_1^\beta\big)$
   (g) $\big(t_2^\mu \cdot t_1^\mu\big)|_\beta \;\to\; t_2^\mu|_\beta \cdot t_1^\mu|_\beta$
   (h) $t_2^\beta \cdot \big(-\,t_1^\beta\big) \;\to\; -\big(t_2^\beta \cdot t_1^\beta\big)$
   (i) $\big(-\,t_2^\beta\big) \cdot t_1^\beta \;\to\; -\big(t_2^\beta \cdot t_1^\beta\big)$
3. Addition:
   (a) $t_3^\beta + \big(t_2^\beta + t_1^\beta\big) \;\to\; \big(t_3^\beta + t_2^\beta\big) + t_1^\beta$
   (b) $t^\beta + \big(0 \otimes \cdots \otimes 0\big) \;\to\; t^\beta$
   (c) $\big(t_1^\beta + t_2^\beta\big) \otimes 0 \;\to\; \big(t_1^\beta \otimes 0\big) + \big(t_2^\beta \otimes 0\big)$
   (d) $t_1^\beta + t_2^\beta \;\to\; t_2^\beta + t_1^\beta$ $\quad\big(\text{if } t_2^\beta \succ_{\text{lrpo}} t_1^\beta\big)$
   (e) $\big(t_3^\beta + t_1^\beta\big) + t_2^\beta \;\to\; \big(t_3^\beta + t_2^\beta\big) + t_1^\beta$ $\quad\big(\text{if } t_2^\beta \succ_{\text{lrpo}} t_1^\beta\big)$
   (f) $\big(t_2^\mu + t_1^\mu\big)|_\beta \;\to\; t_2^\mu|_\beta + t_1^\mu|_\beta$
   (g) $t^\beta + \big(-\,t^\beta\big) \;\to\; 0 \otimes \cdots \otimes 0$
4. Distributivity:
   (a) $t_3^\beta \cdot \big(t_2^\beta + t_1^\beta\big) \;\to\; t_3^\beta \cdot t_2^\beta + t_3^\beta \cdot t_1^\beta$
   (b) $\big(t_3^\beta + t_2^\beta\big) \cdot t_1^\beta \;\to\; t_3^\beta \cdot t_1^\beta + t_2^\beta \cdot t_1^\beta$

the precedence relation defined as the transitive closure of

$$\begin{aligned}
&\mu > \beta \;\Rightarrow\; \circ_\mu \succ \bullet_\beta \quad \text{for all } \circ, \bullet \in \{|., \cdot, \otimes, -, +\},\\
&|_\beta \succ \cdot_\beta \succ \otimes_\beta \succ -_\beta \succ +_\beta \quad \text{for all } \beta \in \mathcal{B} \setminus \{1\},\\
&|_1 \succ \cdot_1 \succ -_1 \succ +_1 \succ \bar{\varrho}_{jb} \succ \varrho_{j'b'} \succ 1 \succ 0,\\
&|_{\beta_\mu} \succ |_{\beta_\nu} \;\Leftrightarrow\; \mu > \nu,\\
&\otimes_{\beta\mu} \succ \otimes_{\beta\nu} \;\Leftrightarrow\; \mu > \nu,\\
&\left.\begin{matrix}\bar{\varrho}_{jb} \succ \bar{\varrho}_{j'b'}\\ \varrho_{jb} \succ \varrho_{j'b'}\end{matrix}\right\} \;\Leftrightarrow\; j > j' \vee (j = j' \wedge b > b')
\end{aligned} \tag{15.2}$$

As in the termination proof of the binary multiplication term normalization Algorithm 14.9, see Section 14.5.2, we define $\succ_{\text{lrpo}}$ to be the *lexicographic recursive path ordering* of Σ with respect to $\succ$. Recall that $\succ_{\text{lrpo}}$ is defined in Definition 14.17 as

$$\begin{aligned}
&g(t_n, \ldots, t_1) \succ_{\text{lrpo}} f(s_m, \ldots, s_1)\\
&\quad\Leftrightarrow\quad \text{(i)}\;\; t_j \succeq_{\text{lrpo}} f(s_m, \ldots, s_1) \text{ for some } j \in \{1, \ldots, n\}, \text{ or}\\
&\qquad\quad \text{(ii)}\;\; g \succ f \text{ and } g(t_n, \ldots, t_1) \succ_{\text{lrpo}} s_i \text{ for all } i \in \{1, \ldots, m\}, \text{ or}\\
&\qquad\quad \text{(iii)}\;\; g = f,\; g(t_n, \ldots, t_1) \succ_{\text{lrpo}} s_i \text{ for all } i \in \{1, \ldots, m\},\\
&\qquad\qquad\quad \text{and } (t_n, \ldots, t_1)\; (\succ_{\text{lrpo}})_{\text{lex}}\; (s_m, \ldots, s_1)
\end{aligned}$$

Now, the commutativity rules are only applicable if $t_2^\beta \succ_{\text{lrpo}} t_1^\beta$. In order to treat Rule 3g in the termination proof below, we need the following lemma:

Lemma 15.6. For all $\beta, \mu \in \mathcal{B}$, $\mu \geq \beta$, and all $t \in \mathcal{T}_{[\mu]}$ the relation $t \succeq_{\text{lrpo}} 0 \otimes \cdots \otimes 0 \in \mathcal{T}_{[\beta]}$ is valid.

Proof. Let $d(t) \in \mathbb{Z}_{\geq 0}$ be the depth of the tree representation of term t. We prove the claim by induction on β and on $d(t)$. For $t = 0 \otimes \cdots \otimes 0 \in \mathcal{T}_{[\beta]}$ nothing has to be shown. Thus, let $\mathcal{T}'_{[\mu]} = \mathcal{T}_{[\mu]} \setminus \{0 \otimes \cdots \otimes 0\}$ and consider $t \in \mathcal{T}'_{[\mu]}$. We have to show that $t \succ_{\text{lrpo}} 0 \otimes \cdots \otimes 0 \in \mathcal{T}_{[\beta]}$.

Assume $\beta = 1$ and let $\mu \geq \beta$ be arbitrary. Because $t \neq 0$ it follows $t \succ_{\text{lrpo}} 0$ due to Condition (ii) of Definition 14.17 since 0 is the smallest symbol with respect to $\succ$.

Now let $\mu \geq \beta \geq 2$. Assume that we have already shown the claim for all $\beta' < \beta$ on arbitrary terms t and for all $\beta' = \beta$ on all terms t' with $d(t') < d(t)$.

Suppose the outermost operation of t is "$\otimes$", i.e., $t = t_2 \otimes t_1$ with $t_1 \in \mathcal{T}_{[\mu_1]}$, $t_2 \in \mathcal{T}_{[\mu_2]}$, and $\mu_1 + \mu_2 = \mu$. Then we have $t_2 \otimes t_1 \succ_{\text{lrpo}} (0 \otimes \cdots \otimes 0) \otimes 0 \in \mathcal{T}_{[\beta]}$ by Condition (iii) of Definition 14.17 since $t_2 \otimes t_1 \succ_{\text{lrpo}} 0 \in \mathcal{T}_{[1]}$ and $t_2 \otimes t_1 \succ_{\text{lrpo}} 0 \otimes \cdots \otimes 0 \in \mathcal{T}_{[\beta-1]}$ by induction and $(t_2, t_1)\; (\succ_{\text{lrpo}})_{\text{lex}}\; (0 \otimes \cdots \otimes 0, 0)$ because either $t_1 \neq 0$ and therefore $t_1 \succ_{\text{lrpo}} 0$ or $t_2 \in \mathcal{T}'_{[\mu-1]}$ and therefore $t_2 \succ_{\text{lrpo}} 0 \otimes \cdots \otimes 0 \in \mathcal{T}_{[\beta-1]}$ by induction.

If "$\otimes$" is not the outermost operation of t, we either have $t = -t_1$, $t = t_1|_\mu$, $t = t_1 \cdot t_2$, or $t = t_1 + t_2$ with $t_1, t_2 \in \mathcal{T}_{[\nu]}$ and $\nu \geq \mu$. In all cases, $t \succ_{\text{lrpo}} 0 \otimes \cdots \otimes 0 \in \mathcal{T}_{[\beta]}$ follows from Condition (i) of Definition 14.17 since $t_1 \succeq_{\text{lrpo}} t$ by induction because $d(t_1) = d(t) - 1 < d(t)$. □

Proposition 15.7. Algorithm 15.1 terminates.

Proof. Obviously, the relation $\succ$ is a well-founded partial ordering (in fact, even a total ordering) on O, compare Definition 14.19. Thus, by Theorem 14.21, the

lexicographic recursive path ordering $\succ_{\text{lrpo}}$ is a well-founded monotonic ordering on $\mathcal{T}_\Sigma$, and we only have to show that for each rewriting rule $t \to t'$ the relation $t \succ_{\text{lrpo}} t'$ holds.

Rule 1a reduces the term order by Lemma 14.18 which states that for subterms t of s we always have $s \succ_{\text{lrpo}} t$. The same holds for Rules 1b, 2b, 2c, and 3b. For Rule 1c we can apply Conditions (ii) and (iii) of Definition 14.17 and Lemma 14.18. The truncation Rule 1d reduces the term order due to iterated application of Condition (ii) and Lemma 14.18. The reduction properties of the associativity Rules 1e, 2a, and 3a follow from a twofold application of Condition (iii) and Lemma 14.18. We have $t_1^\beta \cdot (t_2^{\beta-1} \otimes 0) \succ_{\text{lrpo}} (t_1^\beta|_{\beta-1} \cdot t_2^{\beta-1}) \otimes 0$ in Rule 2d due to Condition (ii), Lemma 14.18, and because $t_1^\beta \cdot (t_2^{\beta-1} \otimes 0) \succ_{\text{lrpo}} t_1^\beta|_{\beta-1} \cdot t_2^{\beta-1}$, which is due to Condition (iii), Condition (ii), and Lemma 14.18 since $\cdot_\beta \succ |_{\beta-1}$. The commutativity Rules 2e, 2f, 3d, and 3e reduce the term order due to Condition (iii) and Lemma 14.18. For the truncation distributivity Rules 2g and 3f we have to apply Condition (ii), Condition (iii), and Lemma 14.18. This is also the case for Rules 2h, 2i, and 3c, and for the distributivity Rules 4a and 4b. Finally, Rule 3g reduces the term order due to Lemma 15.6. □

After having shown how one can normalize bit string arithmetic terms, we are ready to present the term algebra presolving procedure. Each constraint of the property checking problem instance defines an equation between the resultant register and a term which includes the operand registers. The registers are now treated as variables. By substituting the operand registers with their defining terms and by applying the term replacement rules of Algorithm 15.1, we can extract equalities of terms or subterms and thereby equalities of the corresponding resultant register strings or substrings.

The details of this procedure are depicted in Algorithm 15.2. At first in Step 1, we collect all ADD and MULT constraints from the problem instance. We can also use UXOR and UAND constraints since they are equivalent to β_x-ary ADD and MULT constraints on single bit variables, respectively. Note that this includes bitwise XOR and AND constraints since each of these constraints is converted into β_r constraints using the unary operators UXOR and UAND, respectively, see Sections 14.7 and 14.9. The bitwise OR and unary UOR operators are also considered since they are automatically converted to corresponding UAND constraints on negated bit variables, see Sections 14.8 and 14.11. The assembled constraints are stored as term equations $\{r = t\}$ in a set T.

Note. Observe that the term equations are collected only in the first call of the algorithm during the global presolving loop of SCIP, see Section 3.2.5. In all subsequent presolving rounds, we keep the term equation database T as it was at the end of the previous call of the algorithm. Since the term algebra presolving is delayed as long as other presolving methods find problem reductions, the term data base represents an already thoroughly preprocessed constraint set. Therefore, the disregarding of constraints generated after the first call to the term algebra presolving should be of marginal influence. Furthermore, variable fixings and aggregations found in future presolving rounds are exploited in term algebra presolving. We implicitly identify constant symbols $c_1, c_2 \in O_0$ of the bit string arithmetics signature Σ if $c_1 \equiv c_2$, i.e., if the corresponding variables or constants are equivalent in the variable aggregation graph of SCIP (see Section 3.3.4). We replace each constant symbol $c \in O_0$ by a representative of its equivalence class during term normalization.

Algorithm 15.2 Term Algebra Presolving

1. If this is the first call to the algorithm, set $T := \emptyset$ and extend T for all constraints $\mathcal{C} \in \mathfrak{C}$ of the problem instance:
   (a) If $C = \{r = \text{ADD}(x, y)\}$, update $T := T \cup \{r_{\beta_r-1} \otimes \cdots \otimes r_0 = (x_{\beta_r-1} \otimes \cdots \otimes x_0) + (y_{\beta_r-1} \otimes \cdots \otimes y_0)\}$.
   (b) If $C = \{r = \text{MULT}(x, y)\}$, update $T := T \cup \{r_{\beta_r-1} \otimes \cdots \otimes r_0 = (x_{\beta_r-1} \otimes \cdots \otimes x_0) \cdot (y_{\beta_r-1} \otimes \cdots \otimes y_0)\}$.
   (c) If $C = \{r = \text{UXOR}(x)\}$, update $T := T \cup \{r_0 = x_{\beta_x-1} + \ldots + x_0\}$.
   (d) If $C = \{r = \text{UAND}(x)\}$, update $T := T \cup \{r_0 = x_{\beta_x-1} \cdot \ldots \cdot x_0\}$.
2. For all $\{r = t\} \in T$ call Algorithm 15.1 to normalize t. Update T accordingly.
3. For all $\{r = t\} \in T$ call Algorithm 15.3 to process self-references in the term.
4. For all $\{r = t\} \in T$ call Algorithm 15.4 to process term equations with $r|_\mu = 0$.
5. For all $\{r = t\} \in T$ call Algorithm 15.5 to deduce fixings and aggregations for the resultant bits r_b.
6. For all $\{r = t\} \in T$ call Algorithm 15.6 to substitute an operand for a term.
7. If fixings or aggregations were produced, if a term equation was modified in T, or if a new term equation was added to T, goto Step 3.
8. For all pairs $\{r = t\}, \{r' = t'\} \in T$ with $\beta_r \geq \beta_{r'}$:
   (a) Let $\mu \in \{1, \ldots, \beta_{r'}\}$ be the maximal width for which the normalization of $s := t|_\mu$ and $s' := t'|_\mu$ yields equal terms. Set $\mu := 0$ if no such width exists.
   (b) Aggregate $r_b :\stackrel{*}{=} r'_b$ for all $b = 0, \ldots, \mu - 1$.
   (c) If $\mu = \beta_{r'}$, delete $\{r' = t'\}$ from T.

Step 2 normalizes the terms by calling the term normalization Algorithm 15.1. In Step 3 the term equations are inspected for self-references of the resultant, i.e., whether on the least significant bits up to bit $\mu \leq \beta_r$ the term equation has the form $r|_\mu = t|_\mu(r|_\mu)$. However, we can only exploit situations where the resultant's substring $r|_\mu$ appears as addend $r|_\mu \equiv_E a_i|_\mu$ of the summation and not within a multiplication. The procedure is illustrated in Algorithm 15.3. Steps 1 and 2 check

Algorithm 15.3 Term Algebra Presolving – Self-Reference Simplification

Input: Term equation set T and term equation $\{r = t\} \in T$ with t in normal form as in Definition 15.5.

Output: Modified term equation set T.

1. Set $i^\star := 0$ and $q^\star := 0$.
2. For $i = 1, \ldots, m$ with $m_i = 1$ and $a_i = f_{i1} \otimes 0 \otimes \cdots \otimes 0$:
   If $q = \max\{\mu \leq \beta_r \mid r|_\mu \equiv_E a_i|_\mu\} > q^\star$, update $i^\star := i$ and $q^\star := q$.
3. If $q^\star > 0$ and $m \geq 2$, call Algorithm 15.1 to normalize the term equation $\{0 \otimes \cdots \otimes 0 = (a_m + \ldots + a_{i^\star+1} + a_{i^\star-1} + \ldots + a_1)|_{q^\star}\}$ and add it to T.
4. If $q^\star = \beta_r$ and $m = 1$, remove term equation $\{r = t\}$ from T.

Algorithm 15.4 Term Algebra Presolving – Zero Resultant Simplification

Input: Term equation set T and term equation $\{r = t\} \in T$ with t in normal form as in Definition 15.5.

Output: Modified term equation set T.

1. Set $q := \max\{\mu \leq \beta_r \mid r|_\mu \equiv_E 0 \otimes \cdots \otimes 0\}$. If $q = 0$, stop.
2. Select $i \in \{1, \ldots, m\}$ with $m_i = 1$. If no addend a_i with $m_i = 1$ exists, stop.
3. If $m = 1$, set $r^\star := (f_{i1} \otimes 0 \otimes \cdots \otimes 0)|_q$ and $t^\star := 0 \otimes \cdots \otimes 0 \in \mathcal{T}_{[q]}$.
   If $m \geq 2$ and $a_i = f_{i1} \otimes 0 \otimes \cdots \otimes 0$, set
   $$r^\star := a_i|_q \quad \text{and} \quad t^\star := ((-a_m) + \ldots + (-a_{i+1}) + (-a_{i-1}) + \ldots + (-a_1))|_q.$$
   If $m \geq 2$ and $a_i = -f_{i1} \otimes 0 \otimes \cdots \otimes 0$, set
   $$r^\star := -a_i|_q \quad \text{and} \quad t^\star := (a_m + \ldots + a_{i+1} + a_{i-1} + \ldots + a_1)|_q.$$
4. Normalize $r^\star$ and $t^\star$ by calling Algorithm 15.1.
5. If $q = \beta_r$, replace $\{r = t\}$ with $\{r^\star = t^\star\}$ in T.
   If $q < \beta_r$, add term equation $\{r^\star = t^\star\}$ to T.

whether there is an addend a_i which is, if truncated to μ bits, equivalent to the truncated resultant. From all of those candidates, we select the addend $a_{i^\star}$ that is equivalent to the resultant on the largest number of bits. We subtract the addend $a_{i^\star}$ from the truncated resultant and the truncated term, such that the left hand side of the truncated term equation is reduced to zero. If there is at least one other addend, we insert the resulting term equation to the set T in Step 3. If $a_{i^\star}$ was the only addend, the resulting term equation reads $\{0 \otimes \cdots \otimes 0 = 0 \otimes \cdots \otimes 0\}$ which is redundant information. If additionally $\mu = \beta_r$ the original term equation was $\{r = r\}$, and it can be removed from T in Step 4.

Due to fixings of the resultant's bits or due to the processing of self-referencing term equations in Step 3 of Algorithm 15.2 it may happen that some or all of the resultant's bits r_b in a term equation $r = t$ are fixed to zero. Step 4 tries to exploit this situation by calling Algorithm 15.4. In Step 1 of this algorithm, we count the number q of least significant bits in the resultant that are fixed to zero. If $q = 0$, i.e., $r_0 \neq 0$, we cannot process this term equation. Otherwise, we can look at the less significant part $r|_q = t|_q$ of the equation, which is valid due to Corollary 15.3. Since $r|_q \equiv_E 0 \otimes \cdots \otimes 0$, one of the addends a_i in the truncated equation can be moved to the left hand side and take the role of the new resultant. Note, however, that we can only deal with addends consisting of only one factor, i.e., $a_i = f_{i1} \otimes 0 \otimes \cdots \otimes 0$ or $a_i = -f_{i1} \otimes 0 \otimes \cdots \otimes 0$, since otherwise, the left hand side $r^\star$ of the resulting equation $r^\star = t^\star$ would not be a simple bit string.

We select such a "simple" addend in Step 2. If there is no addend with a single factor, we have to abort. If the selection was successful, we remove the addend from the term in Step 3 to yield $t^\star$ and move it to the left hand side of the term equation as new resultant $r^\star$. If $a_i = a_1$ was the only addend, i.e., $m = 1$, we strip a potentially existing minus sign "$-$" from a_i and mark the remaining term $t^\star$ to be zero (implicitly applying rewriting Rules 1a and 1b of the normalization Algorithm 15.1). Otherwise, we have to check whether the addend has a minus

Algorithm 15.5 Term Algebra Presolving – Deductions on Resultant Bits

Input: Term equation set T and term equation $\{r = t\} \in T$ with t in normal form as in Definition 15.5.

Output: Modified term equation set T and fixings and aggregations on resultant bits r_b.

1. While $a_i = a'_i \otimes 0$ or $a_i = -(a'_i \otimes 0)$ for all $i = 1, \ldots, m$ with $a'_i \in \mathcal{T}_\Sigma$, fix $r_0 := 0$, replace $\{r = t\}$ by $\{r_{\beta_r - 1} \otimes \ldots \otimes r_1 = a'_m + \ldots + a'_1\}$, and normalize the term again by calling Algorithm 15.1. Update T accordingly.
2. If $m = 1$, $m_1 = 1$, and a_1 has no minus sign, i.e., $t = c_{\beta_r - 1} \otimes \cdots \otimes c_0$ with $c_b \in O_0$, then aggregate $r_b :\overset{\star}{=} c_b$ for all $b = 0, \ldots, \beta_r - 1$ and delete the term equation from T.
3. If $m = 1$, $m_1 = 1$, and a_1 has a minus sign, i.e., $t = -c_{\beta_r - 1} \otimes \cdots \otimes c_0$ with $c_b \in O_0$, then set $o := 1$ and for all $b = 0, \ldots, \beta_r - 1$:
   (a) If $o = 1$, aggregate $r_b :\overset{\star}{=} c_b$. If $c_b = 1$, set $o := 0$. If $c_b \notin \{0, 1\}$ abort this loop.
   (b) If $o = 0$, aggregate $r_b :\overset{\star}{=} 1 - c_b$.

   If the loop was not aborted prematurely in Step 3a, delete the term equation from T.

sign as first operator. If not, we can use the truncated addend as new resultant $r^\star = a_i|_q$. The term, however, has to be used in its negative form which means that all addends in $t^\star$ must be preceded by a "$-$". On the other hand, if a_i has a minus sign as first operator, we use it in its negative form $a^\star = -a_i|_q$ and leave the remaining term $t^\star$ as it is. Step 4 normalizes both sides of the equation $r^\star = t^\star$. Note that $r^\star$ was defined in such a way that the normalization reduces $r^\star$ to a simple bit concatenation. Finally, in Step 5 the new term equation $r^\star = t^\star$ is inserted into T. If $q = \beta_r$ which means $r_b = 0$ for all resultant bits, the new term equation replaces the old one. Otherwise, the new term equation is only defined on a subset of the bits such that the old equation must remain in T.

The term algebra presolving Algorithm 15.2 continues in Step 5 by inspecting the terms of the term equation in order to deduce fixings and aggregations for the resultant bits. Algorithm 15.5 shows the details of this procedure. If all addends of the term t are proven to be zero on their least significant bit, the resultant bit r_0 can also be fixed to $r_0 := 0$ in Step 1. Additionally, we can prune the least significant bit from the resultant and from each addend of the term which corresponds to dividing the equation by the common divisor 2. By the definition of the normal form of t, this can be applied iteratively $s^\star = \min\{s_i \mid i \in \{1, \ldots, m\}\}$ times.

If a term t has only one addend with a single factor, i.e., $t = [-]c_{\beta_r - 1} \otimes \cdots \otimes c_0$, we can directly exploit the equation $r = t$ and aggregate the bits of r accordingly. In the case that t has no minus sign, we can aggregate $r_b :\overset{\star}{=} c_b$ for all bits $b = 0, \ldots, \beta_r - 1$ in Step 2. If t is preceded by a minus sign, we have to calculate the two's complement of $c = c_{\beta_r - 1} \otimes \cdots \otimes c_0$ manually in Step 3 in order to deduce aggregations on the bits r_b. The two's complement is defined as $-c = \text{NOT}(c) + 1$. Thus, we start with the least significant bit r_0 and aggregate $r_0 :\overset{\star}{=} c_0$ since negation and addition of one cancel each other on the least significant bit. Afterwards, we calculate the overflow o of the "$+1$" addition which is $o = 1 \Leftrightarrow \text{NOT}(c_0) = 1 \Leftrightarrow c_0 = 0$. As long as the

Algorithm 15.6 Term Algebra Presolving – Operand Substitution

Input: Term equation set T and term equation $\{r = t\} \in T$ with t in normal form as in Definition 15.5.

Output: Modified term equation set T.

1. Set $i^\star := 0$, $j^\star := 0$, $M^\star := \infty$, and $q^\star := 0$.
2. For all $i = 1, \ldots, m$, $j = 1, \ldots, m_i$, and $\{r' = t'\} \in T$:
   (a) If $q = \max\left\{\mu \leq \min\{\beta_{r'}, \beta_r - s_i\} \;\middle|\; r'|_\mu \equiv_E f_{ij}|_\mu\right\} > q^\star$, or $q = q^\star \geq 1$ and $M' = \sum_{i'=1}^{m'} m'_{i'} < M^\star$, and
   (b) there is no subterm f'_{ij} of t' with $r'|_q \equiv_E f'_{ij}|_q$,

   update $i^\star := i$, $j^\star := j$, $M^\star := M'$, $q^\star := q$, $r^\star := r'$, and $t^\star = t'$.
3. If $q^\star = 0$, stop.
4. Set $\tilde{r} := r|_{q^\star + s_{i^\star}}$. Set $\tilde{t} := t|_{q^\star + s_{i^\star}}$ and substitute $f_{i^\star j^\star}|_{q^\star} \to t^\star|_{q^\star}$ therein.
5. Normalize $\tilde{r}$ and $\tilde{t}$ by calling Algorithm 15.1.
6. If $q^\star = \beta_r - s_{i^\star}$, replace $\{r = t\}$ in T by $\{r = \tilde{t}\}$. Otherwise, add $\{\tilde{r} = \tilde{t}\}$ to T.

overflow o is uniquely determined because c_b is a fixed value and no variable register bit, we can continue the aggregation of more significant bits. If the overflow becomes zero, all remaining bits can be aggregated as $r_b \overset{\star}{:=} 1 - c_b$ since no additional overflow can appear.

Step 6 of Algorithm 15.2 substitutes operands f_{ij} in term equations $\{r = t\} \in T$ for terms t' that are stored as defining terms $\{f_{ij} = t'\} \in T$. Due to Corollary 15.3, we can also substitute less significant substrings of the operands by corresponding terms, thereby generating a term equation that is valid on this less significant part. The substitution is performed by Algorithm 15.6. Steps 1 and 2 select an operand $f_{i^\star j^\star}$ with $f_{i^\star j^\star}|_\mu \equiv_E r^\star|_\mu$ that should be substituted using $\{r^\star = t^\star\} \in T$. For a substitution candidate pair (f_{ij}, r') we calculate in Step 2a the number q of less significant bits in r' that match the bits of f_{ij}. Note that

$$r'|_\mu \equiv_E f_{ij}|_\mu \quad \Leftrightarrow \quad r'_b = c_{ij,b} \text{ for all } b = 0, \ldots, \mu - 1,$$

which is easy to check. If $r'_0 \neq c_{ij,0}$, we define $q := 0$. From the substitution candidates we select the one with the largest number q of matching bits. If more than one substitution leads to the same number of matching bits, we choose the one with the least total number M of operands f'_{ij} in the term t' in order to keep the substitution as simple as possible. Condition 2b ensures that we do not generate infinite chains of substitutions by excluding self-referencing term equations. If we would allow to substitute f_{ij} for a term $t'(f_{ij})$ we risked the successive generation of an infinite term $r = t(t'(\ldots(t'(f_{ij}))))$.

After having selected a candidate, we truncate the resultant r and the term t to the valid width of the substitution and apply the substitution $f_{i^\star j^\star}|_{q^\star} \to t^\star|_{q^\star}$ in Step 4 which yields the valid equation $\tilde{r} = \tilde{t}$. Both sides of this equation are normalized in Step 5 after which $\tilde{r}$ becomes a simple bit concatenation. If the match of operand bits to resultant bits was exhaustive, we have $r = \tilde{r}$, and Step 6 replaces the original term in T by the substituted version. Otherwise, we extend the term equation set T by $\tilde{r} = \tilde{t}$ which is only valid on a lower significant part.

Steps 3 to 6 of the term algebra presolving Algorithm 15.2 are called iteratively

until no more problem reductions or term substitutions were applied. Afterwards we inspect the final term equation database T in Step 8 to find pairs of term equations $r = t$ and $r' = t'$ for which the terms t and t' are equivalent at least on a subset of the bits. Step 8a calculates the maximal width μ up to which the two terms are equivalent. In Step 8b the corresponding resultant bits are aggregated. If all bits up to the full width $\beta_{r'}$ of the shorter or equal resultant r' were aggregated, the term equation $r' = t'$ can be deleted from T in Step 8c.

Remark 15.8. Due to the fixings and aggregations of register bits ϱ_{jb} and due to the constraint rewritings performed in other presolving algorithms, our term rewriting system is not restricted to ADD, MULT, AND, OR, XOR, UAND, UOR, and UXOR constraints. Instead, we also implicitly consider most of the other circuit operators, namely MINUS, SUB, NOT, ZEROEXT, SIGNEXT, and CONCAT, as well as SHL, SHR, SLICE, READ, and WRITE if the offset operand is fixed.

In the current version of our code, we do not exploit the distributivity law, and we do not mix addition and multiplication constraints in the substitution step. We expect that further preprocessing improvements can be achieved by incorporating other constraints like EQ, LT, ITE or the subword access operators with variable offset operand into the term algebra and by exploiting rewriting rules like the distributivity law which link different operations.

15.2 Irrelevance Detection

In order to prove the validity of a given property, often only a part of the circuit has to be considered. For example, if a property on an arithmetic logical unit (ALU) describes a certain aspect of the addition operation, the other operations of the ALU are irrelevant. Suppose that ITE constraints select the operation of the ALU by routing the output of the desired operation to the output register of the circuit, compare Figure 14.8 of Section 14.15 on page 252. The calculated values of the other operations are linked to the discarded inputs of the ITE constraints and thereby do not contribute to the output of the circuit. These *irrelevant* constraints and the involved intermediate registers have no influence on the validity of the property and can therefore be removed from the problem instance.

The detection of irrelevant parts of the circuit can also be applied to the local subproblems during the branch-and-bound search. In particular, irrelevant register variables need not to be considered as branching candidates. Disregarding locally irrelevant variables in the branching decision can be seen as replacement for the more indirect method of selecting the next branching variable under the literals involved in the recent conflict clauses, which is employed in state-of-the-art SAT solvers [100].

The identification and removal of irrelevant parts of the circuit is also known as *localization reduction* which was developed by Kurshan [137] or *cone of influence reduction*, which is explained, for example, in Biere et al. [43] and Clarke et al. [61]. Similar ideas can be found in the *program slicing* technique of Weiser [210, 211] for decomposing software systems. However, these techniques are employed for SAT based property checking only in presolving and not during the traversal of the search tree. Current state-of-the-art SAT solvers rely on a very fast processing of the individual subproblems, spending most of the time for binary constraint propagation. Incorporating local cone of influence reduction would lead to a large increase in the subproblem processing time. Thus, the performance of SAT solvers would most

likely deteriorate. In contrast, our constraint integer programming solver spends much more time on each individual subproblem with the hope that this will be compensated by a much smaller search tree. The additional overhead to further prune locally irrelevant parts of the circuit is negligible, in particular compared to the time needed to solve the LP relaxations and to execute the domain propagation algorithms.

We implemented the irrelevance detection as part of the domain propagation and presolving algorithms of the circuit constraint handlers, see Chapter 14. The basic reasoning is as follows: whenever a register ϱ_j is only used in a single constraint $\mathcal{C}_i$, and for all values of the other registers involved in $\mathcal{C}_i$ there exists a value for ϱ_j such that the constraint is feasible, we can delete $\mathcal{C}_i$ from the problem. If a counter-example for the property is found, we can calculate a valid value for ϱ_j in a postprocessing stage. Because all circuit operations are totally defined, we can in particular delete constraints for which the resultant r is not used in any other constraint. For certain operators, however, irrelevance detection can also be applied to input registers.

The irrelevance detection procedure is illustrated in Algorithm 15.7. In Step 1 the function graph $G = (V, A)$ is created. Recall that this is a directed bipartite graph $G = (V_\varrho \cup V_\mathfrak{C}, A)$ with two different types of nodes, namely register nodes $V_\varrho = \{\varrho_1, \ldots, \varrho_n\}$ and constraint nodes $V_\mathfrak{C} = \{\mathcal{C}_1, \ldots, \mathcal{C}_m\}$. The arc set is defined as

$$\begin{aligned} A = &\{(\varrho_j, \mathcal{C}_i) \mid \text{register } \varrho_j \text{ is input of circuit operation } \mathcal{C}_i\} \\ &\cup \{(\mathcal{C}_i, \varrho_j) \mid \text{register } \varrho_j \text{ is output of circuit operation } \mathcal{C}_i\}. \end{aligned}$$

Note that although AND and XOR constraints are replaced in presolving by a corresponding number of unary UAND and UXOR constraints, they are still represented as binary AND and XOR constraints in the function graph in order to preserve the structure of the graph. Additionally, whenever a constraint is deleted from the problem because the register bits are fixed in a way such that the constraint will always be feasible, its representative in the function graph is retained.

Constant registers in the property checking problem cannot be set to a matching value while postprocessing a partial solution. Therefore, we have to make sure that they are not used to detect the irrelevance of a constraint. This is achieved by adding an extra node λ to the vertex set of G and linking it to the constant register vertices ϱ_j in Step 2, thereby increasing their degree $d(\varrho_j) := d^+(\varrho_j) + d^-(\varrho_j)$.

Step 3 processes special situations in MULT and ITE constraints. If one of the input registers of a MULT constraint $r = \text{MULT}(x, y)$ is fixed to zero, the resultant r will always be zero, independently from the value of the other operand. Therefore, we can unlink the other operand from the constraint vertex. If the selection operand x of an ITE constraint $r = \text{ITE}(x, y, z)$ is fixed, the operand that is not selected by x does not contribute to the resultant. Again, it can be unlinked from the constraint vertex. If the two input registers y and z are equivalent, the resultant r will also be equal to this single value, independently from the selection operand x. Thus, we can unlink x from the ITE constraint.

Finally, in Step 4 we check for irrelevant constraints. If the degree $d(r)$ of the resultant vertex of a constraint is equal to one, the resultant register is not used in any other constraint and is detected by Condition 4a to be irrelevant for the validity of the property. We can delete the constraint and its resultant register from the problem and calculate the register's value in a postprocessing step if a counter-example has been found.

Algorithm 15.7 Irrelevance Detection

1. Construct the function graph $G = (V, A)$ of the property checking problem, see Section 13.2.
2. Set $V := V \cup \{\lambda\}$. For all constant registers ϱ_j add an arc (λ, ϱ_j) to A.
3. For all constraints $\mathcal{C}_i \in V_{\mathfrak{C}}$ with the indicated constraint type:
   (a) $r = \text{MULT}(x, y)$:
      i. If $x = 0$, remove $(y, \mathcal{C}_i)$ from A.
      ii. If $y = 0$, remove $(x, \mathcal{C}_i)$ from A.
   (b) $r = \text{ITE}(x, y, z)$:
      i. If $x = 1$, remove $(z, \mathcal{C}_i)$ from A.
      ii. If $x = 0$, remove $(y, \mathcal{C}_i)$ from A.
      iii. If $y \stackrel{\star}{=} z$, remove $(x, \mathcal{C}_i)$ from A.
4. For all constraints $\mathcal{C}_i \in V_{\mathfrak{C}}$ with $\mathcal{C}_i = \{r = op(x, y, z)\}$:
   If one of the following holds for the indicated constraint type:
   (a) any constraint: $d(r) = 1$,
   (b) $r = \text{ADD}(x, y)$: $d(x) = 1$ or $d(y) = 1$,
   (c) $r = \text{MULT}(x, y)$: $x, y \in \delta^+(\mathcal{C}_i)$ and $d(x) = d(y) = 1$,
   (d) $r = \text{NOT}(x)$: $d(x) = 1$,
   (e) $r = \text{AND}(x, y)$: $d(x) = d(y) = 1$,
   (f) $r = \text{XOR}(x, y)$: $d(x) = 1$ or $d(y) = 1$,
   (g) $r = \text{UAND}(x)$: $d(x) = 1$,
   (h) $r = \text{UXOR}(x)$: $d(x) = 1$,
   (i) $r = \text{EQ}(x, y)$: $d(x) = 1$ or $d(y) = 1$,
   (j) $r = \text{ITE}(x, y, z)$:
      i. $x, y \in \delta^+(\mathcal{C}_i)$ and $d(x) = d(y) = 1$, or
      ii. $x, z \in \delta^+(\mathcal{C}_i)$ and $d(x) = d(z) = 1$, or
      iii. $y, z \in \delta^+(\mathcal{C}_i)$ and $d(y) = d(z) = 1$,
   (k) $r = \text{CONCAT}(x, y)$: $d(x) = d(y) = 1$,

   the constraint is irrelevant. Delete $\mathcal{C}_i$ from the (sub)problem and from the (local) function graph G.

Conditions 4b to 4k check for additional situations in which the constraint and one or more of the involved registers are irrelevant. If one of the operand registers in an ADD, XOR, or EQ constraint is not used anywhere else, we can choose its value for an arbitrary partial solution in such a way, that the respective constraint becomes feasible. Consider the case $d(x) = 1$. The constraint $r = \text{ADD}(x, y)$ can be made feasible for any given r and y by setting $x = \text{SUB}(r, y)$. If $r = \text{XOR}(x, y)$ we have to calculate $x = \text{XOR}(r, y)$. In the case $r = \text{EQ}(x, y)$ we have to set $x = y$ if $r = 1$, and can choose any $x \neq y$ if $r = 0$. Therefore, constraints of these types are detected to be irrelevant if $d(x) = 1$ or $d(y) = 1$ by Conditions 4b, 4f, and 4i.

If all involved operands of a MULT, NOT, AND, UAND, UXOR, or CONCAT constraint have a degree of one in the function graph G, we can find operand values for any given resultant r such that the constraint is feasible. In the case of MULT constraints, however, we also have to check whether the operands are still linked to the constraint since the corresponding arcs could already have been deleted in Step 3a. If both

links still exist, we can always choose $x = r$ and $y = 1$ to yield a valid constraint $r = \text{MULT}(x, y)$. Analogous postprocessings for a given resultant value r can be performed for the other constraints:

- $x = \text{NOT}(r)$ for NOT constraints,
- $x = y = r$ for AND constraints,
- $x = 0$ if $r = 0$ and $x = (1, \ldots, 1)$ if $r = 1$ for UAND constraints,
- $x = 0$ if $r = 0$ and $x = (0, \ldots, 0, 1)$ if $r = 1$ for UXOR constraints, and
- $x = (r_{\beta_r - 1}, \ldots, r_{\beta_y})$ and $y = (r_{\beta_y - 1}, \ldots, r_0)$ for CONCAT constraints.

For an ITE constraint $r = \text{ITE}(x, y, z)$ we can detect irrelevance in the following situations. If the selection operand x and one of the case operands y or z have both a degree of one and are still linked to the constraint, irrelevance is detected by Conditions 4(j)i or 4(j)ii, respectively. Assume $d(x) = d(y) = 1$. For a given resultant value r and operand value z, we can always set $x = 1$ and $y = r$ to turn $r = \text{ITE}(x, y, z)$ into a feasible constraint. On the other hand, if $d(x) = d(z) = 1$ we just have to set $x = 0$ and $z = r$ and do not have to care about the given value of y. If neither of y and z is appearing in the remaining problem instance, i.e., $y, z \in \delta^+(\mathcal{C}_i)$ and $d(y) = d(z) = 1$, we can postprocess $y = z = r$ independently of the given value of x.

Chapter 16

Search

A branch-and-bound algorithm mainly consists of three steps that are iterated until the given problem instance has been solved, see Section 2.1:

1. Select a subproblem.
2. Process the subproblem.
3. If not pruned, split the subproblem into smaller parts.

The subproblem processing Step 2 for the property checking problem is covered in Chapter 14. The current chapter explains the branching strategy for Step 3 and the node selection rule for Step 1 that we employ.

16.1 Branching

The branching strategy in a branch-and-bound algorithm defines how the problem is recursively split into smaller subproblems, compare Section 2.1 and Chapter 5. Therefore, it is the most important component for determining the shape of the search tree. The ultimate goal of the branching strategy is to partition the problem instance in such a way that it can be solved by processing only a minimal number of subproblems. This global objective, however, is practically impossible to achieve. Instead, since we have to make our branching decision locally at each subproblem, we can at best try to follow some local criteria and hope that this yields a small search tree for the global problem.

As mentioned in Section 2.1, the most popular branching strategy is to *branch on variables*, i.e., split the domain of a single variable into two parts. Since in branch-and-bound solvers the domains of the variables are usually treated implicitly, branching on variables has the advantage that one does not have to explicitly add constraints to the subproblems, thereby avoiding overhead in the subproblem management. In particular, this is true for mixed integer programming and for SAT solving. For the same reason, we also apply branching on variables to the constraint integer programming model of the property checking problem.

In mixed integer programming, the branching selection is mostly guided by the LP relaxation. Branching takes place on integer variables that have a fractional value in the current LP solution. The idea is that the LP relaxation is undecided about these variables and we have to help the LP by forcing a decision. Indeed, one can very often either find an integral LP solution after only a very few (compared to the number of integer variables) branching steps or drive the LP relaxation to infeasibility. Since we also have an LP relaxation at hand for the property checking CIP, we adopt this idea and branch on integer variables with fractional LP solution value. In contrast to mixed integer programming, an integral LP solution is not necessarily a feasible CIP solution for the property checking problem, because not all circuit operators are linearized. Therefore, if the LP solution is integral but still

not feasible, we have to branch on an integer variable with integral LP solution value.

We saw in Chapter 5 that for mixed integer programming the change in the objective value of the LP relaxation is a good indicator for selecting the branching variable. The *full strong branching* strategy yields very good results in the number of branching nodes needed to solve the problem instances. In this strategy, we evaluate the impact of a potential branching decision by solving the two corresponding branching LPs. This is performed for every integer variable with fractional LP value. Then, we choose the variable for which the objective value of the branching LPs increased the most. Unfortunately, full strong branching is very expensive such that the reduction in the number of branching nodes is usually outweighed by the time needed to choose the branching variables. Therefore, one tries to approximate full strong branching by less expensive methods, for example by estimating the LP objective increase instead of calculating it by solving the branching LPs.

In contrast to most mixed integer programs, the property checking problem does not contain an objective function. As we model the problem as constraint integer program, we have to specify objective function coefficients for the variables, but these are just artificial values. Therefore, it seems unlikely that the increase in the objective value of the LP relaxations gives meaningful hints about the quality of the branching candidates.

The satisfiability problem does not contain an objective function either. Current SAT solvers usually select the branching variable by some variant of the *variable state independent decaying sum (VSIDS)* strategy, see Moskewicz et al. [168]. This branching rule basically prefers variables that appear in the recently generated conflict clauses. Goldberg and Novikov [100] improved this scheme in their solver BERKMIN by noting that not only the variables in the final conflict clauses should be regarded as promising branching candidates but also all other variables involved in the conflicts, i.e., all variables on the *conflict side* of the conflict graphs, see Chapter 11.

In our property checking solver, we use a mixture of MIP and SAT branching strategies. Like in MIP, we select the branching variable under all integer variables with fractional LP values. If the LP solution is integral but still not feasible for the CIP, we consider all unfixed integer variables (i.e., integer variables with at least two values in their current domain) as branching candidates. From these candidates we choose the branching variable in a SAT-like manner by applying the VSIDS strategy of BERKMIN. Additionally, we exploit our knowledge about the structure of the circuit by disregarding variables that belong to registers which are irrelevant for the current subproblem. Such registers are identified by the *irrelevance detection* described in Section 15.2.

16.2 Node Selection

The node selection strategy in a branch-and-bound solver determines in which order the nodes of the search tree, defined by the branching rule, are traversed. In mixed integer programming one usually employs a mixture of best first and depth first search, see Chapter 6. Best first search aims to improve the global dual bound as fast as possible by always selecting the subproblem with the lowest dual bound as next subproblem to be processed. In fact, for a fixed branching strategy, a pure best first search traversal of the tree would solve the problem instance with the

minimal number of branching nodes. The disadvantage of best first search is that it entails a large amount of switching between subproblems that are very far away from each other in the search tree. This involves significant subproblem management overhead. Therefore, the best first search strategy is supplemented with depth first search elements: after a best subproblem was selected, the search continues with a few iterations of depth first search, after which again a best subproblem is selected.

Since SAT does not contain an objective function, best first search does not make sense for SAT. Indeed, state-of-the-art SAT solvers employ pure depth first search. This has several advantages in the context of SAT. First, the subproblem management is reduced to a minimum. In fact, SAT solvers completely avoid the explicit generation of the search tree. Instead, they only keep track of the path to the current subproblem and process the tree implicitly by using backtracking and by generating conflict clauses, see Chapter 11. The second advantage is that the management of conflict clauses is much easier. Most conflict clauses are only useful in a certain part of the search tree, namely in the close vicinity of the node for which they were generated. Since depth first search completely solves the nodes in this part of the tree before moving on to other regions, one can effectively identify conflict clauses that are no longer useful and delete them from the clause database, see Goldberg and Novikov [100] for details. Nevertheless, depth first search entails the risk of getting stuck in an infeasible region of the search tree while there are lots of easy to find feasible solutions in other parts of the tree. To avoid such a situation, SAT solvers perform frequent *restarts*, which means to completely undo all branching decisions and start the search from scratch. Since conflict clauses are retained, the information gathered during the search is preserved.

Like the satisfiability problem, the property checking problem does not contain an objective function. Therefore, we mainly employ depth first search. The problem of getting stuck in infeasible or fruitless parts of the tree, however, cannot be resolved by restarts as easily as in SAT solvers. Because the effort in solving the LPs would be lost, restarts were too costly. Therefore, we enrich the depth first search strategy by a small amount of best first search: after every 100 nodes of depth first search, we select a best node from the tree and continue depth first search from this node. The subproblem management overhead for this combined strategy is marginal, the effect is similar to the restarts of SAT solvers, but the LP relaxations of the processed nodes do not have to be solved again. A disadvantage compared to actual restarts is that we cannot undo our branching decisions.

Chapter 17

Computational Results

In this chapter we examine the computational effectiveness of the described constraint integer programming techniques on industrial benchmarks obtained from verification projects conducted by OneSpin Solutions. The instances are described in Appendix A.4.

In a first series of benchmarks, we compare the CIP approach with the current state-of-the-art technology, which is to apply a SAT solver to the gate level representation of the circuit and property. Before the SAT solver is called, a preprocessing step is executed to simplify the instance at the gate level, which is based on binary decision diagrams (BDDs). We use MiniSat 2.0 [82] to solve the resulting SAT instances. We also tried MiniSat 1.14, Siege v4 [196], and zChaff 2004.11.15 [168], but MiniSat 2.0 turned out to perform best on most of the instances of our test set.

A second set of benchmarks evaluates the impact of the problem specific presolving methods that we described in Chapter 15, of probing as explained in Section 10.6, and of conflict analysis, which is described in Chapter 11. We also performed additional benchmark tests to assess the performance impact of other components like the branching and node selection strategies, or the specification of the objective function, but the alternative settings that we tried did not have a strong influence on the performance.

17.1 Comparison of CIP and SAT

Tables 17.1 and 17.2 present the comparison of MiniSat and the CIP approach on an arithmetical logical unit (ALU). This circuit is able to perform ADD, SUB, SHL, SHR, and signed and unsigned MULT operations on two input registers. We consider multiple versions of the circuit which differ in the width of the input registers. The width is depicted in the second column of the tables. Apart from the number of branching nodes and the time in CPU seconds needed to solve each instance, the other entries of the tables show the number of clauses and variables of the SAT representations in conjunctive normal form (CNF) and the number of constraints and variables of the CIP instances. Note that the registers in the CIP formulation are represented as words and bits which are linked via special constraints, see Section 14.1. Thus, each register in the register transfer level description gives rise to one additional constraint, and the number of variables shown in the tables equals the total number of bits in the registers plus the number of 16-bit words that cover the registers.

Overall, we investigated 11 different properties of the ALU circuit. Six of them turned out to be trivial for both, the SAT and the CIP solver, and they are not listed in the tables. From the remaining five properties, two are invalid and three are valid.

The results on the invalid properties are shown in Table 17.1. Recall that for such instances the task is to find a feasible solution, i.e., a counter-example to the

		SAT				CIP			
Property	width	Clauses	Vars	Nodes	Time	Constrs	Vars	Nodes	Time
add_fail	5	181	80	1	0.0	96	293	3	0.0
	10	326	145	1	0.0	96	356	3	0.0
	15	471	210	1	0.0	96	426	3	0.0
	20	616	275	1	0.0	96	491	2	0.0
	25	761	340	1	0.0	96	551	2	0.0
	30	906	405	1	0.0	95	610	2	0.0
	35	1 051	470	1	0.0	93	661	2	0.0
	40	1 196	535	1	0.0	93	731	2	0.0
sub_fail	5	223	94	1	0.0	103	356	3	0.0
	10	428	179	1	0.0	103	459	3	0.0
	15	633	264	1	0.0	103	577	3	0.0
	20	838	349	1	0.0	103	682	3	0.0
	25	1 043	434	1	0.0	103	782	3	0.1
	30	1 248	519	1	0.0	102	886	3	0.1
	35	1 453	604	1	0.0	100	980	4	0.1
	40	1 658	689	1	0.0	100	1 090	4	0.1

Table 17.1. Comparison of SAT and CIP on invalid ALU properties.

property. As can be seen in the table, both solvers accomplish this task very quickly, even for the largest of the considered register widths.

The most interesting numbers in this table are the sizes of the problem instances. Obviously, the BDD preprocessing applied prior to generating the CNF input for the SAT solver did a great job to reduce the size of the instance. The addition and subtraction operations can be encoded quite easily on the gate level, such that the number of clauses in the SAT representation is rather small. The number of remaining variables for SAT is even smaller than for the CIP approach, even if one accounts for the double modeling on word and bit level in the CIP representation. Of course, the presolving of the CIP solver is also able to reduce the size of the instances considerably. For example, the presolved `add_fail` instance on input registers of width $\beta = 5$ consists of only 61 constraints and 34 variables, including the necessary auxiliary variables for the LP relaxation.

Table 17.2 presents the benchmark results on the valid ALU properties. Here, the solver has to prove the infeasibility of the instances. The `muls` property involves the verification of the signed multiplication operation. As one can see, for register widths of 10 bits or larger, the SAT approach cannot prove the validity of the property within the time limit of 2 hours. Additionally, the CNF formula is already quite large with almost 200000 clauses and more than 60000 variables for 40-bit registers. This is due to the fact that the multiplication has to be represented as complex addition network in the gate level description. In contrast, the sizes of the CIP models are similar to the ones for the invalid ALU properties. Because the structural information of the circuit is still available at the RT level, the CIP solver is able to prove the property already in the presolving step, which is denoted by the branching node count of 0.

The `neg_flag` property represents a validity check on the sign flag of the status register. In contrast to the previously described properties, it does not select a specific arithmetical operation but deals with all operations simultaneously. On these instances, the CIP approach is again superior to SAT, although it cannot prove the property in presolving and has to revert to branching. Still, it can solve even the largest instances within a reasonable time, while the SAT approach is already considerably slower for 10-bit registers and fails to solve the instances with registers of width $\beta = 15$ or larger.

		SAT				CIP			
Property	width	Clauses	Vars	Nodes	Time	Constrs	Vars	Nodes	Time
muls	5	3 343	1 140	13 864	0.5	79	302	0	0.0
	10	12 828	4 325	—	>7200.0	99	470	0	0.0
	15	28 463	9 560	—	>7200.0	116	646	0	0.0
	20	50 248	16 845	—	>7200.0	136	947	0	0.1
	25	78 183	26 180	—	>7200.0	156	1 294	0	0.1
	30	112 268	37 565	—	>7200.0	176	1 684	0	0.1
	35	152 503	51 000	—	>7200.0	196	2 144	0	0.2
	40	198 888	66 485	—	>7200.0	216	2 651	0	0.3
neg_flag	5	3 436	1 166	3 081	0.1	318	1 103	45	0.8
	10	12 826	4 306	941 867	100.0	340	1 439	50	3.6
	15	28 366	9 496	—	>7200.0	352	1 693	64	11.6
	20	50 056	16 736	—	>7200.0	374	2 220	42	36.3
	25	77 896	26 026	—	>7200.0	394	2 726	107	81.8
	30	111 886	37 366	—	>7200.0	413	3 274	53	136.6
	35	152 026	50 756	—	>7200.0	422	3 714	35	218.4
	40	198 316	66 196	—	>7200.0	442	4 385	55	383.5
zero_flag	5	3 119	1 109	79	0.0	323	1 127	54	2.3
	10	9 974	3 454	137	0.0	345	1 485	28	0.6
	15	20 729	7 099	176	0.1	357	1 763	37	1.6
	20	35 384	12 044	73	0.1	379	2 322	26	4.0
	25	53 939	18 289	202	0.2	399	2 851	78	6.2
	30	76 394	25 834	221	0.4	418	3 424	73	10.7
	35	102 749	34 679	185	0.5	427	3 895	63	15.6
	40	133 004	44 824	103	0.6	447	4 589	185	379.7

Table 17.2. Comparison of SAT and CIP on valid ALU properties.

Similar to `neg_flag`, the `zero_flag` property is "global" in the sense that it does not focus on a single arithmetical or logical operation in the ALU circuit. It describes the desired behavior of the zero flag in the status register. The performance of the CIP solver on these instances is better than the `neg_flag` performance. Note that the long runtime on the 40-bit instance is an exception, which is due to "bad luck" in probing, see Section 10.6 (97 % of the total time is spent in presolving, most of it in probing). The 39-bit instance (not shown in the table) solves in 39.6 seconds. As shown in Section 17.3, the 40-bit instance can be solved without probing in 9.3 seconds.

In contrast to the `neg_flag` results, the SAT approach is surprisingly efficient on the `zero_flag` property. This result may be related to the difference of signed and unsigned multiplication: the property on unsigned multiplication is trivial for both SAT and CIP and is therefore not listed in the tables, while the signed multiplication instances of larger register widths are intractable for SAT. Since the zero status flag does not depend on the sign of the result, this property is easier to prove for SAT solvers than the `neg_flag` property.

Tables 17.3 and 17.4 show the results for properties of a pipelined adder. The underlying chip design is very similar to the circuit of Example 13.2, which is depicted in Figure 13.1 on page 191. It has one input register and an internal accumulator register, which adds up the values assigned to the input during consecutive time steps. Additionally, it has a reset signal that clears the accumulator. The properties verify certain variants of the commutative and associative law on inputs over four consecutive time steps. Again, we distinguish between the invalid properties of Table 17.3 and the valid properties of Table 17.4.

As can be seen in Table 17.3, finding counter-examples for the invalid PipeAdder instances is very easy for both solvers. Since the CIP data structures are much more complex and involve a larger overhead, the SAT solver is slightly faster. It is

		SAT				CIP			
Property	width	Clauses	Vars	Nodes	Time	Constrs	Vars	Nodes	Time
#1	5	3126	1204	136	0.0	281	706	6	0.1
	10	5691	2189	233	0.0	281	1071	6	0.1
	15	8272	3174	592	0.0	281	1436	6	0.1
	20	10821	4159	436	0.0	281	1874	6	0.2
	25	13394	5144	1209	0.1	281	2239	6	0.2
	30	15967	6129	984	0.1	281	2604	6	0.2
	35	18524	7114	1552	0.1	281	3042	6	0.3
	40	21081	8099	1283	0.1	281	3407	6	0.4
#2	5	2271	898	1	0.0	228	620	4	0.1
	10	4276	1683	62	0.0	228	965	17	0.1
	15	6281	2468	148	0.0	228	1310	60	0.2
	20	8286	3253	1	0.0	228	1724	76	0.3
	25	10291	4038	239	0.0	228	2069	50	0.3
	30	12296	4823	423	0.0	228	2414	109	0.4
	35	14301	5608	1	0.1	228	2828	125	0.6
	40	16306	6393	1	0.1	228	3173	138	0.7
#3	5	2475	966	91	0.0	236	648	5	0.1
	10	4675	1816	389	0.0	236	1009	5	0.1
	15	6875	2666	388	0.0	236	1373	4	0.2
	20	9075	3516	914	0.0	236	1802	34	0.2
	25	11275	4366	1745	0.0	236	2162	4	0.2
	30	13475	5216	1823	0.1	236	2522	104	0.5
	35	15675	6066	2218	0.1	236	2955	5	0.4
	40	17875	6916	3729	0.1	236	3315	31	0.7
#4	5	2478	967	1	0.0	248	663	3	0.0
	10	4678	1817	1	0.0	248	1024	4	0.1
	15	6878	2667	1	0.0	248	1388	3	0.1
	20	9078	3517	1	0.0	248	1817	4	0.1
	25	11278	4367	1	0.0	248	2177	4	0.1
	30	13478	5217	1	0.0	248	2537	4	0.1
	35	15678	6067	1	0.0	248	2970	5	0.1
	40	17878	6917	1	0.0	248	3330	96	0.3
#5	5	2778	1067	1	0.0	264	725	3	0.0
	10	5278	2017	1	0.0	264	1126	3	0.1
	15	7778	2967	1	0.0	264	1536	3	0.1
	20	10278	3917	1	0.0	264	2007	4	0.1
	25	12778	4867	1	0.0	264	2407	4	0.1
	30	15278	5817	1	0.0	264	2807	4	0.1
	35	17778	6767	1	0.0	264	3288	4	0.1
	40	20278	7717	1	0.0	264	3688	4	0.1
#8	5	2781	1068	107	0.0	264	725	5	0.0
	10	5281	2018	247	0.0	264	1126	5	0.1
	15	7781	2968	863	0.0	264	1536	64	0.1
	20	10281	3918	1253	0.0	264	2007	20	0.1
	25	12781	4868	1618	0.0	264	2407	60	0.2
	30	15281	5818	1626	0.0	264	2807	108	0.2
	35	17781	6768	2549	0.1	264	3288	40	0.2
	40	20281	7718	3152	0.1	264	3688	56	0.2

Table 17.3. Comparison of SAT and CIP on invalid PIPEADDER properties.

		SAT				CIP			
Property	width	Clauses	Vars	Nodes	Time	Constrs	Vars	Nodes	Time
#6	5	2 778	1 067	4 321	0.0	264	725	0	0.0
	10	5 278	2 017	24 195	0.2	264	1 126	0	0.0
	15	7 778	2 967	58 839	0.5	264	1 536	0	0.0
	20	10 278	3 917	83 615	0.8	264	2 007	0	0.0
	25	12 778	4 867	145 377	1.3	264	2 407	0	0.0
	30	15 278	5 817	191 388	1.6	264	2 807	0	0.1
	35	17 778	6 767	214 307	2.1	264	3 288	0	0.1
	40	20 278	7 717	291 160	2.9	264	3 688	0	0.1
#7	5	2 778	1 067	5 696	0.0	264	725	0	0.0
	10	5 278	2 017	59 387	0.6	264	1 126	0	0.1
	15	7 778	2 967	128 369	1.3	264	1 536	0	0.1
	20	10 278	3 917	165 433	1.9	264	2 007	0	0.1
	25	12 778	4 867	187 012	2.2	264	2 407	0	0.1
	30	15 278	5 817	149 481	1.7	264	2 807	0	0.1
	35	17 778	6 767	1 343 206	22.5	264	3 288	0	0.1
	40	20 278	7 717	892 946	16.4	264	3 688	0	0.2
#9	5	2 778	1 067	14 901	0.1	261	724	0	0.0
	10	5 278	2 017	630 929	8.9	261	1 115	0	0.1
	15	7 778	2 967	1 109 126	21.3	261	1 515	0	0.1
	20	10 278	3 917	24 485 660	764.0	261	1 974	0	0.1
	25	12 778	4 867	119 922 885	5554.4	261	2 364	0	0.1
	30	15 278	5 817	2 696 270	81.6	261	2 760	0	0.1
	35	17 778	6 767	4 562 830	177.1	261	3 223	0	0.2
	40	20 278	7 717	64 749 573	4087.1	261	3 613	0	0.2
#10	5	2 778	1 067	4 321	0.0	261	724	0	0.0
	10	5 278	2 017	24 195	0.2	261	1 115	0	0.0
	15	7 778	2 967	58 839	0.5	261	1 515	0	0.0
	20	10 278	3 917	83 615	0.8	261	1 974	0	0.0
	25	12 778	4 867	145 377	1.3	261	2 364	0	0.1
	30	15 278	5 817	191 388	1.6	261	2 760	0	0.1
	35	17 778	6 767	214 307	2.1	261	3 223	0	0.1
	40	20 278	7 717	291 160	2.9	261	3 613	0	0.1

Table 17.4. Comparison of SAT and CIP on valid PipeAdder properties.

interesting to note that the number of CIP constraints stays constant for increasing register widths, while the number of clauses in the SAT representation grows in a linear fashion. This is because the number of logical gates that are needed to implement an addition grows linearly with the number of input bits.

The results for the valid properties of the PipeAdder circuit are shown in Table 17.4. All of the instances are solved by the CIP presolving techniques almost immediately. The properties #6, #7, and #10 are also quite easy to prove with SAT techniques. In contrast, property #9 poses some difficulties for MiniSat 2.0. This seems to be an issue which is specific to MiniSat 2.0, since all other SAT solvers we tried show a much more regular behavior. In particular, their runtimes increase monotonously with the widths of the input registers. The fastest SAT solver on property #9 is Siege, which solves the 40-bit version in 146.8 seconds. This is, however, considerably slower than the 0.2 seconds needed with our CIP approach.

Tables 17.5 and 17.6 also deal with a circuit that implements a pipelined arithmetical operation. In contrast to the pipelined adder of the previous tables, it performs a multiplication of the consecutively provided inputs.

The results of the pipelined multiplier are similar in quality as for the adder: counter-examples for invalid properties can be found faster with SAT techniques, while CIP is superior in proving the infeasibility of the instances that model valid properties. The differences in the runtime, however, are much more pronounced: for

		SAT				CIP			
Property	width	Clauses	Vars	Nodes	Time	Constrs	Vars	Nodes	Time
#1	5	4 254	1 578	122	0.0	305	730	6	0.3
	10	11 994	4 288	187	0.0	305	1 095	28	0.6
	15	23 875	8 373	669	0.1	305	1 460	6	0.7
	20	39 849	13 833	516	0.2	305	1 898	6	1.6
	25	59 972	20 668	1 283	0.3	305	2 263	6	2.3
	30	84 220	28 878	4 410	0.4	305	2 628	6	4.7
	35	112 577	38 463	3 876	0.6	305	3 066	6	7.7
	40	145 059	49 423	569	0.7	305	3 431	7	10.5
#2	5	3 627	1 356	73	0.0	252	664	15	0.2
	10	10 997	3 936	143	0.0	252	1 029	21	2.2
	15	22 512	7 895	418	0.1	252	1 394	45	1.2
	20	38 112	13 221	808	0.2	252	1 832	42	2.8
	25	57 867	19 928	2 425	0.3	252	2 197	39	3.5
	30	81 747	28 010	2 144	0.4	252	2 562	32	4.6
	35	109 732	37 463	6 707	0.6	252	3 000	82	7.7
	40	141 842	48 291	4 639	0.8	252	3 365	43	9.4
#3	5	4 053	1 498	72	0.0	261	714	15	0.7
	10	12 653	4 488	281	0.1	261	1 118	15	3.0
	15	26 223	9 132	702	0.1	261	1 519	33	2.6
	20	44 703	15 418	1 817	0.2	261	1 996	20	9.3
	25	68 163	23 360	3 531	0.3	261	2 399	20	13.5
	30	96 573	32 952	6 798	0.6	261	2 799	36	20.4
	35	129 913	44 190	10 432	1.0	261	3 277	45	21.8
	40	168 203	57 078	14 881	1.2	261	3 680	46	28.8
#4	5	4 056	1 499	50	0.0	273	729	11	0.5
	10	12 656	4 489	132	0.0	273	1 133	21	2.8
	15	26 226	9 133	191	0.1	273	1 534	42	6.2
	20	44 706	15 419	880	0.2	273	2 011	28	11.1
	25	68 166	23 361	1 615	0.2	273	2 414	61	23.1
	30	96 576	32 953	1 587	0.3	273	2 814	38	33.0
	35	129 916	44 191	2 517	0.5	273	3 292	120	6.3
	40	168 206	57 079	2 514	0.7	273	3 695	90	7.3
#5	5	4 800	1 747	62	0.0	289	821	16	1.1
	10	15 770	5 527	558	0.0	289	1 301	25	5.4
	15	33 360	11 511	935	0.1	289	1 772	33	15.0
	20	57 510	19 687	2 216	0.2	289	2 327	42	32.5
	25	88 290	30 069	3 435	0.3	289	2 806	85	52.0
	30	125 670	42 651	6 458	0.6	289	3 276	75	92.0
	35	169 630	57 429	8 531	0.8	289	3 832	44	55.1
	40	220 190	74 407	10 970	1.0	289	4 311	422	125.2
#8	5	4 944	1 795	40	0.0	289	821	51	1.2
	10	16 079	5 630	212	0.1	289	1 301	133	8.1
	15	33 834	11 669	587	0.1	289	1 772	55	19.8
	20	58 149	19 900	1 597	0.2	289	2 327	154	43.1
	25	89 094	30 337	1 727	0.3	289	2 806	279	45.5
	30	126 639	42 974	2 530	0.6	289	3 276	73	89.9
	35	170 764	57 807	2 596	0.7	289	3 832	92	118.9
	40	221 489	74 840	6 834	1.1	289	4 311	42	91.4

Table 17.5. Comparison of SAT and CIP on invalid PIPEMULT properties.

		SAT				CIP			
Property	width	Clauses	Vars	Nodes	Time	Constrs	Vars	Nodes	Time
#6	5	4 800	1 747	113 585	2.8	289	821	0	0.0
	10	15 770	5 527	—	>7200.0	289	1 301	0	0.1
	15	33 360	11 511	—	>7200.0	289	1 772	0	0.1
	20	57 510	19 687	—	>7200.0	289	2 327	0	0.2
	25	88 290	30 069	—	>7200.0	289	2 806	0	0.3
	30	125 670	42 651	—	>7200.0	289	3 276	0	0.4
	35	169 630	57 429	—	>7200.0	289	3 832	0	0.6
	40	220 190	74 407	—	>7200.0	289	4 311	0	0.7
#7	5	4 800	1 747	241 341	8.3	289	821	0	0.1
	10	15 770	5 527	—	>7200.0	289	1 301	0	0.2
	15	33 360	11 511	—	>7200.0	289	1 772	0	0.5
	20	57 510	19 687	—	>7200.0	289	2 327	0	1.0
	25	88 290	30 069	—	>7200.0	289	2 806	0	1.8
	30	125 670	42 651	—	>7200.0	289	3 276	0	2.9
	35	169 630	57 429	—	>7200.0	289	3 832	0	4.8
	40	220 190	74 407	—	>7200.0	289	4 311	0	6.8
#9	5	4 800	1 747	885 261	34.5	287	900	0	0.1
	10	15 770	5 527	—	>7200.0	287	1 442	0	0.5
	15	33 360	11 511	—	>7200.0	287	1 976	0	1.6
	20	57 510	19 687	—	>7200.0	287	2 594	0	3.5
	25	88 290	30 069	—	>7200.0	287	3 135	0	7.1
	30	125 670	42 651	—	>7200.0	287	3 672	0	11.6
	35	169 630	57 429	—	>7200.0	287	4 288	0	19.8
	40	220 190	74 407	—	>7200.0	287	4 832	0	27.9
#10	5	4 800	1 747	91 909	2.2	287	900	0	0.1
	10	15 770	5 527	—	>7200.0	287	1 442	0	0.1
	15	33 360	11 511	—	>7200.0	287	1 976	0	0.2
	20	57 510	19 687	—	>7200.0	287	2 594	0	0.4
	25	88 290	30 069	—	>7200.0	287	3 135	0	0.6
	30	125 670	42 651	—	>7200.0	287	3 672	0	0.8
	35	169 630	57 429	—	>7200.0	287	4 288	0	1.1
	40	220 190	74 407	—	>7200.0	287	4 832	0	1.5

Table 17.6. Comparison of SAT and CIP on valid PipeMult properties.

the invalid properties, CIP needs up to two minutes to produce a counter-example, while MiniSat solves almost all of the instances within one second each. The results for the valid properties reveal a runtime difference of a completely different magnitude: while CIP can prove the infeasibility of the instances in presolving within a few seconds, the SAT solver fails to verify the properties within the time limit for register widths $\beta \geq 10$.

Tables 17.7 and 17.8 show the SAT/CIP comparison for a DSP/IIR filter core which we have obtained from the `opencores.org` website. In order to derive property checking instances from the circuit, OneSpin Solutions constructed a set of reasonable properties, which again includes invalid and valid properties. Within the two years of the Valse-XT project, the tool chain of OneSpin Solutions that converts the circuit and property specifications into CIP and SAT input has been under continuous development. At different dates during the project we obtained different versions of the instances, which are marked as 'A', 'B', and 'C' in the tables.

As before, we distinguish between the invalid and valid properties. The former are contained in Table 17.7, which shows the usual behavior: SAT solvers are faster to find counter-examples for invalid properties. The results on the valid Biquad properties shown in Table 17.8 are also supporting the general trend that the CIP approach is superior to SAT for proving the infeasibility of the instances. The most prominent advantage can be observed for the `g2_checkg2` property: these instances

	SAT				CIP			
Property	Clauses	Vars	Nodes	Time	Constrs	Vars	Nodes	Time
g3_checkreg1-A	3 595	1 270	89	0.0	154	1 070	6	1.2
g3_checkreg1-B	3 595	1 270	89	0.0	122	797	25	0.9
g3_checkreg1-C	3 643	1 286	192	0.0	121	795	25	0.9
g3_xtoxmdelay-A	841	447	1	0.0	820	2 833	37	0.5
g3_xtoxmdelay-B	838	445	1	0.0	136	898	31	0.1
g3_xtoxmdelay-C	838	445	1	0.0	135	896	31	0.1
g3_checkgfail-A	298 810	103 204	3 030	0.7	15 258	119 373	504	290.0
g3_checkgfail-B	168 025	58 924	8 528	0.6	3 646	24 381	236	29.2
g3_checkgfail-C	168 331	59 026	23 889	1.2	3 694	24 688	150	50.2

Table 17.7. Comparison of SAT and CIP on invalid BIQUAD properties.

can be solved with CIP in around 4 minutes while the SAT solver hits the time limit for all of the three versions.

Finally, we apply SAT and CIP techniques on a multiplication circuit. There are various possible bit level implementations of a multiplication operation. Two of them are the so-called *boothed* and *non-boothed* variants. Having both signed and unsigned multiplication, we end up with four different circuits, for which the results are shown in Table 17.9.

The property that has to be checked is that the outcome of the bit level implementation of the multiplication is always equal to the register level multiplication constraint. Thus, these instances are rather equivalence checking problems than property checking problems, since the task is to show the equivalence of the register transfer level representation and the chosen bit level implementation.

Since the MULTIPLIER circuits are designed on bit level, all of the structure of the multiplication is already missing. This suggests that almost all of the benefit of the CIP approach vanishes. The only structured constraint in the CIP instance that can be exploited is the single multiplication that represents the property. The results on the smaller sized instances seem to support this concern: SAT is faster in proving infeasibility than CIP for input registers of up to 9 bits. For the larger register widths, however, the situation reverses. With constraint integer programming, one can solve all of the instances within two hours, while the SAT solver fails on register widths $\beta \geq 11$ for the signed variants and for $\beta \geq 12$ for the unsigned circuits.

We conclude from our experiments that applying constraint integer programming

	SAT				CIP			
Property	Clauses	Vars	Nodes	Time	Constrs	Vars	Nodes	Time
g_checkgpre-A	46 357	16 990	463 851	22.2	2 174	13 359	1 163	14.2
g_checkgpre-B	46 916	17 171	1 109 042	57.6	1 616	10 745	2 086	12.3
g_checkgpre-C	46 918	17 172	625 296	29.1	1 694	11 046	1 886	15.3
g2_checkg2-A	52 305	18 978	—	>7200.0	2 293	14 486	15 793	213.9
g2_checkg2-B	52 864	19 159	—	>7200.0	1 714	11 567	21 346	204.8
g2_checkg2-C	52 866	19 160	—	>7200.0	1 792	11 868	23 517	257.6
g25_checkg25-A	—	—	0	0.0	4 569	32 604	699	29.7
g25_checkg25-B	56 988	20 283	55 112	2.4	2 616	18 636	2 643	22.4
g25_checkg25-C	56 994	20 286	54 643	2.5	2 731	19 345	2 632	24.2
g3_negres-A	1 745	619	0	0.0	1 359	7 425	0	0.7
g3_negres-B	1 744	617	0	0.0	79	656	0	0.0
g3_negres-C	1 758	623	0	0.0	82	658	0	0.0
gBIG_checkreg1-A	143 263	49 637	1 558 916	287.2	6 816	37 104	3 293	170.7
gBIG_checkreg1-B	113 729	39 615	1 275 235	157.3	3 095	17 628	46	7.0
gBIG_checkreg1-C	113 729	39 632	1 211 654	159.6	3 255	18 757	68	8.6

Table 17.8. Comparison of SAT and CIP on valid BIQUAD properties.

		SAT				CIP			
Layout	width	Clauses	Vars	Nodes	Time	Constrs	Vars	Nodes	Time
booth	2	175	66	18	0.0	107	143	8	0.0
signed	3	644	227	142	0.0	269	345	71	0.1
	4	1 116	389	549	0.0	350	464	527	0.9
	5	1 924	663	3 023	0.1	568	742	1 771	3.9
	6	2 687	922	12 689	0.4	675	900	7 586	21.3
	7	3 936	1 343	71 154	3.3	1 002	1 312	17 035	70.1
	8	5 002	1 703	340 372	21.0	1 139	1 514	61 116	318.7
	9	6 654	2 259	1 328 631	135.4	1 556	2 045	56 794	384.2
	10	8 023	2 720	5 877 637	935.1	1 723	2 292	116 383	904.1
	11	10 080	3 411	—	>7200.0	2 230	2 933	173 096	1756.2
	12	11 752	3 973	—	>7200.0	2 427	3 223	248 437	2883.7
	13	14 216	4 799	—	>7200.0	3 024	3 977	355 515	4995.9
	14	16 191	5 462	—	>7200.0	3 251	4 310	182 684	3377.9
booth	2	137	54	19	0.0	126	162	0	0.0
unsigned	3	401	146	102	0.0	209	277	35	0.1
	4	894	315	596	0.0	412	547	239	0.5
	5	1 363	476	2 593	0.1	525	698	1 743	3.5
	6	2 117	732	17 854	0.5	816	1 092	5 218	15.7
	7	2 787	960	66 761	2.5	963	1 279	14 939	51.7
	8	3 808	1 305	341 113	17.9	1 342	1 799	51 710	269.1
	9	4 679	1 600	1 433 711	102.9	1 523	2 024	151 270	911.3
	10	5 965	2 034	6 969 778	879.0	1 988	2 675	133 336	1047.6
	11	7 037	2 396	24 247 606	4360.4	2 203	2 925	231 890	2117.7
	12	8 590	2 919	—	>7200.0	2 752	3 706	206 625	2295.1
	13	9 863	3 348	—	>7200.0	3 001	3 980	343 227	4403.4
	14	11 685	3 960	—	>7200.0	3 636	4 897	421 494	7116.8
nonbooth	2	157	58	12	0.0	100	138	0	0.0
signed	3	430	155	70	0.0	186	274	12	0.1
	4	980	343	582	0.0	321	485	274	0.5
	5	1 701	588	2 396	0.1	496	763	1 311	2.8
	6	2 605	894	11 785	0.4	711	1 111	4 283	12.8
	7	3 698	1 263	71 370	3.4	966	1 532	8 493	31.2
	8	4 992	1 699	308 925	21.8	1 261	2 030	21 770	100.6
	9	6 481	2 200	1 317 390	134.1	1 596	2 612	44 695	265.9
	10	8 162	2 765	7 186 499	1344.1	1 971	3 273	76 626	569.8
	11	10 035	3 394	—	>7200.0	2 386	4 019	75 973	690.8
	12	12 091	4 084	—	>7200.0	2 841	4 853	159 132	1873.0
	13	14 336	4 837	—	>7200.0	3 336	5 778	153 857	1976.3
	14	16 773	5 654	—	>7200.0	3 871	6 797	263 266	4308.9
nonbooth	2	0	0	0	0.0	76	110	4	0.0
unsigned	3	280	105	49	0.0	167	252	4	0.1
	4	671	240	416	0.0	298	458	105	0.2
	5	1 179	414	2 288	0.0	469	732	675	1.4
	6	1 807	628	10 862	0.3	680	1 076	1 297	3.6
	7	2 567	886	55 718	1.8	931	1 493	6 315	22.4
	8	3 471	1 192	337 873	16.5	1 222	1 986	25 909	111.2
	9	4 507	1 542	1 212 498	83.1	1 553	2 564	39 668	214.0
	10	5 675	1 936	8 198 899	909.6	1 924	3 221	52 252	335.4
	11	6 975	2 374	28 863 978	5621.5	2 335	3 963	128 326	1040.1
	12	8 395	2 852	—	>7200.0	2 786	4 793	147 940	1507.5
	13	9 947	3 374	—	>7200.0	3 277	5 714	188 797	2347.7
	14	11 631	3 940	—	>7200.0	3 808	6 729	294 927	4500.2

Table 17.9. Comparison of SAT and CIP on Multiplier instances (all properties are valid).

		default			no term algebra			no irrelevance		
Property	width	Vars	Nodes	Time	Vars	Nodes	Time	Vars	Nodes	Time
neg_flag	5	395	45	0.8	395	45	0.7	541	46	1.4
	10	1 010	50	3.6	1 010	50	3.5	1 500	58	6.2
	15	2 027	64	11.6	2 027	64	10.9	3 127	57	17.8
	20	3 009	42	36.3	3 009	42	34.3	4 885	32	42.5
	25	4 424	107	81.8	4 424	107	78.2	7 787	61	84.9
	30	6 900	53	136.6	6 900	53	133.2	11 228	90	152.6
	35	10 214	35	218.4	10 214	35	210.2	15 788	83	267.4
	40	11 690	55	383.5	11 690	55	369.1	20 291	117	353.0
zero_flag	5	579	54	2.3	579	54	2.2	501	27	2.4
	10	312	28	0.6	312	28	0.6	1 131	8	7.5
	15	383	37	1.6	383	37	1.5	2 458	23	31.4
	20	489	26	4.0	489	26	3.3	4 367	14	87.1
	25	695	78	6.2	695	78	6.0	7 814	15	127.5
	30	794	73	10.7	794	73	10.3	11 136	46	235.4
	35	787	63	15.6	787	63	15.0	14 616	24	503.5
	40	20 699	185	379.7	20 699	185	373.4	20 036	277	423.7

Table 17.10. Evaluation of problem specific presolving techniques on valid ALU properties.

on the gate level representation of the circuit can indeed yield significant performance improvements for proving the validity of properties compared to the current state-of-the-art SAT approach. For invalid properties, however, SAT solvers are usually much faster in finding a counter-example. Therefore, we suggest to combine the two techniques, for example by first running a SAT solver for some limited time, and switching to CIP if one gets the impression that the property is valid.

17.2 Problem Specific Presolving

In this section we evaluate the contribution of the problem specific presolving methods to the overall success of the CIP approach, namely the *term algebra preprocessing* of Section 15.1 and the *irrelevance detection* of Section 15.2. In order to assess the impact of the two techniques, we compare the default parameter settings to settings in which the respective presolving method is disabled. Interesting comparison values are the sizes of the instances after presolving and the nodes and time needed to find a counter-example or to prove the validity of the property. In order to retain the structure of the function graph, see Section 13.2, satisfied and irrelevant constraints are only disabled instead of being completely removed from the presolved problem instances. Therefore, the effect of presolving can only be seen in the number of remaining variables, and we omit the number of constraints in the tables.

The results on the invalid properties and the valid `muls` property for the ALU circuit do not differ from the default settings if any of the problem specific presolving methods is disabled. For the `neg_flag` and `zero_flag` properties, however, some differences can be observed as shown in Table 17.10. The term algebra preprocessing does not find any reductions, such that its deactivation saves a few seconds. In contrast, the irrelevance detection can simplify the presolved model significantly, which usually leads to a reduction in the runtime. This is most prominent for the `zero_flag` property, for which the irrelevance detection can reduce the number of variables to a few hundreds, while there remain thousands of variables in the larger instances if irrelevance detection is disabled.

The results on the 40-bit version of the `zero_flag` property shed some light on the abnormal behavior in the default settings. We already explained in the previous

		default			no term algebra			no irrelevance		
Property	width	Vars	Nodes	Time	Vars	Nodes	Time	Vars	Nodes	Time
#7	5	—	0	0.0	317	41	0.1	—	0	0.0
	10	—	0	0.1	543	40	0.1	—	0	0.1
	15	—	0	0.1	767	41	0.1	—	0	0.1
	20	—	0	0.1	1 039	98	0.2	—	0	0.1
	25	—	0	0.1	1 264	161	0.2	—	0	0.1
	30	—	0	0.1	1 489	188	0.3	—	0	0.1
	35	—	0	0.1	1 760	462	0.7	—	0	0.1
	40	—	0	0.2	1 985	304	0.6	—	0	0.2
#9	5	—	0	0.0	333	82	0.1	—	0	0.0
	10	—	0	0.1	569	72	0.1	—	0	0.0
	15	—	0	0.1	806	98	0.1	—	0	0.1
	20	—	0	0.1	1 085	408	0.4	—	0	0.1
	25	—	0	0.1	1 320	231	0.3	—	0	0.1
	30	—	0	0.1	1 555	289	0.4	—	0	0.1
	35	—	0	0.2	1 836	643	1.0	—	0	0.2
	40	—	0	0.2	2 071	1 121	1.8	—	0	0.2

Table 17.11. Evaluation of problem specific presolving techniques on valid PipeAdder properties.

section that this is due to very "bad luck" in probing: for some reason, probing does not find the necessary fixings that enable the irrelevance decection to discard large parts of the circuit. The behavior on all other `zero_flag` instances, including the ones for up to 39 bits which are not shown in the table, is similar to the 5 to 35-bit variants.

For the invalid PipeAdder circuit, there are almost no differences in the results if the presolving techniques are disabled. The same holds for the valid properties `#6` and `#10`. As can be seen in Table 17.11, the term algebra preprocessing has, however, a noticeable impact on properties `#7` and `#9`, since it is able to prove their validity during presolving. Without term algebra preprocessing, we have to revert to branching. Nevertheless, all of the instances can still be solved in less than 2 seconds.

The situation changes for the PipeMult circuit. For the invalid properties shown in Table 17.12, irrelevance detection is able to remove about 10 % of the variables from the problem instances, which improves the performance for property `#4`, while it does not seem to have a clear impact on the other two properties. The term algebra preprocessing is not able to find reductions on the invalid properties.

In contrast, term algebra preprocessing is able to prove the validity of the valid PipeMult properties, but without this technique, properties `#7` and `#9` become intractable for register widths $\beta \geq 15$, as can be seen in Table 17.13. Note, however, that CIP still performs better than SAT solvers: the latter cannot even solve the 10-bit instances of these properties within the time limit.

Table 17.14 shows the impact of the two presolving techniques on the invalid properties of the Biquad circuit. The term algebra preprocessing is able to remove a very small fraction of the variables in the `g3_checkgfail` instances, but this reduction does not compensate for the runtime costs of the presolving method. In contrast, irrelevance detection removes about half of the variables, which leads to a considerable performance improvement. For the `g3_checkreg1` and `g3_xtoxmdelay` properties, there is almost no difference in the presolved model sizes and the runtimes. Therefore, they are not included in the table.

The valid `g3_negres` property of the Biquad circuit is again unaffected by the disabling of the two presolving methods. The differences for the remaining valid properties are shown in Table 17.15. As can be seen, term algebra preprocessing

		default			no term algebra			no irrelevance		
Property	width	Vars	Nodes	Time	Vars	Nodes	Time	Vars	Nodes	Time
#4	5	553	11	0.5	553	11	0.5	498	16	0.6
	10	1 493	21	2.8	1 493	21	2.7	1 591	17	2.9
	15	2 879	42	6.2	2 879	42	6.3	3 187	42	7.8
	20	4 806	28	11.1	4 806	28	11.0	5 435	29	14.4
	25	7 241	61	23.1	7 241	61	23.1	8 370	39	23.8
	30	10 058	38	33.0	10 058	38	32.9	11 649	38	40.4
	35	13 727	120	6.3	13 727	120	6.3	15 665	21	50.4
	40	17 534	90	7.3	17 534	90	7.4	20 257	21	66.1
#5	5	937	16	1.1	937	16	1.1	920	12	1.0
	10	2 947	25	5.4	2 947	25	5.5	3 083	25	5.7
	15	6 092	33	15.0	6 092	33	15.0	6 375	33	16.4
	20	10 375	42	32.5	10 375	42	32.5	10 950	168	30.7
	25	15 828	85	52.0	15 828	85	52.2	16 899	42	52.2
	30	22 058	75	92.0	22 058	75	92.0	23 613	249	90.5
	35	30 378	44	55.1	30 378	44	55.5	32 087	42	136.6
	40	38 503	422	125.2	38 503	422	125.7	41 918	426	94.5
#8	5	789	51	1.2	789	51	1.2	814	66	1.3
	10	2 593	133	8.1	2 593	133	8.1	2 757	143	6.7
	15	5 481	55	19.8	5 481	55	19.9	5 847	55	19.7
	20	9 399	154	43.1	9 399	154	43.3	10 025	180	45.9
	25	14 457	279	45.5	14 457	279	45.5	15 468	69	70.1
	30	19 955	73	89.9	19 955	73	91.0	21 708	72	77.6
	35	27 088	92	118.9	27 088	92	119.3	29 375	56	109.4
	40	34 849	42	91.4	34 849	42	92.2	37 843	42	123.1

Table 17.12. Evaluation of problem specific presolving techniques on invalid PIPEMULT properties.

		default			no term algebra			no irrelevance		
Property	width	Vars	Nodes	Time	Vars	Nodes	Time	Vars	Nodes	Time
#7	5	—	0	0.1	833	2 630	8.2	—	0	0.1
	10	—	0	0.2	2 713	337 950	2827.1	—	0	0.2
	15	—	0	0.5	5 379	>272 513	>7200.0	—	0	0.6
	20	—	0	1.0	9 513	>212 561	>7200.0	—	0	1.2
	25	—	0	1.8	14 368	>93 410	>7200.0	—	0	2.2
	30	—	0	2.9	20 351	>111 827	>7200.0	—	0	3.5
	35	—	0	4.8	27 666	>66 407	>7200.0	—	0	5.8
	40	—	0	6.8	35 436	>38 825	>7200.0	—	0	8.4
#9	5	—	0	0.1	1 628	16 069	92.8	—	0	0.1
	10	—	0	0.5	4 983	77 120	1258.9	—	0	0.6
	15	—	0	1.6	10 740	>138 788	>7200.0	—	0	1.8
	20	—	0	3.5	17 855	>101 771	>7200.0	—	0	3.9
	25	—	0	7.1	27 755	>85 969	>7200.0	—	0	7.9
	30	—	0	11.6	40 295	>75 138	>7200.0	—	0	12.9
	35	—	0	19.8	⋆	⋆	⋆	—	0	21.7
	40	—	0	27.9	69 912	>34 364	>7200.0	—	0	31.2

Table 17.13. Evaluation of problem specific presolving techniques on valid PIPEMULT properties. Note that the run for property #9 with 35-bit registers in the "no bitarith term" setting failed due to a bug in the solver which caused a segmentation fault.

	default			no term algebra			no irrelevance		
Property	Vars	Nodes	Time	Vars	Nodes	Time	Vars	Nodes	Time
g3_checkgfail-A	33 775	504	290.0	35 753	55	268.0	71 640	4 495	1705.7
g3_checkgfail-B	12 365	236	29.2	12 569	376	16.9	23 564	136	205.7
g3_checkgfail-C	12 382	150	50.2	12 517	149	45.6	23 567	189	208.5

Table 17.14. Evaluation of problem specific presolving techniques on invalid BIQUAD properties.

	default			no term algebra			no irrelevance		
Property	Vars	Nodes	Time	Vars	Nodes	Time	Vars	Nodes	Time
g_checkgpre-A	4 819	1 163	14.2	4 821	1 409	14.7	9 419	3 314	83.9
g_checkgpre-B	3 400	2 086	12.3	3 410	1 718	11.2	7 660	2 409	57.4
g_checkgpre-C	3 698	1 886	15.3	3 787	1 260	9.1	8 264	1 375	46.9
g2_checkg2-A	7 285	15 793	213.9	7 296	15 109	216.5	⋆	⋆	⋆
g2_checkg2-B	4 722	21 346	204.8	4 753	11 207	116.4	8 351	28 713	495.6
g2_checkg2-C	4 997	23 517	257.6	5 072	29 710	275.8	8 957	13 411	242.9
g25_checkg25-A	9 410	699	29.7	9 502	835	29.3	20 974	2 715	159.1
g25_checkg25-B	5 187	2 643	22.4	5 238	1 279	13.6	8 834	1 981	56.1
g25_checkg25-C	5 313	2 632	24.2	5 376	850	11.8	8 908	2 422	63.8
gBIG_checkreg1-A	13 264	3 293	170.7	13 393	4 728	181.9	31 683	446	289.6
gBIG_checkreg1-B	5 837	46	7.0	5 929	44	6.4	14 511	821	122.1
gBIG_checkreg1-C	6 271	68	8.6	6 418	47	7.5	15 113	1 045	135.6

Table 17.15. Evaluation of problem specific presolving techniques on valid Biquad properties. Note that the run for property `g2_checkg2-A` in the "no irrelevance" setting failed due to a bug in the solver which caused a segmentation fault.

exhibits the same behavior as for the invalid properties: it produces only a very small number of variable reductions and is therefore not worth its runtime costs. On the other hand, irrelevance detection is again a very useful technique. It removes around half of the variables and improves the overall performance significantly.

As a final test to evaluate the impact of the two problem specific presolving techniques, we performed benchmarks on the Multiplier test set. It turned out that the differences in the presolved model sizes and in the performance are very marginal. Therefore, we omit the tables and the discussion of the results.

17.3 Probing

Probing is a generally applicable presolving technique that tentatively fixes the binary variables of the CIP problem instance to zero and one and evaluates the deductions that are produced by subsequent domain propagation. See Section 10.6 for a more detailed explanation.

Since some of the domain propagation algorithms included in the chip verification constraint handlers are quite expensive, probing itself is a rather time-consuming method. In this section, we want to investigate whether the expenses to apply probing pay off with respect to the overall runtime. We compare the default settings in which probing is activated to an alternative parameter set that disables probing.

For the invalid ALU properties, there are almost no differences. With and without probing, the instances can be solved in virtually no time. The same holds for the valid `muls` property of the circuit. In contrast, there are significant differences for the `neg_flag` and `zero_flag` properties, as can be seen in Table 17.16: although probing, in combination with irrelevance detection, can significantly reduce the sizes of the problem instances, it is way too expensive to yield an improvement in the overall runtime performance. Without probing, the instances, in particular the `neg_flag` instances, are solved in a fraction of the time that is needed for probing.

Compared to SAT, the results on the `neg_flag` properties emphasize the benefits of the CIP approach even more than the numbers presented in Section 17.1: our CIP solver without probing solves the 40-bit instance in less than 4 seconds, while the SAT solver cannot even solve the 15-bit instance within the time limit of 2 hours.

The benchmark tests on the PipeAdder circuit do not show a significant difference if probing is disabled. There are no differences on the valid properties, and

Property	width	default Vars	default Nodes	default Time	no probing Vars	no probing Nodes	no probing Time
neg_flag	5	395	45	0.8	751	37	0.1
	10	1 010	50	3.6	2 007	55	0.2
	15	2 027	64	11.6	3 956	40	0.2
	20	3 009	42	36.3	6 638	39	0.5
	25	4 424	107	81.8	10 013	49	0.8
	30	6 900	53	136.6	13 955	79	1.7
	35	10 214	35	218.4	18 738	69	2.3
	40	11 690	55	383.5	24 147	64	3.9
zero_flag	5	579	54	2.3	792	54	0.1
	10	312	28	0.6	2 084	173	0.5
	15	383	37	1.6	4 068	66	0.3
	20	489	26	4.0	6 795	256	2.0
	25	695	78	6.2	10 206	94	1.4
	30	794	73	10.7	14 183	113	2.6
	35	787	63	15.6	19 011	136	3.5
	40	20 699	185	379.7	24 456	211	9.3

Table 17.16. Impact of probing on valid ALU properties.

also without probing, counter-examples to the invalid properties can be found in less than a second. The tables with the benchmark results are omitted.

For the PIPEMULT test set, the situation is different, as can be seen in Tables 17.17 and 17.18. In both cases, one can observe the same behavior as for the ALU instances: probing turns out to be a waste of time, although the number of variables for the invalid properties can be slightly reduced in most of the cases.

Tables 17.19 and 17.20 show the results on the BIQUAD circuits. There are no differences associated to probing on the valid `g3_negres` property, which is why it is missing in Table 17.20.

Our previous poor impression of probing carries forward to the invalid BIQUAD properties: Table 17.19 clearly shows that probing deteriorates the performance. On the other hand, the results of Table 17.20 are much more promising. In particular for the `g_checkpre` instances, the reduction in the problem size obtained by probing decreases the runtime approximately by 60 to 80 %. Another large performance improvement can be observed for `gBIG_checkreg1-A`, but there are other instances, for example `g2_checkg2-B` and `g25_checkg25-B`, that show a deterioration.

Our final benchmarks on the MULTIPLIER circuit conveys a better impression of probing. As can be seen in Table 17.21, disabling probing yields almost consistently an increase in the runtime. In particular, three of the instances cannot be solved anymore within the time limit. An explanation is that due to the large node counts, the time spent on probing becomes a relatively small portion of the overall time. Additionally, these instances contain only one constraint, namely the multiplication in the property part of the model, that features an expensive domain propagation algorithm. Therefore, probing is, compared to the size of the instance, relatively cheap.

We conclude from the benchmark results of this section that it may be better to disable probing in our CIP based chip design verification algorithm. The results on the valid BIQUAD and MULTIPLIER instances, however, are in favor of this presolving technique. Overall, a better strategy is needed to control the effort which is spent on probing.

Property	width	default Vars	default Nodes	default Time	no probing Vars	no probing Nodes	no probing Time
#1	5	849	6	0.3	959	34	0.1
	10	2 050	28	0.6	2 174	37	0.1
	15	3 745	6	0.7	3 854	56	0.3
	20	6 085	6	1.6	6 218	56	0.4
	25	8 905	6	2.3	9 058	54	0.6
	30	12 088	6	4.7	12 288	43	0.7
	35	16 050	6	7.7	16 276	76	1.6
	40	20 495	7	10.5	20 741	53	1.3
#2	5	869	15	0.2	922	23	0.1
	10	2 050	21	2.2	2 137	34	0.1
	15	3 746	45	1.2	3 817	29	0.2
	20	6 104	42	2.8	6 181	59	0.5
	25	8 946	39	3.5	9 021	61	0.7
	30	12 178	32	4.6	12 251	52	0.7
	35	16 167	82	7.7	16 239	52	0.9
	40	20 629	43	9.4	20 704	52	1.1
#3	5	1 029	15	0.7	1 123	18	0.0
	10	2 708	15	3.0	2 801	35	0.1
	15	5 135	33	2.6	5 225	22	0.2
	20	8 481	20	9.3	8 561	59	0.6
	25	12 622	20	13.5	12 703	53	0.7
	30	17 298	36	20.4	17 392	53	0.9
	35	23 123	45	21.8	23 217	64	1.6
	40	29 570	46	28.8	29 665	25	1.0
#4	5	553	11	0.5	579	20	0.0
	10	1 493	21	2.8	1 566	12	0.1
	15	2 879	42	6.2	3 019	33	0.2
	20	4 806	28	11.1	4 984	54	0.5
	25	7 241	61	23.1	7 460	73	0.9
	30	10 058	38	33.0	10 249	38	0.7
	35	13 727	120	6.3	13 727	34	0.8
	40	17 534	90	7.3	17 534	33	1.0
#5	5	937	16	1.1	1 044	26	0.1
	10	2 947	25	5.4	3 134	31	0.2
	15	6 092	33	15.0	6 346	40	0.4
	20	10 375	42	32.5	10 646	34	0.6
	25	15 828	85	52.0	16 228	33	0.8
	30	22 058	75	92.0	22 515	72	2.2
	35	30 378	44	55.1	30 382	33	1.4
	40	38 503	422	125.2	38 982	33	1.8
#8	5	789	51	1.2	975	21	0.1
	10	2 593	133	8.1	2 888	132	0.6
	15	5 481	55	19.8	5 829	21	0.3
	20	9 399	154	43.1	9 736	61	1.0
	25	14 457	279	45.5	14 816	66	1.5
	30	19 955	73	89.9	20 523	49	1.7
	35	27 088	92	118.9	27 671	25	1.3
	40	34 849	42	91.4	35 444	25	1.7

Table 17.17. Impact of probing on invalid PIPEMULT properties.

Property	width	default Vars	default Nodes	default Time	no probing Vars	no probing Nodes	no probing Time
#7	5	—	0	0.1	—	0	0.0
	10	—	0	0.2	—	0	0.1
	15	—	0	0.5	—	0	0.1
	20	—	0	1.0	—	0	0.2
	25	—	0	1.8	—	0	0.3
	30	—	0	2.9	—	0	0.5
	35	—	0	4.8	—	0	0.6
	40	—	0	6.8	—	0	0.8
#9	5	—	0	0.1	—	0	0.1
	10	—	0	0.5	—	0	0.1
	15	—	0	1.6	—	0	0.2
	20	—	0	3.5	—	0	0.4
	25	—	0	7.1	—	0	0.7
	30	—	0	11.6	—	0	0.9
	35	—	0	19.8	—	0	1.2
	40	—	0	27.9	—	0	1.6

Table 17.18. Impact of probing on valid PIPEMULT properties.

Property	default Vars	default Nodes	default Time	no probing Vars	no probing Nodes	no probing Time
g3_checkreg1-A	1 164	6	1.2	1 185	8	0.1
g3_checkreg1-B	661	25	0.9	679	19	0.0
g3_checkreg1-C	661	25	0.9	679	22	0.0
g3_xtoxmdelay-A	1 243	37	0.5	1 253	38	0.4
g3_xtoxmdelay-B	248	31	0.1	253	17	0.0
g3_xtoxmdelay-C	248	31	0.1	253	15	0.0
g3_checkgfail-A	33 775	504	290.0	34 311	66	95.4
g3_checkgfail-B	12 365	236	29.2	12 595	79	4.9
g3_checkgfail-C	12 382	150	50.2	12 631	72	5.0

Table 17.19. Impact of probing on invalid BIQUAD properties.

Property	default Vars	default Nodes	default Time	no probing Vars	no probing Nodes	no probing Time
g_checkgpre-A	4 819	1 163	14.2	5 553	5 952	60.5
g_checkgpre-B	3 400	2 086	12.3	4 115	7 383	57.9
g_checkgpre-C	3 698	1 886	15.3	4 413	6 029	45.3
g2_checkg2-A	7 285	15 793	213.9	7 724	17 004	216.5
g2_checkg2-B	4 722	21 346	204.8	5 119	7 320	52.9
g2_checkg2-C	4 997	23 517	257.6	5 397	29 385	285.7
g25_checkg25-A	9 410	699	29.7	9 709	1 043	22.0
g25_checkg25-B	5 187	2 643	22.4	5 361	894	7.8
g25_checkg25-C	5 313	2 632	24.2	5 487	2 515	18.5
gBIG_checkreg1-A	13 264	3 293	170.7	14 919	13 372	418.0
gBIG_checkreg1-B	5 837	46	7.0	6 083	68	2.8
gBIG_checkreg1-C	6 271	68	8.6	6 516	105	3.2

Table 17.20. Impact of probing on valid BIQUAD properties.

Property	width	default Vars	default Nodes	default Time	no probing Vars	no probing Nodes	no probing Time
booth	2	37	8	0.0	72	41	0.0
signed	3	126	71	0.1	177	180	0.2
	4	202	527	0.9	265	789	1.1
	5	352	1 771	3.9	446	2 378	5.0
	6	460	7 586	21.3	581	7 779	23.9
	7	669	17 035	70.1	832	19 244	76.5
	8	806	61 116	318.7	1 013	43 452	230.6
	9	1 224	56 794	384.2	1 365	64 564	431.5
	10	1 432	116 383	904.1	1 596	248 085	1898.2
	11	1 799	173 096	1756.2	1 999	268 999	2932.8
	12	2 045	248 437	2883.7	2 276	402 279	4444.0
	13	2 502	355 515	4995.9	2 780	402 389	6191.6
	14	2 790	182 684	3377.9	3 105	309 863	5270.3
booth	2	—	0	0.0	67	31	0.0
unsigned	3	101	35	0.1	129	149	0.1
	4	224	239	0.5	260	749	0.9
	5	336	1 743	3.5	356	1 519	2.6
	6	516	5 218	15.7	552	8 864	26.4
	7	651	14 939	51.7	683	14 904	52.0
	8	886	51 710	269.1	942	56 758	275.3
	9	1 075	151 270	911.3	1 125	171 556	995.5
	10	1 376	133 336	1047.6	1 450	140 821	1113.8
	11	1 582	231 890	2117.7	1 652	171 404	1550.8
	12	1 941	206 625	2295.1	2 041	323 834	3630.9
	13	2 179	343 227	4403.4	2 274	>532 643	>7200.0
	14	2 601	421 494	7116.8	2 732	>426 186	>7200.0
nonbooth	2	—	0	0.0	70	23	0.0
signed	3	63	12	0.1	144	100	0.1
	4	174	274	0.5	259	457	0.5
	5	296	1 311	2.8	421	2 564	4.5
	6	441	4 283	12.8	610	6 081	15.0
	7	609	8 493	31.2	835	26 652	95.3
	8	802	21 770	100.6	1 096	53 957	240.7
	9	1 172	44 695	265.9	1 420	106 346	625.1
	10	1 455	76 626	569.8	1 757	115 635	893.8
	11	1 767	75 973	690.8	2 130	141 005	1222.8
	12	2 107	159 132	1873.0	2 539	230 977	2585.3
	13	2 503	153 857	1976.3	3 011	393 617	5236.1
	14	2 903	263 266	4308.9	3 494	405 893	6706.1
nonbooth	2	28	4	0.0	47	12	0.0
unsigned	3	40	4	0.1	114	98	0.1
	4	160	105	0.2	217	298	0.3
	5	294	675	1.4	344	1 509	2.7
	6	439	1 297	3.6	510	3 885	9.0
	7	601	6 315	22.4	696	21 230	75.8
	8	799	25 909	111.2	925	82 398	423.5
	9	1 024	39 668	214.0	1 187	117 653	601.7
	10	1 291	52 252	335.4	1 482	152 011	1004.9
	11	1 557	128 326	1040.1	1 787	162 844	1333.7
	12	1 871	147 940	1507.5	2 146	436 984	4105.9
	13	2 183	188 797	2347.7	2 506	436 908	4834.4
	14	2 558	294 927	4500.2	2 934	>517 781	>7200.0

Table 17.21. Impact of probing on verifying the MULTIPLIER circuit.

		default		no conflict analysis	
Property	width	Nodes	Time	Nodes	Time
neg_flag	5	45	0.8	309	0.9
	10	50	3.6	619	4.1
	15	64	11.6	597	11.7
	20	42	36.3	939	36.9
	25	107	81.8	573	80.9
	30	53	136.6	885	139.0
	35	35	218.4	1665	225.5
	40	55	383.5	1991	387.3
zero_flag	5	54	2.3	141	2.3
	10	28	0.6	45	0.7
	15	37	1.6	83	1.6
	20	26	4.0	57	3.3
	25	78	6.2	185	6.2
	30	73	10.7	197	10.6
	35	63	15.6	129	15.1
	40	185	379.7	2715	421.4

Table 17.22. Impact of conflict analysis on valid ALU properties.

17.4 Conflict Analysis

Conflict analysis is a technique to analyze infeasible subproblems that are encountered during the branch-and-bound search. It can produce additional valid constraints that can be used later in the solving process to prune the search tree. Conflict analysis has originated in the SAT community and is particularly suited for proving the infeasibility of a problem instance. Chapter 11 explains how conflict analysis can be generalized to mixed integer programming, which easily extends to constraint integer programming.

In order to use conflict analysis for arbitrary constraints, each constraint handler has to implement a so-called *reverse propagation* method. This method has to provide the "reasons" for a deduction that the constraint handler's domain propagation method has produced earlier during the search, compare Section 11.1. Given the rather complex domain propagation algorithms of our chip verification constraint handlers, this is not an easy task. Nevertheless, we equipped the constraint handlers with the necessary algorithms such that almost all of the possible propagations can be "reverse propagated".

In this section, we evaluate the impact of conflict analysis on solving property checking problems. Since counter-examples to the invalid ALU properties can be found in at most 4 branch-and-bound nodes, conflict analysis does not have any influence on these instances. The same holds for all instances that can already be solved in the presolving stage, which includes the `muls` property of the ALU circuit. The results for the remaining two valid ALU instances are shown in Table 17.22. It turns out that the number of branching nodes needed to prove the infeasibility of the instances gets significantly larger if conflict analysis is disabled. This does not, however, increase the runtime by a significant amount. In fact, almost all of the runtime is spent in presolving, such that the time that can be saved by conflict analysis is very limited.

Similar to the invalid ALU instances, finding counter-examples for the invalid PIPEADDER and PIPEMULT properties is so easy that conflict analysis cannot yield significant performance improvements. The correctness of the valid properties of these circuits is already shown in presolving. Therefore, we skip the tables with the PIPEADDER and PIPEMULT results.

	default		no conflict analysis	
Property	Nodes	Time	Nodes	Time
g_checkgpre-A	1 163	14.2	>2023338	>7200.0
g_checkgpre-B	2 086	12.3	>2864290	>7200.0
g_checkgpre-C	1 886	15.3	1558967	4398.1
g2_checkg2-A	15 793	213.9	>1097670	>7200.0
g2_checkg2-B	21 346	204.8	>1842786	>7200.0
g2_checkg2-C	23 517	257.6	>1808079	>7200.0
g25_checkg25-A	699	29.7	109175	662.5
g25_checkg25-B	2 643	22.4	34397	143.4
g25_checkg25-C	2 632	24.2	39845	162.7
gBIG_checkreg1-A	3 293	170.7	>486990	>7200.0
gBIG_checkreg1-B	46	7.0	653	9.7
gBIG_checkreg1-C	68	8.6	465	10.3

Table 17.23. Impact of conflict analysis on valid Biquad properties.

For the invalid Biquad properties, conflict analysis yields only minor improvements. The only noteworthy result is the one on the `g3_checkgfail-A` instance: in the default settings, a counter-example can be found in 504 nodes and 290.0 seconds. Disabling conflict analysis deteriorates the performance, such that 2612 nodes and 357.4 seconds are needed.

The results for the valid Biquad properties are shown in Table 17.23. Here, the benefit of including conflict analysis becomes clearly evident: using the default settings, all of the instances can be solved in a few minutes while six of them become intractable if conflict analysis is disabled.

A similar conclusion can be derived from the results on the Multiplier circuit, which are given in Table 17.24. While in default settings all instances can be solved within two hours, without conflict analysis the solver hits the time limit if the width of the input registers becomes larger or equal to $\beta = 10$. Note that for both, the valid Biquad properties and the Multiplier circuit, there is not a single instance that can be solved without conflict analysis in less time or with a fewer number of branching nodes than in the default settings.

We conclude from our experiments, that conflict analysis is a very important tool to prove the validity of properties for more complex circuits. Having already produced a noticeable performance improvement for mixed integer programming, see the computational results of Section 11.3, conflict analysis is an indispensable ingredient of our CIP based chip design verification solver.

		default		no conflict analysis	
Property	width	Nodes	Time	Nodes	Time
booth	2	8	0.0	9	0.0
signed	3	71	0.1	83	0.1
	4	527	0.9	1 603	1.4
	5	1 771	3.9	5 581	6.1
	6	7 586	21.3	23 717	34.1
	7	17 035	70.1	101 119	181.3
	8	61 116	318.7	499 117	1129.7
	9	56 794	384.2	1 435 175	4322.2
	10	116 383	904.1	>2 080 575	>7200.0
	11	173 096	1756.2	>1 712 966	>7200.0
	12	248 437	2883.7	>1 298 920	>7200.0
	13	355 515	4995.9	>1 146 821	>7200.0
	14	182 684	3377.9	> 729 553	>7200.0
booth	2	0	0.0	0	0.0
unsigned	3	35	0.1	55	0.1
	4	239	0.5	477	0.6
	5	1 743	3.5	5 821	6.2
	6	5 218	15.7	21 389	31.1
	7	14 939	51.7	96 173	151.2
	8	51 710	269.1	384 581	858.5
	9	151 270	911.3	1 811 671	4469.6
	10	133 336	1047.6	>2 005 291	>7200.0
	11	231 890	2117.7	>1 964 676	>7200.0
	12	206 625	2295.1	>1 266 363	>7200.0
	13	343 227	4403.4	>1 232 579	>7200.0
	14	421 494	7116.8	> 834 663	>7200.0
nonbooth	2	0	0.0	0	0.0
signed	3	12	0.1	25	0.1
	4	274	0.5	563	0.6
	5	1 311	2.8	3 273	3.6
	6	4 283	12.8	16 951	22.8
	7	8 493	31.2	74 541	124.5
	8	21 770	100.6	298 209	640.3
	9	44 695	265.9	1 044 537	3014.9
	10	76 626	569.8	>2 048 638	>7200.0
	11	75 973	690.8	>1 491 337	>7200.0
	12	159 132	1873.0	>1 075 690	>7200.0
	13	153 857	1976.3	>1 026 471	>7200.0
	14	263 266	4308.9	> 675 644	>7200.0
nonbooth	2	4	0.0	5	0.0
unsigned	3	4	0.1	5	0.1
	4	105	0.2	237	0.3
	5	675	1.4	2 535	2.5
	6	1 297	3.6	12 341	15.7
	7	6 315	22.4	67 581	96.6
	8	25 909	111.2	247 323	459.2
	9	39 668	214.0	1 082 421	2691.3
	10	52 252	335.4	>2 421 634	>7200.0
	11	128 326	1040.1	>1 776 217	>7200.0
	12	147 940	1507.5	>1 348 860	>7200.0
	13	188 797	2347.7	>1 074 452	>7200.0
	14	294 927	4500.2	> 937 928	>7200.0

Table 17.24. Impact of conflict analysis on verifying the MULTIPLIER circuit.

Appendix A

Computational Environment

In the following we characterize the computational infrastructure and the environment under which our computational experiments have been conducted. Afterwards, we describe the various test sets on which the computations have been performed.

A.1 Computational Infrastructure

All benchmark runs have been executed on Dell Precision 370 desktop PCs with 2 GB RAM and a 3.8 GHz Intel Pentium 4 CPU with 1 MB second-level cache. They have been distributed to eight equivalent machines. We used Suse Linux 10.2 as operating system and Gcc 4.1.2 with the `-O3` optimizer option to compile the source code.

All timings are reported in CPU seconds. As time measurements, in particular for hyperthreading CPUs, are very dependent on the CPU load caused by other processes running in the background, the author did his best to run the tests only on unloaded machines and to rerun suspicious benchmarks. Small variations in the runtime, however, cannot be avoided. Our experience for the reported mean values is that differences of up to 2 % in the average runtime can be attributed to variations in the timing.

For the MIP benchmarks, we used a time limit of one hour, while we allowed two hour runs on the chip verification test sets. If the time limit was hit, we treat the time limit and the current number of processed nodes as the resulting benchmark values. Thus, we pretend that the instance was solved at this moment. Note that this provides an advantage to the runs in which the time limit was hit, since the accounted values are smaller than the actual time and number of nodes needed to solve the respective instances. On the other hand, it has the benefit that the influence on the overall statistics due to a failure on a single instance is limited.

We imposed a memory limit of 1.5 GB in order to avoid swapping. If the memory limit was hit, we treat the run as if it hit the time limit. The accounted number of nodes is scaled accordingly, as if the nodes per time ratio stayed constant throughout the solving process. An excess of the memory limit happened only very infrequently in our computations. In total, only 180 out of the 28060 MIP runs have been aborted due to the memory limit, most of them for the mik test set, in particular with *random branching*, *most infeasible branching*, *least infeasible branching*, and *inference branching*. All of the chip verification runs stayed within the memory limit.

We used two unofficial versions of SCIP in the benchmark runs that the author will make available on request. For the MIP benchmarks, we employed SCIP 0.90f, while the chip verification benchmarks have been conducted using SCIP 0.90i as CIP framework. SCIP 0.90i is the more recent version of the two. It is very similar to version 1.0, which will be published shortly after this thesis. We linked SCIP to Cplex 10.0.1 [118] in order to solve the underlying LP relaxations.

test set	type	size	problem class	ref	origin
MIPLIB	mixed	30	mixed	[6]	http://miplib.zib.de
CORAL	mixed	38	mixed	[145]	http://coral.ie.lehigh.edu/mip-instances/
MILP	mixed	37	mixed		http://plato.asu.edu/ftp/milp/
ENLIGHT	IP	7	combinatorial game		http://miplib.zib.de/contrib/AdrianZymolka/
ALU	IP	25	infeasible chip verification	[1]	http://miplib.zib.de/contrib/ALU/
FCTP	MBP	16	fixed charge transportation	[106]	http://plato.asu.edu/ftp/fctp/
ACC	BP	7	sports scheduling	[173]	http://www.ps.uni-sb.de/~walser/acc/acc.html
FC	MBP	20	fixed charge netfork flow	[22]	http://www.ieor.berkeley.edu/~atamturk/data/
ARCSET	IP/MIP	23	capacitated network design	[25]	http://www.ieor.berkeley.edu/~atamturk/data/
MIK	MIP	41	mixed integer knapsack	[23]	http://www.ieor.berkeley.edu/~atamturk/data/

Table A.1. Mixed integer programming test sets.

For reasons of comparison, we also solved the MIP instances with CPLEX 10.0.1 as stand-alone MIP solver. We used default settings, except that we set the gap dependent abort criteria to `mipgap` $= 0$ and `absmipgap` $= 10^{-9}$, which are the corresponding values in SCIP.

A.2 Mixed Integer Programming Test Set

In order to evaluate the impact of the various algorithms related to MIP solving by computational experiments, we collected several publicly available MIP instances from the web. Table A.1 gives an overview of these test sets.

Overall, we collected 575 instances, from which we selected subsets of the sizes given in the table using the following procedure:

1. First, we ran CPLEX 10.0.1 and SCIP 0.82d with and without conflict analysis on all of the instances. If none of the two solvers could solve the instance within one hour CPU time, we discarded the instance.

2. As the CORAL and MILP test set share some instances, we removed the duplicates from CORAL.

3. From the remaining CORAL test set, we removed instances that can be solved within 10 seconds by both, CPLEX and SCIP 0.82d in default settings. Then, we sorted the instances by non-decreasing CPLEX time and removed every second instance.

4. From the ALU test set, we removed the `_3`, `_4`, `_5`, and `_9` instances since they are trivial. Furthermore, we only used the instances that model circuits with register widths of at most 8 bits, because the instances for larger register widths are too difficult in terms of numerics.

5. From the MIK test set, we removed instances that can be solved within 10 seconds by both, CPLEX and SCIP 0.82d in default settings.

This procedure left a total of 244 instances with the sizes of the individual test sets shown in Table A.1.

A.3 Computing Averages

Since we consider a large number of MIP instances in our benchmark tests, it is unpractical to show the results for all of the individual instances. Additionally, in order to compare different solvers or parameter settings, it is convenient to subsume the timings and node counts for the instances of a single test set by calculating an average value. One has to keep in mind, however, that it is important how the average values are defined.

Let $v_1, \ldots, v_k \in \mathbb{R}_{\geq 0}$ be non-negative values, for example the times or node counts for the individual instances of a test set. We consider three different types of averages: the geometric mean

$$\gamma(v_1, \ldots, v_k) = \Big(\prod_{i=1}^{k} \max\{v_i, 1\} \Big)^{\frac{1}{k}},$$

the shifted geometric mean

$$\gamma_s(v_1, \ldots, v_k) = \Big(\prod_{i=1}^{k} \max\{v_i + s, 1\} \Big)^{\frac{1}{k}} - s$$

with $s \in \mathbb{R}_{\geq 0}$, and the arithmetic mean

$$\varnothing(v_1, \ldots, v_k) = \frac{1}{k} \sum_{i=1}^{k} v_i.$$

We set the shifting parameter s in the shifted geometric mean to $s = 10$ for the time and $s = 100$ for the node counts.

For a set of benchmark results, the above mean values differ considerably if the scales of the individual values v_i are divers. This is illustrated in the following example:

Example A.1 (mean value calculation). Consider two different settings A and B that have been applied on the same test set consisting of 10 instances. The instances in the test set are of varying difficulty. Setting A usually needs 10 % fewer nodes than B, but there are two outliers.

setting	n_1	n_2	n_3	n_4	n_5	n_6	n_7	n_8	n_9	n_{10}
A	100	10	50	100	500	1 000	5 000	10 000	50 000	500 000
B	1	11	55	110	550	1 100	5 500	11 000	55 000	1 000 000
ratio B/A	0.01	1.1	1.1	1.1	1.1	1.1	1.1	1.1	1.1	2.0

Table A.2. Hypothetical node counts for two settings applied on a test set with 10 instances.

Assume that we have observed the node counts given in Table A.2. Then, the mean values for these node counts using $s = 100$ are approximately as follows:

$$\begin{aligned}
&\text{geometric mean:} && \gamma(A) = 1120.7, && \gamma(B) = 817.9, && \frac{\gamma(B)}{\gamma(A)} = 0.73,\\
&\text{shifted geometric mean:} && \gamma_s(A) = 1783.6, && \gamma_s(B) = 1890.2, && \frac{\gamma_s(B)}{\gamma_s(A)} = 1.06,\\
&\text{arithmetic mean:} && \varnothing(A) = 56676.0, && \varnothing(B) = 107332.7, && \frac{\varnothing(B)}{\varnothing(A)} = 1.89.
\end{aligned}$$

Thus, the geometric mean reports a 27 % improvement in the node count for setting B, while the shifted geometric mean and arithmetic mean show a 6 % and 89 % deterioration, respectively.

Example A.1 shows that the performance ratios reported by the three different means are influenced by different characteristics of the benchmark values. Using the geometric mean, one computes a (geometric) average value of the per instance ratios. Small absolute variations in the benchmark values for an easy instance can lead to a large difference in the corresponding ratio. Therefore, ratios for easy instances usually have a much larger range than the ratios for hard instances, and the geometric mean is sometimes dominated by these small absolute variations for the easy instances. In contrast, the arithmetic mean ignores the easy instances almost completely if there are other instances with performance values of a larger scale.

The shifted geometric mean is a compromise between the two extreme cases. It is a variant of the geometric mean that focuses on ratios instead of totals, which prevents the hard instances from dominating the reported result. On the other hand, the shifting of the nominators and denominators of the ratios reduces the strong influence of the very small node counts and time values. Therefore, we use the shifted geometric mean for the summary tables in the main part of the thesis. The geometric means and the totals (which are equivalent to arithmetic means for a test set of fixed size) can be found in the detailed tables in Appendix B.

In the previous Section A.2 we specified the test sets used for our MIP benchmarks. Note that the MIPLIB, CORAL, and MILP test sets are collections of MIP instances that model a variety of different problem classes. Therefore, these test sets are well suited to assess the general performance of MIP solvers. In contrast, each of the other test sets contains different instances of only a single problem class. Therefore, in order to calculate a reasonable total average performance value from the 244 instances, we have to reduce the influence of the problem class specific test sets in the average calculations. Otherwise, these problem classes would be overrepresented in the totals, in particular the ones of larger size.

We decided to treat each of the special problem class test sets as if they represented three instances in the overall test set. This is achieved by calculating the total average performance value as

$$\begin{aligned}\gamma_s(\text{TOTAL}) = \Big(&\gamma_s(\text{MIPLIB})^{30} + \gamma_s(\text{CORAL})^{38} + \gamma_s(\text{MILP})^{37} + \\ &\gamma_s(\text{ENLIGHT})^3 + \gamma_s(\text{ALU})^3 + \gamma_s(\text{FCTP})^3 + \gamma_s(\text{ACC})^3 + \\ &\gamma_s(\text{FC})^3 + \gamma_s(\text{ARCSET})^3 + \gamma_s(\text{MIK})^3\Big)^{\frac{1}{126}}.\end{aligned}$$

Hence, our TOTAL test set consists of 105 individual instances and 21 "pseudo instances" that represent a particular problem class.

A.4 Chip Design Verification Test Set

To compare the SAT and CIP approaches for the chip design verification problem, and to evaluate the impact of the various CIP components, we conducted

experiments on the following sets of property checking instances, which have been generated by OneSpin Solutions:

- ALU: an arithmetical logical unit which is able to perform ADD, SUB, SHL, SHR, and signed and unsigned MULT operations.
- PipeAdder: an adder with a 4-stage pipeline to add up four values.
- PipeMult: a multiplier with a 4-stage pipeline to multiply four values.
- Biquad: a DSP/IIR filter core obtained from [177] in different representations with some valid and invalid properties.
- Multiplier: boothed and non-boothed gate level architectures of signed and unsigned multiplication networks, for which the correctness has to be proven. These instances are rather equivalence checking instances than property checking problems.

All test sets except the Multiplier set involve valid and invalid properties. The width of the input registers in the ALU, PipeAdder and PipeMult sets range from 5 to 40 bits. For the Multiplier instances the width ranges from 2 to 14 bits.

We converted these instances into a register transfer level description and an equivalent gate level representation using proprietary tools of OneSpin Solutions. The former representation is used as input for our CIP solver, the latter for a SAT solver.

Note that the tools of OneSpin Solutions perform an additional presolving step during the transformation to the gate level, which can simplify or sometimes even completely solve the instances. Therefore, some instances are not available as SAT input, which is marked by reporting a node count of 0 and a runtime of 0.0 in the benchmark results. The time needed for the gate level transformation and the presolving is not included in the timings reported in the result tables. However, this time is usually neglectable.

Appendix B

Tables

This appendix chapter presents detailed result tables for our benchmark tests on mixed integer programming. Each table contains statistics for using various different parameter settings on the instances of a single test set. The column "setting" lists the settings that we have tried. In column "T" one can find the number of instances for which the solver hit the time or memory limit if called with the respective parameter settings. The columns "fst" and "slw" indicate, how many instances have been solved at least 10 % faster or slower, respectively, than with the reference settings of the first row (which usually are the default settings).

The n_{gm}, n_{sgm}, and n_{tot} values represent the geometric means, shifted geometric means, and total numbers of branching nodes, respectively, that have been spent on solving the instances of the test set. The numbers are given as percental relative differences to the reference settings: if S_0 are the reference settings, the number in column n_{gm} for setting i is equal to

$$n_{gm}^i = 100 \cdot \left(\frac{\gamma(n_1^i, \ldots, n_k^i)}{\gamma(n_1^0, \ldots, n_k^0)} - 1 \right), \tag{B.1}$$

with n_j^i being the node count on instance j for settings S_i. Analogous formulas are used for the other entries of the table.

The t_{gm}, t_{sgm}, and t_{tot} values are the geometric means, shifted geometric means, and totals of the time needed to solve the instances of the test set. Again, we list percental relative differences to the reference settings that are calculated in a similar way as in Equation (B.1). Note that the numbers shown in the summary tables in the main part of the thesis are the shifted geometric means, i.e., we have copied columns n_{sgm} and t_{sgm} of the "all instances" block into the summary tables.

In order to evaluate the impact and the computational costs of a certain strategy, it is sometimes useful to distinguish between the models that are affected by the change in the solving strategy and the ones for which the solving process proceeds in exactly the same way as for the reference settings. This distinction yields the "different path" and "equal path" blocks of the tables. While the "all instances" block shows the average values for all instances in the test set (with the number of instances indicated in brackets), the average values in the "different path" block are calculated on only those instances for which the solving path for the respective settings is different to the one of the reference settings. Since it is very hard to actually compare the paths in the search tree, we assume that the solving path of two runs is equal if and only if both runs spent the same number of branching nodes and the same total number of simplex iterations to process the instance. Of course, this may declare some runs to have the same path although they were different. However, this seems to be very unlikely.

The "#" columns in these blocks show the number of instances that fall into the respective category. Note that we only include instances that were solved to optimality within the time and memory limit by both, the reference settings and the settings of the respective row. Therefore, the sum of the "different" and "equal path"

instances is only equal to the total number of instances in the test set if none of the two involved settings hit the time or memory limit. Note also that the comparison of the different settings does not make sense in the "different path" and "equal path" blocks, since the values are usually averages over a different subset of the instances.

The full tables with all results of the individual instances and the complete log files of the runs can be found in the web at http://scip.zib.de.

Memory Management

				all instances (30)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	1	0	0	0	0	0	0	0	0	0	—	—	—	29	0	0	0
no block	2	0	9	−1	−1	−7	+9	+8	+16	0	—	—	—	28	+8	+7	+15
no buffer	1	1	5	0	0	−1	+6	+3	+1	0	—	—	—	29	+6	+3	+2
none	2	0	7	−1	−1	−7	+7	+6	+15	0	—	—	—	28	+6	+5	+12

Table B.1. Evaluation of memory management on test set MIPLIB.

				all instances (38)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	3	0	0	0	0	0	0	0	0	0	—	—	—	35	0	0	0
no block	3	0	13	0	0	−1	+10	+10	+10	0	—	—	—	35	+11	+11	+19
no buffer	3	0	0	0	0	0	0	0	0	0	—	—	—	35	0	0	0
none	3	0	13	0	0	−1	+12	+11	+12	0	—	—	—	35	+13	+12	+24

Table B.2. Evaluation of memory management on test set CORAL.

				all instances (36)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	7	0	0	0	0	0	0	0	0	0	—	—	—	29	0	0	0
no block	7	0	10	−1	−1	−7	+8	+7	+4	0	—	—	—	29	+10	+9	+10
no buffer	7	0	0	0	0	0	−1	−1	0	0	—	—	—	29	−1	−1	−1
none	7	0	10	−1	−1	−7	+7	+7	+3	0	—	—	—	29	+9	+8	+9

Table B.3. Evaluation of memory management on test set MILP.

				all instances (7)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—	7	0	0	0
no block	0	0	0	0	0	0	+4	+3	+8	0	—	—	—	7	+4	+3	+8
no buffer	0	0	0	0	0	0	+1	−1	−2	0	—	—	—	7	+1	−1	−2
none	0	0	3	0	0	0	+11	+7	+19	0	—	—	—	7	+11	+7	+19

Table B.4. Evaluation of memory management on test set ENLIGHT.

				all instances (25)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	1	0	0	0	0	0	0	0	0	0	—	—	—	24	0	0	0
no block	1	0	2	0	0	−7	+4	+4	+4	0	—	—	—	24	+4	+5	+10
no buffer	1	0	0	0	0	0	+1	+1	+1	0	—	—	—	24	+1	+1	+2
none	1	0	5	−1	−1	−10	+8	+6	+6	0	—	—	—	24	+8	+6	+14

Table B.5. Evaluation of memory management on test set ALU.

				all instances (16)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—	16	0	0	0
no block	0	0	2	0	0	0	+6	+5	+9	0	—	—	—	16	+6	+5	+9
no buffer	0	0	1	0	0	0	+6	+4	+6	0	—	—	—	16	+6	+4	+6
none	0	0	0	0	0	0	+4	+3	+5	0	—	—	—	16	+4	+3	+5

Table B.6. Evaluation of memory management on test set FCTP.

				all instances (7)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—	7	0	0	0
no block	0	0	0	0	0	0	+2	+2	+1	0	—	—	—	7	+2	+2	+1
no buffer	0	0	0	0	0	0	0	0	0	0	—	—	—	7	0	0	0
none	0	0	0	0	0	0	+3	+2	+2	0	—	—	—	7	+3	+2	+2

Table B.7. Evaluation of memory management on test set ACC.

				all instances (20)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—	20	0	0	0
no block	0	0	0	0	0	0	+4	+3	+4	0	—	—	—	20	+4	+3	+4
no buffer	0	0	0	0	0	0	+4	+3	+2	0	—	—	—	20	+4	+3	+2
none	0	0	1	0	0	0	+6	+5	+5	0	—	—	—	20	+6	+5	+5

Table B.8. Evaluation of memory management on test set FC.

				all instances (23)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—	23	0	0	0
no block	0	0	15	0	0	0	+10	+9	+9	0	—	—	—	23	+10	+9	+9
no buffer	0	0	0	0	0	0	+2	+1	+1	0	—	—	—	23	+2	+1	+1
none	0	0	7	0	0	0	+9	+8	+6	0	—	—	—	23	+9	+8	+6

Table B.9. Evaluation of memory management on test set ARCSET.

				all instances (41)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—	41	0	0	0
no block	0	0	24	0	0	0	+11	+11	+18	0	—	—	—	41	+11	+11	+18
no buffer	0	0	0	0	0	0	+1	+1	+1	0	—	—	—	41	+1	+1	+1
none	0	0	29	0	0	0	+12	+11	+19	0	—	—	—	41	+12	+11	+19

Table B.10. Evaluation of memory management on test set MIK.

BRANCHING

				all instances (30)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	1	0	0	0	0	0	0	0	0	0	—	—	—	29	0	0	0
random	8	1	22	+808	+475	+42	+162	+139	+180	19	+134	+112	+227	3	−2	−1	−1
most inf	7	2	20	+569	+341	+16	+153	+139	+169	20	+147	+139	+356	3	−2	−2	−2
least inf	10	1	22	+1804	+1096	+100	+325	+266	+268	17	+322	+268	+790	3	0	0	0
pseudocost	1	5	13	+183	+87	−11	+20	+16	+27	26	+23	+19	+40	3	0	0	0
full strong	3	1	18	−71	−65	−82	+95	+92	+128	24	+107	+105	+165	3	−1	0	0
strong	2	1	13	−64	−62	−78	+38	+38	+46	19	+52	+49	+36	9	+1	+1	0
hybr strong	1	2	11	−24	−18	+3	+19	+20	+23	22	+27	+27	+34	7	−1	−1	−2
psc strinit	1	8	6	+31	+13	−2	+5	+5	−3	25	+6	+6	−5	4	−2	−1	−2
reliability	1	4	4	−10	−7	−9	−1	−1	−6	21	−1	−1	−8	8	−1	−1	−1
inference	7	1	20	+483	+269	+53	+108	+101	+171	20	+84	+79	+377	3	0	0	0

Table B.11. Evaluation of branching on test set MIPLIB.

				all instances (37)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	2	0	0	0	0	0	0	0	0	0	—	—	—	35	0	0	0
random	18	4	27	+758	+694	+148	+332	+332	+298	17	+114	+123	+222	2	0	0	0
most inf	15	4	26	+529	+517	+125	+312	+314	+275	20	+132	+141	+232	2	+1	+1	+1
least inf	21	2	29	+1586	+1380	+195	+598	+575	+369	14	+242	+238	+330	2	−2	−2	−1
pseudocost	3	9	19	+84	+79	+68	+38	+40	+50	31	+37	+39	+70	2	0	0	+1
full strong	4	3	28	−85	−79	−34	+98	+97	+86	30	+115	+114	+115	2	−2	−2	−1
strong	3	2	23	−73	−68	−35	+59	+59	+65	31	+68	+69	+96	3	−1	−1	−1
hybr strong	4	8	14	−19	−12	+32	+27	+27	+43	31	+16	+16	+28	2	+1	+1	+2
psc strinit	1	14	9	+17	+18	−18	+2	+2	−7	33	+6	+6	+11	2	−2	−2	−1
reliability	2	11	8	+14	+16	−14	+6	+7	+13	31	+7	+8	+23	4	+2	+2	+2
inference	12	4	25	+381	+329	+105	+179	+177	+192	23	+103	+103	+153	2	−1	0	0

Table B.12. Evaluation of branching on test set CORAL.

				all instances (35)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	6	0	0	0	0	0	0	0	0	0	—	—	—	29	0	0	0
random	14	5	16	+304	+194	+19	+86	+81	+65	16	+48	+41	+47	5	+1	0	+1
most inf	16	5	16	+269	+187	+22	+93	+86	+75	14	+14	+6	−6	5	0	0	0
least inf	16	5	16	+488	+306	+32	+113	+109	+84	14	+47	+42	+53	5	0	0	0
pseudocost	10	7	13	+105	+71	+18	+23	+23	+30	20	+17	+15	+35	5	−1	−1	−1
full strong	12	4	19	−79	−72	−68	+115	+107	+72	15	+198	+177	+212	5	0	0	+1
strong	7	4	18	−58	−59	−80	+46	+44	+22	17	+111	+105	+99	9	+2	+1	+1
hybr strong	10	1	18	+45	+41	+24	+47	+43	+33	17	+73	+64	+66	8	0	0	−1
psc strinit	8	7	8	+37	+40	+9	+8	+9	+14	18	−1	0	−1	9	0	0	0
reliability	6	7	8	+11	+7	−4	+5	+6	+7	17	+4	+6	+6	10	+1	+1	+2
inference	8	8	11	+122	+76	+1	+20	+20	+25	21	+25	+26	+67	5	−2	−1	0

Table B.13. Evaluation of branching on test set MILP.

				all instances (7)							different path				equal path		
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—	7	0	0	0
random	2	0	5	+164	+163	+144	+147	+115	+202	4	+174	+203	+615	1	0	0	0
most inf	0	4	0	−29	−29	−65	−35	−40	−71	6	−39	−43	−71	1	0	0	0
least inf	2	0	5	+221	+219	+194	+191	+149	+212	4	+266	+325	+1235	1	0	0	0
pseudocost	0	2	3	+3	+3	−70	−8	−27	−74	6	−9	−30	−74	1	0	0	0
full strong	0	0	5	−84	−83	−91	+68	+45	+83	6	+83	+51	+83	1	0	0	0
strong	0	1	4	−86	−85	−93	+20	+11	−2	6	+24	+13	−2	1	0	0	0
hybr strong	0	1	4	0	+1	+21	+21	+9	+21	6	+25	+11	+21	1	0	0	0
psc strinit	0	2	3	+22	+23	+68	+26	+27	+66	6	+31	+30	+66	1	0	0	0
reliability	0	2	1	−9	−8	+13	0	+5	+11	6	0	+5	+11	1	0	0	0
inference	0	3	1	−48	−49	−91	−56	−70	−93	6	−62	−73	−93	1	0	0	0

Table B.14. Evaluation of branching on test set ENLIGHT.

				all instances (24)						different path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—
random	6	1	22	+23951	+6987	+1249	+2655	+1271	+1061	18	+3070	+1559	+1149
most inf	10	0	19	+9715	+5127	+1577	+2897	+1991	+1611	14	+1412	+1567	+4680
least inf	8	0	23	+30929	+9084	+1487	+3504	+1891	+1482	16	+4071	+2655	+6360
pseudocost	5	0	19	+3169	+1659	+804	+907	+619	+818	19	+676	+472	+500
full strong	4	1	17	−65	−60	−51	+157	+180	+572	15	+116	+123	+179
strong	0	5	10	−79	−78	−78	+7	+13	+93	17	+10	+16	+93
hybr strong	1	6	8	−43	−31	+217	−2	+11	+198	18	−15	−4	+87
psc strinit	0	3	12	+149	+120	+50	+67	+55	+64	19	+92	+66	+64
reliability	0	5	11	+6	+17	+246	+23	+36	+182	19	+30	+43	+183
inference	0	9	7	+176	+6	−39	−29	−35	−45	24	−29	−35	−45

Table B.15. Evaluation of branching on test set ALU.

				all instances (16)							different path				equal path		
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—	16	0	0	0
random	3	0	10	+867	+511	+1352	+247	+288	+1121	10	+209	+256	+773	3	0	0	0
most inf	3	0	11	+757	+443	+1139	+234	+267	+1095	10	+191	+223	+634	3	0	0	0
least inf	3	0	11	+1684	+931	+1974	+355	+379	+1252	10	+378	+415	+1469	3	0	0	0
pseudocost	0	2	9	+221	+103	+93	+33	+35	+65	13	+42	+40	+65	3	0	0	0
full strong	0	1	10	−72	−73	−92	+33	+36	+81	13	+43	+40	+81	3	0	0	0
strong	0	1	9	−65	−68	−90	+24	+25	+48	11	+36	+31	+49	5	+2	+3	+3
hybr strong	0	1	6	+7	+6	+14	+3	+4	+10	11	+5	+5	+10	5	0	0	0
psc strinit	0	0	7	+71	+39	+39	+13	+14	+29	12	+16	+17	+29	4	+3	+4	+4
reliability	0	0	0	0	0	−1	+3	+2	+3	8	+4	+3	+3	8	+1	+2	+2
inference	1	0	10	+631	+364	+818	+170	+187	+683	12	+221	+225	+801	3	0	0	0

Table B.16. Evaluation of branching on test set FCTP.

				all instances (7)							different path				equal path		
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—	7	0	0	0
random	1	1	3	+292	+393	+1639	+48	+52	+175	3	−3	−1	+70	3	−3	−3	−3
most inf	3	1	3	+396	+513	+2057	+81	+85	+264	1	−82	−82	−82	3	−2	−2	−2
least inf	3	0	3	+1110	+1422	+5881	+132	+138	+293	1	+8	+8	+8	3	−3	−3	−3
pseudocost	0	3	1	+69	+88	+314	−40	−41	−34	4	−59	−58	−35	3	−2	−2	−2
full strong	3	0	4	−90	−95	−98	+167	+174	+346	1	+390	+390	+390	3	−1	−1	−1
strong	2	0	4	−43	−52	−62	+78	+82	+202	2	+80	+79	+63	3	−3	−3	−3
hybr strong	3	0	4	+316	+392	+778	+147	+153	+309	1	+59	+59	+59	3	−1	−1	−1
psc strinit	1	2	2	+6	+31	+459	+10	+11	+90	3	−31	−31	−13	3	−3	−3	−3
reliability	1	1	3	+325	+404	+864	+81	+84	+163	3	+184	+183	+150	3	−2	−2	−2
inference	0	3	1	+29	+33	+110	−23	−24	−27	4	−36	−36	−28	3	−2	−2	−2

Table B.17. Evaluation of branching on test set ACC.

				all instances (19)						different path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—
random	2	1	14	+11564	+5542	+22505	+684	+912	+5925	17	+456	+634	+3346
most inf	2	0	14	+9640	+5060	+20634	+840	+1152	+6688	17	+581	+842	+4359
least inf	2	1	14	+12830	+6039	+28965	+618	+837	+6129	17	+404	+569	+3617
pseudocost	0	0	13	+1230	+603	+490	+77	+98	+154	19	+77	+98	+154
full strong	0	1	8	−74	−81	−92	+11	+14	+21	19	+11	+14	+21
strong	0	0	10	−66	−73	−89	+14	+18	+24	14	+18	+21	+26
hybr strong	0	0	9	−27	−28	−31	+13	+14	+17	15	+15	+16	+18
psc strinit	0	3	1	+69	+54	+17	−4	−5	−7	14	−4	−5	−7
reliability	0	1	0	0	0	−1	−2	−2	−2	9	−2	−2	−2
inference	0	4	13	+2204	+1137	+1964	+137	+188	+568	19	+137	+188	+568

Table B.18. Evaluation of branching on test set FC.

				all instances (23)						different path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—
random	11	0	22	+3802	+3219	+2518	+1348	+1276	+1080	12	+660	+662	+1431
most inf	12	0	23	+2863	+2434	+2543	+1163	+1114	+1078	11	+397	+404	+876
least inf	13	0	23	+4221	+3573	+2868	+1370	+1296	+1134	10	+380	+376	+577
pseudocost	1	1	22	+301	+248	+180	+110	+106	+118	22	+106	+102	+79
full strong	2	2	19	−62	−60	−37	+112	+112	+259	21	+105	+107	+316
strong	2	1	17	−52	−51	−59	+74	+72	+172	20	+63	+62	+69
hybr strong	0	2	18	−6	−6	+21	+39	+38	+48	23	+39	+38	+48
psc strinit	0	6	12	+46	+37	+60	+18	+18	+31	23	+18	+18	+31
reliability	0	5	7	+4	0	−14	0	−1	−8	19	−1	−2	−13
inference	1	2	21	+875	+742	+656	+331	+317	+355	22	+325	+314	+331

Table B.19. Evaluation of branching on test set ARCSET.

				all instances (41)						different path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—
random	41	0	41	+9016	+8994	+3851	+11976	+10606	+5543	0	—	—	—
most inf	41	0	41	+7415	+7397	+3189	+11976	+10606	+5543	0	—	—	—
least inf	34	0	41	+9218	+9195	+4088	+10168	+9009	+5024	7	+10977	+10504	+11773
pseudocost	0	0	40	+123	+123	+125	+113	+102	+143	41	+113	+102	+143
full strong	0	2	36	−90	−90	−87	+59	+59	+140	41	+59	+59	+140
strong	0	14	20	−86	−86	−83	+5	+8	+38	41	+5	+8	+38
hybr strong	0	7	20	+1	+1	+18	+11	+11	+24	41	+11	+11	+24
psc strinit	0	4	28	+32	+32	+62	+35	+35	+75	41	+35	+35	+75
reliability	0	4	6	−1	−1	+8	+2	+2	+10	41	+2	+2	+10
inference	30	0	41	+4663	+4652	+2650	+6524	+5841	+4271	11	+2721	+2604	+3467

Table B.20. Evaluation of branching on test set MIK.

BRANCHING SCORE FUNCTION

setting	T	fst	slw	all instances (30) n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	different path t_{gm}	t_{sgm}	t_{tot}	#	equal path t_{gm}	t_{sgm}	t_{tot}
default	1	0	0	0	0	0	0	0	0	0	—	—	—	29	0	0	0
min ($\mu = 0$)	2	1	12	+22	+24	−5	+26	+26	+40	24	+32	+32	+66	4	+1	+1	0
weighted ($\mu = \frac{1}{6}$)	2	4	9	+27	+21	+2	+12	+12	+23	24	+10	+10	+5	4	−1	−1	−1
weighted ($\mu = \frac{1}{3}$)	3	4	10	+58	+46	+24	+26	+28	+61	23	+21	+23	+35	4	−1	−1	−1
avg ($\mu = \frac{1}{2}$)	2	0	14	+88	+73	+9	+29	+30	+50	24	+29	+29	+42	4	−1	−1	−2
max ($\mu = 1$)	4	1	16	+109	+92	+41	+53	+53	+80	22	+57	+58	+92	4	−1	−1	−1

Table B.21. Evaluation of branching score functions on test set MIPLIB.

setting	T	fst	slw	all instances (37) n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	different path t_{gm}	t_{sgm}	t_{tot}	#	equal path t_{gm}	t_{sgm}	t_{tot}
default	2	0	0	0	0	0	0	0	0	0	—	—	—	35	0	0	0
min ($\mu = 0$)	3	7	19	+23	+25	+4	+26	+25	+30	31	+31	+31	+51	2	+1	+1	+1
weighted ($\mu = \frac{1}{6}$)	5	8	13	+49	+48	+7	+21	+20	+43	30	+11	+10	+8	2	−1	−1	−1
weighted ($\mu = \frac{1}{3}$)	4	12	12	+47	+53	+6	+26	+27	+42	30	+23	+25	+40	2	−3	−3	−2
avg ($\mu = \frac{1}{2}$)	5	7	20	+119	+102	+27	+50	+49	+66	30	+45	+44	+62	2	+1	+1	+1
max ($\mu = 1$)	7	5	23	+220	+199	+94	+113	+113	+135	28	+87	+88	+155	2	+1	+1	+1

Table B.22. Evaluation of branching score functions on test set CORAL.

setting	T	fst	slw	all instances (35) n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	different path t_{gm}	t_{sgm}	t_{tot}	#	equal path t_{gm}	t_{sgm}	t_{tot}
default	6	0	0	0	0	0	0	0	0	0	—	—	—	29	0	0	0
min ($\mu = 0$)	8	4	11	+28	+19	−13	+18	+18	+15	18	+26	+27	+34	8	0	0	0
weighted ($\mu = \frac{1}{6}$)	7	5	10	+36	+26	−11	+10	+10	+6	19	+10	+8	−8	8	0	0	0
weighted ($\mu = \frac{1}{3}$)	8	3	12	+80	+69	+5	+37	+36	+25	19	+52	+50	+62	7	+2	+2	+3
avg ($\mu = \frac{1}{2}$)	10	4	13	+67	+68	+22	+36	+35	+33	20	+39	+37	+47	5	+1	+1	+1
max ($\mu = 1$)	11	2	16	+106	+97	+17	+59	+58	+48	19	+74	+73	+94	5	+2	+2	+3

Table B.23. Evaluation of branching score functions on test set MILP.

setting	T	fst	slw	all instances (7) n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	different path t_{gm}	t_{sgm}	t_{tot}	#	equal path t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—	7	0	0	0
min ($\mu = 0$)	0	1	4	+10	+12	+67	+26	+34	+73	6	+31	+38	+73	1	0	0	0
weighted ($\mu = \frac{1}{6}$)	0	3	2	−13	−13	−48	−10	−19	−52	6	−11	−21	−52	1	0	0	0
weighted ($\mu = \frac{1}{3}$)	0	2	3	+14	+15	−23	+18	−1	−30	6	+21	−1	−30	1	0	0	0
avg ($\mu = \frac{1}{2}$)	0	3	2	+2	+2	−9	0	−9	−22	6	0	−10	−22	1	0	0	0
max ($\mu = 1$)	0	1	4	+115	+116	+194	+113	+115	+172	6	+141	+135	+172	1	0	0	0

Table B.24. Evaluation of branching score functions on test set ENLIGHT.

setting	T	fst	slw	all instances (25) n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	different path t_{gm}	t_{sgm}	t_{tot}	#	equal path t_{gm}	t_{sgm}	t_{tot}
default	1	0	0	0	0	0	0	0	0	0	—	—	—	24	0	0	0
min ($\mu = 0$)	3	3	13	+70	+85	+108	+78	+88	+161	17	+79	+86	+188	5	0	0	0
weighted ($\mu = \frac{1}{6}$)	1	3	12	+117	+104	+53	+76	+66	+48	19	+111	+90	+111	5	0	0	0
weighted ($\mu = \frac{1}{3}$)	2	2	13	+94	+96	+92	+79	+85	+116	18	+73	+71	+139	5	0	0	0
avg ($\mu = \frac{1}{2}$)	2	3	14	+195	+165	+86	+116	+101	+108	19	+116	+88	+121	4	0	0	0
max ($\mu = 1$)	3	1	16	+499	+357	+135	+239	+187	+185	19	+256	+183	+250	3	0	0	0

Table B.25. Evaluation of branching score functions on test set ALU.

setting	T	fst	slw	all instances (16) n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	different path t_{gm}	t_{sgm}	t_{tot}	#	equal path t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—	16	0	0	0
min ($\mu = 0$)	0	1	9	+70	+71	+277	+57	+72	+286	12	+83	+89	+287	4	0	0	0
weighted ($\mu = \frac{1}{6}$)	0	3	3	+13	+8	+10	+1	−2	+1	12	+1	−2	+1	4	0	0	0
weighted ($\mu = \frac{1}{3}$)	0	2	6	+14	+12	+45	+8	+6	+24	12	+10	+7	+24	4	+3	+4	+4
avg ($\mu = \frac{1}{2}$)	0	1	7	+32	+34	+125	+21	+26	+89	13	+26	+29	+89	3	0	0	0
max ($\mu = 1$)	0	0	10	+88	+68	+245	+48	+56	+176	13	+62	+64	+177	3	0	0	0

Table B.26. Evaluation of branching score functions on test set FCTP.

setting	T	fst	slw	all instances (7) n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	different path t_{gm}	t_{sgm}	t_{tot}	#	equal path t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—	7	0	0	0
min ($\mu = 0$)	1	0	4	+77	+100	+455	+41	+43	+120	3	+22	+22	+25	3	−2	−2	−2
weighted ($\mu = \frac{1}{6}$)	2	1	3	+312	+390	+1151	+67	+70	+186	2	+40	+39	+20	3	−2	−2	−2
weighted ($\mu = \frac{1}{3}$)	1	1	3	+176	+222	+836	+27	+29	+136	3	+25	+26	+103	3	−2	−2	−2
avg ($\mu = \frac{1}{2}$)	1	0	3	+191	+235	+730	+39	+41	+107	3	+17	+17	+8	3	−2	−2	−2
max ($\mu = 1$)	0	0	4	+316	+391	+733	+48	+50	+81	4	+101	+101	+84	3	−2	−2	−2

Table B.27. Evaluation of branching score functions on test set ACC.

setting	T	fst	slw	all instances (20) n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	different path t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—
min ($\mu = 0$)	0	1	12	+135	+147	+292	+43	+58	+111	19	+45	+60	+113
weighted ($\mu = \frac{1}{6}$)	0	3	1	−8	−12	−29	−6	−7	−11	20	−6	−7	−11
weighted ($\mu = \frac{1}{3}$)	0	4	1	−1	−6	−16	−4	−6	−9	20	−4	−6	−9
avg ($\mu = \frac{1}{2}$)	0	6	4	−31	−35	−30	−3	−4	−6	20	−3	−4	−6
max ($\mu = 1$)	0	6	4	+4	−10	+7	−2	−2	−2	20	−2	−2	−2

Table B.28. Evaluation of branching score functions on test set FC.

setting	T	fst	slw	all instances (23) n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	different path t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—
min ($\mu = 0$)	0	3	14	+66	+57	+87	+37	+35	+42	23	+37	+35	+42
weighted ($\mu = \frac{1}{6}$)	0	4	12	+23	+20	+13	+12	+11	+6	23	+12	+11	+6
weighted ($\mu = \frac{1}{3}$)	0	3	14	+44	+37	+43	+22	+22	+23	23	+22	+22	+23
avg ($\mu = \frac{1}{2}$)	1	3	12	+69	+60	+116	+32	+32	+90	22	+27	+26	+39
max ($\mu = 1$)	1	2	16	+117	+109	+232	+61	+62	+161	22	+56	+57	+138

Table B.29. Evaluation of branching score functions on test set ARCSET.

setting	T	fst	slw	all instances (41) n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	different path t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—
min ($\mu = 0$)	0	1	36	+131	+131	+267	+132	+134	+274	41	+132	+134	+274
weighted ($\mu = \frac{1}{6}$)	0	4	20	+16	+16	+37	+12	+13	+36	41	+12	+13	+36
weighted ($\mu = \frac{1}{3}$)	0	1	34	+34	+34	+58	+31	+31	+64	41	+31	+31	+64
avg ($\mu = \frac{1}{2}$)	0	0	38	+63	+63	+84	+54	+52	+97	41	+54	+52	+97
max ($\mu = 1$)	3	0	41	+193	+193	+291	+180	+169	+459	38	+156	+149	+191

Table B.30. Evaluation of branching score functions on test set MIK.

NODE SELECTION

				all instances (30)							different path				equal path		
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	1	0	0	0	0	0	0	0	0	0	—	—	—	29	0	0	0
dfs	2	5	8	+60	+64	+81	+22	+22	+36	18	+26	+22	+4	9	−2	−2	−2
bfs	1	3	15	−24	−18	−15	+30	+28	+25	25	+37	+34	+37	4	−2	−2	−2
bfs/plunge	1	2	6	+5	+3	+24	+4	+3	+4	20	+7	+5	+7	9	0	0	0
estimate	1	3	13	−18	−15	+24	+15	+14	+12	23	+20	+18	+18	6	0	0	−1
estim/plunge	1	6	3	−3	−3	+15	−3	−3	−10	18	−6	−5	−15	11	0	0	0
hybrid	2	3	7	+7	+7	+26	+8	+8	+18	18	+5	+4	−10	10	−2	−1	−2

Table B.31. Evaluation of node selection on test set MIPLIB.

				all instances (38)							different path				equal path		
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	3	0	0	0	0	0	0	0	0	0	—	—	—	35	0	0	0
dfs	7	13	15	+145	+143	+38	+51	+49	+69	28	+11	+12	+31	3	+1	+1	+1
bfs	5	6	20	−41	−28	−1	+37	+38	+46	30	+31	+33	+52	3	+3	+3	+4
bfs/plunge	2	6	18	−1	0	−12	+2	+2	−10	30	+15	+15	+10	5	+2	+2	+2
estimate	4	9	19	−23	−9	−5	+19	+19	+17	30	+24	+25	+29	3	0	0	0
estim/plunge	2	8	10	+3	+3	−9	−3	−3	−11	30	+1	+1	+4	5	+2	+2	+2
hybrid	2	7	10	−9	−9	0	−5	−5	−10	30	+4	+4	+10	5	+3	+2	+2

Table B.32. Evaluation of node selection on test set CORAL.

				all instances (35)							different path				equal path		
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	6	0	0	0	0	0	0	0	0	0	—	—	—	29	0	0	0
dfs	6	5	13	+57	+41	+8	+12	+12	+7	18	+13	+13	+10	9	+1	+1	+1
bfs	6	5	12	−24	−12	−6	+20	+20	+13	20	+39	+38	+57	8	0	0	−1
bfs/plunge	4	4	8	+3	+2	+9	+2	+1	−4	17	+8	+8	+8	11	+1	+1	+2
estimate	5	6	12	−19	−9	+9	+9	+8	+3	20	+17	+15	+16	8	−1	0	−1
estim/plunge	4	3	7	+9	+9	+18	+5	+5	+2	16	+11	+11	+14	13	+3	+2	+2
hybrid	6	2	8	+9	+8	+28	+4	+3	0	18	+6	+5	−1	11	+2	+2	+3

Table B.33. Evaluation of node selection on test set MILP.

				all instances (7)							different path				equal path		
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—	7	0	0	0
dfs	0	1	4	+109	+109	+51	+73	+53	+9	6	+90	+61	+9	1	0	0	0
bfs	0	1	4	+14	+16	+61	+78	+74	+136	6	+95	+86	+136	1	0	0	0
bfs/plunge	1	1	4	+35	+36	+200	+42	+53	+173	5	+39	+57	+275	1	0	0	0
estimate	0	2	3	+1	+2	−52	+24	0	−49	6	+28	0	−49	1	0	0	0
estim/plunge	0	2	1	+7	+7	−11	−5	−11	−29	6	−5	−12	−29	1	0	0	0
hybrid	0	2	3	+12	+13	+113	+20	+25	+108	6	+24	+28	+108	1	0	0	0

Table B.34. Evaluation of node selection on test set ENLIGHT.

				all instances (24)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—	24	0	0	0
dfs	0	12	1	−43	−43	−52	−36	−40	−59	16	−49	−47	−59	8	0	0	0
bfs	0	2	13	+55	+50	+121	+91	+101	+250	19	+127	+125	+250	5	0	0	0
bfs/plunge	0	6	5	+27	+29	+56	+20	+25	+97	16	+32	+34	+97	8	0	0	0
estimate	1	2	15	+97	+87	+189	+108	+113	+273	18	+131	+120	+171	5	0	0	0
estim/plunge	1	6	6	+16	+17	+93	+18	+25	+127	14	−4	−3	−5	9	0	0	0
hybrid	0	6	4	+6	+6	+2	+2	+3	+9	16	+3	+4	+9	8	0	0	0

Table B.35. Evaluation of node selection on test set ALU.

				all instances (16)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—	16	0	0	0
dfs	0	1	8	+71	+74	+219	+46	+59	+145	11	+74	+78	+145	5	0	0	0
bfs	0	1	8	−12	−6	+12	+20	+25	+63	12	+28	+29	+63	4	0	0	0
bfs/plunge	0	1	6	−1	+2	+19	+7	+9	+27	12	+9	+10	+27	4	0	0	0
estimate	0	2	7	−6	−5	+24	+17	+22	+66	12	+24	+26	+66	4	0	0	0
estim/plunge	0	3	1	−2	−3	−5	−1	−2	−6	9	−3	−3	−6	7	+1	+2	+2
hybrid	0	1	6	+1	+2	+21	+6	+7	+26	12	+8	+9	+26	4	0	0	0

Table B.36. Evaluation of node selection on test set FCTP.

				all instances (7)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—	7	0	0	0
dfs	2	0	4	+340	+425	+1069	+98	+102	+233	2	+249	+248	+289	3	−3	−3	−3
bfs	0	2	2	−22	−30	−24	−7	−7	−23	4	−11	−11	−24	3	−2	−2	−2
bfs/plunge	0	2	1	−16	−7	+229	−7	−7	+8	3	−15	−14	+9	4	−2	−2	−2
estimate	0	2	2	−21	−29	−20	−7	−7	−23	4	−11	−11	−24	3	−2	−2	−2
estim/plunge	1	2	1	+10	+25	+458	+7	+8	+68	2	−49	−49	−45	4	−2	−2	−2
hybrid	0	2	1	+11	+27	+495	+7	+7	+62	3	+19	+19	+69	4	−1	−1	−1

Table B.37. Evaluation of node selection on test set ACC.

				all instances (20)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—	20	0	0	0
dfs	0	3	1	−9	−9	−2	−3	−4	−4	15	−4	−4	−4	5	−2	−2	−2
bfs	0	1	10	−29	−31	−36	+11	+14	+20	15	+16	+17	+21	5	−2	−2	−2
bfs/plunge	0	0	2	+8	+9	+8	+2	+2	+3	15	+3	+3	+3	5	−2	−2	−2
estimate	0	1	5	−29	−32	−35	+4	+6	+10	15	+6	+7	+11	5	−2	−2	−2
estim/plunge	0	0	1	+8	+9	+6	0	+1	+1	13	+1	+1	+1	7	−1	−1	−1
hybrid	0	0	2	+11	+12	+23	+3	+5	+9	15	+5	+6	+10	5	−1	−1	−1

Table B.38. Evaluation of node selection on test set FC.

				all instances (23)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—	23	0	0	0
dfs	3	5	15	+253	+249	+1007	+152	+152	+287	20	+95	+97	+116	0	—	—	—
bfs	1	0	21	+7	+6	+44	+55	+53	+108	22	+50	+48	+64	0	—	—	—
bfs/plunge	0	4	10	0	0	+7	+10	+11	+21	22	+10	+11	+21	1	+4	+4	+4
estimate	0	2	17	+22	+19	+23	+43	+42	+42	23	+43	+42	+42	0	—	—	—
estim/plunge	0	5	4	+7	+7	+8	+5	+5	−4	20	+6	+6	−4	3	+2	+2	+3
hybrid	0	3	10	+4	+3	+18	+13	+13	+30	22	+13	+13	+30	1	+3	+3	+3

Table B.39. Evaluation of node selection on test set ARCSET.

setting	T	fst	slw	all instances (41) n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	different path t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—
dfs	0	16	11	+19	+19	−1	−5	−7	−21	41	−5	−7	−21
bfs	0	1	36	+3	+3	+19	+46	+44	+65	41	+46	+44	+65
bfs/plunge	0	5	11	−4	−4	+6	+2	+3	+6	41	+2	+3	+6
estimate	0	0	38	+6	+6	+12	+42	+39	+52	41	+42	+39	+52
estim/plunge	0	9	3	+2	+2	+5	−3	−2	−2	41	−3	−2	−2
hybrid	0	5	14	0	0	+5	+6	+6	+5	41	+6	+6	+5

Table B.40. Evaluation of node selection on test set MIK.

CHILD SELECTION

setting	T	fst	slw	all instances (30) n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	different path t_{gm}	t_{sgm}	t_{tot}	#	equal path t_{gm}	t_{sgm}	t_{tot}
default	1	0	0	0	0	0	0	0	0	0	—	—	—	29	0	0	0
down	1	2	11	+20	+18	+5	+22	+23	+27	22	+31	+32	+40	7	−1	−1	−1
up	1	5	4	+3	−1	−6	0	0	+2	23	0	0	+3	6	−1	−1	−2
pseudocost	2	6	7	−11	−8	−15	+11	+12	+20	22	+13	+15	+24	6	−2	−2	−2
LP value	1	9	9	+5	+1	−20	+7	+7	+3	22	+10	+10	+5	7	−1	0	−1
root LP value	1	3	6	+2	−2	−6	+1	0	+1	25	+2	0	+1	4	0	0	0
inference	1	4	4	−5	−4	+6	+4	+4	+1	19	+6	+6	+2	10	+1	0	0

Table B.41. Evaluation of child selection on test set MIPLIB.

setting	T	fst	slw	all instances (38) n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	different path t_{gm}	t_{sgm}	t_{tot}	#	equal path t_{gm}	t_{sgm}	t_{tot}
default	3	0	0	0	0	0	0	0	0	0	—	—	—	35	0	0	0
down	3	10	13	+11	+14	+5	+11	+10	+12	30	+6	+4	+10	4	0	0	0
up	3	14	4	−27	−19	+65	−12	−11	+3	31	−14	−13	+7	4	+1	+1	+1
pseudocost	4	14	9	−7	−1	+20	+5	+6	+17	30	−4	−3	−2	3	−2	−2	−2
LP value	2	11	10	−13	−9	+5	−2	−2	−5	32	+1	+1	+10	3	−1	−1	0
root LP value	4	15	5	−19	−14	−11	−6	−5	+9	31	−10	−9	+1	3	0	0	0
inference	3	13	5	+3	+1	−7	−3	−4	−4	30	−4	−5	−8	5	−1	−1	−1

Table B.42. Evaluation of child selection on test set CORAL.

setting	T	fst	slw	all instances (35) n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	different path t_{gm}	t_{sgm}	t_{tot}	#	equal path t_{gm}	t_{sgm}	t_{tot}
default	6	0	0	0	0	0	0	0	0	0	—	—	—	29	0	0	0
down	5	4	10	+7	+8	−6	+15	+14	+11	20	+31	+30	+45	9	+1	+1	+1
up	4	8	3	−11	−9	+5	−5	−5	0	19	−9	−9	+10	10	+3	+2	+2
pseudocost	5	8	10	+7	+10	−9	+11	+10	+7	20	+24	+23	+50	8	+1	+1	+1
LP value	5	5	12	+11	+15	−4	+13	+12	+14	21	+23	+22	+48	8	0	+1	+1
root LP value	5	9	4	−10	−9	−1	−8	−8	−5	20	−13	−13	−12	9	0	+1	0
inference	3	5	6	−15	−12	+3	−4	−5	−8	17	−6	−7	−10	12	+1	+1	+2

Table B.43. Evaluation of child selection on test set MILP.

setting	T	fst	slw	all instances (7) n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	different path t_{gm}	t_{sgm}	t_{tot}	#	equal path t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—	7	0	0	0
down	0	1	3	−5	−4	+16	+2	−1	+11	6	+3	−1	+11	1	0	0	0
up	0	1	3	+20	+20	−12	+17	+3	−16	6	+20	+4	−16	1	0	0	0
pseudocost	0	2	2	−3	−3	+34	+1	−2	+22	6	+1	−2	+22	1	0	0	0
LP value	0	3	1	−9	−9	−17	−9	−11	−28	6	−10	−12	−28	1	0	0	0
root LP value	0	2	1	−3	−3	−11	−5	−8	−16	6	−6	−8	−16	1	0	0	0
inference	0	2	3	+7	+7	+27	+10	+3	+22	6	+11	+4	+22	1	0	0	0

Table B.44. Evaluation of child selection on test set ENLIGHT.

				all instances (24)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—	24	0	0	0
down	0	9	7	−1	−6	−19	−6	−12	−24	19	−7	−13	−24	5	0	0	0
up	0	7	6	−26	−20	+67	−8	−3	+88	17	−11	−4	+88	7	0	0	0
pseudocost	0	6	8	+33	+27	+14	+19	+11	+12	19	+25	+13	+12	5	0	0	0
LP value	0	6	6	−1	+3	+10	+5	+5	+7	19	+7	+5	+7	5	0	0	0
root LP value	0	5	8	+20	+29	+66	+30	+41	+94	18	+42	+51	+94	6	0	0	0
inference	0	8	7	+8	+6	−19	−7	−12	−30	16	−10	−15	−30	8	0	0	0

Table B.45. Evaluation of child selection on test set ALU.

				all instances (16)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—	16	0	0	0
down	0	1	4	+2	+2	+5	+3	+4	+11	9	+5	+5	+11	7	+1	+2	+2
up	0	4	1	−14	−11	−6	−7	−7	−11	12	−9	−9	−11	4	0	0	0
pseudocost	0	2	4	0	−2	+10	+6	+4	+24	11	+8	+5	+24	5	+2	+3	+3
LP value	0	1	6	−5	−1	+8	+6	+6	+20	11	+7	+8	+20	5	+2	+3	+3
root LP value	0	4	2	−11	−9	0	−4	−6	−2	12	−6	−7	−2	4	0	0	0
inference	0	1	0	−2	−2	+1	−1	−2	+3	8	−4	−3	+3	8	+1	+1	+1

Table B.46. Evaluation of child selection on test set FCTP.

				all instances (7)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—	7	0	0	0
down	0	2	2	−10	−13	0	−18	−19	−39	4	−29	−29	−40	3	−2	−2	−2
up	0	2	2	+24	+35	+326	−1	−1	+28	4	+1	+1	+29	3	−3	−3	−3
pseudocost	0	1	3	+37	+64	+372	+19	+20	+100	4	+37	+37	+103	3	−2	−2	−2
LP value	0	2	0	−22	−18	0	−15	−15	−24	4	−23	−23	−24	3	−3	−3	−3
root LP value	0	1	2	−6	+21	+175	+12	+13	+56	4	+23	+24	+58	3	−1	−1	−1
inference	0	2	2	+19	+27	+159	−7	−7	−16	4	−11	−11	−17	3	−1	−1	−1

Table B.47. Evaluation of child selection on test set ACC.

				all instances (20)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—	20	0	0	0
down	0	0	2	+4	+4	+5	0	+1	+2	15	+1	+1	+2	5	−2	−2	−2
up	0	3	2	−2	−1	−9	−2	−3	−5	15	−3	−4	−6	5	−1	−1	−1
pseudocost	0	4	2	−6	−9	−25	−4	−5	−8	15	−4	−6	−9	5	−2	−2	−2
LP value	0	1	1	+1	+1	−13	−2	−3	−5	15	−2	−3	−5	5	−1	−1	−1
root LP value	0	2	2	−5	−7	−17	−3	−4	−8	15	−4	−5	−8	5	−1	−1	−1
inference	0	0	1	+4	+3	−3	0	−1	−2	14	0	−1	−2	6	−1	−1	−1

Table B.48. Evaluation of child selection on test set FC.

				all instances (23)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—	23	0	0	0
down	0	7	6	+8	+5	−2	−1	−2	−9	23	−1	−2	−9	0	—	—	—
up	0	7	9	−3	−2	+10	+3	+3	+20	23	+3	+3	+20	0	—	—	—
pseudocost	0	4	7	+7	+5	+18	+6	+6	+28	23	+6	+6	+28	0	—	—	—
LP value	0	4	5	+2	0	+1	+1	+1	+8	23	+1	+1	+8	0	—	—	—
root LP value	0	4	10	+8	+7	+8	+6	+6	+6	23	+6	+6	+6	0	—	—	—
inference	0	3	9	+4	+3	+3	+4	+4	+2	22	+4	+4	+2	1	+3	+3	+3

Table B.49. Evaluation of child selection on test set ARCSET.

				all instances (41)							different path		
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—
down	0	24	9	+13	+13	+4	−9	−8	−15	41	−9	−8	−15
up	0	6	18	−5	−5	−4	+8	+8	+9	41	+8	+8	+9
pseudocost	0	4	23	+12	+12	+18	+19	+19	+34	41	+19	+19	+34
LP value	0	1	24	+20	+20	+19	+17	+17	+24	41	+17	+17	+24
root LP value	0	11	14	+12	+12	+25	+1	+2	+15	41	+1	+2	+15
inference	0	9	3	−5	−5	−3	−1	−1	+2	41	−1	−1	+2

Table B.50. Evaluation of child selection on test set MIK.

DOMAIN PROPAGATION

				all instances (30)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	1	0	0	0	0	0	0	0	0	0	—	—	—	29	0	0	0
none	1	5	6	+12	+1	−16	+5	+6	+1	27	+6	+6	+2	2	−2	−1	−1
aggr linear	1	2	5	0	0	+2	+3	+3	+9	17	+5	+5	+16	12	+1	+1	+1
no obj prop	1	6	1	−4	−4	−11	−3	−3	−8	13	−6	−6	−16	16	−1	−1	−3
no root redcost	1	2	1	−1	−3	−7	−2	−2	−5	16	−3	−4	−8	13	−1	−1	−1

Table B.51. Evaluation of domain propagation on test set MIPLIB.

				all instances (38)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	3	0	0	0	0	0	0	0	0	0	—	—	—	35	0	0	0
none	5	11	12	+11	+12	+15	+15	+15	+33	31	+8	+8	+19	2	−2	−2	−1
aggr linear	3	14	6	−12	−10	−29	−7	−8	−9	28	−10	−11	−19	7	+1	+1	+2
no obj prop	3	7	2	−5	−5	−1	−4	−4	0	20	−7	−7	+1	15	0	0	−2
no root redcost	2	6	1	−9	−8	−8	−7	−7	−11	16	−9	−9	−2	19	0	0	−1

Table B.52. Evaluation of domain propagation on test set CORAL.

				all instances (36)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	7	0	0	0	0	0	0	0	0	0	—	—	—	29	0	0	0
none	6	6	10	+35	+20	−32	+18	+13	−3	23	+31	+22	−7	6	0	0	+1
aggr linear	5	7	3	−11	−9	−4	−4	−3	−10	15	−1	+1	−4	14	+1	+1	+1
no obj prop	6	3	3	+28	+15	−37	+7	+5	−5	8	+63	+53	+18	21	+1	+1	+1
no root redcost	8	1	6	+1	0	−1	+2	+1	+1	13	+1	0	−4	15	+2	+2	+1

Table B.53. Evaluation of domain propagation on test set MILP.

				all instances (7)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—	7	0	0	0
none	1	0	5	+185	+183	+118	+167	+104	+142	5	+238	+140	+184	1	0	0	0
aggr linear	0	5	0	−61	−59	−54	−46	−42	−65	6	−51	−45	−65	1	0	0	0
no obj prop	0	0	1	+2	+2	0	0	0	−4	4	+2	+2	−2	3	−2	−2	−5
no root redcost	0	0	0	0	0	0	−1	−1	−1	0	—	—	—	7	−1	−1	−1

Table B.54. Evaluation of domain propagation on test set ENLIGHT.

				all instances (24)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—	24	0	0	0
none	2	0	16	+287	+283	+583	+209	+222	+508	18	+255	+254	+411	4	0	0	0
aggr linear	0	13	3	−26	−24	−64	−30	−36	−66	19	−36	−40	−66	5	0	0	0
no obj prop	0	0	0	0	0	0	0	0	0	0	—	—	—	24	0	0	0
no root redcost	0	0	0	0	0	0	+2	+1	+2	0	—	—	—	24	+2	+1	+2

Table B.55. Evaluation of domain propagation on test set ALU.

				all instances (16)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—	16	0	0	0
none	0	4	1	−3	−5	−3	−3	−7	−9	13	−3	−7	−9	3	0	0	0
aggr linear	0	0	0	−2	−2	−2	0	0	0	7	0	0	0	9	+1	+1	+1
no obj prop	0	0	0	0	0	0	0	0	0	0	—	—	—	16	0	0	0
no root redcost	0	0	0	0	0	0	+1	0	0	7	+1	0	0	9	0	0	0

Table B.56. Evaluation of domain propagation on test set FCTP.

				all instances (7)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—	7	0	0	0
none	1	1	2	+7	+25	+239	+29	+31	+142	3	−1	−1	+53	3	−2	−2	−2
aggr linear	0	2	2	−2	−5	+37	−18	−18	−34	4	−28	−28	−35	3	−2	−2	−2
no obj prop	0	0	0	0	0	0	−2	−1	−1	0	—	—	—	7	−2	−1	−1
no root redcost	0	0	0	0	0	0	−1	−1	−1	0	—	—	—	7	−1	−1	−1

Table B.57. Evaluation of domain propagation on test set ACC.

				all instances (20)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—	20	0	0	0
none	0	0	15	+94	+75	+184	+33	+42	+79	20	+33	+42	+79	0	—	—	—
aggr linear	0	0	3	−1	0	−1	+3	+3	+4	14	+4	+4	+4	6	+3	+3	+3
no obj prop	0	0	13	+73	+63	+209	+31	+40	+87	19	+33	+42	+89	1	+2	+2	+2
no root redcost	0	0	0	+3	+4	+1	+3	+3	+3	9	+3	+3	+3	11	+2	+2	+3

Table B.58. Evaluation of domain propagation on test set FC.

				all instances (23)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—	23	0	0	0
none	0	3	11	+12	+10	+21	+13	+13	+18	23	+13	+13	+18	0	—	—	—
aggr linear	0	5	6	+4	+2	−2	+1	+1	0	23	+1	+1	0	0	—	—	—
no obj prop	0	4	4	+1	+1	−1	+1	+1	0	11	+1	+1	−1	12	+1	+1	+1
no root redcost	0	3	1	−3	−4	−5	0	0	+3	11	−5	−4	+1	12	+4	+3	+3

Table B.59. Evaluation of domain propagation on test set ARCSET.

				all instances (41)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—	41	0	0	0
none	0	14	22	+66	+66	+94	+17	+19	+57	41	+17	+19	+57	0	—	—	—
aggr linear	0	27	3	−34	−34	−42	−15	−15	−23	41	−15	−15	−23	0	—	—	—
no obj prop	0	10	15	+38	+38	+94	+17	+21	+66	41	+17	+21	+66	0	—	—	—
no root redcost	0	1	0	−1	−1	−1	+1	+1	+1	7	−2	−2	−1	34	+1	+1	+1

Table B.60. Evaluation of domain propagation on test set MIK.

CUTTING PLANE SEPARATION

				all instances (30)							different path				equal path		
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	1	0	0	0	0	0	0	0	0	0	—	—	—	29	0	0	0
none	6	6	22	+6599	+2958	+159	+592	+443	+220	24	+335	+273	+195	0	—	—	—
no knap	1	5	4	+19	+13	+1	+3	+5	+1	13	+11	+14	+3	16	−2	−2	−2
no c-MIR	1	8	6	+302	+157	0	+51	+35	+13	13	+167	+133	+89	16	−2	−1	−1
no Gom	1	17	4	+11	−1	−15	−18	−15	−19	24	−22	−19	−31	5	−5	−3	−3
no SCG	1	10	1	−10	−2	−4	−6	−5	−4	12	−13	−11	−14	17	−2	−1	−1
no flow	1	13	3	+32	+3	−16	−14	−9	−7	17	−19	−13	−31	12	−8	−6	−2
no impl	1	2	4	+36	+10	0	+1	+1	+5	9	+7	+9	+29	20	−2	−2	−2
no cliq	1	4	2	+15	+7	+1	−6	−6	+3	7	−19	−17	+19	22	−2	−2	−1
no rdcost	3	2	13	+38	+20	+28	+22	+21	+38	25	+21	+20	+30	2	−2	−2	−2
cons Gom	1	15	3	+14	+2	−15	−8	−6	−15	24	−11	−8	−25	5	+4	+3	+3

Table B.61. Evaluation of cut separation on test set MIPLIB.

				all instances (37)							different path				equal path		
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	3	0	0	0	0	0	0	0	0	0	—	—	—	34	0	0	0
none	9	20	13	+342	+290	+164	+48	+51	+106	27	−4	−2	+45	1	−46	−46	−46
no knap	3	4	2	+15	+6	+32	+4	+4	+15	7	+24	+27	+158	27	−1	−1	0
no c-MIR	2	9	7	+16	+15	+7	+4	+3	+18	19	+44	+41	+109	15	0	0	+2
no Gom	4	14	11	+64	+54	+29	+10	+12	+31	31	+9	+11	+56	2	−2	−2	−1
no SCG	3	9	3	−1	+1	−19	−2	−3	−4	12	−2	−6	−19	21	0	0	+1
no flow	3	14	4	+15	+6	+14	−3	−1	+16	17	−1	+1	+61	17	−5	−4	−2
no impl	2	5	5	−19	−19	−20	−12	−12	−14	12	−3	−2	+13	22	+2	+2	+2
no cliq	3	3	4	+17	+12	+2	+1	+2	+3	9	+4	+7	+41	25	0	0	0
no rdcost	3	6	9	+2	+3	+29	+1	+1	+6	28	+1	+1	+15	6	+2	+2	+3
cons Gom	2	17	8	+3	0	−14	−16	−15	−17	32	−14	−13	−7	2	−1	−1	0

Table B.62. Evaluation of cut separation on test set CORAL.

				all instances (34)							different path				equal path		
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	6	0	0	0	0	0	0	0	0	0	—	—	—	28	0	0	0
none	6	13	14	+191	+81	−2	+8	+6	−5	25	+15	+12	+4	1	−3	−3	−3
no knap	6	0	3	+9	+9	+1	+4	+4	+2	3	+52	+52	+114	25	0	0	0
no c-MIR	6	7	7	+66	+48	+10	+20	+18	+8	12	+82	+75	+77	15	−7	−6	−5
no Gom	6	10	8	+29	+3	+6	−3	−4	−7	21	−10	−12	−25	5	+1	+1	+1
no SCG	5	7	3	+4	−5	−1	−7	−7	−9	11	−17	−19	−41	17	0	0	−1
no flow	4	7	1	−4	−3	−5	−4	−4	−8	8	−10	−9	−31	20	0	0	+1
no impl	4	5	1	−3	−4	−12	−8	−9	−16	8	−7	−8	−22	20	−1	−1	−1
no cliq	5	5	5	+22	+5	+1	−3	−7	−8	12	−11	−20	−46	16	+3	+2	+2
no rdcost	5	5	6	+5	+12	−9	+5	+5	+2	20	+9	+9	+8	8	−1	0	+1
cons Gom	4	11	8	0	−4	−9	−5	−5	−16	23	−1	0	−14	5	+3	+2	+2

Table B.63. Evaluation of cut separation on test set MILP.

				all instances (7)						different path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—
none	0	3	1	+144	+34	−63	−35	−47	−82	7	−35	−47	−82
no knap	1	0	5	+280	+108	+103	+61	+51	+111	6	+53	+47	+94
no c-MIR	0	2	0	−21	−21	−61	−27	−33	−68	3	−52	−53	−68
no Gom	0	3	1	−18	−18	−64	−29	−40	−76	7	−29	−40	−76
no SCG	0	3	2	+14	+13	−32	+2	−13	−41	6	+2	−14	−41
no flow	0	2	1	−15	−15	−61	−18	−30	−68	7	−18	−30	−68
no impl	0	3	2	−32	−31	−78	−39	−49	−85	7	−39	−49	−85
no cliq	0	0	0	0	0	0	−2	−3	−5	0	—	—	—
no rdcost	0	1	0	+6	+6	+3	0	−1	−6	5	−1	−1	−6
cons Gom	0	1	2	+19	+19	−24	+4	−4	−41	7	+4	−4	−41

Table B.64. Evaluation of cut separation on test set ENLIGHT.

				all instances (24)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—	24	0	0	0
none	0	11	6	+17	+7	−27	−17	−21	−32	24	−17	−21	−32	0	—	—	—
no knap	0	8	7	−28	−13	+61	−9	+4	+40	24	−9	+4	+40	0	—	—	—
no c-MIR	0	7	9	−14	−14	+3	−7	−16	−6	22	−8	−17	−6	2	0	0	0
no Gom	0	10	5	−30	−25	−21	−16	−17	−27	24	−16	−17	−27	0	—	—	—
no SCG	0	5	7	−6	+4	+17	+14	+16	+26	18	+19	+20	+32	6	0	+1	+2
no flow	0	8	6	+18	+18	−16	+3	−6	−24	22	+4	−6	−25	2	−2	−2	−2
no impl	0	12	7	−55	−50	−10	−35	−34	−20	23	−36	−35	−20	1	0	0	0
no cliq	0	10	6	−10	−14	−21	−11	−15	−30	21	−13	−16	−30	3	0	0	0
no rdcost	0	4	6	+14	+14	+7	+10	+11	+6	15	+16	+15	+6	9	0	0	0
cons Gom	0	8	5	−36	−27	−2	−23	−16	−10	23	−24	−16	−10	1	0	0	0

Table B.65. Evaluation of cut separation on test set ALU.

				all instances (16)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—	16	0	0	0
none	1	4	8	+3051	+898	+1259	+103	+152	+484	15	+88	+146	+383	0	—	—	—
no knap	0	2	4	+25	+28	+71	+13	+12	+56	14	+15	+13	+56	2	0	0	0
no c-MIR	0	5	4	+30	+25	+94	−4	+9	+71	13	−4	+10	+72	3	−3	−4	−5
no Gom	0	7	5	+46	+21	+26	−9	−5	+6	16	−9	−5	+6	0	—	—	—
no SCG	0	1	0	−2	−2	−5	−1	−1	−3	1	−20	−20	−20	15	0	+1	+2
no flow	0	3	7	+53	+26	+119	+16	+14	+86	15	+17	+15	+86	1	0	0	0
no impl	0	0	0	0	0	0	0	0	0	1	0	0	0	15	0	0	0
no cliq	0	0	0	0	0	0	+4	+3	+4	0	—	—	—	16	+4	+3	+4
no rdcost	0	2	2	+4	+3	+16	+1	0	+9	13	+1	0	+9	3	0	0	0
cons Gom	0	6	3	0	+10	−4	−9	−8	−15	15	−9	−8	−15	1	0	0	0

Table B.66. Evaluation of cut separation on test set FCTP.

				all instances (7)						different path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—
none	1	4	3	+285	+164	+593	−9	+26	+106	6	−35	−5	+7
no knap	0	0	0	0	0	0	−1	−1	−1	0	—	—	—
no c-MIR	0	0	0	0	0	0	−3	−2	−2	0	—	—	—
no Gom	0	6	1	+42	−22	−41	−30	−28	−43	7	−30	−28	−43
no SCG	0	4	3	+117	+31	+61	+13	+13	−1	7	+13	+13	−1
no flow	0	0	0	0	0	0	−2	−2	−1	0	—	—	—
no impl	1	0	2	+82	+98	+418	+36	+38	+102	1	+30	+30	+30
no cliq	1	0	6	+903	+304	+383	+155	+150	+161	6	+150	+145	+140
no rdcost	0	2	1	+2	+9	+103	−7	−6	−5	3	−11	−11	−5
cons Gom	0	4	3	+293	+45	+57	+10	+8	−13	7	+10	+8	−13

Table B.67. Evaluation of cut separation on test set ACC.

setting	T	fst	slw	all instances (19) n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	different path t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—
none	6	0	19	+170321	+81239	+112359	+2382	+2433	+11127	13	+797	+826	+2767
no knap	0	2	9	+46	+39	+51	+12	+11	+16	19	+12	+11	+16
no c-MIR	0	8	9	+1409	+855	+1442	+42	+99	+269	19	+42	+99	+269
no Gom	0	13	2	+11	−7	−18	−18	−17	−21	19	−18	−17	−21
no SCG	0	2	6	+15	+16	+24	+5	+5	+6	10	+10	+8	+9
no flow	0	9	8	+310	+169	+145	+8	+22	+32	19	+8	+22	+32
no impl	0	1	4	+14	+17	+14	+4	+5	+6	10	+7	+8	+8
no cliq	0	0	0	0	0	0	0	0	0	0	—	—	—
no rdcost	0	2	2	0	+1	+10	+1	+1	+2	19	+1	+1	+2
cons Gom	0	8	5	−9	−11	−26	−7	−9	−12	19	−7	−9	−12

Table B.68. Evaluation of cut separation on test set FC.

setting	T	fst	slw	all instances (23) n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	different path t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—
none	3	9	11	+860	+717	+1509	+104	+104	+241	20	+20	+17	−23
no knap	0	2	0	0	0	0	−3	−2	−1	0	—	—	—
no c-MIR	0	7	11	+176	+141	+369	+46	+50	+97	23	+46	+50	+97
no Gom	0	11	10	+110	+96	+78	+17	+17	−6	23	+17	+17	−6
no SCG	0	4	5	+14	+4	−7	+1	+1	−8	12	0	0	−18
no flow	0	6	7	−6	−9	−9	−6	−5	−7	16	−9	−8	−10
no impl	0	2	6	+43	+28	+12	+13	+12	−3	11	+25	+23	−9
no cliq	0	0	0	0	0	0	0	0	−1	0	—	—	—
no rdcost	0	4	9	+30	+27	+27	+12	+11	−3	23	+12	+11	−3
cons Gom	0	7	9	+74	+57	+51	+8	+9	+13	23	+8	+9	+13

Table B.69. Evaluation of cut separation on test set ARCSET.

setting	T	fst	slw	all instances (41) n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	different path t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—
none	41	0	41	+8597	+8575	+3751	+11976	+10606	+5543	0	—	—	—
no knap	0	0	0	0	0	0	0	0	0	0	—	—	—
no c-MIR	0	0	35	+211	+211	+226	+137	+131	+188	39	+148	+142	+198
no Gom	0	4	33	+64	+64	+115	+46	+46	+104	41	+46	+46	+104
no SCG	0	5	12	+5	+5	+2	+6	+6	+4	26	+7	+8	+9
no flow	0	14	21	+39	+39	+131	+21	+32	+114	41	+21	+32	+114
no impl	0	2	19	+32	+32	+8	+29	+24	+11	26	+46	+42	+38
no cliq	0	0	0	0	0	0	−2	−2	−3	0	—	—	—
no rdcost	3	0	41	+238	+237	+471	+431	+404	+767	38	+411	+399	+802
cons Gom	0	7	23	+23	+23	+57	+15	+17	+49	41	+15	+17	+49

Table B.70. Evaluation of cut separation on test set MIK.

				all instances (30)							different path				equal path		
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
none	6	0	0	0	0	0	0	0	0	0	—	—	—	24	0	0	0
knapsack	5	8	1	−53	−51	−17	−41	−35	−16	11	−73	−67	−60	13	0	0	0
c-MIR	3	12	0	−92	−89	−50	−79	−72	−45	10	−97	−95	−95	14	+2	+2	+2
Gomory	5	9	9	−75	−70	−28	−53	−50	−32	19	−48	−49	−48	4	+4	+4	+4
strong CG	5	7	6	−52	−44	−12	−30	−28	−18	12	−26	−30	−57	12	+3	+3	+3
flow cover	4	10	2	−61	−57	−26	−45	−40	−22	13	−74	−69	−63	11	+2	+2	+1
impl bds	5	5	1	−23	−23	−3	−17	−16	−11	8	−42	−41	−28	16	0	0	0
clique	6	2	2	−12	−7	−1	0	0	0	6	−4	−4	−5	18	+1	+1	+1
redcost	4	13	3	−23	−24	−20	−23	−20	−17	21	−27	−22	−24	3	+2	+2	+3
cons Gom	4	10	3	−67	−61	−23	−53	−48	−30	14	−62	−58	−45	10	+5	+3	+1

Table B.71. Evaluation of only using individual cut separators on test set MIPLIB.

				all instances (37)							different path				equal path		
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
none	9	0	0	0	0	0	0	0	0	0	—	—	—	28	0	0	0
knapsack	10	1	2	+16	+15	0	+16	+15	+9	4	+6	+6	+8	23	+2	+2	+2
c-MIR	6	7	6	−39	−39	−32	−17	−18	−26	13	−16	−16	−40	15	+6	+5	+4
Gomory	6	13	12	−50	−49	−29	−24	−25	−31	27	−1	−3	−32	1	+2	+2	+2
strong CG	9	2	8	+20	+19	+2	+14	+13	0	11	+51	+47	+2	17	+2	+2	+2
flow cover	7	4	9	−10	−10	−34	+9	+7	−13	13	+59	+53	−12	15	+5	+5	+3
impl bds	9	7	3	−15	−15	−11	−4	−5	−6	10	−29	−30	−58	17	+3	+2	+3
clique	8	1	3	−11	−11	−2	−1	−1	−6	5	+127	+123	+84	23	+2	+2	+2
redcost	9	9	6	−14	−15	−22	−5	−5	−2	21	−8	−9	−16	6	+1	+1	+3
cons Gom	5	11	9	−37	−36	−20	−26	−27	−37	19	−7	−8	−37	9	+2	+2	+2

Table B.72. Evaluation of only using individual cut separators on test set CORAL.

				all instances (37)							different path				equal path		
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
none	8	0	0	0	0	0	0	0	0	0	—	—	—	29	0	0	0
knapsack	8	1	2	+2	+2	−1	+4	+5	+3	5	+30	+35	+76	24	+1	+1	+1
c-MIR	8	5	6	−33	−29	−6	−12	−12	−5	12	−26	−27	−4	16	+1	+1	+1
Gomory	7	8	13	−17	−5	−22	+5	+4	+4	21	+49	+47	+64	7	0	0	0
strong CG	8	6	6	−9	−4	+1	+2	+2	+2	16	+3	+4	+13	13	0	0	0
flow cover	8	3	2	+2	+2	−21	−4	−4	−3	4	−48	−51	−72	24	−1	−1	−1
impl bds	8	3	4	−9	−8	+4	−1	−2	0	9	−4	−9	−3	19	−2	−1	−2
clique	8	3	6	−17	−4	0	+7	+11	+8	12	+23	+40	+80	17	0	0	0
redcost	8	9	5	−8	−8	+45	−2	−2	+3	23	−7	−6	−4	5	0	0	0
cons Gom	8	5	13	−15	−1	+1	+8	+8	+7	18	+27	+25	+33	10	+1	+2	+3

Table B.73. Evaluation of only using individual cut separators on test set MILP.

				all instances (7)						different path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
none	0	0	0	0	0	0	0	0	0	0	—	—	—
knapsack	0	0	0	0	0	0	+2	0	0	0	—	—	—
c-MIR	0	3	0	−53	−53	−83	−51	−65	−83	3	−82	−81	−85
Gomory	0	3	2	−49	−7	+164	+27	+52	+238	7	+27	+52	+238
strong CG	0	4	1	−89	−61	−60	−25	−35	−57	7	−25	−35	−57
flow cover	0	3	1	−42	−41	−70	−37	−50	−71	7	−37	−50	−71
impl bds	0	0	0	0	0	0	+3	+2	+2	0	—	—	—
clique	0	0	0	0	0	0	+1	0	−1	0	—	—	—
redcost	0	0	3	+14	+14	+42	+24	+30	+58	5	+35	+38	+59
cons Gom	0	2	2	−62	−30	+3	+3	+9	+19	7	+3	+9	+19

Table B.74. Evaluation of only using individual cut separators on test set ENLIGHT.

				all instances (25)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
none	1	0	0	0	0	0	0	0	0	0	—	—	—	24	0	0	0
knapsack	1	5	7	−24	−14	−7	−9	−3	−10	24	−9	−3	−31	0	—	—	—
c-MIR	1	8	7	−51	−39	+16	−25	−10	+11	24	−26	−12	+32	0	—	—	—
Gomory	2	7	8	−33	−21	+48	−3	+16	+65	19	−18	−2	+7	4	−1	−1	−2
strong CG	1	7	5	−68	−57	−4	−39	−25	−6	20	−47	−34	−23	4	0	0	0
flow cover	1	6	8	−6	−2	+1	−6	−7	−7	22	−6	−8	−21	2	−3	−3	−4
impl bds	1	7	9	−22	−17	+9	−7	0	+6	24	−7	0	+17	0	—	—	—
clique	1	8	8	−38	−30	−6	−23	−14	−11	24	−23	−16	−33	0	—	—	—
redcost	1	4	4	+4	+4	0	+6	+8	+2	16	+9	+11	+7	8	0	0	0
cons Gom	1	8	6	−60	−54	−4	−41	−32	−16	20	−48	−44	−64	4	−1	−2	−3

Table B.75. Evaluation of only using individual cut separators on test set ALU.

				all instances (16)						different path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
none	1	0	0	0	0	0	0	0	0	0	—	—	—
knapsack	0	9	1	−84	−75	−59	−50	−52	−44	15	−52	−58	−75
c-MIR	1	7	2	−46	−45	−34	−18	−22	−12	11	−25	−29	−31
Gomory	1	0	9	−20	−10	−11	+34	+15	+9	15	+37	+17	+23
strong CG	1	0	0	−1	0	−1	+5	+2	+1	1	0	0	0
flow cover	0	8	2	−92	−82	−74	−54	−57	−59	15	−54	−61	−80
impl bds	1	0	0	−1	0	−1	+1	+1	0	1	0	0	0
clique	1	0	0	0	0	−2	+5	+2	+1	0	—	—	—
redcost	1	0	0	0	0	0	+1	+1	0	0	—	—	—
cons Gom	1	3	6	−24	−15	−10	+7	−3	+2	15	+8	−3	+6

Table B.76. Evaluation of only using individual cut separators on test set FCTP.

				all instances (7)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
none	1	0	0	0	0	0	0	0	0	0	—	—	—	6	0	0	0
knapsack	1	0	0	0	0	0	−1	−1	−1	0	—	—	—	6	−1	−1	−1
c-MIR	1	0	0	0	0	−2	+3	+2	+1	0	—	—	—	6	+3	+2	+3
Gomory	1	1	5	+19	+22	+19	+110	+54	+31	6	+137	+67	+73	0	—	—	—
strong CG	1	2	4	−52	−50	−28	+11	−15	−15	6	+13	−18	−36	0	—	—	—
flow cover	1	0	0	0	0	+1	0	0	0	0	—	—	—	6	0	0	−1
impl bds	0	2	0	−53	−58	−78	−42	−43	−57	1	−93	−93	−93	5	−1	−1	−1
clique	0	3	4	−70	−50	−68	−32	−39	−33	6	−24	−33	+10	0	—	—	—
redcost	0	1	0	−6	−8	−27	−3	−3	−10	0	—	—	—	6	0	0	0
cons Gom	1	0	3	+5	+2	−1	+27	+9	+1	3	+73	+31	+22	3	+1	+1	+1

Table B.77. Evaluation of only using individual cut separators on test set ACC.

				all instances (20)						different path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
none	7	0	0	0	0	0	0	0	0	0	—	—	—
knapsack	0	20	0	−95	−95	−96	−93	−90	−96	13	−89	−84	−92
c-MIR	0	20	0	−99	−99	−99	−96	−95	−98	13	−88	−86	−94
Gomory	5	8	6	−38	−38	−35	−8	−7	−13	13	−3	−1	−19
strong CG	7	0	0	0	0	−1	+2	+2	0	0	—	—	—
flow cover	1	18	1	−75	−75	−74	−60	−55	−65	13	−51	−44	−60
impl bds	7	1	1	−1	−1	−2	+2	+1	0	2	+4	+4	−4
clique	7	0	0	−1	−1	−2	+3	+2	0	0	—	—	—
redcost	7	4	2	−5	−5	−1	−2	−2	−1	13	−3	−4	−8
cons Gom	5	8	2	−25	−25	−18	−11	−7	−7	13	−14	−9	−25

Table B.78. Evaluation of only using individual cut separators on test set FC.

				all instances (23)						different path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
none	3	0	0	0	0	0	0	0	0	0	—	—	—
knapsack	3	0	0	+1	+1	+4	−2	−2	−1	0	—	—	—
c-MIR	0	14	5	−56	−56	−87	−36	−38	−76	20	−9	−9	−10
Gomory	1	12	6	−67	−66	−39	−39	−36	−29	20	−30	−26	+22
strong CG	2	10	3	−52	−51	−19	−33	−28	−16	13	−46	−39	+24
flow cover	3	5	7	−6	−6	+3	+6	+7	+3	17	+9	+9	+14
impl bds	3	2	4	0	0	+7	0	0	0	8	+2	+2	+5
clique	3	0	0	+1	+1	+4	−2	−2	0	0	—	—	—
redcost	3	5	10	−11	−11	−6	+17	+17	+19	19	+8	+8	+8
cons Gom	2	13	4	−65	−64	−29	−49	−45	−29	20	−44	−40	−14

Table B.79. Evaluation of only using individual cut separators on test set ARCSET.

				all instances (41)						different path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
none	41	0	0	0	0	0	0	0	0	0	—	—	—
knapsack	41	0	0	−1	−1	−1	0	0	0	0	—	—	—
c-MIR	41	0	0	+2	+2	+2	0	0	0	0	—	—	—
Gomory	12	29	0	−69	−69	−56	−69	−69	−52	0	—	—	—
strong CG	41	0	0	−3	−3	−2	0	0	0	0	—	—	—
flow cover	10	31	0	−62	−62	−47	−66	−65	−51	0	—	—	—
impl bds	41	0	0	+1	+1	+2	0	0	0	0	—	—	—
clique	41	0	0	+2	+2	+2	0	0	0	0	—	—	—
redcost	33	8	0	−19	−19	+10	−45	−44	−18	0	—	—	—
cons Gom	10	30	0	−66	−66	−52	−67	−67	−51	0	—	—	—

Table B.80. Evaluation of only using individual cut separators on test set MIK.

				all instances (30)							different path				equal path		
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	1	0	0	0	0	0	0	0	0	0	—	—	—	29	0	0	0
all (1)	7	0	24	−65	−63	−96	+182	+162	+178	19	+195	+173	+317	4	−2	−1	−1
all (1*)	2	0	18	−31	−29	−73	+47	+45	+39	23	+57	+55	+50	5	+19	+24	+129
all (10*)	1	1	6	−6	−7	−36	+8	+7	+9	16	+13	+13	+16	13	+2	+1	+1
impl bds (1*)	1	2	0	−7	−7	−10	−4	−4	−7	9	−10	−10	−21	20	−1	−1	−1
knapsack (1*)	1	3	1	−8	−5	−8	−2	−3	−6	7	−8	−9	−41	22	−1	−1	−1
impl/knap (1*)	1	4	1	−12	−10	−14	−3	−3	−8	13	−7	−8	−26	16	0	0	0

Table B.81. Evaluation of local cut separation on test set MIPLIB.

				all instances (38)							different path				equal path		
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	3	0	0	0	0	0	0	0	0	0	—	—	—	35	0	0	0
all (1)	8	3	29	−78	−73	−82	+156	+171	+122	28	+160	+185	+183	2	+3	+3	+3
all (1*)	3	4	23	−54	−45	−68	+19	+28	+27	32	+24	+36	+62	2	−3	−3	−2
all (10*)	3	6	6	−19	−13	−41	+2	+2	0	27	+2	+3	−1	8	0	0	+1
impl bds (1*)	2	2	3	−6	−5	−7	−1	−1	−9	15	+3	+4	+11	20	+1	+1	+1
knapsack (1*)	3	5	1	−6	−5	−15	−3	−3	0	10	−12	−11	−1	25	+1	0	0
impl/knap (1*)	2	5	2	−13	−12	−23	−7	−7	−12	19	−9	−8	−1	16	+2	+1	0

Table B.82. Evaluation of local cut separation on test set CORAL.

				all instances (36)							different path				equal path		
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	7	0	0	0	0	0	0	0	0	0	—	—	—	29	0	0	0
all (1)	14	1	17	−68	−70	−89	+103	+95	+55	14	+175	+151	+103	8	+1	+1	0
all (1*)	9	2	15	−37	−38	−74	+41	+37	+24	18	+61	+53	+39	9	+4	+4	+10
all (10*)	8	0	9	−8	−10	−44	+18	+17	+14	15	+34	+34	+40	13	+3	+2	+3
impl bds (1*)	7	3	2	0	0	−2	0	−1	−3	8	+1	−3	−20	21	0	0	0
knapsack (1*)	6	1	2	−4	−4	−25	0	0	−3	2	+21	+21	+19	27	0	0	0
impl/knap (1*)	7	2	3	−5	−5	−28	+2	+1	−2	9	+2	−1	−16	20	+3	+3	+4

Table B.83. Evaluation of local cut separation on test set MILP.

				all instances (7)							different path				equal path		
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—	7	0	0	0
all (1)	2	0	6	−96	−95	−99	+270	+136	+198	4	+457	+279	+385	1	0	0	0
all (1*)	0	0	6	−54	−53	−73	+132	+66	+103	6	+167	+76	+104	1	0	0	0
all (10*)	0	1	5	+5	+5	−20	+27	+11	0	6	+32	+12	0	1	0	0	0
impl bds (1*)	0	2	0	−34	−33	−54	−14	−18	−40	6	−16	−20	−40	1	0	0	0
knapsack (1*)	0	1	2	−11	−11	−40	−7	−13	−42	6	−8	−14	−42	1	0	0	0
impl/knap (1*)	0	2	2	−30	−30	−12	−5	−6	+31	6	−6	−6	+31	1	0	0	0

Table B.84. Evaluation of local cut separation on test set ENLIGHT.

				all instances (22)							different path				equal path		
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—	22	0	0	0
all (1)	2	2	13	−51	−47	−47	+201	+196	+537	15	+290	+249	+510	5	0	0	0
all (1*)	0	1	14	−4	+1	−3	+37	+29	+37	17	+50	+34	+37	5	0	0	0
all (10*)	0	2	3	+3	+3	+3	+2	+2	+3	13	+4	+3	+3	9	0	0	0
impl bds (1*)	0	4	8	−15	−17	+13	−10	−13	+7	17	−13	−14	+7	5	0	0	0
knapsack (1*)	0	6	4	−10	−16	−18	−16	−25	−27	17	−20	−28	−27	5	0	0	0
impl/knap (1*)	0	6	8	+26	+21	+17	+26	+14	+31	17	+35	+16	+31	5	0	0	0

Table B.85. Evaluation of local cut separation on test set ALU.

				all instances (16)							different path				equal path		
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—	16	0	0	0
all (1)	0	0	12	−76	−72	−92	+115	+123	+310	12	+177	+158	+311	4	0	0	0
all (1*)	0	2	9	−26	−22	−40	+25	+21	+35	12	+35	+25	+35	4	0	0	0
all (10*)	0	0	3	−1	−1	−5	+7	+8	+8	9	+11	+10	+8	7	+1	+2	+2
impl bds (1*)	0	0	0	0	0	0	+1	+1	+2	3	+1	+1	+1	13	+1	+1	+2
knapsack (1*)	0	2	0	−12	−9	−5	−4	−5	−3	11	−7	−7	−3	5	0	0	+1
impl/knap (1*)	0	2	0	−12	−8	−5	−4	−5	−2	11	−6	−6	−2	5	0	0	+1

Table B.86. Evaluation of local cut separation on test set FCTP.

				all instances (7)							different path				equal path		
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—	7	0	0	0
all (1)	0	2	2	−46	−45	−2	+8	+9	+57	4	+16	+17	+58	3	−2	−2	−2
all (1*)	1	2	2	−44	−41	+25	+8	+9	+66	3	−34	−34	−43	3	−3	−3	−3
all (10*)	0	2	1	−21	−23	−20	−1	−1	−3	4	0	0	−3	3	−2	−2	−2
impl bds (1*)	0	0	1	+21	+43	+288	+27	+28	+75	2	+124	+127	+307	5	+1	+1	+1
knapsack (1*)	0	0	0	0	0	0	−3	−2	−2	0	—	—	—	7	−3	−2	−2
impl/knap (1*)	0	0	1	+21	+43	+288	+24	+25	+71	2	+120	+122	+301	5	−2	−2	−2

Table B.87. Evaluation of local cut separation on test set ACC.

				all instances (20)							different path				equal path		
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—	20	0	0	0
all (1)	0	0	13	−72	−78	−91	+53	+62	+86	15	+77	+77	+93	5	0	0	0
all (1*)	0	1	6	−37	−36	−51	+6	+6	+5	15	+8	+7	+6	5	−1	−1	−1
all (10*)	0	0	0	−2	−2	−1	+1	+1	+2	10	+2	+2	+3	10	0	0	0
impl bds (1*)	0	0	1	−4	−4	0	0	0	0	11	0	0	+1	9	0	0	0
knapsack (1*)	0	0	1	−10	−7	0	0	+1	+1	8	+1	+2	+4	12	0	0	0
impl/knap (1*)	0	0	4	−13	−9	−1	+6	+6	+7	14	+6	+6	+8	6	+3	+3	+4

Table B.88. Evaluation of local cut separation on test set FC.

				all instances (23)							different path				equal path		
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—	23	0	0	0
all (1)	4	0	23	−47	−45	−47	+681	+631	+600	19	+697	+671	+1223	0	—	—	—
all (1*)	0	1	22	−10	−11	−16	+75	+69	+106	22	+75	+70	+106	1	+65	+65	+65
all (10*)	0	0	4	−6	−6	−6	+6	+6	+14	13	+8	+8	+15	10	+2	+2	+3
impl bds (1*)	0	2	1	0	−1	−10	+1	+1	−3	11	+1	0	−8	12	+1	+1	+1
knapsack (1*)	0	1	2	−3	−3	−1	+7	+7	+7	1	−40	−40	−40	22	+10	+10	+7
impl/knap (1*)	0	3	6	−3	−4	−11	+8	+8	+2	11	+13	+13	+3	12	+4	+3	+2

Table B.89. Evaluation of local cut separation on test set ARCSET.

setting	T	fst	slw	all instances (41) n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	different path #	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—
all (1)	0	6	29	−91	−91	−93	+63	+56	+30	41	+63	+56	+30
all (1*)	0	26	5	−69	−69	−74	−28	−29	−42	41	−28	−29	−42
all (10*)	0	21	3	−30	−30	−25	−12	−12	−13	41	−12	−12	−13
impl bds (1*)	0	0	0	0	0	0	+2	+2	+2	12	+2	+2	+2
knapsack (1*)	0	24	4	−43	−42	−38	−25	−23	−29	41	−25	−23	−29
impl/knap (1*)	0	22	11	−43	−43	−38	−20	−18	−24	41	−20	−18	−24

Table B.90. Evaluation of local cut separation on test set MIK.

CUTTING PLANE SELECTION

				all instances (30)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	1	0	0	0	0	0	0	0	0	0	—	—	—	29	0	0	0
one per round	4	2	24	+36	+3	−1	+116	+97	+116	23	+55	+40	+63	3	+45	+18	0
take all	4	2	24	+9	−8	−6	+137	+105	+110	25	+105	+72	+45	1	−2	−2	−2
no obj paral	1	3	5	+14	+2	−1	+5	+4	+10	19	+7	+7	+38	10	+1	+1	+1
no ortho	1	1	18	+8	−7	+2	+39	+28	+15	26	+46	+35	+35	3	0	0	+1

Table B.91. Evaluation of cut selection on test set MIPLIB.

				all instances (37)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	3	0	0	0	0	0	0	0	0	0	—	—	—	34	0	0	0
one per round	8	8	19	−11	+3	−17	+99	+98	+96	26	+41	+40	+27	2	+2	+2	+3
take all	4	7	19	−24	−7	−22	+39	+38	+21	31	+36	+36	+19	1	0	0	0
no obj paral	3	9	5	−4	−9	−20	−8	−8	−4	21	−9	−9	−2	12	+3	+2	+3
no ortho	4	10	19	−12	−3	−12	+18	+19	+25	30	+8	+9	+19	2	−1	0	+2

Table B.92. Evaluation of cut selection on test set CORAL.

				all instances (35)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	7	0	0	0	0	0	0	0	0	0	—	—	—	28	0	0	0
one per round	9	5	15	−16	−4	−42	+32	+27	+18	22	+47	+39	+40	2	+1	+1	+2
take all	12	7	17	+7	−1	−17	+51	+45	+34	21	+23	+15	−8	1	−3	−3	−3
no obj paral	6	8	4	−4	−8	−3	−7	−7	−10	16	−14	−14	−25	12	+1	+1	+1
no ortho	8	9	10	−25	−21	−42	+14	+11	+7	23	+14	+9	+2	3	+1	+1	+1

Table B.93. Evaluation of cut selection on test set MILP.

				all instances (7)						different path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—
one per round	1	0	5	+220	+95	+152	+97	+85	+187	6	+94	+94	+318
take all	0	1	5	−29	−29	−44	+37	−17	−34	7	+37	−17	−34
no obj paral	0	1	2	−2	−2	−14	+1	−3	−7	6	+2	−4	−7
no ortho	0	1	5	+9	+10	−2	+28	+8	−4	7	+28	+8	−4

Table B.94. Evaluation of cut selection on test set ENLIGHT.

				all instances (24)						different path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—
one per round	0	6	11	−20	−13	−16	−10	−3	−13	24	−10	−3	−13
take all	0	7	8	−29	−24	+10	+6	+1	+18	24	+6	+1	+18
no obj paral	0	8	9	+11	+18	+17	+14	+18	+22	23	+15	+19	+22
no ortho	0	11	6	−33	−25	−14	−14	−16	−17	24	−14	−16	−17

Table B.95. Evaluation of cut selection on test set ALU.

				all instances (16)						different path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—
one per round	0	1	6	−2	−10	+11	+18	+7	+28	15	+20	+7	+28
take all	0	0	14	−21	−7	−5	+170	+69	+58	16	+170	+69	+58
no obj paral	0	1	7	+14	+10	+12	+8	+6	+16	13	+10	+6	+16
no ortho	0	2	11	−7	+2	+2	+54	+17	+16	16	+54	+17	+16

Table B.96. Evaluation of cut selection on test set FCTP.

				all instances (7)						different path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—
one per round	6	0	7	+828	+258	+637	+2032	+1853	+702	1	+1393	+1393	+1393
take all	1	0	6	+231	+128	+453	+387	+351	+195	6	+363	+325	+116
no obj paral	0	1	0	−32	−32	−21	−21	−22	−27	2	−55	−56	−77
no ortho	2	0	6	+130	+85	+591	+345	+313	+218	5	+348	+312	+95

Table B.97. Evaluation of cut selection on test set ACC.

				all instances (19)						different path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—
one per round	0	0	18	+362	+140	+87	+87	+80	+86	19	+87	+80	+86
take all	0	0	19	−38	−24	−27	+822	+694	+756	19	+822	+694	+756
no obj paral	0	4	6	−2	+6	+60	+5	+7	+20	19	+5	+7	+20
no ortho	0	0	19	−3	+11	+18	+117	+102	+108	19	+117	+102	+108

Table B.98. Evaluation of cut selection on test set FC.

				all instances (23)						different path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—
one per round	0	7	12	+70	+50	+26	+37	+35	+16	23	+37	+35	+16
take all	0	5	16	−55	−42	−29	+40	+49	+53	23	+40	+49	+53
no obj paral	0	4	7	+16	+12	0	+6	+6	−6	17	+8	+7	−6
no ortho	0	6	9	+8	0	−8	+2	+1	−6	23	+2	+1	−6

Table B.99. Evaluation of cut selection on test set ARCSET.

				all instances (41)						different path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—
one per round	3	2	38	+312	+312	+680	+246	+247	+639	38	+222	+230	+546
take all	0	0	41	+27	+27	+6	+745	+676	+570	41	+745	+676	+570
no obj paral	0	5	15	+4	+4	+8	+7	+7	+13	39	+8	+8	+13
no ortho	0	9	19	−6	−6	+26	+6	+8	+26	41	+6	+8	+26

Table B.100. Evaluation of cut selection on test set MIK.

PRIMAL HEURISTICS

				all instances (30)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	1	0	0	0	0	0	0	0	0	0	—	—	—	29	0	0	0
none	2	9	13	+291	+111	+23	+39	+39	+68	27	+26	+25	+56	1	−4	−4	−4
no round	1	3	10	+79	+26	−2	+13	+10	0	22	+20	+16	+1	7	−3	−3	−4
no diving	2	3	6	−5	−3	+10	+5	+5	+19	18	+2	+2	+1	10	+2	+2	+1
no objdiving	1	5	6	+58	+30	−11	+14	+17	+32	19	+25	+27	+52	10	−2	−3	−13
no improvement	1	2	3	0	−1	−5	+1	0	−3	12	+5	+2	−6	17	−2	−2	−1

Table B.101. Evaluation of primal heuristics on test set MIPLIB.

				all instances (38)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	3	0	0	0	0	0	0	0	0	0	—	—	—	35	0	0	0
none	3	16	8	+37	+18	−31	−8	−9	−5	34	−9	−10	−10	1	+2	+2	+2
no round	2	8	8	−3	−3	−39	−3	−4	−10	25	+6	+4	+11	10	−1	−1	0
no diving	4	13	8	−6	−1	+29	−3	−3	+17	29	−7	−7	+19	5	+1	+1	+1
no objdiving	2	11	5	+4	−3	−22	−6	−6	−15	28	+1	+1	−1	7	0	0	−1
no improvement	2	4	2	−6	−6	−5	−3	−3	−5	10	−10	−10	−5	25	+2	+2	+2

Table B.102. Evaluation of primal heuristics on test set CORAL.

				all instances (36)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	7	0	0	0	0	0	0	0	0	0	—	—	—	29	0	0	0
none	8	7	9	+103	+42	−11	+10	+7	+4	23	+12	+8	+2	4	+3	+2	+2
no round	5	5	6	+26	+9	−23	+7	+5	−4	20	+13	+10	−11	9	+1	+1	+2
no diving	7	3	11	+14	+5	+5	+3	+2	−1	19	+12	+11	+7	9	+2	+1	+1
no objdiving	6	4	5	+25	+9	−1	+4	+3	0	19	+9	+7	+3	10	−2	−2	−3
no improvement	7	0	2	+7	+3	−1	+2	+2	+2	6	+10	+10	+11	23	+1	+1	+2

Table B.103. Evaluation of primal heuristics on test set MILP.

				all instances (7)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—	7	0	0	0
none	0	2	1	−14	−13	−2	−10	−8	+3	6	−12	−9	+3	1	0	0	0
no round	0	0	0	0	0	0	−4	−2	−4	0	—	—	—	7	−4	−2	−4
no diving	0	2	1	−17	−17	−19	−12	−13	−17	6	−14	−14	−17	1	0	0	0
no objdiving	0	1	3	+19	+19	+7	+15	+14	0	6	+18	+16	0	1	0	0	0
no improvement	0	0	0	0	0	0	+1	+1	0	0	—	—	—	7	+1	+1	0

Table B.104. Evaluation of primal heuristics on test set ENLIGHT.

				all instances (25)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	1	0	0	0	0	0	0	0	0	0	—	—	—	24	0	0	0
none	1	12	3	−10	−12	0	−14	−13	−2	18	−19	−17	−5	6	0	0	0
no round	1	0	0	0	0	+1	−1	−1	0	0	—	—	—	24	−1	−1	0
no diving	1	10	5	−2	−3	0	−4	−2	+1	17	−5	−3	+3	7	0	0	0
no objdiving	1	6	6	−4	−4	+22	0	+3	+26	16	0	+5	+60	8	−1	−2	−2
no improvement	1	0	0	0	0	+1	+2	0	0	0	—	—	—	24	+2	0	0

Table B.105. Evaluation of primal heuristics on test set ALU.

				all instances (16)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—	16	0	0	0
none	0	2	6	+208	+84	+19	+13	+3	−2	13	+16	+3	−2	3	0	0	0
no round	0	4	4	+71	+35	−1	+9	+1	−2	12	+12	+1	−2	4	0	0	0
no diving	0	1	6	0	0	+5	+6	+4	+3	9	+8	+6	+3	7	+3	+5	+5
no objdiving	0	3	0	−2	−2	−2	0	−2	0	8	−3	−3	0	8	+3	+4	+4
no improvement	0	1	3	+4	+4	0	+5	+4	+5	6	+8	+6	+5	10	+4	+3	+5

Table B.106. Evaluation of primal heuristics on test set FCTP.

				all instances (7)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—	7	0	0	0
none	0	2	5	+957	+151	+427	+42	+35	+48	7	+42	+35	+48	0	—	—	—
no round	0	0	0	0	0	0	−3	−3	−4	0	—	—	—	7	−3	−3	−4
no diving	0	2	1	−15	−5	+330	−9	−9	+35	3	−18	−17	+38	4	−1	−1	−1
no objdiving	0	2	5	+1094	+158	+262	+70	+62	+33	7	+70	+62	+33	0	—	—	—
no improvement	0	0	0	0	0	0	−2	−2	−2	0	—	—	—	7	−2	−2	−2

Table B.107. Evaluation of primal heuristics on test set ACC.

				all instances (20)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—	20	0	0	0
none	0	0	20	+1761	+774	+536	+83	+82	+107	20	+83	+82	+107	0	—	—	—
no round	0	1	13	+274	+139	+60	+19	+21	+24	18	+22	+23	+26	2	−6	−6	−6
no diving	0	0	0	−2	−2	−1	+3	+3	+3	10	+3	+3	+3	10	+3	+3	+3
no objdiving	0	1	0	−2	−2	−6	0	−1	−3	8	−3	−3	−4	12	+2	+2	+2
no improvement	0	2	1	−10	−10	−7	0	0	0	12	−1	−1	0	8	+2	+2	+2

Table B.108. Evaluation of primal heuristics on test set FC.

				all instances (23)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—	23	0	0	0
none	0	10	6	+72	+53	+42	+2	+2	−14	23	+2	+2	−14	0	—	—	—
no round	0	5	5	+18	+10	−9	+1	+1	−9	23	+1	+1	−9	0	—	—	—
no diving	0	4	7	+6	+5	−5	0	−1	−18	23	0	−1	−18	0	—	—	—
no objdiving	0	6	5	+5	+4	+12	+1	+1	+6	19	+1	+1	+6	4	−1	−1	0
no improvement	0	5	5	0	+1	+5	+1	+1	−1	13	+3	+3	−2	10	−1	−1	+1

Table B.109. Evaluation of primal heuristics on test set ARCSET.

				all instances (41)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—	41	0	0	0
none	2	0	41	+380	+379	+551	+524	+481	+832	39	+519	+487	+950	0	—	—	—
no round	0	16	21	+24	+24	−8	+29	+23	−1	41	+29	+23	−1	0	—	—	—
no diving	0	11	6	+10	+10	+22	0	+2	+16	41	0	+2	+16	0	—	—	—
no objdiving	0	6	10	0	0	+7	+4	+4	+11	41	+4	+4	+11	0	—	—	—
no improvement	0	1	3	+1	+1	+8	+1	+2	+10	23	+1	+2	+11	18	+1	+1	+1

Table B.110. Evaluation of primal heuristics on test set MIK.

				all instances (29)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	1	0	0	0	0	0	0	0	0	0	—	—	—	28	0	0	0
no RENS	1	4	3	+74	+19	−1	+2	+2	0	11	+8	+9	+8	17	−1	−1	−1
no simple rnd	1	2	1	−1	−1	+1	−2	−2	−3	5	−4	−5	−19	23	−2	−2	−3
no rounding	1	2	0	+3	−1	0	−2	−2	−3	6	−4	−5	−23	22	−1	−1	−2
no shifting	1	3	3	−2	+1	+2	0	−1	−8	10	+2	0	−19	18	−2	−2	−3
no int shifting	1	1	1	+2	+3	+3	+1	+2	0	10	+8	+9	+9	18	−2	−2	−2
octane	1	1	3	−1	−1	0	+3	+2	+2	1	+19	+19	+19	27	+3	+2	+2

Table B.111. Evaluation of rounding heuristics on test set MIPLIB.

				all instances (38)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	3	0	0	0	0	0	0	0	0	0	—	—	—	35	0	0	0
no RENS	3	1	3	+12	+11	−1	+8	+6	+2	8	+36	+26	+2	27	+2	+2	+3
no simple rnd	3	0	2	0	0	0	0	0	0	0	—	—	—	35	0	0	0
no rounding	3	0	0	+2	+2	−1	−1	−1	0	6	0	+1	+3	29	−1	−1	0
no shifting	3	5	2	−5	−4	−29	−3	−2	−2	11	−6	−6	−9	24	−1	−1	−1
no int shifting	3	3	2	−7	−6	+1	−2	−2	+1	12	−7	−7	+2	23	+1	+1	+1
octane	3	0	0	0	0	0	0	0	0	2	0	0	+1	33	0	0	0

Table B.112. Evaluation of rounding heuristics on test set CORAL.

				all instances (35)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	6	0	0	0	0	0	0	0	0	0	—	—	—	29	0	0	0
no RENS	6	1	2	+31	+17	+1	+12	+10	+1	3	+229	+183	+14	26	+2	+2	+2
no simple rnd	6	0	0	+1	+1	0	0	0	+1	2	+8	+8	+9	27	−1	−1	0
no rounding	6	1	0	+4	0	−2	−2	−2	−2	3	−12	−12	−26	26	−1	−1	−1
no shifting	6	2	1	−7	−9	−4	−4	−5	−4	6	−23	−24	−33	23	0	0	−1
no int shifting	5	3	4	+2	+3	+1	+1	+1	−2	12	+5	+5	−2	17	−1	0	0
octane	6	0	0	0	0	+3	0	0	0	0	—	—	—	29	0	0	−1

Table B.113. Evaluation of rounding heuristics on test set MILP.

				all instances (7)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—	7	0	0	0
no RENS	0	0	0	0	0	0	−1	−2	−3	0	—	—	—	7	−1	−2	−3
no simple rnd	0	0	0	0	0	0	−1	−1	−4	0	—	—	—	7	−1	−1	−4
no rounding	0	0	0	0	0	0	−2	−2	−5	0	—	—	—	7	−2	−2	−5
no shifting	0	0	0	0	0	0	−2	−2	−5	0	—	—	—	7	−2	−2	−5
no int shifting	0	0	0	0	0	0	−1	−1	−2	0	—	—	—	7	−1	−1	−2
octane	0	0	0	0	0	0	−2	−2	−5	0	—	—	—	7	−2	−2	−5

Table B.114. Evaluation of rounding heuristics on test set ENLIGHT.

				all instances (24)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—	24	0	0	0
no RENS	0	0	0	0	0	0	+1	+1	+2	0	—	—	—	24	+1	+1	+2
no simple rnd	0	0	0	0	0	0	−1	−1	−2	0	—	—	—	24	−1	−1	−2
no rounding	0	0	0	0	0	0	0	0	+1	0	—	—	—	24	0	0	+1
no shifting	0	0	0	0	0	0	−1	−1	0	0	—	—	—	24	−1	−1	0
no int shifting	0	0	0	0	0	0	+1	+1	+2	0	—	—	—	24	+1	+1	+2
octane	0	0	0	0	0	0	+4	+2	+3	0	—	—	—	24	+4	+2	+3

Table B.115. Evaluation of rounding heuristics on test set ALU.

				all instances (16)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—	16	0	0	0
no RENS	0	0	4	+74	+34	+4	+10	+4	+5	12	+13	+5	+5	4	0	0	0
no simple rnd	0	0	6	+14	+13	+7	+12	+12	+12	9	+19	+16	+12	7	+4	+5	+5
no rounding	0	0	0	0	0	0	+2	+2	+3	0	—	—	—	16	+2	+2	+3
no shifting	0	0	0	0	0	0	0	0	+1	0	—	—	—	16	0	0	+1
no int shifting	0	0	2	+1	+1	+1	+4	+3	+4	8	+5	+5	+4	8	+3	+4	+4
octane	0	0	0	0	0	0	0	0	0	0	—	—	—	16	0	0	0

Table B.116. Evaluation of rounding heuristics on test set FCTP.

				all instances (7)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—	7	0	0	0
no RENS	0	0	0	0	0	0	+1	0	0	0	—	—	—	7	+1	0	0
no simple rnd	0	0	0	0	0	0	−2	−1	−1	0	—	—	—	7	−2	−1	−1
no rounding	0	0	0	0	0	0	−1	−1	−1	0	—	—	—	7	−1	−1	−1
no shifting	0	0	0	0	0	0	−2	−2	−1	0	—	—	—	7	−2	−2	−1
no int shifting	0	0	0	0	0	0	0	0	+1	0	—	—	—	7	0	0	+1
octane	0	0	0	0	0	0	−3	−2	−2	0	—	—	—	7	−3	−2	−2

Table B.117. Evaluation of rounding heuristics on test set ACC.

				all instances (20)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—	20	0	0	0
no RENS	0	2	15	+370	+149	+65	+22	+21	+24	20	+22	+21	+24	0	—	—	—
no simple rnd	0	0	1	+7	+8	+5	+2	+3	+4	5	+8	+8	+7	15	+1	+1	+1
no rounding	0	0	0	+5	+3	0	+1	0	0	2	+6	+6	+6	18	0	0	0
no shifting	0	2	1	−6	−6	−7	−2	−3	−4	7	−6	−6	−8	13	0	0	0
no int shifting	0	2	2	−10	−11	−1	−1	0	+3	9	0	0	+3	11	−1	−1	−1
octane	0	0	0	0	0	0	+2	+1	+1	0	—	—	—	20	+2	+1	+1

Table B.118. Evaluation of rounding heuristics on test set FC.

				all instances (23)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—	23	0	0	0
no RENS	0	1	3	+30	+18	−2	+3	+3	−3	9	+9	+9	−10	14	0	0	+1
no simple rnd	0	3	5	−1	−1	−2	+3	+2	+2	11	+2	+1	+2	12	+3	+3	+3
no rounding	0	1	2	+3	+2	+2	+1	+1	+4	5	+4	+4	+12	18	0	0	0
no shifting	0	1	2	0	0	+1	+4	+4	+6	6	+6	+7	+12	17	+3	+3	+3
no int shifting	0	1	2	−1	−1	−3	+2	+2	−1	10	+1	0	−5	13	+3	+3	+3
octane	0	0	0	0	0	0	0	0	0	0	—	—	—	23	0	0	0

Table B.119. Evaluation of rounding heuristics on test set ARCSET.

				all instances (41)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—	41	0	0	0
no RENS	0	3	16	+13	+13	+12	+9	+9	+16	37	+10	+10	+21	4	+2	+2	+2
no simple rnd	0	0	23	+1	+1	+6	+10	+9	+14	38	+10	+9	+14	3	+11	+10	+13
no rounding	0	0	0	0	0	0	0	0	+1	1	+2	+2	+2	40	0	0	+1
no shifting	0	0	0	0	0	+1	0	0	+1	8	0	0	+2	33	0	0	0
no int shifting	0	2	7	−1	−1	+5	+2	+2	+8	41	+2	+2	+8	0	—	—	—
octane	0	0	0	0	0	0	0	0	0	0	—	—	—	41	0	0	0

Table B.120. Evaluation of rounding heuristics on test set MIK.

				all instances (30)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	1	0	0	0	0	0	0	0	0	0	—	—	—	29	0	0	0
no coef	1	3	6	+1	+1	+3	+1	0	+3	19	+2	+1	+4	10	−1	−1	−2
no frac	1	5	2	−3	−3	−14	−6	−6	−13	18	−9	−9	−20	11	−1	−1	−1
no guided	1	3	1	0	0	−7	−2	−2	−9	14	−3	−3	−13	15	−2	−2	−7
no linesearch	1	1	1	0	0	+1	−1	−1	+1	14	0	0	+3	15	−2	−2	−8
no pscost	1	5	2	−3	−3	−7	−4	−4	−9	19	−6	−6	−13	10	−2	−1	−1
no veclen	1	5	1	−3	−3	−3	−3	−3	−3	16	−5	−4	−4	13	−2	−2	−8
no backtrack	1	6	3	−6	−6	−6	−4	−4	−6	18	−5	−6	−9	11	−2	−1	−1

Table B.121. Evaluation of diving heuristics on test set MIPLIB.

				all instances (38)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	3	0	0	0	0	0	0	0	0	0	—	—	—	35	0	0	0
no coef	3	15	4	−18	−13	−10	−8	−8	−1	30	−10	−10	−2	5	0	0	0
no frac	3	7	4	−1	0	−12	0	0	0	28	−1	−1	−1	7	+3	+3	+3
no guided	2	9	2	−4	−4	−14	−6	−6	−5	24	−9	−9	−9	11	−1	−1	−1
no linesearch	2	5	6	+4	+4	−4	+4	+4	−1	26	+7	+6	+5	9	+3	+3	+3
no pscost	3	10	3	−9	−5	−12	−3	−3	−3	29	−4	−4	−7	6	−1	−1	−1
no veclen	2	10	4	−8	−7	−22	−7	−7	−15	27	+1	+2	+1	8	0	0	0
no backtrack	3	11	4	+3	+2	−16	−1	−3	−4	29	−2	−3	−7	6	−1	−1	−1

Table B.122. Evaluation of diving heuristics on test set CORAL.

				all instances (35)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	6	0	0	0	0	0	0	0	0	0	—	—	—	29	0	0	0
no coef	6	5	4	−7	−7	+2	−2	−2	+1	18	−8	−8	−10	10	0	0	0
no frac	5	4	4	+4	+4	+2	+1	+1	−1	16	+4	+4	+3	13	−1	0	0
no guided	5	1	8	+16	+16	+2	+11	+10	+9	15	+25	+25	+33	14	+2	+2	+2
no linesearch	6	0	8	+9	+10	0	+6	+6	+5	15	+15	+14	+17	14	0	0	0
no pscost	6	2	4	0	0	−1	+1	0	−1	16	+2	+1	−2	13	−1	−1	−1
no veclen	6	2	2	+2	+2	+6	+1	+2	+2	15	+3	+3	+7	14	0	0	0
no backtrack	5	2	5	+5	+5	+4	+3	+3	0	17	+7	+6	+1	12	+1	+1	+2

Table B.123. Evaluation of diving heuristics on test set MILP.

				all instances (7)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—	7	0	0	0
no coef	0	2	3	+3	+3	+6	+4	+4	+1	6	+5	+4	+1	1	0	0	0
no frac	0	1	2	+6	+6	−12	+3	−4	−17	6	+4	−5	−17	1	0	0	0
no guided	0	0	0	+2	+2	+2	+2	+2	+2	4	+4	+4	+8	3	0	−1	−1
no linesearch	0	1	2	+12	+13	+30	+10	+13	+28	6	+11	+14	+28	1	0	0	0
no pscost	0	0	3	+19	+19	+11	+19	+18	+11	6	+22	+20	+11	1	0	0	0
no veclen	0	0	3	+12	+12	+17	+10	+9	+14	6	+12	+10	+14	1	0	0	0
no backtrack	0	2	3	+10	+11	+29	+16	+19	+30	6	+19	+21	+30	1	0	0	0

Table B.124. Evaluation of diving heuristics on test set ENLIGHT.

				all instances (25)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	1	0	0	0	0	0	0	0	0	0	—	—	—	24	0	0	0
no coef	1	2	10	+54	+52	+24	+35	+36	+42	17	+56	+53	+97	7	0	0	0
no frac	1	2	8	+3	+3	+3	0	−1	+3	16	0	−2	+6	8	0	0	0
no guided	1	0	0	0	0	+1	0	0	0	0	—	—	—	24	0	0	+1
no linesearch	1	4	9	+25	+22	+3	+11	+8	+4	16	+17	+11	+9	8	0	0	0
no pscost	1	5	6	+8	+6	−1	0	−2	−6	16	+1	−3	−14	8	0	0	0
no veclen	1	4	7	+11	+11	+4	+6	+5	+8	16	+9	+7	+19	8	0	0	0
no backtrack	1	8	5	−1	−2	0	−4	−5	+2	16	−6	−7	+4	8	0	0	0

Table B.125. Evaluation of diving heuristics on test set ALU.

				all instances (16)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—	16	0	0	0
no coef	0	2	1	−3	−3	−1	−2	−3	−3	9	−5	−5	−3	7	+1	+2	+2
no frac	0	0	3	+1	+1	+4	+5	+4	+8	8	+5	+5	+8	8	+4	+5	+6
no guided	0	2	0	−3	−4	−1	−2	−3	−1	8	−5	−5	−1	8	+1	+1	+1
no linesearch	0	0	1	0	0	0	+2	+2	+2	8	+2	+2	+2	8	+2	+3	+3
no pscost	0	0	1	+1	+1	+3	+1	+1	+2	8	+2	+2	+2	8	+1	+1	+1
no veclen	0	2	1	−2	−2	−3	+1	0	+1	8	0	−1	+1	8	+3	+4	+4
no backtrack	0	1	0	−3	−3	−7	0	−2	−4	8	−2	−3	−4	8	+3	+4	+4

Table B.126. Evaluation of diving heuristics on test set FCTP.

				all instances (7)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—	7	0	0	0
no coef	0	1	2	+37	+72	+498	+18	+19	+95	3	+53	+54	+105	4	−3	−3	−3
no frac	1	1	2	+28	+48	+458	+12	+13	+77	2	−39	−39	−31	4	−2	−2	−2
no guided	0	0	0	0	0	0	−2	−2	−2	0	—	—	—	7	−2	−2	−2
no linesearch	1	2	1	+29	+48	+582	+14	+15	+78	2	−34	−34	−30	4	−3	−3	−3
no pscost	1	1	2	+49	+73	+441	+40	+42	+142	2	+30	+30	+57	4	−2	−2	−2
no veclen	0	0	2	+35	+50	+135	+16	+16	+30	3	+40	+40	+33	4	+1	+1	0
no backtrack	0	2	1	+14	+26	+314	+12	+12	+39	3	+27	+27	+43	4	+1	+1	+1

Table B.127. Evaluation of diving heuristics on test set ACC.

				all instances (20)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—	20	0	0	0
no coef	0	1	0	−2	−2	−8	−1	−1	−2	10	−2	−2	−3	10	0	0	0
no frac	0	0	0	0	−1	0	0	0	0	10	0	0	0	10	0	0	0
no guided	0	0	0	0	0	−1	0	0	0	6	0	0	−1	14	0	0	0
no linesearch	0	0	0	0	0	−1	0	0	0	6	0	0	0	14	0	0	0
no pscost	0	0	0	+1	+1	+2	0	+1	+1	10	+1	+1	+2	10	0	0	0
no veclen	0	0	1	+1	+1	+8	+1	+1	+2	8	+1	+1	+2	12	+1	+1	0
no backtrack	0	1	0	−1	−1	−3	0	0	−1	8	0	−1	−1	12	+1	+1	+1

Table B.128. Evaluation of diving heuristics on test set FC.

				all instances (23)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—	23	0	0	0
no coef	0	6	6	+7	+6	+12	+6	+5	−7	23	+6	+5	−7	0	—	—	—
no frac	0	5	2	−6	−6	−9	−3	−3	−8	20	−3	−3	−8	3	+1	+1	+1
no guided	0	4	3	0	0	−4	+1	+1	−4	16	0	0	−4	7	+2	+2	+2
no linesearch	0	4	4	+2	+2	0	+1	+1	0	16	0	+1	0	7	+1	+1	+1
no pscost	0	3	5	+8	+8	+18	+9	+10	+5	20	+10	+11	+5	3	+3	+3	+3
no veclen	0	6	3	−7	−8	−3	−5	−5	+4	20	−6	−6	+4	3	0	0	0
no backtrack	0	5	6	+11	+11	+12	+6	+6	+4	22	+6	+6	+4	1	+2	+2	+2

Table B.129. Evaluation of diving heuristics on test set ARCSET.

				all instances (41)						different path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—
no coef	0	5	9	0	0	+6	+2	+2	+7	41	+2	+2	+7
no frac	0	6	9	0	0	+3	+2	+1	+6	41	+2	+1	+6
no guided	0	2	1	0	0	+1	+1	0	+1	20	0	0	+2
no linesearch	0	5	6	0	0	+11	+3	+4	+15	41	+3	+4	+15
no pscost	0	4	7	−1	−1	+8	+2	+2	+11	41	+2	+2	+11
no veclen	0	7	6	−2	−2	+1	0	0	+3	41	0	0	+3
no backtrack	0	4	10	+4	+4	+7	+2	+2	+7	41	+2	+2	+7

Table B.130. Evaluation of diving heuristics on test set MIK.

				all instances (30)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	1	0	0	0	0	0	0	0	0	0	—	—	—	29	0	0	0
no obj pscost diving	1	3	0	−1	−1	−1	−2	−3	−5	15	−4	−4	−7	14	−1	−2	−7
no rootsol diving	1	3	3	+3	+3	−4	+1	+1	−4	14	+5	+4	−6	15	−2	−2	−8
no feaspump	2	3	5	+66	+37	+3	+17	+19	+47	10	+18	+19	+42	18	−3	−3	−6

Table B.131. Evaluation of objective diving heuristics on test set MIPLIB.

				all instances (38)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	3	0	0	0	0	0	0	0	0	0	—	—	—	35	0	0	0
no obj pscost diving	2	12	6	−1	−1	−22	−3	−4	−14	27	0	0	−3	8	+3	+3	+3
no rootsol diving	2	12	6	−3	−3	−18	−3	−3	−10	26	−2	−2	−3	9	+3	+3	+3
no feaspump	3	6	4	+3	−3	0	0	0	+1	17	+1	+1	+3	18	−1	−1	+1

Table B.132. Evaluation of objective diving heuristics on test set CORAL.

				all instances (36)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	7	0	0	0	0	0	0	0	0	0	—	—	—	29	0	0	0
no obj pscost diving	7	1	6	+12	+12	−14	+6	+6	+5	14	+10	+10	+6	14	+1	+1	+1
no rootsol diving	7	2	4	+7	+7	−1	+2	+2	+4	14	+2	+2	+5	14	−1	0	0
no feaspump	7	5	2	+13	−2	0	0	−1	−1	11	+1	−2	−4	18	−1	−1	−1

Table B.133. Evaluation of objective diving heuristics on test set MILP.

				all instances (7)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—	7	0	0	0
no obj pscost diving	0	1	2	+10	+10	+21	+11	+15	+19	6	+13	+16	+19	1	0	0	0
no rootsol diving	0	2	2	+1	+1	−2	+3	+4	−4	6	+4	+5	−4	1	0	0	0
no feaspump	0	0	1	+5	+5	+13	+3	+3	+7	5	+4	+4	+7	2	0	0	0

Table B.134. Evaluation of objective diving heuristics on test set ENLIGHT.

				all instances (25)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	1	0	0	0	0	0	0	0	0	0	—	—	—	24	0	0	0
no obj pscost diving	1	1	7	+17	+18	+3	+15	+15	+6	16	+24	+21	+14	8	0	0	0
no rootsol diving	1	3	8	+18	+19	0	+13	+13	+1	16	+21	+18	+3	8	0	0	0
no feaspump	1	1	5	+17	+17	+4	+10	+10	+10	16	+16	+15	+24	8	0	0	0

Table B.135. Evaluation of objective diving heuristics on test set ALU.

				all instances (16)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—	16	0	0	0
no obj pscost diving	0	1	1	−2	−2	+4	0	−1	+5	8	−2	−2	+5	8	+2	+3	+3
no rootsol diving	0	0	0	+1	+1	+2	+1	0	+4	8	+1	+1	+4	8	+1	+1	+1
no feaspump	0	0	0	0	0	0	+2	+2	+2	0	—	—	—	16	+2	+2	+2

Table B.136. Evaluation of objective diving heuristics on test set FCTP.

				all instances (7)						different path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—
no obj pscost diving	0	2	1	+14	+24	+217	+8	+9	+31	3	+20	+20	+34
no rootsol diving	0	2	1	+16	+28	+262	+14	+15	+66	3	+38	+39	+73
no feaspump	0	0	7	+1447	+239	+440	+117	+107	+126	7	+117	+107	+126

Table B.137. Evaluation of objective diving heuristics on test set ACC.

				all instances (20)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—	20	0	0	0
no obj pscost diving	0	0	0	0	0	0	−1	−1	−1	7	−2	−2	−2	13	0	0	0
no rootsol diving	0	0	0	0	0	−2	0	0	0	7	0	0	−1	13	+1	+1	0
no feaspump	0	0	0	0	0	0	0	0	0	0	—	—	—	20	0	0	0

Table B.138. Evaluation of objective diving heuristics on test set FC.

				all instances (23)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—	23	0	0	0
no obj pscost diving	0	5	5	+6	+6	+6	+5	+6	−3	18	+6	+6	−3	5	+3	+3	+3
no rootsol diving	0	3	7	+7	+7	+12	+8	+8	+11	18	+9	+10	+12	5	+4	+4	+4
no feaspump	0	1	4	+9	+9	+10	+8	+8	+9	8	+22	+21	+20	15	+1	+1	+1

Table B.139. Evaluation of objective diving heuristics on test set ARCSET.

				all instances (41)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—	41	0	0	0
no obj pscost diving	0	9	5	0	0	+12	−2	−1	+10	41	−2	−1	+10	0	—	—	—
no rootsol diving	0	6	7	−1	−1	+3	+2	+2	+7	41	+2	+2	+7	0	—	—	—
no feaspump	0	0	0	0	0	0	+2	+1	+1	1	−9	−9	−9	40	+2	+2	+2

Table B.140. Evaluation of objective diving heuristics on test set MIK.

				all instances (30)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	1	0	0	0	0	0	0	0	0	0	—	—	—	29	0	0	0
no oneopt	1	2	2	−3	−4	−11	0	−1	−6	8	+4	−1	−22	21	−2	−2	−3
no crossover	1	2	1	−2	−2	−6	−4	−4	−7	5	−15	−17	−18	24	−1	−1	−1
local branching	1	2	5	−4	−4	−7	+1	0	−4	7	−8	−10	−16	22	+4	+4	+7
RINS	1	1	2	−1	−1	−8	−1	−1	−7	8	−1	−3	−15	21	−1	−1	−2
mutation	1	3	1	−2	−2	−9	0	−1	−7	9	−5	−7	−16	20	+2	+1	+3

Table B.141. Evaluation of improvement heuristics on test set MIPLIB.

				all instances (37)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	3	0	0	0	0	0	0	0	0	0	—	—	—	34	0	0	0
no oneopt	3	1	1	−1	−1	−2	+1	+1	+1	4	−6	−7	−13	30	+2	+2	+2
no crossover	3	3	1	−4	−4	0	−2	−2	0	8	−12	−12	−3	26	+1	+1	+2
local branching	3	2	3	−4	−4	0	+1	+1	+1	8	−6	−6	+2	26	+3	+3	+2
RINS	2	4	0	−9	−9	−10	−5	−5	−13	10	−8	−7	−3	24	+2	+2	+3
mutation	3	3	0	−4	−4	−1	−3	−3	−1	10	−10	−9	−5	24	0	0	+1

Table B.142. Evaluation of improvement heuristics on test set CORAL.

				all instances (35)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	6	0	0	0	0	0	0	0	0	0	—	—	—	29	0	0	0
no oneopt	6	0	0	+5	+2	+1	−1	0	0	2	+2	+2	+1	27	−1	−1	−1
no crossover	6	0	2	+2	+2	0	+1	+1	+2	5	+9	+9	+11	24	0	0	0
local branching	6	0	8	+1	+1	−2	+6	+5	+4	6	+14	+14	+16	23	+5	+5	+4
RINS	6	0	2	0	0	−2	+2	+2	+2	7	+5	+5	+9	22	+1	+1	+1
mutation	6	0	1	+1	+1	+7	+2	+1	+1	5	+4	+4	+3	24	+1	+1	+2

Table B.143. Evaluation of improvement heuristics on test set MILP.

				all instances (7)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—	7	0	0	0
no oneopt	0	0	0	0	0	0	−3	−3	−6	0	—	—	—	7	−3	−3	−6
no crossover	0	0	0	0	0	0	+1	−1	−2	0	—	—	—	7	+1	−1	−2
local branching	0	0	1	0	0	0	+6	+1	−1	0	—	—	—	7	+6	+1	−1
RINS	0	0	0	0	0	0	+2	+1	+2	0	—	—	—	7	+2	+1	+2
mutation	0	0	0	0	0	0	−2	−2	−5	0	—	—	—	7	−2	−2	−5

Table B.144. Evaluation of improvement heuristics on test set ENLIGHT.

				all instances (24)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—	24	0	0	0
no oneopt	0	0	0	0	0	0	−1	−1	−1	0	—	—	—	24	−1	−1	−1
no crossover	0	0	0	0	0	0	+1	0	+1	0	—	—	—	24	+1	0	+1
local branching	0	0	0	0	0	0	−1	−1	−1	0	—	—	—	24	−1	−1	−1
RINS	0	0	0	0	0	0	0	0	0	0	—	—	—	24	0	0	0
mutation	0	0	0	0	0	0	+1	0	+1	0	—	—	—	24	+1	0	+1

Table B.145. Evaluation of improvement heuristics on test set ALU.

				all instances (16)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—	16	0	0	0
no oneopt	0	0	0	0	0	+2	+4	+3	+6	1	+9	+9	+9	15	+4	+2	+3
no crossover	0	1	2	+3	+3	−4	+3	+2	0	6	+5	+3	0	10	+3	+2	+3
local branching	0	1	4	+3	+3	−4	+7	+6	+4	6	+13	+10	+4	10	+4	+5	+4
RINS	0	2	2	−5	−5	−3	0	−1	+2	8	−2	−1	+2	8	+2	+3	+3
mutation	0	1	1	+3	+3	−1	+1	0	0	6	+2	0	0	10	0	0	0

Table B.146. Evaluation of improvement heuristics on test set FCTP.

				all instances (7)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—	7	0	0	0
no oneopt	0	0	0	0	0	0	−2	−2	−2	0	—	—	—	7	−2	−2	−2
no crossover	0	0	0	0	0	0	−1	−1	−1	0	—	—	—	7	−1	−1	−1
local branching	0	0	0	0	0	0	−2	−1	−1	0	—	—	—	7	−2	−1	−1
RINS	0	0	0	0	0	0	−2	−2	−2	0	—	—	—	7	−2	−2	−2
mutation	0	0	0	0	0	0	−1	−1	−1	0	—	—	—	7	−1	−1	−1

Table B.147. Evaluation of improvement heuristics on test set ACC.

				all instances (20)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—	20	0	0	0
no oneopt	0	2	1	−10	−10	−7	0	0	+1	12	−1	−1	+1	8	0	+1	+1
no crossover	0	0	0	+1	+1	+2	+1	+1	+1	3	+2	+2	+2	17	0	0	0
local branching	0	0	8	+1	+1	+2	+10	+12	+16	3	+26	+24	+22	17	+7	+9	+13
RINS	0	0	0	+1	+1	0	+2	+2	+2	4	+2	+2	+2	16	+2	+2	+2
mutation	0	0	0	+1	+1	+2	+2	+2	+2	3	+4	+4	+4	17	+2	+1	+1

Table B.148. Evaluation of improvement heuristics on test set FC.

				all instances (23)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—	23	0	0	0
no oneopt	0	2	3	−1	−1	−1	+4	+3	+1	7	+5	+4	0	16	+3	+3	+3
no crossover	0	7	2	−1	−1	−4	−4	−4	−9	8	−6	−6	−14	15	−4	−3	−1
local branching	0	6	4	−1	−1	−4	−1	−1	−6	8	−1	−2	−12	15	−2	−1	+1
RINS	0	3	5	−3	−3	−7	+5	+4	0	8	+2	0	−5	15	+7	+7	+6
mutation	0	3	2	−1	−1	−5	+1	+1	−2	8	0	0	−6	15	+2	+2	+2

Table B.149. Evaluation of improvement heuristics on test set ARCSET.

				all instances (41)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—	41	0	0	0
no oneopt	0	0	4	0	0	+3	+3	+3	+7	11	+2	+3	+11	30	+3	+3	+4
no crossover	0	3	0	−1	−1	+2	−4	−4	−1	16	−6	−5	0	25	−3	−3	−3
local branching	0	2	8	−1	−1	+2	+2	+2	+3	16	+4	+4	+6	25	+1	+1	+1
RINS	0	2	2	−1	−1	+1	−1	−1	+1	17	−1	−1	+2	24	−1	−1	−1
mutation	0	2	4	−1	−1	+2	+4	+3	+7	16	+3	+3	+9	25	+4	+4	+4

Table B.150. Evaluation of improvement heuristics on test set MIK.

				all instances (30)							different path				equal path		
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	1	0	0	0	0	0	0	0	0	0	—	—	—	29	0	0	0
all (5%)	0	23	0	−97	−89	−92	−72	−62	−79	27	−66	−55	−71	2	+1	+1	0
none (5%)	1	21	4	−6	−39	−88	−35	−34	−43	27	−33	−33	−62	1	−4	−4	−4
all (20%)	0	25	0	−99	−95	−98	−82	−72	−84	28	−78	−67	−78	1	−2	−2	−2
none (20%)	1	19	4	−50	−64	−96	−53	−48	−47	27	−52	−48	−68	1	+4	+4	+4

Table B.151. Evaluation of primal heuristics to reach a predefined solution quality on test set MIPLIB.

				all instances (38)							different path				equal path		
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	3	0	0	0	0	0	0	0	0	0	—	—	—	35	0	0	0
all (5%)	1	19	0	−94	−83	−94	−66	−59	−56	25	−75	−66	−56	10	+2	+2	+3
none (5%)	1	24	3	−63	−64	−93	−63	−57	−58	34	−61	−54	−56	1	−5	−5	−5
all (20%)	1	28	0	−98	−93	−97	−81	−73	−62	31	−83	−74	−64	4	+4	+4	+5
none (20%)	1	26	2	−79	−78	−95	−70	−65	−62	34	−68	−62	−63	1	−3	−3	−3

Table B.152. Evaluation of primal heuristics to reach a predefined solution quality on test set CORAL.

				all instances (36)							different path				equal path		
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	7	0	0	0	0	0	0	0	0	0	—	—	—	29	0	0	0
all (5%)	5	12	0	−74	−51	−25	−52	−48	−32	19	−61	−58	−53	10	+1	+1	+2
none (5%)	5	15	7	−5	−27	−40	−35	−36	−32	25	−25	−27	−40	3	0	0	0
all (20%)	2	20	0	−95	−88	−44	−79	−75	−63	22	−78	−75	−65	7	−1	0	0
none (20%)	3	17	7	−72	−75	−50	−64	−62	−49	25	−50	−51	−52	3	−2	−1	0

Table B.153. Evaluation of primal heuristics to reach a predefined solution quality on test set MILP.

				all instances (7)							different path				equal path		
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—	7	0	0	0
all (5%)	0	0	0	0	0	0	+1	0	−1	1	−2	−2	−2	6	+1	0	−1
none (5%)	0	2	1	−14	−13	−2	−12	−10	−2	6	−14	−11	−2	1	0	0	0
all (20%)	0	0	0	−3	−3	−1	−2	−2	−2	4	−3	−2	−2	3	−1	−1	−2
none (20%)	0	4	1	−18	−18	−2	−16	−11	−2	6	−19	−12	−2	1	0	0	0

Table B.154. Evaluation of primal heuristics to reach a predefined solution quality on test set ENLIGHT.

				all instances (25)							different path				equal path		
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	1	0	0	0	0	0	0	0	0	0	—	—	—	24	0	0	0
all (5%)	1	0	0	0	0	0	+1	+1	+1	0	—	—	—	24	+1	+1	+2
none (5%)	1	12	3	−10	−12	+1	−15	−14	−3	18	−21	−18	−6	6	0	0	0
all (20%)	1	0	0	0	0	+2	−1	−1	−1	0	—	—	—	24	−1	−1	−1
none (20%)	1	12	3	−10	−12	0	−15	−14	−3	18	−21	−18	−6	6	0	0	0

Table B.155. Evaluation of primal heuristics to reach a predefined solution quality on test set ALU.

				all instances (16)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—	16	0	0	0
all (5 %)	0	12	0	−96	−82	−90	−62	−65	−83	14	−67	−67	−83	2	0	0	0
none (5 %)	0	9	4	+23	−25	−78	−40	−56	−80	13	−47	−59	−80	3	0	0	0
all (20 %)	0	12	0	−100	−100	−100	−89	−95	−98	16	−89	−95	−98	0	—	—	—
none (20 %)	0	9	4	−22	−54	−90	−54	−70	−89	13	−62	−73	−89	3	0	0	0

Table B.156. Evaluation of primal heuristics to reach a predefined solution quality on test set FCTP.

				all instances (7)						different path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—
all (5 %)	0	0	0	0	0	0	+1	+1	+1	0	—	—	—
none (5 %)	0	2	5	+957	+151	+427	+40	+34	+45	7	+40	+34	+45
all (20 %)	0	0	0	0	0	0	−3	−3	−3	0	—	—	—
none (20 %)	0	2	5	+957	+151	+427	+44	+37	+49	7	+44	+37	+49

Table B.157. Evaluation of primal heuristics to reach a predefined solution quality on test set ACC.

				all instances (20)						different path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—
all (5 %)	0	17	0	−90	−69	−70	−29	−29	−33	20	−29	−29	−33
none (5 %)	0	1	19	+1316	+571	+372	+54	+51	+58	20	+54	+51	+58
all (20 %)	0	20	0	−99	−100	−100	−49	−52	−58	20	−49	−52	−58
none (20 %)	0	2	18	+1234	+531	+313	+48	+43	+44	20	+48	+43	+44

Table B.158. Evaluation of primal heuristics to reach a predefined solution quality on test set FC.

				all instances (23)						different path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—
all (5 %)	0	23	0	−100	−97	−87	−86	−77	−68	23	−86	−77	−68
none (5 %)	0	20	0	−75	−73	−64	−62	−57	−67	23	−62	−57	−67
all (20 %)	0	23	0	−100	−100	−100	−95	−89	−88	23	−95	−89	−88
none (20 %)	0	20	0	−85	−82	−77	−66	−62	−76	23	−66	−62	−76

Table B.159. Evaluation of primal heuristics to reach a predefined solution quality on test set ARCSET.

				all instances (41)						different path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—
all (5 %)	0	41	0	−100	−100	−100	−97	−96	−98	41	−97	−96	−98
none (5 %)	0	27	9	−57	−57	−73	−40	−41	−58	41	−40	−41	−58
all (20 %)	0	41	0	−100	−100	−100	−98	−98	−99	41	−98	−98	−99
none (20 %)	0	36	4	−80	−80	−85	−69	−67	−78	41	−69	−67	−78

Table B.160. Evaluation of primal heuristics to reach a predefined solution quality on test set MIK.

PRESOLVING

setting	T	fst	slw	all instances (30) n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	different path #	t_{gm}	t_{sgm}	t_{tot}	equal path #	t_{gm}	t_{sgm}	t_{tot}
default	1	0	0	0	0	0	0	0	0	0	—	—	—	29	0	0	0
none	2	7	18	+314	+163	+1	+80	+65	+47	27	+50	+39	+23	1	−30	−30	−30
no linear pairs	1	4	1	+5	+2	0	−3	−2	+1	9	−1	−1	+8	20	−4	−2	−3
aggreg linear pairs	1	4	4	−2	−2	0	+1	+2	+2	13	+2	+4	+7	16	+1	0	+1
no knap disaggreg	1	1	1	+4	+3	+1	−1	−1	−1	6	+2	+3	−2	23	−2	−2	−2

Table B.161. Evaluation of constraint specific presolving methods on test set MIPLIB.

setting	T	fst	slw	all instances (35) n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	different path #	t_{gm}	t_{sgm}	t_{tot}	equal path #	t_{gm}	t_{sgm}	t_{tot}
default	3	0	0	0	0	0	0	0	0	0	—	—	—	32	0	0	0
none	7	7	21	+164	+126	+33	+96	+93	+96	27	+45	+42	+69	0	—	—	—
no linear pairs	3	7	5	−7	−4	−2	−3	−3	−1	17	−6	−5	−4	15	−1	−1	−1
aggreg linear pairs	3	6	9	−5	−6	+29	−5	−5	−3	17	0	0	−4	14	+1	+1	+2
no knap disaggreg	2	2	3	+14	+11	+3	+9	+10	+3	4	+215	+239	+1111	28	−1	−1	−1

Table B.162. Evaluation of constraint specific presolving methods on test set CORAL.

setting	T	fst	slw	all instances (35) n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	different path #	t_{gm}	t_{sgm}	t_{tot}	equal path #	t_{gm}	t_{sgm}	t_{tot}
default	7	0	0	0	0	0	0	0	0	0	—	—	—	28	0	0	0
none	11	9	16	+288	+128	+7	+99	+82	+31	23	+44	+35	+11	0	—	—	—
no linear pairs	7	7	7	+21	+10	0	+7	+4	−1	17	+16	+10	−5	11	−2	−2	−1
aggreg linear pairs	6	4	8	+27	+14	−2	+12	+10	−4	15	+37	+30	+8	13	+3	+2	+3
no knap disaggreg	7	1	6	+3	+1	−1	+10	+8	+1	6	+61	+48	+49	22	+3	+2	+1

Table B.163. Evaluation of constraint specific presolving methods on test set MILP.

setting	T	fst	slw	all instances (7) n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	different path #	t_{gm}	t_{sgm}	t_{tot}	equal path #	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—	7	0	0	0
none	0	3	2	+51	+50	−26	+7	−7	−54	7	+7	−7	−54	0	—	—	—
no linear pairs	0	0	0	0	0	0	−1	−1	−3	0	—	—	—	7	−1	−1	−3
aggreg linear pairs	0	0	0	0	0	0	−2	−2	−4	0	—	—	—	7	−2	−2	−4
no knap disaggreg	0	0	0	0	0	0	−2	−2	−4	0	—	—	—	7	−2	−2	−4

Table B.164. Evaluation of constraint specific presolving methods on test set ENLIGHT.

setting	T	fst	slw	all instances (23) n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	different path #	t_{gm}	t_{sgm}	t_{tot}	equal path #	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—	23	0	0	0
none	2	4	17	+1058	+398	+373	+366	+248	+429	21	+297	+191	+206	0	—	—	—
no linear pairs	0	7	10	−31	−23	+13	−8	−6	+11	22	−8	−6	+11	1	0	0	0
aggreg linear pairs	0	13	4	−53	−43	+6	−27	−10	+33	22	−28	−10	+33	1	0	0	0
no knap disaggreg	0	0	0	0	0	0	−1	−1	−2	0	—	—	—	23	−1	−1	−2

Table B.165. Evaluation of constraint specific presolving methods on test set ALU.

				all instances (16)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—	16	0	0	0
none	1	0	13	+1002	+402	+821	+139	+130	+421	15	+123	+121	+253	0	—	—	—
no linear pairs	0	0	0	0	0	0	+3	+3	+4	0	—	—	—	16	+3	+3	+4
aggreg linear pairs	0	0	0	0	0	0	+2	+2	+2	0	—	—	—	16	+2	+2	+2
no knap disaggreg	0	0	0	0	0	0	−1	0	0	0	—	—	—	16	−1	0	0

Table B.166. Evaluation of constraint specific presolving methods on test set FCTP.

				all instances (7)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—	7	0	0	0
none	2	3	4	+373	+233	+525	+109	+112	+224	5	+7	+6	+16	0	—	—	—
no linear pairs	0	1	1	+31	+35	+91	+6	+7	+15	2	+33	+34	+70	5	−3	−2	−2
aggreg linear pairs	0	2	3	+99	+33	+116	+17	+16	+40	6	+20	+19	+41	1	0	0	0
no knap disaggreg	0	0	2	+58	+62	+77	+18	+19	+17	2	+85	+85	+77	5	−1	−1	−2

Table B.167. Evaluation of constraint specific presolving methods on test set ACC.

				all instances (20)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—	20	0	0	0
none	0	1	19	+5728	+2589	+2101	+341	+372	+601	20	+341	+372	+601	0	—	—	—
no linear pairs	0	0	0	−8	−8	+1	+1	+1	+2	11	+1	+2	+4	9	0	0	0
aggreg linear pairs	0	0	2	−3	−4	−1	+1	+1	+1	12	+1	+2	+3	8	0	0	0
no knap disaggreg	0	0	0	+1	0	0	0	0	0	2	0	0	0	18	0	0	0

Table B.168. Evaluation of constraint specific presolving methods on test set FC.

				all instances (23)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—	23	0	0	0
none	0	4	18	+135	+114	+331	+73	+72	+96	23	+73	+72	+96	0	—	—	—
no linear pairs	0	6	15	+63	+60	+21	+26	+24	+14	23	+26	+24	+14	0	—	—	—
aggreg linear pairs	0	4	5	+18	+8	−7	0	−1	−10	9	−6	−6	−24	14	+4	+3	0
no knap disaggreg	0	0	0	0	0	0	+2	+2	+2	0	—	—	—	23	+2	+2	+2

Table B.169. Evaluation of constraint specific presolving methods on test set ARCSET.

				all instances (41)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—	41	0	0	0
none	1	0	41	+204	+204	+215	+509	+458	+587	40	+506	+459	+571	0	—	—	—
no linear pairs	0	1	18	−1	−1	0	+10	+9	+10	41	+10	+9	+10	0	—	—	—
aggreg linear pairs	0	0	0	0	0	0	+2	+2	+3	0	—	—	—	41	+2	+2	+3
no knap disaggreg	0	0	0	0	0	0	−1	−1	−1	0	—	—	—	41	−1	−1	−1

Table B.170. Evaluation of constraint specific presolving methods on test set MIK.

				all instances (30)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	1	0	0	0	0	0	0	0	0	0	—	—	—	29	0	0	0
no int to binary	0	2	1	+2	−1	−10	−3	−3	−14	2	+11	+12	+14	27	−3	−2	−2
no probing	2	9	10	+11	+16	−9	+8	+9	+25	24	+7	+8	+9	4	−7	−9	−11
full probing	2	4	4	−12	−9	−9	+15	+16	+38	8	+5	+4	+20	20	−1	0	−2
no impl graph	1	2	0	−4	−2	0	−2	−2	−1	2	−8	−7	−7	27	−2	−1	−2
no dual fixing	1	4	5	+6	+2	−9	0	−1	−4	10	+5	0	−7	19	−2	−2	−4

Table B.171. Evaluation of generic presolving methods on test set MIPLIB.

				all instances (36)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	2	0	0	0	0	0	0	0	0	0	—	—	—	34	0	0	0
no int to binary	2	1	0	−1	−1	−11	−1	−1	−2	1	−18	−18	−18	33	0	0	0
no probing	3	8	21	+26	+14	+3	+21	+20	+40	29	+16	+15	+49	3	+19	+19	+19
full probing	2	5	12	−34	−17	0	+5	+4	+4	18	−5	−6	−13	16	+17	+17	+19
no impl graph	3	0	1	+2	+2	+3	+5	+5	+16	0	—	—	—	33	0	0	0
no dual fixing	2	3	4	−18	−6	−28	+5	+7	+8	16	+13	+16	+26	18	0	0	0

Table B.172. Evaluation of generic presolving methods on test set CORAL.

				all instances (35)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	7	0	0	0	0	0	0	0	0	0	—	—	—	28	0	0	0
no int to binary	7	0	1	+4	+4	+2	+1	+1	0	1	+62	+62	+62	27	−1	−1	−1
no probing	6	12	10	+108	+52	−6	+15	+12	−7	27	+24	+20	−3	1	−3	−3	−3
full probing	8	3	5	+9	+5	+1	+5	+5	+5	10	+6	+7	−4	17	+2	+2	+1
no impl graph	6	1	2	+2	0	0	0	0	−4	6	+10	+10	+12	22	0	0	0
no dual fixing	8	4	9	−3	−5	+2	+5	+3	+5	14	+2	−3	−18	13	+2	+2	+1

Table B.173. Evaluation of generic presolving methods on test set MILP.

				all instances (7)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—	7	0	0	0
no int to binary	1	0	3	+26	+27	+74	+41	+35	+81	6	+31	+26	+6	0	—	—	—
no probing	0	1	3	+44	+44	+28	+12	+3	−2	7	+12	+3	−2	0	—	—	—
full probing	0	0	0	0	0	0	0	0	−1	0	—	—	—	7	0	0	−1
no impl graph	0	0	0	0	0	0	−2	−2	−3	0	—	—	—	7	−2	−2	−3
no dual fixing	0	0	0	0	0	0	−2	−2	−5	0	—	—	—	7	−2	−2	−5

Table B.174. Evaluation of generic presolving methods on test set ENLIGHT.

				all instances (25)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	1	0	0	0	0	0	0	0	0	0	—	—	—	24	0	0	0
no int to binary	1	6	7	−3	−2	+3	+3	−2	+5	23	+3	−2	+12	1	0	0	0
no probing	1	8	13	+58	−8	+12	+14	−2	+16	24	+15	−2	+37	0	—	—	—
full probing	1	0	0	0	0	0	−1	−1	0	0	—	—	—	24	−1	−1	0
no impl graph	1	11	5	−45	−37	−8	−25	−22	−16	24	−26	−25	−36	0	—	—	—
no dual fixing	1	0	0	0	0	+1	−1	−1	−1	0	—	—	—	24	−1	−1	−1

Table B.175. Evaluation of generic presolving methods on test set ALU.

				all instances (16)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—	16	0	0	0
no int to binary	0	0	0	0	0	0	0	0	0	0	—	—	—	16	0	0	0
no probing	0	2	2	−3	−6	−3	−3	−4	−3	11	−4	−5	−3	5	0	0	0
full probing	0	2	1	−1	−1	−1	+1	0	+3	8	+1	0	+3	8	+1	+2	+2
no impl graph	0	0	0	0	0	0	+2	+2	+3	0	—	—	—	16	+2	+2	+3
no dual fixing	0	0	0	0	0	0	0	0	0	0	—	—	—	16	0	0	0

Table B.176. Evaluation of generic presolving methods on test set FCTP.

				all instances (7)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—	7	0	0	0
no int to binary	0	0	0	0	0	0	+1	+1	+1	0	—	—	—	7	+1	+1	+1
no probing	3	1	3	+212	+286	+940	+95	+100	+268	1	−67	−67	−67	3	−3	−3	−3
full probing	0	0	2	+265	+32	+7	+41	+36	+4	3	+129	+139	+195	4	−2	−2	−2
no impl graph	0	0	0	0	0	0	+1	+1	+2	0	—	—	—	7	+1	+1	+2
no dual fixing	0	0	0	0	0	0	0	0	0	0	—	—	—	7	0	0	0

Table B.177. Evaluation of generic presolving methods on test set ACC.

				all instances (20)						different path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—
no int to binary	0	0	0	0	0	0	0	0	0	0	—	—	—
no probing	0	3	7	+13	+15	+29	+7	+9	+10	20	+7	+9	+10
full probing	0	0	0	0	0	0	0	0	0	0	—	—	—
no impl graph	0	0	0	0	0	0	0	0	0	0	—	—	—
no dual fixing	0	5	8	+9	+10	0	+7	+4	+2	20	+7	+4	+2

Table B.178. Evaluation of generic presolving methods on test set FC.

				all instances (23)						different path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—
no int to binary	0	0	3	+6	+5	+2	+8	+7	+6	4	+26	+26	+30
no probing	0	5	7	+27	+13	+3	−3	−3	−12	14	−5	−5	−24
full probing	0	4	6	+19	+7	−14	+4	+2	−12	12	+5	+2	−30
no impl graph	0	0	0	0	0	0	+1	+1	+1	0	—	—	—
no dual fixing	0	9	10	+23	+11	+6	+7	+7	−5	23	+7	+7	−5

Table B.179. Evaluation of generic presolving methods on test set ARCSET.

				all instances (41)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—	41	0	0	0
no int to binary	0	11	5	−2	−2	+1	−4	−3	−1	26	−6	−4	−1	15	−2	−1	−1
no probing	0	6	25	+44	+44	+15	+34	+28	+16	41	+34	+28	+16	0	—	—	—
full probing	0	0	0	0	0	0	+3	+3	+2	0	—	—	—	41	+3	+3	+2
no impl graph	0	0	0	0	0	0	0	0	0	0	—	—	—	41	0	0	0
no dual fixing	0	0	0	0	0	0	+2	+2	+3	1	+8	+8	+8	40	+2	+2	+3

Table B.180. Evaluation of generic presolving methods on test set MIK.

				all instances (30)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	1	0	0	0	0	0	0	0	0	0	—	—	—	29	0	0	0
no restart	1	3	4	+8	−4	−12	+5	+3	−6	12	+15	+10	−15	17	−1	−1	−2
sub restart	1	1	5	+6	+3	+9	+2	+2	+11	10	+8	+7	+29	19	0	0	0
aggr sub restart	1	1	7	+6	+2	+10	+4	+4	+15	10	+11	+10	+37	19	+1	+1	+1

Table B.181. Evaluation of restarts on test set MIPLIB.

				all instances (38)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	3	0	0	0	0	0	0	0	0	0	—	—	—	35	0	0	0
no restart	3	3	1	−5	−4	−31	−1	−1	−2	6	−10	−11	−18	29	0	0	0
sub restart	3	1	1	+3	+3	+13	0	0	0	3	+7	+6	+2	32	0	0	0
aggr sub restart	3	2	2	+2	+2	+11	+2	+2	+1	4	+5	+4	−9	31	+2	+2	+2

Table B.182. Evaluation of restarts on test set CORAL.

				all instances (36)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	7	0	0	0	0	0	0	0	0	0	—	—	—	29	0	0	0
no restart	7	0	2	+25	+19	0	+12	+10	+2	5	+113	+98	+23	24	+1	+1	+1
sub restart	8	1	6	+14	+15	+1	+11	+12	+10	6	+26	+27	+40	22	0	+1	0
aggr sub restart	8	0	8	+20	+21	+1	+14	+14	+12	7	+41	+40	+77	21	0	0	0

Table B.183. Evaluation of restarts on test set MILP.

				all instances (7)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—	7	0	0	0
no restart	0	0	0	0	0	0	−1	−1	−3	0	—	—	—	7	−1	−1	−3
sub restart	0	0	0	0	0	0	0	−1	−1	0	—	—	—	7	0	−1	−1
aggr sub restart	0	0	0	0	0	0	0	−1	−1	0	—	—	—	7	0	−1	−1

Table B.184. Evaluation of restarts on test set ENLIGHT.

				all instances (25)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	1	0	0	0	0	0	0	0	0	0	—	—	—	24	0	0	0
no restart	1	0	0	0	0	0	0	0	0	0	—	—	—	24	0	0	+1
sub restart	1	0	0	−1	0	−1	+2	+1	+1	1	+7	+7	+7	23	+2	+2	+3
aggr sub restart	1	1	1	−16	−13	+8	−4	0	+6	4	−23	−2	+57	20	0	0	+1

Table B.185. Evaluation of restarts on test set ALU.

				all instances (16)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—	16	0	0	0
no restart	0	0	0	0	0	0	+3	+3	+4	0	—	—	—	16	+3	+3	+4
sub restart	0	0	0	−1	0	0	+2	+2	+3	1	0	0	0	15	+2	+2	+3
aggr sub restart	0	0	0	−1	0	0	+1	+1	+1	1	0	0	0	15	+1	+1	+1

Table B.186. Evaluation of restarts on test set FCTP.

				all instances (7)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—	7	0	0	0
no restart	0	0	1	0	0	0	+3	+3	+4	0	—	—	—	7	+3	+3	+4
sub restart	0	0	0	0	0	0	−2	−1	−1	0	—	—	—	7	−2	−1	−1
aggr sub restart	0	0	0	0	0	0	+1	+1	+1	0	—	—	—	7	+1	+1	+1

Table B.187. Evaluation of restarts on test set ACC.

				all instances (20)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—	20	0	0	0
no restart	0	3	2	+28	+8	+6	−1	+1	+4	12	−2	+2	+10	8	0	0	0
sub restart	0	0	1	−3	−2	0	+1	+1	+1	2	+11	+11	+12	18	0	0	0
aggr sub restart	0	0	2	−10	−8	−14	+3	+2	+1	5	+7	+3	−1	15	+2	+2	+2

Table B.188. Evaluation of restarts on test set FC.

				all instances (23)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—	23	0	0	0
no restart	0	0	1	+4	+4	+5	+10	+9	+6	1	+238	+238	+238	22	+4	+4	+4
sub restart	0	2	1	−8	−6	−7	−1	−1	+1	3	−27	−27	−40	20	+3	+3	+3
aggr sub restart	0	1	1	−5	−3	+1	0	+1	+2	4	−4	+1	+10	19	+1	+1	+1

Table B.189. Evaluation of restarts on test set ARCSET.

				all instances (41)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—	41	0	0	0
no restart	3	0	41	+311	+310	+403	+553	+500	+697	38	+533	+494	+604	0	—	—	—
sub restart	0	0	0	0	0	0	0	0	0	0	—	—	—	41	0	0	0
aggr sub restart	0	0	0	0	0	0	+2	+1	+1	0	—	—	—	41	+2	+1	+1

Table B.190. Evaluation of restarts on test set MIK.

CONFLICT ANALYSIS

				all instances (30)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	1	0	0	0	0	0	0	0	0	0	—	—	—	29	0	0	0
prop	2	4	1	−7	−6	+7	−1	0	+15	13	−6	−5	−7	15	−2	−1	−1
prop/inflp	2	1	7	−1	−1	+14	+7	+6	+19	14	+4	+3	+1	14	+5	+2	0
prop/inflp/age	1	4	2	−9	−9	−33	0	0	+7	14	−2	−2	+1	14	−1	−1	−1
prop/lp	3	2	9	−15	−14	−22	+13	+12	+33	18	+13	+11	+14	9	+2	+2	+3
all	3	1	8	−11	−10	−19	+14	+14	+35	21	+14	+13	+20	6	−1	−1	−2
full	2	1	17	−15	−15	−27	+26	+24	+29	22	+33	+31	+42	6	+2	+1	+1

Table B.191. Evaluation of conflict analysis on test set MIPLIB.

				all instances (37)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	2	0	0	0	0	0	0	0	0	0	—	—	—	35	0	0	0
prop	1	10	3	−21	−20	−23	−10	−10	−8	21	−9	−8	+22	14	+3	+3	+2
prop/inflp	2	12	5	−16	−13	−21	−1	−1	+7	26	−3	−4	+3	8	0	0	0
prop/inflp/age	2	11	7	−31	−29	−48	−7	−7	0	26	−2	−3	+12	8	+3	+3	+2
prop/lp	3	11	8	−25	−22	−42	+3	+2	+19	30	−1	−2	+10	4	+1	+1	+2
all	2	9	16	−29	−27	−46	+3	+2	+8	32	+11	+10	+25	2	+2	+2	+2
full	2	5	21	−38	−36	−48	+9	+8	+15	32	+18	+17	+38	2	+2	+2	+2

Table B.192. Evaluation of conflict analysis on test set CORAL.

				all instances (35)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	6	0	0	0	0	0	0	0	0	0	—	—	—	29	0	0	0
prop	4	2	2	−16	−16	−21	−13	−13	−11	11	−12	−12	−7	18	+1	+1	+2
prop/inflp	3	4	1	−29	−29	−35	−26	−26	−23	12	−14	−15	−14	17	0	0	0
prop/inflp/age	3	7	1	−35	−36	−40	−28	−28	−25	12	−19	−20	−18	17	+2	+2	+2
prop/lp	4	8	5	−47	−48	−51	−31	−31	−32	16	−21	−21	−33	12	+3	+2	+3
all	5	5	8	−44	−45	−46	−24	−24	−23	18	−6	−6	−18	10	+1	+1	+1
full	3	7	12	−46	−47	−58	−19	−18	−23	18	+16	+15	+12	10	−1	0	0

Table B.193. Evaluation of conflict analysis on test set MILP.

				all instances (7)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—	7	0	0	0
prop	0	2	1	−30	−29	−25	−12	−11	−7	6	−14	−12	−7	1	0	0	0
prop/inflp	0	4	1	−49	−48	−53	−25	−21	−37	6	−29	−23	−37	1	0	0	0
prop/inflp/age	0	5	0	−79	−79	−96	−46	−49	−79	6	−51	−52	−79	1	0	0	0
prop/lp	0	4	1	−50	−49	−55	−24	−21	−38	6	−28	−23	−38	1	0	0	0
all	0	5	0	−50	−49	−63	−26	−26	−52	6	−30	−28	−52	1	0	0	0
full	0	5	0	−50	−49	−64	−26	−26	−53	6	−29	−28	−53	1	0	0	0

Table B.194. Evaluation of conflict analysis on test set ENLIGHT.

				all instances (25)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	1	0	0	0	0	0	0	0	0	0	—	—	—	24	0	0	0
prop	1	9	3	−63	−61	−39	−30	−28	−4	16	−42	−37	−8	8	0	0	0
prop/inflp	0	13	2	−78	−74	−52	−42	−35	−5	16	−57	−45	0	8	0	0	0
prop/inflp/age	2	10	6	−83	−80	−85	−26	−11	+89	15	−47	−28	+122	8	0	0	0
prop/lp	1	13	2	−79	−75	−56	−44	−39	−10	16	−60	−50	−24	8	0	0	0
all	1	13	1	−83	−78	−57	−45	−39	+2	19	−55	−46	+5	5	0	0	0
full	1	13	1	−82	−77	−53	−45	−38	+4	19	−54	−46	+10	5	0	0	0

Table B.195. Evaluation of conflict analysis on test set ALU.

				all instances (16)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—	16	0	0	0
prop	0	0	1	+1	+1	+1	+1	+1	+1	6	+2	+2	+1	10	+1	+1	+1
prop/inflp	0	0	1	0	0	−2	+3	+2	+1	6	+3	+2	+1	10	+3	+3	+4
prop/inflp/age	0	0	0	0	0	−2	+1	+1	0	6	0	0	0	10	+2	+2	+2
prop/lp	0	0	7	+2	+2	−1	+18	+23	+41	9	+34	+33	+41	7	+1	+2	+2
all	0	1	9	−3	−3	+1	+21	+20	+54	12	+28	+24	+54	4	0	0	0
full	0	1	10	−8	−8	−5	+39	+41	+121	12	+54	+50	+121	4	+3	+4	+4

Table B.196. Evaluation of conflict analysis on test set FCTP.

				all instances (7)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—	7	0	0	0
prop	0	0	2	+35	+51	+179	+18	+19	+38	3	+49	+49	+42	4	−1	−1	−1
prop/inflp	0	2	2	+48	+57	+225	+20	+21	+49	4	+39	+39	+50	3	−2	−2	−2
prop/inflp/age	0	2	2	+46	+52	+188	+16	+17	+28	4	+31	+30	+29	3	−1	−1	−1
prop/lp	0	2	2	+13	+20	+177	−2	−2	+17	4	−3	−3	+18	3	−1	−1	−1
all	0	2	2	−14	−21	−29	−18	−19	−38	4	−29	−29	−39	3	−2	−2	−2
full	0	2	2	+21	+17	+6	+5	+5	+15	4	+10	+10	+15	3	−2	−2	−2

Table B.197. Evaluation of conflict analysis on test set ACC.

				all instances (20)						different path				equal path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—	20	0	0	0
prop	0	0	0	−1	−1	−2	+1	+1	+1	4	+1	0	0	16	+1	+1	+1
prop/inflp	0	0	0	−3	−3	−9	+1	+1	0	9	0	−1	−2	11	+2	+2	+2
prop/inflp/age	0	1	0	−3	−3	−9	0	−1	−2	9	−1	−2	−3	11	+1	+1	+1
prop/lp	0	0	6	+1	+1	0	+7	+9	+14	14	+10	+11	+15	6	+2	+2	+2
all	0	1	6	−14	−15	−16	+8	+9	+13	15	+10	+11	+14	5	+1	+2	+2
full	0	0	8	−22	−24	−35	+14	+18	+25	15	+19	+22	+27	5	−1	−1	−1

Table B.198. Evaluation of conflict analysis on test set FC.

				all instances (23)						different path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—
prop	0	2	1	−5	−5	−14	+1	+1	−7	13	0	−1	−10
prop/inflp	0	11	4	−16	−16	−21	−6	−6	−18	22	−6	−6	−18
prop/inflp/age	0	9	4	−17	−17	−24	−5	−6	−16	22	−6	−6	−16
prop/lp	0	7	7	−29	−29	−34	−3	−4	−11	23	−3	−4	−11
all	0	6	7	−29	−29	−29	−1	−1	−6	23	−1	−1	−6
full	0	6	7	−34	−34	−38	+1	+1	+21	23	+1	+1	+21

Table B.199. Evaluation of conflict analysis on test set ARCSET.

setting	T	fst	slw	all instances (41) n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	different path #	t_{gm}	t_{sgm}	t_{tot}
default	0	0	0	0	0	0	0	0	0	0	—	—	—
prop	0	0	12	+1	+1	+13	+7	+7	+22	34	+8	+8	+24
prop/inflp	0	4	11	−6	−5	+3	+4	+4	+19	41	+4	+4	+19
prop/inflp/age	0	5	14	−12	−12	−17	+6	+7	+15	41	+6	+7	+15
prop/lp	0	19	13	−53	−53	−57	−10	−11	−17	41	−10	−11	−17
all	0	22	10	−56	−55	−60	−15	−15	−23	41	−15	−15	−23
full	0	20	13	−60	−60	−59	−16	−17	−13	41	−16	−17	−13

Table B.200. Evaluation of conflict analysis on test set MIK.

COMPARISON TO CPLEX

				all instances (30)						different path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
CPLEX 10.0.1	1	0	0	0	0	0	0	0	0	0	—	—	—
SCIP 0.90f	1	5	25	−68	−38	−6	+131	+51	+1	28	+202	+94	+25
SCIP 0.90i	1	5	24	−62	−41	−21	+108	+44	−2	28	+164	+81	+12

Table B.201. Evaluation of CPLEX 10.0.1, SCIP 0.90f, and SCIP 0.90i on test set MIPLIB.

				all instances (37)						different path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
CPLEX 10.0.1	4	0	0	0	0	0	0	0	0	0	—	—	—
SCIP 0.90f	3	8	27	+30	−1	+58	+237	+117	−1	30	+289	+147	+27
SCIP 0.90i	2	13	24	−35	−43	−66	+121	+54	−29	31	+196	+103	+3

Table B.202. Evaluation of CPLEX 10.0.1, SCIP 0.90f, and SCIP 0.90i on test set CORAL.

				all instances (36)						different path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
CPLEX 10.0.1	1	0	0	0	0	0	0	0	0	0	—	—	—
SCIP 0.90f	7	5	28	−5	+40	+220	+295	+256	+240	28	+183	+151	+152
SCIP 0.90i	3	9	25	−43	−20	+62	+176	+149	+115	32	+170	+143	+129

Table B.203. Evaluation of CPLEX 10.0.1, SCIP 0.90f, and SCIP 0.90i on test set MILP.

				all instances (7)						different path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
CPLEX 10.0.1	1	0	0	0	0	0	0	0	0	0	—	—	—
SCIP 0.90f	0	1	4	−56	−26	−55	+9	−12	−49	6	+42	+20	+31
SCIP 0.90i	0	3	1	−97	−89	−98	−61	−73	−95	6	−43	−66	−94

Table B.204. Evaluation of CPLEX 10.0.1, SCIP 0.90f, and SCIP 0.90i on test set ENLIGHT.

				all instances (24)						different path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
CPLEX 10.0.1	1	0	0	0	0	0	0	0	0	0	—	—	—
SCIP 0.90f	0	9	12	−90	−70	−75	−14	−10	−64	23	−8	−1	−48
SCIP 0.90i	0	11	8	−99	−97	−97	−68	−63	−71	23	−59	−50	−45

Table B.205. Evaluation of CPLEX 10.0.1, SCIP 0.90f, and SCIP 0.90i on test set ALU.

				all instances (16)						different path			
setting	T	fst	slw	n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	t_{gm}	t_{sgm}	t_{tot}
CPLEX 10.0.1	0	0	0	0	0	0	0	0	0	0	—	—	—
SCIP 0.90f	0	0	12	−1	+10	+1	+109	+82	+49	16	+109	+82	+49
SCIP 0.90i	0	0	12	−12	+11	+11	+90	+72	+50	16	+90	+72	+50

Table B.206. Evaluation of CPLEX 10.0.1, SCIP 0.90f, and SCIP 0.90i on test set FCTP.

setting	T	fst	slw	all instances (7) n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	different path t_{gm}	t_{sgm}	t_{tot}
CPLEX 10.0.1	0	0	0	0	0	0	0	0	0	0	—	—	—
SCIP 0.90f	0	1	6	+155	+234	+316	+665	+390	+454	7	+665	+390	+454
SCIP 0.90i	0	1	6	+407	+287	+435	+619	+350	+294	7	+619	+350	+294

Table B.207. Evaluation of CPLEX 10.0.1, SCIP 0.90f, and SCIP 0.90i on test set ACC.

setting	T	fst	slw	all instances (20) n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	different path t_{gm}	t_{sgm}	t_{tot}
CPLEX 10.0.1	0	0	0	0	0	0	0	0	0	0	—	—	—
SCIP 0.90f	0	0	20	−34	−7	+35	+149	+137	+158	20	+149	+137	+158
SCIP 0.90i	0	0	19	−39	−22	−12	+126	+107	+110	20	+126	+107	+110

Table B.208. Evaluation of CPLEX 10.0.1, SCIP 0.90f, and SCIP 0.90i on test set FC.

setting	T	fst	slw	all instances (23) n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	different path t_{gm}	t_{sgm}	t_{tot}
CPLEX 10.0.1	0	0	0	0	0	0	0	0	0	0	—	—	—
SCIP 0.90f	0	4	18	−39	−35	−44	+182	+93	−35	23	+182	+93	−35
SCIP 0.90i	0	7	16	−60	−53	−60	+116	+48	−59	23	+116	+48	−59

Table B.209. Evaluation of CPLEX 10.0.1, SCIP 0.90f, and SCIP 0.90i on test set ARCSET.

setting	T	fst	slw	all instances (41) n_{gm}	n_{sgm}	n_{tot}	t_{gm}	t_{sgm}	t_{tot}	#	different path t_{gm}	t_{sgm}	t_{tot}
CPLEX 10.0.1	0	0	0	0	0	0	0	0	0	0	—	—	—
SCIP 0.90f	0	31	7	−66	−66	−81	−49	−48	−71	41	−49	−48	−71
SCIP 0.90i	0	30	9	−70	−70	−82	−48	−47	−67	41	−48	−47	−67

Table B.210. Evaluation of CPLEX 10.0.1, SCIP 0.90f, and SCIP 0.90i on test set MIK.

Appendix C

SCIP versus Cplex

In order to give an indication of the performance of SCIP as a black-box MIP solver, we compare SCIP to the commercial code Cplex 10.0.1 [118], which is one of the fastest MIP solvers that is currently available. We used Cplex in its default settings, except that we set the gap dependent abort criteria to `mipgap` $= 0$ and `absmipgap` $= 10^{-9}$, which are the corresponding values in SCIP. Note that SCIP uses Cplex to solve the LP relaxations of the subproblems. This means that our benchmarks only compare the MIP part of the solver. SCIP is usually much slower if a non-commercial LP solver, for example SoPlex [219], is employed.

Table C.1 shows a summary of the results on our MIP test sets. One can see how the performance of the two SCIP versions considered in this thesis, SCIP 0.90f and SCIP 0.90i, compares to Cplex. As usual, we consider the shifted geometric mean of the results for the instances of a test set, see Appendix A.3. The numbers in the table, calculated as in Equation (B.1) on page 325, are the percental differences of the SCIP results compared to the reference mean values of Cplex. The results for the individual instances of the test sets are listed in Tables C.2 to C.11.

Overall, SCIP 0.90f is 115 % slower than Cplex, i.e., it takes (in the shifted geometric mean) a little more than twice as long to solve a MIP instance with SCIP 0.90f than with Cplex. In contrast, SCIP 0.90i is only 63 % slower than

	test set	SCIP 0.90f	SCIP 0.90i
time	MIPLIB	+51	+44
	CORAL	+117	+54
	MILP	+256	+149
	ENLIGHT	−12	−73
	ALU	−10	−63
	FCTP	+82	+72
	ACC	+390	+350
	FC	+137	+107
	ARCSET	+93	+48
	MIK	−48	−47
	total	**+115**	**+63**
nodes	MIPLIB	−38	−41
	CORAL	−1	−43
	MILP	+40	−20
	ENLIGHT	−26	−89
	ALU	−70	−97
	FCTP	+10	+11
	ACC	+234	+287
	FC	−7	−22
	ARCSET	−35	−53
	MIK	−66	−70
	total	**−6**	**−40**

Table C.1. Comparison of Cplex 10.0.1 to SCIP 0.90f and SCIP 0.90i. The values denote the percental changes in the shifted geometric mean of the runtime (top) and number of branching nodes (bottom) compared to Cplex. Positive values indicate instances on which SCIP is inferior, negative values represent superior results.

	Cplex 10.0.1		SCIP 0.90f		SCIP 0.90i	
	Nodes	Time	Nodes	Time	Nodes	Time
modglob	245	0.5	13	2.0	241	2.6
fiber	72	0.5	29	2.8	10	1.7
p2756	16	0.5	95	5.1	37	4.6
gesa2	39	0.5	3	5.9	8	5.6
set1ch	326	0.5	10	3.3	23	2.6
fixnet6	132	1.0	11	5.3	23	2.6
pp08a	618	1.1	1 196	3.0	1 20	2.9
vpm2	3 269	1.1	5 846	6.8	723	2.2
pp08aCUTS	1 372	2.0	817	3.7	426	2.8
10teams	22	4.4	1 374	43.1	538	39.8
gesa2-o	3 380	4.8	1	8.6	7	8.3
misc07	11 49	9.7	20 982	30.8	25 256	39.3
air05	288	12.4	291	124.8	342	90.9
air04	128	14.8	19	233.6	258	214.6
cap6000	4 565	15.8	3 165	8.6	3 65	8.8
nw04	126	20.1	7	73.1	5	77.7
aflow30a	11 832	30.5	2 180	33.8	2 426	30.2
qiu	2 371	31.6	10 567	185.4	8 697	110.0
mzzv42z	183	52.4	1 404	599.0	1 91	823.8
mod011	45	70.1	2 769	279.7	2 731	200.5
pk1	338 108	83.3	233 736	113.7	228 745	136.4
rout	41 345	91.8	12 542	55.0	28 669	79.1
mas76	660 320	122.5	414 423	142.3	347 197	134.9
disctom	50	195.2	1	98.8	1	94.7
mzzv11	1 873	219.0	1 786	1291.6	3 271	1148.3
harp2	1 71 588	1033.8	1 727 702	1415.0	>4 925 65	>3600.0
mas74	4 451 916	1267.3	4 86 233	2515.1	3 272 670	1477.8
noswot	4 717 721	1528.3	>5 203 78	>3600.0	983 43	2210.9
fast0507	13 997	3141.7	1 424	749.4	1 445	772.8
manna81	>1 177 281	>3600.0	1	4.6	2	5.6
geom. mean	2 556	20.7	818	47.8	963	43.1
sh. geom. mean	3 662	46.8	2 271	70.9	2 178	67.7
arithm. mean	417 142	385.2	391 56	388.1	327 901	377.7

Table C.2. Detailed benchmark results on the miplib test set. Values printed in red and blue indicate a difference of at least a factor of 2 to the reference Cplex results.

Cplex, which is an almost 25 % performance improvement with respect to the older version SCIP 0.90f. Thus, the conclusions we draw from our computational experiments and the resulting changes in the default settings indeed lead to a substantial enhancement in the MIP solving capabilities of SCIP. As a conclusion of the comparison to Cplex, we take the results as evidence that the algorithms of SCIP are close to the state-of-the-art in MIP solving, which substantiates the significance of our computational experiments on SCIP.

	Cplex 10.0.1		SCIP 0.90f		SCIP 0.90i	
	Nodes	Time	Nodes	Time	Nodes	Time
neos-583731	1	0.5	2 71	134.8	185	63.2
neos-612125	2	0.9	215	67.6	341	55.7
neos-633273	1	1.1	4 48	1209.2	2 673	481.5
neos-612143	15	1.2	441	92.7	407	56.3
neos-494568	40	1.4	142	22.4	91	45.5
neos-584851	40	2.3	1 408	101.1	521	76.4
neos-717614	734	2.7	>932 392	>3600.0	174 879	710.1
neos-498623	80	3.7	79	38.4	306	54.9
neos-555001	470	4.7	140	7.0	1	4.1
neos-807639	5 90	7.0	16 284	150.6	13 650	147.6
neos-585467	973	9.9	347	23.2	189	17.0
neos-565815	8	11.3	3	26.9	1	41.7
aligninq	1 918	15.0	715	31.6	1 905	24.3
neos-570431	1 124	17.6	1 456	31.6	2 105	38.1
neos-504815	11 72	28.2	10 420	70.5	6 896	52.1
prod1	61 292	34.6	26 784	33.6	25 439	43.0
neos-738098	1	49.9	>199	>3600.0	>68	>3600.0
neos-480878	17 603	57.1	11 512	109.5	7 266	82.7
binkar10_1	7 992	63.4	205 358	844.6	259 288	974.3
neos-512201	21 202	69.5	15 772	179.2	10 720	131.1
neos-538916	55 938	72.6	40 693	113.7	12 731	61.2
neos-530627	823 990	77.6	2 405 10	1146.3	3	0.5
neos-796608	701 625	96.5	4 366 657	1691.7	—	—
neos-538867	124 697	119.5	155 214	298.8	42 468	162.2
neos-791021	129	140.0	1	147.7	34	230.6
neos-686190	12 11	155.9	6 320	237.2	6 21	328.3
neos-506422	6 113	163.1	7 233	246.4	1 894	112.0
bienst1	7 918	219.6	12 24	339.0	12 473	55.8
prod2	276 396	299.3	107 299	207.3	69 191	173.9
neos-585192	28 262	476.1	1 690	69.5	903	62.4
neos-476283	3 710	587.8	1 66	1154.3	1 294	1209.1
neos-808072	53 464	1109.4	478	106.1	444	151.6
neos-787933	>20 469	>1967.8	1	102.6	1	99.7
neos-503737	50 438	2087.1	2 53	192.6	4 838	347.6
lrn	365 796	2742.3	>49 410	>3600.0	>66 579	>3600.0
bienst2	>109 661	>3600.0	113 376	1168.9	81 708	344.9
neos-551991	>8 19	>3600.0	3 444	496.4	6 542	785.2
neos-595925	>691 192	>3600.0	232 871	531.9	111 587	348.5
geom. mean	2 323	52.9	2 995	176.4	1 486	116.0
sh. geom. mean	4 720	88.2	4 608	189.3	2 659	134.7
arithm. mean	74 807	578.4	118 53	555.0	25 17	399.3

Table C.3. Detailed benchmark results on the coral test set. Values printed in red and blue indicate a difference of at least a factor of 2 to the reference Cplex results.

	Cplex 10.0.1		SCIP 0.90f		SCIP 0.90i	
	Nodes	Time	Nodes	Time	Nodes	Time
neos648910	370	0.7	476	4.3	125	4.1
neos1	1	1.2	1	4.9	14	20.2
neos8	1	4.0	1	150.1	1	152.1
neos4	55	4.9	1	3.2	1	3.1
neos2	951	6.1	13 954	134.8	17 595	76.0
neos10	28	8.7	5	187.2	5	252.1
markshare1_1	179 832	12.5	—	—	1 168 136	186.1
swath1	5 69	16.6	2 115	91.5	1 262	85.5
nug08	54	17.3	2	238.8	1	488.3
neos3	4 104	20.7	>537 967	>3600.0	217 360	758.9
neos22	8 41	27.4	2	7.2	1	6.3
neos20	3 516	33.4	1 604	22.9	986	19.6
markshare2_1	468 378	34.6	>12 973 500	>3600.0	3 551 769	545.5
neos21	3 140	43.5	2 634	52.3	1 796	44.7
neos7	14 159	52.6	2	11.2	2	10.2
swath2	26 492	70.8	4 318	131.6	1 960	94.3
mkc1	14 265	71.7	>612 556	>3600.0	>639 263	>3600.0
30:70:4.5:0.5:100	1	78.6	6 221	1315.2	441	1504.5
neos6	1 456	98.4	2 282	333.6	5 877	864.1
bc1	5 834	103.1	5 28	164.8	5 315	290.6
dano3_4	27	116.7	61	296.9	42	240.5
dano3_3	15	122.9	36	205.6	41	225.1
30:70:4.5:0.95:98	59	129.7	16	253.3	29	465.3
30:70:4.5:0.95:100	1	133.8	16	283.9	16	537.8
neos17	92 121	159.3	>1 937 629	>3600.0	16 65	42.6
qap10	10	160.4	2	1760.9	4	656.3
neos11	2 827	167.6	12 214	1021.4	8 523	684.8
neos12	151	215.6	>3 621	>3600.0	>5 35	>3600.0
neos23	214 230	282.8	>2 246 374	>3600.0	2 281	10.4
neos18	46 439	298.6	107 197	1007.0	16 403	202.7
swath3	106 882	327.5	47 271	469.8	107 348	1376.8
dano3_5	367	526.8	287	530.1	278	501.7
seymour1	6 419	740.0	4 167	780.4	5 956	960.4
neos14	375 642	769.5	700 591	3161.5	533 485	1850.8
neos13	1 697	1568.8	22 787	1565.4	9 351	1057.0
neos5	6 233 26	1657.2	>5 259 593	>3600.0	>7 231 78	>3600.0
neos9	>22 794	>3600.0	1	267.4	1	244.7
geom. mean	1 242	74.7	1 176	295.0	711	206.3
sh. geom. mean	2 794	96.7	3 926	344.2	2 241	241.0
arithm. mean	212 739	324.2	680 681	1101.6	343 880	696.6

Table C.4. Detailed benchmark results on the MILP test set. Values printed in red and blue indicate a difference of at least a factor of 2 to the reference Cplex results.

	Cplex 10.0.1		SCIP 0.90f		SCIP 0.90i	
	Nodes	Time	Nodes	Time	Nodes	Time
enlight4	66	0.5	1	0.5	1	0.5
enlight5	940	0.5	1 75	0.5	1	0.5
enlight6	3 628	0.7	3 34	1.3	447	0.7
enlight7	2 951	1.1	4 773	2.5	2 213	2.0
enlight8	80 928	25.0	59 959	36.8	8 855	10.0
enlight9	3 871 450	1253.3	2 748 499	1635.4	43 56	59.1
enlight10	>5 161 291	>3600.0	1 278 955	803.0	121 763	155.5
geom. mean	19 505	11.2	8 528	12.2	643	4.4
sh. geom. mean	22 676	47.5	16 731	41.6	2 398	12.8
arithm. mean	1 303 36	697.3	585 185	354.3	25 190	32.6

Table C.5. Detailed benchmark results on the ENLIGHT test set. Values printed in red and blue indicate a difference of at least a factor of 2 to the reference Cplex results.

	Cplex 10.0.1		SCIP 0.90f		SCIP 0.90i	
	Nodes	Time	Nodes	Time	Nodes	Time
alu4_2	54	0.5	1	0.5	1	0.5
alu6_6	247	0.5	11	0.7	33	0.8
alu5_6	357	0.5	123	1.4	13	0.5
alu5_2	741	0.5	1	0.5	1	0.5
alu4_6	1883	1.0	13	0.5	19	1.1
alu6_2	2584	1.4	1	0.5	1	0.5
alu4_7	1933	1.6	2719	3.5	1	0.5
alu4_1	1944	2.9	8247	18.2	244	4.1
alu4_8	4578	2.9	17873	23.0	1218	3.5
alu5_7	7497	4.2	8457	7.1	1723	3.4
alu5_8	18746	11.7	13501	16.2	2466	5.9
alu7_2	27238	14.1	1	0.5	1	0.5
alu7_6	183	0.5	19	0.6	11	0.8
alu6_8	1956	14.2	417365	425.3	10747	26.8
alu6_7	20237	13.3	133159	103.2	15311	32.5
alu5_1	17954	14.3	22115	39.0	33	4.1
alu6_1	35789	27.5	2901	12.3	215	5.7
alu8_2	37954	21.3	1	0.5	1	0.5
alu7_7	87605	63.5	59337	483.1	36619	169.2
alu7_8	238302	220.1	251409	313.8	53143	224.2
alu8_6	75085	314.8	8469	10.3	21	0.9
alu8_7	238617	1450.3	772187	577.4	194977	1702.1
alu7_1	>462957	>3600.0	400995	630.2	90	4.6
alu8_1	2240568	1807.5	14835	34.6	53	4.1
alu8_8	—	—	>4116714	>3600.0	>471965	>3600.0
geom. mean	13179	11.5	1271	9.8	106	3.7
sh. geom. mean	14498	29.4	4357	26.4	496	10.8
arithm. mean	438775	316.2	111143	112.6	13205	91.6

Table C.6. Detailed benchmark results on the alu test set. Values printed in red and blue indicate a difference of at least a factor of 2 to the reference Cplex results.

	Cplex 10.0.1		SCIP 0.90f		SCIP 0.90i	
	Nodes	Time	Nodes	Time	Nodes	Time
bk4x3	1	0.5	1	0.5	1	0.5
gr4x6	1	0.5	1	0.5	1	0.5
bal8x12	1	0.5	1	0.5	1	0.5
ran10x10b	22	0.5	13	0.7	17	0.6
ran10x10a	43	0.5	28	1.6	3	1.0
ran4x64	63	0.5	29	1.2	17	1.0
ran10x12	1	0.5	1	0.9	1	0.8
ran6x43	122	0.5	78	1.3	139	1.5
ran10x10c	25	1.4	1682	3.6	197	3.0
ran17x17	4651	5.4	1667	13.0	2108	11.4
ran12x12	13218	9.3	22851	33.5	20316	28.0
ran8x32	8993	12.3	1482	41.8	2390	52.1
ran13x13	14535	15.4	69697	95.0	48698	65.5
ran10x26	2286	34.2	3090	72.3	30276	62.7
ran12x21	53156	84.0	110622	223.7	11575	201.2
ran16x16	392768	520.7	263773	532.3	32687	601.1
geom. mean	351	3.1	347	6.5	309	5.9
sh. geom. mean	1381	9.9	1516	18.1	1528	17.1
arithm. mean	31979	42.9	32163	63.9	35432	64.5

Table C.7. Detailed benchmark results on the fctp test set. Values printed in red and blue indicate a difference of at least a factor of 2 to the reference Cplex results.

	Cplex 10.0.1		SCIP 0.90f		SCIP 0.90i	
	Nodes	Time	Nodes	Time	Nodes	Time
acc-0	1	0.5	1	20.3	1	17.7
acc-1	1	5.0	1	31.2	115	90.9
acc-2	1	7.5	1	38.6	1	44.4
acc-3	29	50.3	572	988.5	115	256.1
acc-4	37	69.7	672	1230.3	418	716.3
acc-5	86	94.4	1219	546.6	2322	869.3
acc-6	453	322.4	63	188.7	280	173.7
geom. mean	12	21.3	31	163.1	62	153.2
sh. geom. mean	51	36.4	173	178.0	201	163.6
arithm. mean	86	78.5	361	434.9	464	309.8

Table C.8. Detailed benchmark results on the acc test set. Values printed in red and blue indicate a difference of at least a factor of 2 to the reference Cplex results.

	Cplex 10.0.1		SCIP 0.90f		SCIP 0.90i	
	Nodes	Time	Nodes	Time	Nodes	Time
fc.30.50.7	1	0.8	4	2.7	35	2.9
fc.60.20.9	59	1.1	5	5.9	5	7.0
fc.60.20.5	62	1.2	3	4.1	7	3.9
fc.30.50.10	81	1.4	6	1.9	12	2.0
fc.30.50.1	176	1.5	62	3.8	28	3.6
fc.30.50.5	6	1.5	4	2.9	3	2.7
fc.30.50.4	210	2.1	266	4.4	383	4.4
fc.30.50.9	82	2.1	43	5.1	36	5.3
fc.30.50.3	275	2.4	425	4.0	216	4.9
fc.30.50.6	264	2.8	436	6.4	138	4.9
fc.30.50.8	45	2.8	47	5.6	40	5.7
fc.30.50.2	334	2.9	35	5.2	13	3.9
fc.60.20.2	1180	3.9	5177	33.8	1385	15.8
fc.60.20.10	139	4.3	581	16.7	1380	21.3
fc.60.20.3	132	6.5	1837	20.2	1513	17.7
fc.60.20.8	1782	6.7	677	13.0	1207	12.4
fc.60.20.6	2743	8.4	1124	14.7	2502	16.4
fc.60.20.1	2474	10.7	1601	20.4	548	14.7
fc.60.20.4	585	11.1	10773	35.9	5695	19.7
fc.60.20.7	2544	12.9	1950	18.0	1133	13.8
geom. mean	228	3.1	152	7.8	139	7.1
sh. geom. mean	380	4.0	355	9.4	297	8.2
arithm. mean	928	4.4	1252	11.2	813	9.2

Table C.9. Detailed benchmark results on the fc test set. Values printed in red and blue indicate a difference of at least a factor of 2 to the reference Cplex results.

	Cplex 10.0.1		SCIP 0.90f		SCIP 0.90i	
	Nodes	Time	Nodes	Time	Nodes	Time
ns4-pr6	20	0.5	12 683	77.7	12 175	72.3
ns25-pr6	858	2.5	737	12.6	699	13.2
ns60-pr12	1 17	3.7	657	15.7	446	13.3
nu120-pr12	571	4.3	160	29.8	32	23.2
ns60-pr6	1 506	4.8	1 593	25.9	745	16.0
nu120-pr6	383	4.9	11	65.2	5	21.3
ns4-pr12	2 599	6.8	3 993	44.0	3 722	43.4
ns60-pr4	478	8.6	544	32.9	358	16.8
nu60-pr6	1 667	8.9	724	31.5	2 271	52.6
ns4-pr4	3 724	9.2	2 989	24.9	2 483	21.4
nu4-pr12	3 152	9.9	362	15.4	3 95	42.3
nu60-pr12	1 874	9.9	184	34.6	192	34.2
nu25-pr6	2 811	11.1	152	18.5	52	17.2
nu4-pr6	3 983	12.8	5 147	46.4	11 67	122.3
ns25-pr12	4 749	13.0	13 876	117.3	3 131	44.1
ns25-pr4	5 125	36.1	2 914	38.0	2 309	27.9
nu25-pr12	18 837	67.9	7 724	82.5	149	22.6
nu4-pr4	29 936	99.6	24 965	176.4	21 142	168.9
nu60-pr4	7 320	180.7	3 916	125.8	1 848	83.9
nu25-pr4	25 308	335.7	4 660	109.7	3 274	74.3
nu120-pr4	20 117	349.2	31 790	1118.2	9 834	205.2
ns25-pr9	106 732	2235.7	19 971	615.2	15 815	461.2
ns60-pr9	47 746	2759.4	22 802	1167.5	21 453	905.9
geom. mean	3 285	22.5	2 3	63.5	1 314	48.6
sh. geom. mean	3 641	35.9	2 368	69.3	1 721	53.1
arithm. mean	12 631	268.5	7 67	175.0	5 56	108.8

Table C.10. Detailed benchmark results on the arcset test set. Values printed in red and blue indicate a difference of at least a factor of 2 to the reference Cplex results.

	Cplex 10.0.1		SCIP 0.90f		SCIP 0.90i	
	Nodes	Time	Nodes	Time	Nodes	Time
mik.250-10-100.3	22 779	11.4	11 56	11.0	10 600	11.3
mik.250-20-75.4	29 335	12.3	157 870	95.2	175 60	104.0
mik.500-5-75.1	22 724	13.6	6 918	6.7	6 654	6.9
mik.250-5-100.1	38 95	15.4	29 666	25.5	15 132	17.9
mik.500-5-75.2	27 549	15.8	18 116	16.2	9 96	8.7
mik.500-10-100.5	23 602	15.9	73 160	50.6	63 810	55.3
mik.250-1-75.4	49 826	16.4	30 470	24.6	14 316	15.5
mik.250-1-100.2	46 81	16.8	14 288	9.2	8 353	8.0
mik.250-20-75.5	55 853	18.6	73 131	35.9	86 365	42.0
mik.500-5-100.3	33 967	19.6	1 870	4.5	2 870	4.5
mik.250-5-75.1	61 46	20.5	14 11	12.0	12 491	10.4
mik.250-5-75.4	65 293	20.9	15 208	13.3	15 708	12.8
mik.250-10-75.2	56 824	21.8	25 286	19.1	29 213	27.0
mik.500-10-100.1	34 205	22.0	55 462	41.0	61 905	52.1
mik.500-20-75.5	27 473	22.7	13 276	11.6	17 998	14.4
mik.250-10-75.1	72 785	24.4	28 938	22.7	45 435	37.4
mik.500-10-75.4	41 72	25.2	9 468	9.7	10 256	11.3
mik.500-20-75.1	33 728	27.1	39 788	31.7	37 244	32.3
mik.500-20-75.4	39 78	29.8	31 639	22.4	55 784	37.2
mik.250-1-75.5	95 260	30.5	32 585	18.7	16 374	14.5
mik.500-1-75.5	71 600	36.0	21 686	15.6	16 183	14.9
mik.500-5-100.2	58 191	36.6	17 778	18.2	13 888	15.9
mik.500-10-75.1	45 868	37.8	13 572	11.7	11 78	11.8
mik.500-5-75.4	76 343	40.4	10 20	8.0	11 958	10.3
mik.500-5-100.5	69 350	46.7	10 21	13.3	16 840	19.3
mik.500-5-75.3	110 762	58.6	21 921	18.7	14 848	13.0
mik.500-1-75.4	127 482	64.0	44 112	34.4	21 573	23.5
mik.250-1-100.4	175 563	64.2	68 722	39.1	37 384	29.6
mik.500-10-75.2	108 113	75.9	61 818	45.5	53 239	49.0
mik.500-10-75.3	132 467	98.8	56 702	44.8	37 204	35.9
mik.250-10-75.4	377 697	119.5	23 7	21.4	25 898	22.6
mik.500-1-100.3	228 923	145.9	82 456	54.4	81 488	74.2
mik.500-1-100.4	232 536	155.8	69 940	43.1	46 481	40.7
mik.500-1-100.2	360 457	232.4	37 624	26.0	27 138	26.2
mik.500-1-75.1	539 310	296.8	179 74	101.7	68 531	56.2
mik.250-1-100.3	1 691 811	710.8	153 606	98.9	80 514	81.1
mik.250-1-100.5	1 649 356	714.0	419 843	317.7	523 143	489.8
mik.500-20-75.3	943 717	805.8	36 765	32.8	60 38	53.2
mik.500-1-100.1	1 337 54	909.1	323 300	208.4	229 752	219.0
mik.500-1-100.5	1 402 912	1039.0	594 77	477.3	697 817	658.9
mik.250-1-100.1	8 98 779	3020.0	656 934	502.9	538 680	529.8
geom. mean	111 50	58.0	37 669	29.8	32 868	30.2
sh. geom. mean	111 133	65.2	37 763	33.6	32 942	34.4
arithm. mean	456 460	222.2	87 443	63.8	80 691	73.1

Table C.11. Detailed benchmark results on the mik test set. Values printed in red and blue indicate a difference of at least a factor of 2 to the reference Cplex results.

Appendix D

Notation

The notation used in the thesis is specified in Tables D.1 to D.5. Due to the finiteness of the set of available symbols, some are used with multiple meanings. It should be clear from the context, however, in which meaning the symbol is used in each case. Finally, Table D.6 contains a list of common abbreviations that we use throughout the thesis.

	sets
$\mathbb{R}$	set of real numbers
$\mathbb{Q}$	set of rational numbers
$\mathbb{Z}$	set of integer numbers
$\mathbb{N}$	set of natural numbers: $\mathbb{N} = \{1, 2, 3, \ldots\}$
$\mathbb{R}^S$	vectors indexed by a set: $x \in \mathbb{R}^S \Leftrightarrow x = (x_j)_{j \in S}$ with $x_j \in \mathbb{R}$
$S_{\geq 0}$	subset of non-negative numbers: $S_{\geq 0} = \{s \in S \mid s \geq 0\}$
$S_{>0}$	subset of positive numbers: $S_{>0} = \{s \in S \mid s > 0\}$
$S_{\leq 0}$	subset of non-positive numbers: $S_{\geq 0} = \{s \in S \mid s \leq 0\}$
$S_{<0}$	subset of negative numbers: $S_{>0} = \{s \in S \mid s < 0\}$
$\mathbb{1}$	vector of all ones: $\mathbb{1} = (1, \ldots, 1)^T$
$\text{conv}(S)$	convex hull of set S
P	polyhedron; linear programming polyhedron
P_I	integer hull: convex hull of integer points in P
X	set of solutions to a feasibility or optimization problem instance
X_K	set of solutions to a knapsack problem instance
P_K	knapsack polyhedron
X_{MK}	set of solutions to a mixed knapsack problem instance
P_{MK}	mixed knapsack polyhedron
X_{SNF}	set of solutions to a single node flow problem instance
	runtime
$\mathcal{P}$	class of polynomial solvable problems
$\mathcal{NP}$	class of nondeterministic polynomial solvable problems
$\mathcal{O}(f(n))$	class of algorithms with asymptotic runtime of at most $f(n)$, $n \to \infty$
	operators
$x \leq y$	vector comparison for $x, y \in \mathbb{R}^n$: $x \leq y :\Leftrightarrow \forall j = 1, \ldots, n : x_j \leq y_j$
$x \in [a, b]$	vector range for $x, a, b \in \mathbb{R}^n$: $x \in [a, b] :\Leftrightarrow a \leq x \leq b$
$a \oplus b$	exclusive or: $a \oplus b :\Leftrightarrow (a \vee b) \wedge \neg(a \wedge b)$
$c = a \bmod b$	modulus operator for $a \in \mathbb{Z}$, $b \in \mathbb{Z}_{>0}$: $c = a \bmod b \Leftrightarrow a = \lfloor \frac{a}{b} \rfloor b + c$
$\text{scm}(q_1, \ldots, q_k)$	smallest common multiple of $q_1, \ldots, q_k \in \mathbb{Z}_{>0}$
$\text{gcd}(p_1, \ldots, p_k)$	greatest common divisor of $p_1, \ldots, p_k \in \mathbb{Z}$
$\gamma(v_1, \ldots, v_k)$	geometric mean of $v_1, \ldots, v_k$ with $v_i \in \mathbb{R}_{\geq 0}$
$\gamma_s(v_1, \ldots, v_k)$	shifted geometric mean of $v_1, \ldots, v_k$ with $v_i \in \mathbb{R}_{\geq 0}$
$\varnothing(v_1, \ldots, v_k)$	arithmetic mean of $v_1, \ldots, v_k$ with $v_i \in \mathbb{R}$

Table D.1. Notation.

	graphs
$d(v)$	degree of node v in a graph, total degree of node v in a digraph
$d^+(v)$	in-degree of node v in digraph $D = (V, A)$: $d^+(v) = \|\{(u, v) \in A\}\|$
$d^-(v)$	out-degree of node v in digraph $D = (V, A)$: $d^-(v) = \|\{(v, w) \in A\}\|$
$\delta(v)$	set of neighbors of of v in a graph or digraph
$\delta^+(v)$	set of predecessors of v in a digraph
$\delta^-(v)$	set of successors of v in a digraph
	optimization problems
$\mathcal{D}$	a domain of a variable: $\mathcal{D} \subseteq \mathbb{R}$
$\mathfrak{D}$	the set of domains: $\mathfrak{D} = \{\mathcal{D}_1, \ldots, \mathcal{D}_n\}$
$\mathcal{C}$	a constraint of the problem instance
$\mathfrak{C}$	the set of constraints in the problem instance
N	the set of variables: $N = \{1, \ldots, n\}$
I	the set of integer variables: $I \subseteq N$
B	the set of binary variables: $B \subseteq I$
C	the set of continuous variables: $C = N \setminus I$
x	the vector of variables of a general optimization problem
$\bar{x}_j$	the negation of $x_j \in \{0, 1\}$
ℓ_j	a literal: $\ell_j \in \{x_j, \bar{x}_j\}$
L	the set of literals
l, u	global bounds of the variables
$\tilde{l}$, $\tilde{u}$	local bounds of the variables
f	the objective function of a general optimization problem
c	the objective vector of an MIP, LP, or CIP
A	the coefficient matrix of an MIP or LP
b	the right hand side vector of an MIP or LP
x	the primal variable vector vector of an MIP or LP
s	the slack vector of an MIP or LP
y	the dual variable vector of an LP
r	the reduced cost vector of an LP
	solutions
X	the set of feasible solutions of an optimization or feasibility problem
$x^\star$	an optimal solution to a problem instance
$c^\star$	optimal objective value of an MIP or CIP
$\hat{x}$	the current incumbent solution; a feasible solution; a partial solution
$\hat{c}$	objective value of current incumbent solution of an MIP or CIP
$\dot{x}$	an infeasible solution vector
$\underline{\check{c}}$	current global dual (lower) bound of an MIP or CIP
γ	relative primal-dual gap
$\check{c}$	optimal objective value of an LP or a relaxation of an MIP or CIP
$\check{x}$	optimal primal solution of an LP; an optimal solution to a relaxation
$\check{s}$	optimal slack solution of an LP
$\check{y}$	optimal dual solution of an LP
$\check{r}$	optimal reduced costs of an LP
$\check{x}_R$	optimal primal solution of the root node LP
$\check{r}_R$	optimal reduced costs of the root node LP
$\check{c}_R$	objective value of the root node LP relaxation

Table D.2. Notation (continued).

	numerics
$\hat{\delta}$	primal feasibility tolerance of SCIP (default is $\hat{\delta} = 10^{-6}$)
$\check{\delta}$	dual feasibility (LP optimality) tolerance of SCIP (default is $\check{\delta} = 10^{-9}$)
ϵ	zero tolerance of SCIP (default is $\epsilon = 10^{-9}$)
$x \doteq y$	equality within feasibility tolerance: $x \doteq y \;\Leftrightarrow\; \frac{\lvert x-y\rvert}{\max\{\lvert x\rvert,\lvert y\rvert,1\}} \leq \hat{\delta}$
	constraints
ALLDIFF	all-different constraint: all variables must take a different value
ELEMENT	element constraint: selects a single element of a list
NOSUBTOUR	no-subtour constraint: forbids subtours in a TSP instance
SPPC	set packing, partitioning, or covering constraint
$\underline{\beta}$, $\overline{\beta}$	left and right hand sides of a linear constraint $\underline{\beta} \leq a^T x \leq \overline{\beta}$
a	coefficient vector of a linear constraint
α	current activity of a linear constraint: $\alpha(x) = a^T x$
$\underline{\alpha}$, $\overline{\alpha}$	activity bounds of a linear constraint w.r.t. the variables' bounds
F_0	current number of variables of a SPPC constraint that are fixed to 0
F_1	current number of variables of a SPPC constraint that are fixed to 1
	branch-and-bound
R	global problem, root of the search tree
Q	current subproblem
S	sibling of the current subproblem
$p(Q)$	parent node of subproblem Q
L	leaf subproblem in the search tree
$\mathcal{S}$	list of siblings of the current subproblem
$\mathcal{C}$	list of children of the current subproblem
$\mathcal{L}$	list of open subproblems; leaves in the search tree
$\mathcal{A}$	active path from the root node to the current subproblem
Q_{relax}	relaxation of subproblem Q
Q_{LP}	LP relaxation of subproblem Q
	simplex algorithm
$\mathcal{B}$	set of basic variables
$\mathcal{N}$	set of non-basic variables
$\bar{a}$	row in the simplex tableau $\bar{A} = A_{\mathcal{B}}^{-1} A$
	data structures
Q	clique: set of binary variables of which at most one can set to 1
$\mathcal{Q}$	clique table

Table D.3. Notation (continued).

	branching
F	branching candidate set: integer variables with fractional LP value
f^-, f^+	distance to rounded values: $f_j^- = \check{x}_j - \lfloor \check{x}_j \rfloor$, $f_j^+ = \lceil \check{x}_j \rceil - \check{x}_j$
Δ_j^-, Δ_j^+	objective increase in child nodes after branching on x_j
Ψ_j^-, Ψ_j^+	downwards and upwards pseudocost values for x_j
Φ_j^-, Φ_j^+	downwards and upwards inference values for x_j
λ	strong branching lookahead
κ	maximal number of strong branching candidates
γ	maximal number of simplex iterations for each strong branching LP
η_{rel}	reliability parameter
score	score function to map the two child score values to a single score
μ	weight of the maximum in the linear score function
s_j	branching score of variable x_j
$s_\varnothing$	current average score values of all variables in the problem instance
	node selection
$\gamma_{\max}$	gap closed on the active path for which the current plunge is aborted
e_Q	pseudocost objective estimate for node Q
e_Q^{proj}	projection objective estimate for node Q
	propagation and presolving
sig_i^+	positive signature vector of a linear constraint $\mathcal{C}_i$: $\text{sig}_i^+ \in \{0,1\}^{64}$
sig_i^-	negative signature vector of a linear constraint $\mathcal{C}_i$: $\text{sig}_i^- \in \{0,1\}^{64}$
$x \stackrel{\star}{=} y$	x is always equal to y
$x \stackrel{\star}{\neq} y$	x is always unequal to y
$x \stackrel{\star}{\neq} y$	x and y are a pair of negated binary variables
$x :\stackrel{\star}{=} y$	aggregation of x and y
$\tau(g)$	crushed form of the affine linear function g
ζ_j^-	number of constraints which down-lock x_j
ζ_j^+	number of constraints which up-lock x_j
$\omega(a)$	weighted support of a linear constraint with coefficient vector a
redl_j ,redu_j	redundancy bounds for a linear constraint
	cutting plane separation
$\mathcal{R}$	list of cutting planes
r	cutting plane $r : \underline{\gamma} \leq d^T x \leq \overline{\gamma}$
$\underline{\gamma}$, $\overline{\gamma}$	left and right hand side of a cutting plane
e	efficacy of a cutting plane: Euclidean distance to current LP solution
o	orthogonality of a cutting plane w.r.t. to all other cuts in $\mathcal{R}$
p	parallelism of a cutting plane w.r.t. the objective function
s	total score of a cutting plane
V	knapsack cover
L_0	set of variables selected for up-lifting
L_1	set of variables selected for down-lifting
f_j	fraction of a coefficient value: $f_j = a_j - \lfloor a_j \rfloor$
	primal heuristics
$\tilde{x}$	rounded solution vector; working solution vector of heuristic
$\Delta(x, \tilde{x})$	Manhattan distance of x and $\tilde{x}$
$\phi(\tilde{x}_j)$	fractionality of $\tilde{x}_j$

Table D.4. Notation (continued).

	conflict analysis
λ	conflict vertex in the conflict graph
V_r	reason side of the conflict graph
V_c	conflict side of the conflict graph
V_B	branching variable vertices in the conflict graph
V_f	conflict set
$\mathcal{C}_f$	conflict clause
$\mathcal{C}_u$	reconvergence clause
$\mathcal{B}_L$	set of local bound changes
$\mathcal{B}_f$	conflicting subset of local bound changes
$\mathcal{B}_C$	subset of local bound changes that renders the LP infeasible
	chip design verification
P	property constraint
ϱ	register
β	width of a register (number of bits)
W	global size of words
L	global size of nibbles
γ	width of a word: $\gamma \leq W$
δ	width of a nibble: $\delta \leq L$
ω	number of words in a register
η	number of nibbles in a register
$x[p]$	bit p in register x
$x[q,p]$	subword ranging from bits q to p in register x, $q \geq p$
$x\langle[p] \leftarrow y\rangle$	replacing bit p of x by $y \in \{0,1\}$
$x\langle[q,p] \leftarrow y\rangle$	replacing a subword $x[q,p]$ of x by the bits of y
$\mathcal{T}$	set of bit string terms
$t_1 \equiv t_2$	equivalence of two terms $t_1, t_2 \in \mathcal{T}$
$t\|_\beta$	truncating bit string term $t \in \mathcal{T}$ to β bits
$t_1 \otimes t_2$	concatenation of two bit string terms $t_1, t_2 \in \mathcal{T}$
$\beta_{\max}$	maximal widths of the bit string terms
$\mathcal{B}$	set of possible term widths: $\mathcal{B} = \{1, \ldots, \beta_{\max}\}$
$\succ_{\text{lex}}$	lexicographic (right-left) ordering w.r.t. $\succ$
$\succ_{\text{lrpo}}$	lexicographic recursive path ordering w.r.t. $\succ$

Table D.5. Notation (continued).

MIP	mixed integer program
MBP	mixed binary program
BP	binary program
IP	integer program
LP	linear program
CP	constraint program
CSP	constraint satisfaction problem
CP(FD)	finite domain constraint program
CSP(FD)	finite domain constraint satisfaction problem
CIP	constraint integer program
SAT	satisfiability problem
CNF	conjunctive normal form
BCP	Boolean constraint propagation
DPLL	Davis-Putnam-Logemann-Loveland Algorithm
MIR	mixed integer rounding
c-MIR	complemented mixed integer rounding
GMI	Gomory mixed integer cut
CG	Chvátal-Gomory cut
RINS	relaxation induced neighborhood search heuristic
RENS	relaxation enforced neighborhood search heuristic
UIP	unique implication point
FUIP	first unique implication point
IIS	irreducible inconsistent subsystem
SoC	Systems-on-Chips
PBC	pseudo-Boolean constraint
BDD	binary decision diagram
PCP	property checking problem

Table D.6. Abbreviations.

List of Algorithms

Bibliography

[1] T. Achterberg, *Conflict analysis in mixed integer programming*, Discrete Optimization **4**, no. 1 (2007), pp. 4–20. Special issue: Mixed Integer Programming.

[2] T. Achterberg and T. Berthold, *Improving the feasibility pump*, Discrete Optimization **4**, no. 1 (2007), pp. 77–86. Special issue: Mixed Integer Programming.

[3] T. Achterberg, M. Grötschel, and T. Koch, *Teaching MIP modeling and solving*, ORMS Today **33**, no. 6 (2006), pp. 14–15.

[4] T. Achterberg, T. Koch, and A. Martin, *The mixed integer programming library:* Miplib 2003. Zuse Institute Berlin, http://miplib.zib.de.

[5] T. Achterberg, T. Koch, and A. Martin, *Branching rules revisited*, Operations Research Letters **33** (2005), pp. 42–54.

[6] T. Achterberg, T. Koch, and A. Martin, *MIPLIB 2003*, Operations Research Letters **34**, no. 4 (2006), pp. 1–12.

[7] A. Aiba, K. Sakai, Y. Sato, D. J. Hawley, and R. Hasegawa, *Constraint logic programming language CAL*, in FGCS-88: Proceedings of the International Conference on Fifth Generation Computer Systems, Tokyo, 1988, pp. 263–276.

[8] S. B. Akers, *Binary decision diagrams*, IEEE Transactions on Computers **C-27**, no. 6 (1978), pp. 509–516.

[9] R. B. J. T. Allenby, *Rings, Fields and Groups*, Arnold, 1991.

[10] E. Althaus, A. Bockmayr, M. Elf, M. Jünger, T. Kasper, and K. Mehlhorn, *SCIL – symbolic constraints in integer linear programming*, Tech. Report ALCOMFT-TR-02-133, MPI Saarbrücken, May 2002.

[11] E. Amaldi, M. E. Pfetsch, and L. E. Trotter, Jr., *On the maximum feasible subsystem problem, IISs, and IIS-hypergraphs*, Mathematical Programming **95**, no. 3 (2003), pp. 533–554.

[12] C. Anders, *Das Chordalisierungspolytop und die Berechnung der Baumweite eines Graphen*, master's thesis, Technische Universität Berlin, 2006.

[13] G. Andreello, A. Caprara, and M. Fischetti, *Embedding cuts in a branch&cut framework: a computational study with* $\{0, \frac{1}{2}\}$*-cuts*, INFORMS Journal on Computing (2007). to appear.

[14] D. Applegate, R. E. Bixby, V. Chvátal, and W. Cook, *Finding cuts in the TSP*, Tech. Report 95-05, DIMACS, March 1995.

[15] D. Applegate, R. E. Bixby, V. Chvátal, and W. Cook, *On the solution of traveling salesman problems*, Documenta Mathematica **ICM III** (1998), pp. 645–656.

[16] D. Applegate, R. E. Bixby, V. Chvátal, and W. Cook, *TSP cuts which do not conform to the template paradigm*, in Computational Combinatorial Optimization, M. Jünger and D. Naddef, eds., Springer, Heidelberg, Germany, 2001, pp. 261–304.

[17] K. R. Apt, *Principles of Constraint Programming*, Cambridge University Press, 2003.

[18] M. Armbruster, *Branch-and-Cut for a Semidefinite Relaxation of the Minimum Bisection Problem*, PhD thesis, Technische Universität Chemnitz, 2007.

[19] M. Armbruster, M. Fügenschuh, C. Helmberg, and A. Martin, *Experiments with linear and semidefinite relaxations for solving the minimum graph bisection problem*, tech. report, Darmstadt University of Technology, November 2006.

[20] M. ARMBRUSTER, M. FÜGENSCHUH, C. HELMBERG, AND A. MARTIN, *On the bisection cut polytope*, preprint, Darmstadt University of Technology, November 2006.

[21] I. D. ARON, J. N. HOOKER, AND T. H. YUNES, *SIMPL: A system for integrating optimization techniques*, in Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, First International Conference, CPAIOR 2004, J.-C. Régin and M. Rueher, eds., Lecture Notes in Computer Science 3011, Nice, France, April 2004, Springer, pp. 21–36.

[22] A. ATAMTÜRK, *Flow pack facets of the single node fixed-charge flow polytope*, Operations Research Letters **29** (2001), pp. 107–114.

[23] A. ATAMTÜRK, *On the facets of the mixed–integer knapsack polyhedron*, Mathematical Programming **98** (2003), pp. 145–175.

[24] A. ATAMTÜRK, G. L. NEMHAUSER, AND M. W. P. SAVELSBERGH, *Conflict graphs in integer programming*, European Journal of Operations Research **121** (2000), pp. 40–55.

[25] A. ATAMTÜRK AND D. RAJAN, *On splittable and unsplittable capacitated network design arc-set polyhedra*, Mathematical Programming **92** (2002), pp. 315–333.

[26] A. ATAMTÜRK AND M. W. P. SAVELSBERGH, *Integer-programming software systems*, Annals of Operations Research **140**, no. 1 (2005), pp. 67–124.

[27] G. AUDEMARD, P. BERTOLI, A. CIMATTI, A. KORNILOWICZ, AND R. SEBASTIANI, *A SAT-based approach for solving formulas over boolean and linear mathematical propositions*, in Proc. Conference on Automated Deduction (CADE), 2002, pp. 195–210.

[28] E. BALAS, *Facets of the knapsack polytope*, Mathematical Programming **8** (1975), pp. 146–164.

[29] E. BALAS, S. CERIA, AND G. CORNUÉJOLS, *A lift-and-project cutting plane algorithm for mixed 0-1 programs*, Mathematical Programming **58** (1993), pp. 295–324.

[30] E. BALAS, S. CERIA, AND G. CORNUÉJOLS, *Mixed 0-1 programming by lift-and-project in a branch-and-cut framework*, Management Science **42** (1996), pp. 1229–1246.

[31] E. BALAS, S. CERIA, G. CORNUÉJOLS, AND N. NATRAJ, *Gomory cuts revisited*, Operations Research Letters **19** (1996), pp. 1–9.

[32] E. BALAS, S. CERIA, M. DAWANDE, F. MARGOT, AND G. PATAKI, *OCTANE: A new heuristic for pure 0-1 programs*, Operations Research **49**, no. 2 (2001), pp. 207–225.

[33] E. BALAS AND C. H. MARTIN, *Pivot-and-complement: A heuristic for 0-1 programming*, Management Science **26**, no. 1 (1980), pp. 86–96.

[34] E. BALAS, S. SCHMIETA, AND C. WALLACE, *Pivot and shift - a mixed integer programming heuristic*, Discrete Optimization **1**, no. 1 (2004), pp. 3–12.

[35] E. BALAS AND E. ZEMEL, *Facets of the knapsack polytope from minimal covers*, SIAM Journal on Applied Mathematics **34** (1978), pp. 119–148.

[36] E. BALAS AND E. ZEMEL, *Lifting and complementing yields all the facets of positive zero-one programming polytopes*, in Proceedings of the International Conference on Mathematical Programming, Rio de Janeiro, 1981, R. W. Cottle, M. I. Kelmanson, and B. Korte, eds., Elsevier Science Publishers, 1984, pp. 13–24.

[37] E. M. L. BEALE, *Branch and bound methods for mathematical programming systems*, in Discrete Optimization II, P. L. Hammer, E. L. Johnson, and B. H. Korte, eds., North Holland Publishing Co., 1979, pp. 201–219.

[38] E. M. L. BEALE AND J. A. TOMLIN, *Special facilities in a general mathematical programming system for non-convex problems using ordered sets of variables*, in OR 69: Proceedings of the Fifth International Conference on Operations Research, J. Lawrence, ed., London, 1970, Tavistock Publications, pp. 447–454.

[39] M. Bénichou, J. M. Gauthier, P. Girodet, G. Hentges, G. Ribière, and O. Vincent, *Experiments in mixed-integer linear programming*, Mathematical Programming **1** (1971), pp. 76–94.

[40] L. Bertacco, M. Fischetti, and A. Lodi, *A feasibility pump heuristic for general mixed-integer problems*, Tech. Report OR/05/5, DEIS – Università di Bologna, Italy, May 2005.

[41] T. Berthold, *Primal heuristics for mixed integer programs*, master's thesis, Technische Universität Berlin, 2006.

[42] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu, *Symbolic model checking using SAT procedures instead of BDDs*, in Proceedings of the International Design Automation Conference (DAC-99), June 1999, pp. 317–320.

[43] A. Biere, E. M. Clarke, R. Raimi, and Y. Zhu, *Verifying safety properties of a Power PC microprocessor using symbolic model checking without BDDs*, in Computer-Aided Verification, Lecture Notes in Computer Science 1633, Springer, 1999, pp. 60–71.

[44] A. Biere and W. Kunz, *SAT and ATPG: Boolean engines for formal hardware verification*, in ACM/IEEE Intl. Conf. on Computer-Aided Design (ICCAD), San Jose, November 2002.

[45] E. Bilgen, *Personalkostenminimierung bei der Einsatzplanung von parallelen identischen Bearbeitungszentren in der Motorradproduktion*, master's thesis, Technische Universität Chemnitz, 2007. to appear.

[46] R. E. Bixby, M. Fenelon, Z. Gu, E. Rothberg, and R. Wunderling, *MIP: Theory and practice – closing the gap*, in Systems Modelling and Optimization: Methods, Theory, and Applications, M. Powell and S. Scholtes, eds., Kluwer, 2000, pp. 19–49.

[47] P. Bjesse, T. Leonard, and A. Mokkedem, *Finding bugs in an Alpha microprocessor using satisfiability solvers*, in Computer-Aided Verification, Lecture Notes in Computer Science 2102, Springer, 2001, pp. 454–464.

[48] A. Bley, F. Kupzog, and A. Zymolka, *Auslegung heterogener Kommunikationsnetze nach Performance und Wirtschaftlichkeit*, in Proceedings of 11th Kasseler Symposium Energie-Systemtechnik: Energie und Kommunikation, Kassel, November 2006, pp. 84–97.

[49] A. Bockmayr and T. Kasper, *Branch-and-infer: A unifying framework for integer and finite domain constraint programming*, INFORMS Journal on Computing **10**, no. 3 (1998), pp. 287–300.

[50] A. Bockmayr and N. Pisaruk, *Solving assembly line balancing problems by combining IP and CP*. Sixth Annual Workshop of the ERCIM Working Group on Constraints, June 2001.

[51] R. Borndörfer, C. E. Ferreira, and A. Martin, *Decomposing matrices into blocks*, SIAM Journal on Optimization **9** (1998), pp. 236–269.

[52] R. Borndörfer and Z. Kormos, *An algorithm for maximum cliques*. Manuscript, 1997.

[53] V. J. Bowman and G. L. Nemhauser, *A finiteness proof for the modified Dantzig cuts in integer programming*, Naval Research Logistics Quarterly **17** (1970), pp. 309–313.

[54] R. Brinkmann, *Preprocessing for Property Checking of Sequential Circuit on the Register Transfer Level*, PhD thesis, University of Kaiserslautern, Kaiserslautern, Germany, 2003.

[55] R. Brinkmann and R. Drechsler, *RTL-datapath verification using integer linear programming*, in Proceedings of the IEEE VLSI Design Conference, 2002, pp. 741–746.

[56] A. BROOKE, D. KENDRICK, A. MEERAUS, R. RAMAN, AND R. E. ROSENTHAL, *GAMS - a user's guide*, December 1998. GAMS Developement Corporation, http://www.gams.com.

[57] R. E. BRYANT, *Graph-based algorithms for Boolean function manipulation*, IEEE Transactions on Computers **C-35**, no. 8 (1986), pp. 677–691.

[58] A. CESELLI, M. GATTO, M. LÜBBECKE, M. NUNKESSER, AND H. SCHILLING, *Optimizing the cargo express service of swiss federal railways*. submitted to Transportation Science, 2007.

[59] D. CHAI AND A. KUEHLMANN, *A fast pseudo-boolean constraint solver*, in Proceedings of the Design Automation Conference (DAC-03), 2003, pp. 830–835.

[60] V. CHVÁTAL, *Edmonds polytopes and a hierarchy of combinatorial problems*, Discrete Mathematics **4** (1973), pp. 305–337.

[61] E. CLARKE, A. BIERE, R. RAIMI, AND Y. ZHU, *Bounded model checking using satisfiability solving*, Formal Methods in System Design **19**, no. 1 (2001).

[62] J. M. CLOCHARD AND D. NADDEF, *Using path inequalities in a branch-and-cut code for the symmetric traveling salesman problem*, in Proceedings on the Third IPCO Conference, L. Wolsey and G. Rinaldi, eds., 1993, pp. 291–311.

[63] COIN-OR, *Computational infrastructure for operations research*. http://www.coin-or.org.

[64] A. COLMERAUER, *Total precedence relations*, Journal of the ACM **17**, no. 1 (1970), pp. 14–30.

[65] A. COLMERAUER, *An introduction to Prolog III*, Communications of the ACM **33**, no. 7 (1990), pp. 69–90.

[66] A. COLMERAUER, H. KANOUI, R. PASERO, AND P. ROUSSEL, *Un système de communication en français*, tech. report, Groupe Intelligence Artificielle, Université Aix-Marseille II, France, 1972.

[67] W. T. COMFORT, *Multiword list items*, Communications of the ACM **7**, no. 6 (1964), pp. 357–362.

[68] S. A. COOK, *The complexity of theorem proving procedures*, in Proceedings of 3rd Annual ACM Symposium on the Theory of Computing, 1971, pp. 151–158.

[69] G. CORNUÉJOLS AND Y. LI, *Elementary closures for integer programs*, Operations Research Letters **28**, no. 1 (2001), pp. 1–8.

[70] H. CROWDER, E. L. JOHNSON, AND M. W. PADBERG, *Solving large scale zero-one linear programming problems*, Operations Research **31** (1983), pp. 803–834.

[71] R. J. DAKIN, *A tree search algorithm for mixed integer programs*, Computer Journal **8**, no. 3 (1965), pp. 250–255.

[72] E. DANNA, E. ROTHBERG, AND C. LE PAPE, *Exploring relaxation induced neighborhoods to improve MIP solutions*, Mathematical Programming **102**, no. 1 (2005), pp. 71–90.

[73] G. B. DANTZIG, *Maximization of a linear function of variables subject to linear inequalities*, in Activity Analysis of Production and Allocation, T. Koopmans, ed., John Wiley & Sons, New York, 1951, pp. 339–347.

[74] G. B. DANTZIG, *Linear programming and extensions*, Princeton University Press, Princeton, New Jersey, 1963.

[75] DASH OPTIMIZATION, XPRESS-MOSEL. http://www.dashoptimization.com.

[76] DASH OPTIMIZATION, XPRESS-MP. http://www.dashoptimization.com.

[77] M. DAVIS, G. LOGEMANN, AND D. LOVELAND, *A machine program for theorem proving*, Communications of the ACM **5** (1962), pp. 394–397.

[78] M. Davis and H. Putnam, *A computing procedure for quantification theory*, Journal of the Association for Computing Machinery **7** (1960), pp. 201–215.

[79] N. Dershowitz, *Orderings for term-rewriting systems*, Theoretical Computer Science **17**, no. 3 (1982), pp. 279–301.

[80] M. Dincbas, P. van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier, *The constraint logic programming language CHiP*, in FGCS-88: Proceedings of the International Conference on Fifth Generation Computer Systems, Tokyo, 1988, pp. 693–702.

[81] A. Dix, *Das Statische Linienplanungsproblem*, master's thesis, Technische Universität Berlin, 2007.

[82] N. Eén and N. Sörensson, *An extensible SAT-solver*, in Proceedings of SAT 2003, E. Giunchiglia and A. Tacchella, eds., Springer, 2003, pp. 502–518.

[83] F. Fallah, S. Devadas, and K. Keutzer, *Functional vector generation for HDL models using linear programming and boolean satisfiability*, IEEE Transactions on CAD **CAD-20**, no. 8 (2001), pp. 994–1002.

[84] M. Fischetti, F. Glover, and A. Lodi, *The feasibility pump*, Mathematical Programming **104**, no. 1 (2005), pp. 91–104.

[85] M. Fischetti and A. Lodi, *Local branching*, Mathematical Programming **98**, no. 1-3 (2003), pp. 23–47.

[86] J. J. H. Forrest, *COIN branch and cut*. COIN-OR, http://www.coin-or.org.

[87] J. J. H. Forrest, D. de la Nuez, and R. Lougee-Heimer, *CLP user guide*. COIN-OR, http://www.coin-or.org/Clp/userguide.

[88] J. J. H. Forrest, J. P. H. Hirst, and J. A. Tomlin, *Practical solution of large scale mixed integer programming problems with UMPIRE*, Management Science **20**, no. 5 (1974), pp. 736–773.

[89] R. Fourer, D. M. Gay, and B. W. Kernighan, *AMPL: A Modelling Language for Mathematical Programming*, Duxbury Press, Brooks/Cole Publishing Company, 2nd ed., November 2002.

[90] A. Fügenschuh and A. Martin, *Computational integer programming and cutting planes*, in Discrete Optimization, K. Aardal, G. L. Nemhauser, and R. Weismantel, eds., Handbooks in Operations Research and Management Science 12, Elsevier, 2005, ch. 2, pp. 69–122.

[91] H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli, *Dpll(t): Fast decision procedures*, in Proceedings of the International Conference on Computer Aided Verification (CAV-04), July 2004, pp. 26–37.

[92] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W.H. Freeman and Company, New York, 1979.

[93] J. M. Gauthier and G. Ribière, *Experiments in mixed-integer linear programming using pseudocosts*, Mathematical Programming **12**, no. 1 (1977), pp. 26–47.

[94] M. L. Ginsberg, *Dynamic backtracking*, Journal of Artificial Intelligence Research **1** (1993), pp. 25–46.

[95] F. Glover and M. Laguna, *General purpose heuristics for integer programming - part I*, Journal of Heuristics **3** (1997).

[96] F. Glover and M. Laguna, *General purpose heuristics for integer programming - part II*, Journal of Heuristics **3** (1997).

[97] F. Glover and M. Laguna, *Tabu Search*, Kluwer Academic Publisher, Boston, Dordrecht, London, 1997.

[98] F. GLOVER, A. LØKKETANGEN, AND D. L. WOODRUFF, *Scatter search to generate diverse MIP solutions*, in OR Computing Tools for Modeling, Optimization and Simulation: Interfaces in Computer Science and Operations Research, M. Laguna and J. González-Velarde, eds., Kluwer Academic Publishers, 2000, pp. 299–317.

[99] GNU, *The GNU linear programming kit*. http://www.gnu.org/software/glpk/.

[100] E. GOLDBERG AND Y. NOVIKOV, *Berkmin: A fast and robust SAT solver*, in Design Automation and Test in Europe (DATE), 2002, pp. 142–149.

[101] C. GOMES, B. SELMAN, AND H. KAUTZ, *Boosting combinatorial search through randomization*, in Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI-98), July 1998.

[102] R. E. GOMORY, *Outline of an algorithm for integer solutions to linear programs*, Bulletin of the American Society **64** (1958), pp. 275–278.

[103] R. E. GOMORY, *Solving linear programming problems in integers*, in Combinatorial Analysis, R. Bellman and J. M. Hall, eds., Symposia in Applied Mathematics X, Providence, RI, 1960, American Mathematical Society, pp. 211–215.

[104] R. E. GOMORY, *An algorithm for integer solutions to linear programming*, in Recent Advances in Mathematical Programming, R. L. Graves and P. Wolfe, eds., New York, 1963, McGraw-Hill, pp. 269–302.

[105] R. E. GOMORY, *Early integer programming*, Operations Research **50**, no. 1 (2002), pp. 78–81.

[106] J. GOTTLIEB AND L. PAULMANN, *Genetic algorithms for the fixed charge transportation problem*, in Proceedings of the 1998 IEEE International Conference on Evolutionary Computation, IEEE Press, 1998, pp. 330–335.

[107] M. GRÖTSCHEL AND O. HOLLAND, *Solution of large-scale symmetric travelling salesman problems*, Mathematical Programming **51**, no. 2 (1991), pp. 141–202.

[108] M. GRÖTSCHEL AND M. W. PADBERG, *On the symmetric traveling salesman problem I: Inequalities*, Mathematical Programming **16** (1979), pp. 265–280.

[109] M. GRÖTSCHEL AND M. W. PADBERG, *On the symmetric traveling salesman problem II: Lifting theorems and facets*, Mathematical Programming **16** (1979), pp. 281–302.

[110] Z. GU, G. L. NEMHAUSER, AND M. W. P. SAVELSBERGH, *Lifted flow covers for mixed 0-1 integer programs*, Mathematical Programming **85**, no. 3 (1999), pp. 439–467.

[111] Z. GU, G. L. NEMHAUSER, AND M. W. P. SAVELSBERGH, *Sequence independent lifting in mixed integer programming*, Journal of Combinatorial Optimization **4**, no. 1 (2000), pp. 109–129.

[112] P. L. HAMMER, E. L. JOHNSON, AND U. N. PELED, *Facets of regular 0-1 polytopes*, Mathematical Programming **8** (1975), pp. 179–206.

[113] P. HANSEN, M. LABBÉ, AND D. SCHINDL, *Set covering and packing formulations of graph coloring: algorithms and first polyhedral results*, tech. report, GERAD, 2005.

[114] S. HEIPCKE, *Applications of Optimization with Xpress-MP*, Dash Optimization, Blisworth, U.K., 2002.

[115] P. V. HENTENRYCK, *Constraint satisfaction in logic programming*, MIT Press, Cambridge, MA, USA, 1989.

[116] F. S. HILLIER, *Efficient heuristic procedures for integer linear programming with an interior*, Operations Research **17** (1969), pp. 600–637.

[117] IBM CORP., *Mathematical Programming System Extended/370 (MPSX/370) program reference manual*, 1979. SH19-1095-3, 4th Ed.

[118] ILOG, CPLEX. http://www.ilog.com/products/cplex.

[119] ILOG, OPL: *Optimization programming language*. http://www.ilog.com/products/oplstudio.

[120] *International technology roadmap for semiconductors*, 2005. http://public.itrs.net.

[121] J. Jaffar and J.-L. Lassez, *Constraint logic programming*, in Proceedings of the 14th ACM Symposium on Principles of Programming Languages, Munich, 1987, pp. 111–119.

[122] V. Jain and I. E. Grossmann, *Algorithms for hybrid MILP/CP models for a class of optimization problems*, INFORMS Journal on Computing **13**, no. 4 (2001), pp. 258–276.

[123] A. A. Jerraya and W. Wolf, *Multiprocessor Systems-on-Chips*, The Morgan Kaufmann Series in Systems on Silicon, Elsevier / Morgan Kaufman, 2004.

[124] Y. Jiang, T. Richards, and B. Richards, *No-good backmarking with min-conflict repair in constraint satisfaction and optimization*, in Principles and Practice of Constraint Programming, Lecture Notes in Computer Science 874, 1994, pp. 21–39.

[125] E. L. Johnson and M. W. Padberg, *Degree-two inequalities, clique facets, and biperfect graphs*, Annals of Discrete Mathematics **16** (1982), pp. 169–187.

[126] M. Joswig and M. E. Pfetsch, *Computing optimal morse matchings*, SIAM Journal on Discrete Mathematics **20**, no. 1 (2006), pp. 11–25.

[127] V. Kaibel, M. Peinhardt, and M. E. Pfetsch, *Orbitopal fixing*, in Proceedings of the 12th Integer Programming and Combinatorial Optimization conference (IPCO), M. Fischetti and D. Williamson, eds., LNCS 4513, Springer-Verlag, 2007, pp. 74–88.

[128] S. Kamin and J.-J. Lévy, *Two generalizations of the recursive path ordering*. Unpublished note, Department of Computer Science, University of Illinois, Urbana, IL. Available from http://www.ens-lyon.fr/LIP/REWRITING/OLD_PUBLICATIONS_ON_TERMINATION/KAMIN_LEVY/kamin-levy80spo.p February 1980.

[129] H. Keller, U. Pferschy, and D. Pisinger, *Knapsack Problems*, Springer-Verlag, Berlin, 2004.

[130] L. G. Khachiyan, *A polynomial algorithm in linear programming*, Doklady Akademii Nauk SSSR **244** (1979), pp. 1093–1096. in Russian.

[131] L. G. Khachiyan, *A polynomial algorithm in linear programming*, Soviet Mathematics Doklady **20** (1979), pp. 191–194. English translation.

[132] A. Klar, *Cutting planes in mixed integer programming*, master's thesis, Technische Universität Berlin, 2006.

[133] T. Koch, Zimpl *user guide*, Tech. Report 01-20, Zuse Institute Berlin, 2001. http://www.zib.de/koch/zimpl.

[134] T. Koch, *Rapid Mathematical Prototyping*, PhD thesis, Technische Universität Berlin, 2004.

[135] T. Koch, *Rapid mathematical programming or how to solve sudoku puzzles in a few seconds*, in Operations Research Proceedings 2005, H.-D. Haasis, H. Kopfer, and J. Schönberger, eds., 2006, pp. 21–26.

[136] R. A. Kowalski, *Predicate logic as programming language*, in Proceedings of the Sixth IFIP Congress, J. L. Rosenfeld, ed., Information Processing 74, Stockholm, Sweden, August 1974, pp. 569–574.

[137] R. P. Kurshan, *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach*, Princeton Series in Computer Science, Princeton University Press, Princeton, New Jersey, 1994, ch. 8, pp. 170–172.

[138] M. Kutschka, *Algorithmen zur Separierung von* $\{0, \frac{1}{2}\}$*-Schnitten*, master's thesis, Technische Universität Berlin, 2007. to appear.

[139] A. LAND AND S. POWELL, *Computer codes for problems of integer programming*, Ann. of Discrete Math. **5** (1979), pp. 221–269.

[140] M. LECONTE, *A bounds-based reduction scheme for constraints of difference*, in Proceedings of the Constraint-96, Second International Workshop on Constraint-based Reasoning, Key West, Florida, 1996, pp. 19–28.

[141] S. LEIPERT, *The tree interface – version 1.0 user manual*, Tech. Report 96.242, Institut für Informatik, Universität zu Köln, 1996. http://www.informatik.uni-koeln.de/ls_juenger/research/vbctool.

[142] A. N. LETCHFORD AND A. LODI, *Strengthening Chvátal-Gomory cuts and Gomory fractional cuts*, Operations Research Letters **30**, no. 2 (2002), pp. 74–82.

[143] C. M. LI AND ANBULAGAN, *Heuristics based on unit propagation for satisfiability problems*, in Proceedings of 15th International Joint Conference on Artificial Interlligence (IJCAI 1997), Japan, 1997, Morgan Kaufmann, pp. 366–371.

[144] C. M. LI AND ANBULAGAN, *Look-ahead versus look-back for satisfiability problems*, in Proceedings of third international conference on Principles and Practice of Constraint Programming (CP 1997), Autriche, 1997, Springer, pp. 342–356.

[145] J. T. LINDEROTH AND T. K. RALPHS, *Noncommercial software for mixed-integer linear programming*, in Integer Programming: Theory and Practice, J. Karlof, ed., Operations Research Series, CRC Press, 2005, pp. 253–303.

[146] J. T. LINDEROTH AND M. W. P. SAVELSBERGH, *A computational study of search strategies for mixed integer programming*, INFORMS Journal on Computing **11** (1999), pp. 173–187.

[147] LINDO SYSTEMS, INC., LINDO. http://www.lindo.com.

[148] LINDO SYSTEMS, INC., LINGO. http://www.lindo.com.

[149] J. D. C. LITTLE, K. G. MURTY, D. W. SWEENEY, AND C. KAREL, *An algorithm for the traveling salesman problem*, Operations Research **21** (1963), pp. 972–989.

[150] A. LØKKETANGEN AND F. GLOVER, *Solving zero/one mixed integer programming problems using tabu search*, European Journal of Operations Research **106** (1998), pp. 624–658.

[151] A. LÓPEZ-ORTIZ, C.-G. QUIMPER, J. TROMP, AND P. VAN BEEK, *A fast and simple algorithm for bounds consistency of the alldifferent constraint*, in Proceedings of the 18th International Joint Conference on Artificial Intelligence, Acapulco, Mexico, August 2003, pp. 245–250.

[152] J. C. MADRE AND J. P. BILLON, *Proving circuit correctness using formal comparison between expected and extracted behavior*, in Proceedings of the 25th Design Automation Conference, June 1988, pp. 205–210.

[153] H. MARCHAND, *A polyhedral study of the mixed knapsack set and its use to solve mixed integer programs*, PhD thesis, Faculté des Sciences Appliquées, Université catholique de Louvain, 1998.

[154] H. MARCHAND, A. MARTIN, R. WEISMANTEL, AND L. A. WOLSEY, *Cutting planes in integer and mixed integer programming*, Discrete Applied Mathematics **123/124** (2002), pp. 391–440.

[155] H. MARCHAND AND L. A. WOLSEY, *Aggregation and mixed integer rounding to solve MIPs*, Operations Research **49**, no. 3 (2001), pp. 363–371.

[156] H. M. MARKOWITZ AND A. S. MANNE, *On the solution of discrete programming problems*, Econometrica **25** (1957), pp. 84–110.

[157] J. P. MARQUES-SILVA AND K. A. SAKALLAH, *GRASP: A search algorithm for propositional satisfiability*, IEEE Transactions of Computers **48** (1999), pp. 506–521.

[158] K. MARRIOTT AND P. J. STUCKEY, *Programming with Constraints: An Introduction*, MIT Press, 1998.

[159] A. Martin, *Integer programs with block structure*. Habilitations-Schrift, Technische Universität Berlin, 1998. http://www.zib.de/Publications/abstracts/SC-99-03/.

[160] A. Martin and R. Weismantel, *The intersection of knapsack polyhedra and extensions*, in Integer Programming and Combinatorial Optimization, R. E. Bixby, E. Boyd, and R.Z.Ríos-Mercado, eds., Proceedings of the 6th IPCO Conference, 1998, pp. 243–256. http://www.zib.de/Publications/abstracts/SC-97-61/.

[161] G. T. Martin, *An accelerated euclidean algorithm for integer linear programming*, in Recent Advances in Mathematical Programming, R. Graves and P. Wolfe, eds., New York, 1963, McGraw-Hill, pp. 311–317.

[162] Maximal Software, Inc., MPL *modelling system*. http://www.maximal-usa.com/mpl.

[163] A. Mehrotra and M. A. Trick, *A column generation approach for graph coloring*, INFORMS Journal on Computing **8**, no. 4 (1996), pp. 344–354.

[164] M. Milano, G. Ottosson, P. Refalo, and E. S. Thorsteinsson, *The role of integer programming techniques in constraint programming's global constraints*, INFORMS Journal on Computing **14**, no. 4 (2002).

[165] G. Mitra, *Investigations of some branch and bound strategies for the solution of mixed integer linear programs*, Mathematical Programming **4** (1973), pp. 155–170.

[166] H. Mittelmann, *Decision tree for optimization software: Benchmarks for optimization software*. http://plato.asu.edu/bench.html.

[167] Mosek, Mosek *optimization tools*. http://www.mosek.com.

[168] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, *Chaff: Engineering an efficient SAT solver*, in Proceedings of the Design Automation Conference, July 2001.

[169] D. Naddef, *Polyhedral theory and branch-and-cut algorithms for the symmetric TSP*, in The Traveling Salesman Problem and its Variations, G. Gutin and A. Punnen, eds., Kluwer, 2002.

[170] M. Nediak and J. Eckstein, *Pivot, cut, and dive: A heuristic for 0-1 mixed integer programming*, Tech. Report RRR 53-2001, Rutgers University, 2001.

[171] G. L. Nemhauser and M. W. P. Savelsbergh, *MINTO*. http://coral.ie.lehigh.edu/~minto.

[172] G. L. Nemhauser, M. W. P. Savelsbergh, and G. C. Sigismondi, *MINTO, a Mixed INTeger Optimizer*, Operations Research Letters **15** (1994), pp. 47–58.

[173] G. L. Nemhauser and M. A. Trick, *Scheduling a major college basketball conference*, Operations Research **46**, no. 1 (1998), pp. 1–8.

[174] G. L. Nemhauser and L. A. Wolsey, *Integer and Combinatorial Optimization*, John Wiley & Sons, 1988.

[175] G. L. Nemhauser and L. A. Wolsey, *A recursive procedure to generate all cuts for 0-1 mixed integer programs*, Mathematical Programming **46**, no. 3 (1990), pp. 379–390.

[176] M. Nunkesser, *Algorithm design and analysis of problems in manufacturing, logistic, and telecommunications: An algorithmic jam session*, PhD thesis, Eidgenössische Technische Hochschule ETH Zürich, 2006.

[177] http://www.opencores.org.

[178] S. Orlowski, A. M. C. A. Koster, C. Raack, and R. Wessäly, *Two-layer network design by branch-and-cut featuring MIP-based heuristics*, in Proceedings of the Third International Network Optimization Conference (INOC 2007), Spa, Belgium, 2007.

[179] M. W. PADBERG, *A note on zero-one programming*, Operations Research **23** (1975), pp. 833–837.

[180] M. W. PADBERG, *(1,k)-configurations and facets for packing problems*, Mathematical Programming **18** (1980), pp. 94–99.

[181] M. W. PADBERG, T. J. VAN ROY, AND L. A. WOLSEY, *Valid inequalities for fixed charge problems*, Operations Research **33**, no. 4 (1985), pp. 842–861.

[182] PARAGON DECISION TECHNOLOGY, AIMMS. http://www.aimms.com.

[183] G. PARTHASARATHY, M. K. IYER, K. T. CHENG, AND F. BREWER, *RTL SAT simplification by boolean and interval arithmetic reasoning*, in Proceedings of the International Conference on Computer-Aided Design (ICCAD-05), 2005.

[184] G. PARTHASARATHY, M. K. IYER, K. T. CHENG, AND L. C. WANG, *An efficient finite-domain constraint solver for rtl circuits*, in Proceedings of the International Design Automation Conference (DAC-04), June 2004.

[185] J. PATEL AND J. W. CHINNECK, *Active-constraint variable ordering for faster feasibility of mixed integer linear programs*, Mathematical Programming (2006). to appear.

[186] M. E. PFETSCH, *The Maximum Feasible Subsystem Problem and Vertex-Facet Incidences of Polyhedra*, PhD thesis, Technische Universität Berlin, 2002.

[187] M. E. PFETSCH, *Branch-and-cut for the maximum feasible subsystem problem*, Report 05–46, ZIB, 2005.

[188] J.-F. PUGET, *A C++ implementation of CLP*, Tech. Report 94-01, ILOG S.A., Gentilly, France, 1994.

[189] J.-F. PUGET, *A fast algorithm for the bound consistency of alldiff constraints*, in Proceedings of the Fifteenth National Conference on Artificial Intelligence, Madison, WI, July 1998, pp. 359–366.

[190] T. K. RALPHS, *SYMPHONY version 5.0 user's manual*, Tech. Report 04T-020, Lehigh University Industrial and Systems Engineering, 2004. http://branchandcut.org/SYMPHONY.

[191] RAVENBROOK, *The memory management reference*. http://www.memorymanagement.org/.

[192] J.-C. RÉGIN, *A filtering algorithm for constraints of difference in CSP*, in AAAI-94:Proceedings of the 12th National Conference on Artificial Intelligence, 1994, pp. 362–367.

[193] G. REINELT, *TSPLIB 95*, 1995. Institut für Angewandte Mathematik, Universität Heidelberg, http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95.

[194] E. ROTHBERG, *An evolutionary algorithm for polishing mixed integer programming solutions*, INFORMS Journal on Computing (2007). to appear.

[195] D. S. RUBIN AND R. L. GRAVES, *Strengthened Dantzig cuts for integer programming*, ORSA **20** (1972), pp. 178–182.

[196] L. RYAN, *Efficient algorithms for clause-learning SAT solvers*, master's thesis, Simon Fraser University, 2004.

[197] R. M. SALTZMAN AND F. S. HILLIER, *A heuristic ceiling point algorithm for general integer linear programming*, Management Science **38**, no. 2 (1992), pp. 263–283.

[198] T. SANDHOLM AND R. SHIELDS, *Nogood learning for mixed integer programming*, Tech. Report CMU-CS-06-155, Carnegie Mellon University, Computer Science Department, September 2006.

[199] M. W. P. SAVELSBERGH, *Preprocessing and probing techniques for mixed integer programming problems*, ORSA Journal on Computing **6** (1994), pp. 445–454.

[200] C. Schulte, *Comparing trailing and copying for constraint programming*, in Proceedings of the 1999 International Conference on Logic Programming, D. D. Schreye, ed., Las Cruces, NM, USA, November 1999, MIT Press, pp. 275–289.

[201] R. M. Stallman and G. J. Sussman, *Forward reasoning and dependency directed backtracking in a system for computer-aided circuit analysis*, Artificial Intelligence **9** (1977), pp. 135–196.

[202] I. E. Sutherland, *Sketchpad: A Man-Machine Graphical Communication System*, PhD thesis, Massachusetts Institute of Technology, Lincoln Lab, 1963.

[203] I. E. Sutherland, *Sketchpad: A man-machine graphical communication system*, in Proceedings of the 1963 Spring Joint Computer Conference, E. Johnson, ed., AFIPS Conference Proceedings 23, Baltimore, MD, 1963, American Federation of Information Processing Societies, Spartan Books Inc., pp. 329–346.

[204] S. Thienel, *ABACUS - A Branch-and-Cut System*, PhD thesis, Institut für Informatik, Universität zu Köln, 1995.

[205] C. Timpe, *Solving planning and scheduling problems with combined integer and constraint programming*, OR Spectrum **24**, no. 4 (2002), pp. 431–448.

[206] K. Truemper, *Design of Logic-based Intelligent Systems*, John Wiley & Sons, Hoboken, New Jersey, 2004.

[207] *VALSE-XT: Eine integrierte Lösung für die SoC-Verifikation*, 2005. http://www.edacentrum.de/ekompass/projektflyer/pf-valse-xt.pdf.

[208] T. J. van Roy and L. A. Wolsey, *Valid inequalities for mixed 0-1 programs*, Discrete Applied Mathematics **14**, no. 2 (1986), pp. 199–213.

[209] M. Wedler, D. Stoffel, and W. Kunz, *Arithemetik reasoning in DPLL-based SAT solving*, in Proc. Conference on Design, Automation and Test in Europe (DATE-04), Paris, France, February 2004.

[210] M. Weiser, *Program slices, formal, psychological and practical investigations of an automatic program abstraction method*, PhD thesis, University of Michigan, Ann Arbor, MI, 1979.

[211] M. Weiser, *Program slicing*, IEEE Transactions on Software Engineering **SE-10**, no. 4 (1984), pp. 352–357.

[212] R. Weismantel, *On the 0/1 knapsack polytope*, Mathematical Programming **77** (1997), pp. 49–68.

[213] R. Williams, C. Gomes, and B. Selman, *Backdoors to typical case complexity*, in Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence, IJCAI 2003, 2003.

[214] R. Williams, C. Gomes, and B. Selman, *On the connections between backdoors, restarts, and heavytailedness in combinatorial search*, in Sixth International Conference on Theory and Applications of Satisfiability Testing, SAT 2003, Informal Proceedings, 2003, pp. 222–230.

[215] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles, *Dynamic storage allocation: A survey and critical review*, in Proceedings of the International Workshop on Memory Management, Kinross Scotland (UK), 1995.

[216] L. A. Wolsey, *Faces for a linear inequality in 0-1 variables*, Mathematical Programming **8** (1975), pp. 165–178.

[217] L. A. Wolsey, *Valid inequalities for 0-1 knapsacks and MIPs with generalized upper bound constraints*, Discrete Applied Mathematics **29** (1990), pp. 251–261.

[218] K. Wolter, *Implementation of cutting plane separators for mixed integer programs*, master's thesis, Technische Universität Berlin, 2006.

[219] R. WUNDERLING, *Paralleler und objektorientierter Simplex-Algorithmus*, PhD thesis, Technische Universität Berlin, 1996. http://www.zib.de/Publications/abstracts/TR-96-09/.

[220] R. ZABIH AND D. A. MCALLESTER, *A rearrangement search strategy for determining propositional satisfiability*, in Proceedings of the National Conference on Artificial Intelligence, 1988, pp. 155–160.

[221] Z. ZENG, M. CIESIELSKI, AND B. ROUZEYRE, *Functional test generation using constraint logic programming*, in Proceedings of IFIP International Conference on Very Large Scale Integration (IFIP VLSI-SOC 2001), Montpellier, France, 2001.

[222] Z. ZENG, P. KALLA, AND M. CIESIELSKI, *LPSAT: A unified approach to RTL satisfiability*, in Proceedings of Conference on Design, Automation and Test in Europe (DATE-01), Munich, Germany, March 2001.

[223] H. ZHANG, *SATO: An efficient propositional prover*, in Proceedings of the International Conference on Automated Deduction, July 1997, pp. 272–275.

[224] L. ZHANG, C. F. MADIGAN, M. W. MOSKEWICZ, AND S. MALIK, *Efficient conflict driven learning in boolean satisfiability solver*, in ICCAD, 2001, pp. 279–285.