Linear Programming in MILP Solving

A Computational Perspective

vorgelegt von Diplom-Mathematiker

Matthias Miltenberger

ORCID: 0000-0002-0784-0964

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

Doktor der Naturwissenschaften
Dr. rer. nat.

genehmigte Dissertation

Promotionsausschuss:

Vorsitzender:	Prof. Dr. John M. Sullivan
1. Gutachter:	Prof. Dr. Thorsten Koch
2. Gutachter:	Dr. Julian Hall

Tag der wissenschaftlichen Aussprache: 31.03.2023

Berlin, 2023

Für Marlene

Acknowledgments

This thesis would not have been possible without the help of many excellent companions along the way.

First and foremost, I would like to thank the developers who have worked on the SCIP Optimization Suite alongside me: Timo Berthold, Leon Eifler, Gerald Gamrath, Ambros Gleixner, Leona Gottwald, Stefan Heinz, Gregor Hendel, Kati Jarck, Marco Lübbecke, Stephen Maher, Benjamin Müller, Marc Pfetsch, Daniel Rehfeldt, Franziska Schlösser, Felipe Serrano, Yuji Shinano, Stefan Vigerske, Dieter Weniger, Michael Winkler, and Jakob Witzig. You all have created a unique and stimulating atmosphere and productive working environment at the Zuse Institute Berlin (ZIB).

It has been a great honor to meet all of you and I am grateful to consider many of you close friends who I am hoping to keep connected with for many years to come. I am very pleased with the opportunity to keep working on your side as a guest at ZIB and to remain a part of this exciting community.

Most notably, I want to thank Gerald, my longstanding office mate: You have been accompanying me at ZIB since day one and I could not have wished for another person to have taken your place.

Ambros, you have been a source of inspiration and a person to look up to with your disciplined approach to research. Thank you for always finding the time to answer my questions and providing new perspectives.

Felipe, thank you for the time and energy you put into the intensive development and long-term maintenance of PySCIPOpt and the many interesting challenges we had to overcome.

All this would not have been possible without Thorsten Koch. Thank you for leading and growing our department over the years and especially for supervising this thesis. The landscape of discrete optimization—not just in Germany—would have been very different without you.

Thank you, Julian Hall, for interesting simplex discussions, for agreeing to be on my committee, and for adopting my naming suggestion concerning HiGHS—I am sure this project has a great future under your guidance and I am honored to have helped with some of the early steps and design choices.

I also want to thank Volker Mehrmann for pointing me towards ZIB after supervising

my diploma thesis. You anticipated that I was going to thrive in the fascinating field of operations research and mathematical programming.

I am grateful to Ted Ralphs and Dan Steffy for countless fruitful discussions both in-person during your visits to Berlin as well as in video calls during the pandemic and before. I left every meeting with new striking ideas and more insight—not only regarding numerical topics.

I want to say thanks to SAP for providing a challenging research project on supply network planning that has kept us on our toes for many years. Similarly, I am grateful to the MODAL Research Campus[1] for providing the financial resources to support our work.

I also want to show my appreciation to Sonja Mars, my manager at Gurobi Optimization, for your continued trust and the final push to finish this dissertation.

Most of all, I want to thank my loving wife Ina for always being at my side and providing a different perspective outside of the constraints of mathematics and computer science.

Matthias Miltenberger, Berlin 2023

[1] Research Campus Modal (`https://forschungscampus-modal.de/`) funded by the German Federal Ministry of Education and Research (fund number `05M14ZAM`).

Zusammenfassung

Die gemischt-ganzzahlige Programmierung (MILP) ist ein wichtiges Teilgebiet im Bereich der mathematischen Optimierung und kommt bei der Modellierung und Lösung einer Vielzahl von verschiedenen Anwendungsproblemen mit oft weitreichenden wirtschaftlichen Folgen zum Einsatz. Die am weitesten verbreiteten Löser für solche MILP-Probleme verwenden den LP-basierten *branch-and-cut*-Ansatz. Hierbei wird die Ganzzahligkeitsbedingung relaxiert und das sich daraus ergebende lineare Programm (LP) dazu verwendet, die Lösungsqualität abzuschätzen und den Lösungsraum schrittweise aufzuteilen und zu verkleinern.

In dieser Arbeit vergleichen und analysieren wir verschiedene algorithmische Techniken und Software-Implementierungen, die zur Optimierung dieser LP-Relaxierungen herangezogen werden können. Wir benutzen dazu die SCIP Optimization Suite[2], die es aufgrund ihrer modularen Struktur erlaubt, solche Vergleiche zwischen freien Lösern wie Clp oder SoPlex und kommerziellen Hochleistungscodes wie CPLEX, Gurobi oder Xpress durchzuführen.

Wir untersuchen speziell, wie die von uns entwickelten Methoden *LP solution polishing* und *persistent scaling* im LP-Löser SoPlex das Verhalten von SCIP bei der Lösung von MILPs beeinflussen. Das erstere Verfahren dient dabei der Verminderung der Fraktionalität der errechneten LP-Lösungen durch Ausnutzen von multiplen optimalen Lösungen, während das zweite zu einer Verbesserung der numerischen Eigenschaften beiträgt, indem die initialen Skalierungsfaktoren über den gesamten Lösungsprozess weiterverwendet werden. Beide Verfahren erhöhen signifikant die Leistungsfähigkeit von SCIP und sind standardmäßig aktiviert.

Außerdem beinhaltet diese Abhandlung eine Übersicht über sämtliche Eigenschaften und mathematischen Techniken, die im LP-Löser SoPlex implementiert sind und diesen von anderen vergleichbaren Implementierungen des Simplex-Algorithmus abheben. Obwohl SoPlex im direkten Vergleich weniger performant ist, zeugen diese Erweiterungen vom wissenschaftlichen Fortschritt auf dem Gebiet der linearen Programmierung.

Weiterhin präsentieren wir Ergebnisse von Studien zur numerischen Stabilität von

[2] https://scipopt.org/

SCIP während der unterschiedlichen Phasen des MILP-Lösens. Hierbei beleuchten wir die Stabilität aus der Perspektive des LP-Lösers und stellen mit dem von uns entwickelten Python Paket TreeD[3] eine neue Möglichkeit vor, wie der Suchbaum interaktiv und animiert im dreidimensionalen Raum dargestellt werden kann. Diese Visualisierungstechnik eignet sich zur anschaulichen Darstellung des MILP-Löseprozesses von SCIP und kann somit zu einem besseren Verständnis dessen beitragen.

Darüber hinaus zeigen wir die schnelle und intuitive Erarbeitung algorithmischer Prototypen auf, die mit der von uns entwickelten SCIP Optimization Suite-Erweiterung PySCIPOpt[4] möglich sind. Hiermit kann mittels der anwendungsfreundlichen Programmiersprache Python auf viele interne Datenstrukturen von SCIP zugegriffen werden, um in kurzer Zeit ohne C/C++-Kenntnisse neue Ideen zu implementieren, wie wir am Beispiel von TreeD zeigen. Auch die intuitive Modellierung ganzer Optimierungsprobleme ist möglich, ohne die zugrundeliegenden Daten in eine weitere Modellierungsumgebung zu überführen.

Sämtliche Entwicklungen und Ergebnisse sind entweder bereits in die quelloffene und für nicht-kommerzielle Nutzung verfügbare SCIP Optimization Suite eingeflossen oder als separate Pakete über die Code-Plattform GitHub[5] verfügbar und frei verwendbar.

[3] https://github.com/mattmilten/TreeD
[4] https://scipopt.github.io/PySCIPOpt
[5] https://github.com/

Abstract

Mixed-integer linear programming (MILP) plays a crucial role in the field of mathematical optimization and is especially relevant for practical applications due to the broad range of problems that can be modeled in that fashion. The vast majority of MILP solvers employ the LP-based branch-and-cut approach. As the name suggests, the linear programming (LP) subproblems that need to be solved therein influence their behavior and performance significantly.

This thesis explores the impact of various LP solvers as well as LP solving techniques on the constraint integer programming framework SCIP Optimization Suite[6]. SCIP allows for comparisons between academic and open-source LP solvers like Clp and SoPlex, as well as commercially developed, high-end codes like CPLEX, Gurobi, and Xpress.

We investigate how the overall performance and stability of an MILP solver can be improved by new algorithmic enhancements like *LP solution polishing* and *persistent scaling* that we have implemented in the LP solver SoPlex. The former decreases the fractionality of LP solutions by selecting another vertex on the optimal hyperplane of the LP relaxation, exploiting degeneracy. The latter provides better numerical properties for the LP solver throughout the MILP solving process by preserving and extending the initial scaling factors, effectively also improving the overall performance of SCIP. Both enhancement techniques are activated by default in the SCIP Optimization Suite.

Additionally, we provide an analysis of numerical conditions in SCIP through the lens of the LP solver by comparing different measures and how these evolve during the different stages of the solving process.

A side effect of our work on this topic was the development of TreeD[7]: a new and convenient way of presenting the search tree interactively and animated in the three-dimensional space. This visualization technique facilitates a better understanding of the MILP solving process of SCIP.

Furthermore, this thesis presents the various algorithmic techniques like the row representation and iterative refinement that are implemented in SoPlex and that distinguish the solver from other simplex-based codes. Although it is often not as per-

[6] https://scipopt.org/
[7] https://github.com/mattmilten/TreeD

formant as its competitors, SoPlex demonstrates the ongoing research efforts in the field of linear programming with the simplex method.

Aside from that, we demonstrate the rapid prototyping of algorithmic ideas and modeling approaches via PySCIPOpt[8], the Python interface to the SCIP Optimization Suite. This tool allows for convenient access to SCIP's internal data structures from the user-friendly Python programming language to implement custom algorithms and extensions without any prior knowledge of SCIP's programming language C. TreeD is one such example, demonstrating the use of several Python libraries on top of SCIP. PySCIPOpt also provides an intuitive modeling layer to formulate problems directly in the code without having to utilize another modeling language or framework.

All contributions presented in this thesis are readily accessible in source code in SCIP Optimization Suite or as separate projects on the public code-sharing platform GitHub[9].

[8] https://scipopt.github.io/PySCIPOpt
[9] https://github.com/

Contents

Notation

$\mathbb{Z}$	Set of integers
$\mathbb{N}$	Set of non-negative integers
$\mathbb{R}$	Set of real numbers
n	Number of variables in the problem
m	Number of constraints in the problem
$\mathcal{C} = \{1, \ldots, n\}$	Set of problem variable indices
$\mathcal{R} = \{1, \ldots, m\}$	Set of slack variable indices
$\mathcal{B}$	Set of basic indices
$\mathcal{N} = (\mathcal{R} \cup \mathcal{C}) \setminus \mathcal{B}$	Set of non-basic indices
$\mathcal{I} \subseteq \mathcal{C}$	Set of integer variable indices
$A \in \mathbb{R}^{m,n}$	Constraint matrix of the problem
$A^\top$	Transpose of matrix A, $A^\top \in \mathbb{R}^{n,m}$
a_q	qth column of A, $a_q = A_{\cdot q}$
B	Basic part of matrix A, $B = A_{\mathcal{B}}$
N	Non-basic part of matrix A, $N = A_{\mathcal{N}}$
LP	Linear programming problem (also refers to the problem class)
MILP	Mixed-integer linear programming problem (also refers to the problem class)
κ	Condition number of a square matrix, $\kappa(A) = \|A\| \cdot \left\|A^{-1}\right\|$
κ_{LP}	LP condition number (see Definition 6 in Chapter 6)

Chapter 1

Introduction

In all my scientific presentations that dealt with the interaction between and the influence and importance of LP solving in the context of mixed-integer linear programming (MILP), I learnt that there is a great interest in understanding this relationship or at the very least in learning something new about it. Everyone from first-year PhD students to experts with decades of experience showed interest in how SCIP behaves with various different LP solvers and what takeaways we can get from that. This topic has been fueled by numerous benchmarks comparing different LP and MILP solvers over many years conducted by Prof. Hans Mittelmann as in Figure 1.1:

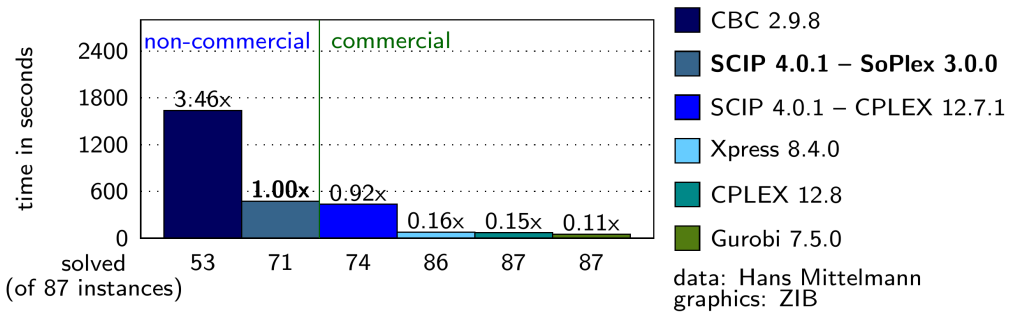


Figure 1.1.: Mittelmann benchmark results on MIPLIB 2010, November 2017.

This provided plenty of motivation to put in the time and effort of investigating the impact of an LP solver in SCIP.

Prof. Thorsten Koch posed the provocative question *"Why LP does not matter for MILP"* and motivated us to dive deep into investigating why a performance increase of an LP code often does not seem to affect the respective MILP performance accordingly. This is by no means a SCIP-specific question: The LP performance improvements of new versions of commercial solvers may not carry over to the respective MILP benchmarks. It takes more than just plugging a fast LP solver into SCIP to get a decent MILP solver.

Nevertheless, implementing new and improved LP techniques remains important and can significantly impact how an MILP is solved. In Chapter 3 we present several

such examples, ranging from more straightforward ideas like a stable sum implementation to improve the accuracy of long summations to more involved concepts like the bound flipping ratio test or the row representation for the simplex algorithm.

A technique that is mostly relevant for solving LP relaxations in the MILP context but still entirely implemented within the domain of the simplex solver SoPlex is LP solution polishing (Chapter 5).

Numerics play an important role both in LP and MILP solving and may decide whether an instance can be solved correctly or not. In Chapter 6 we explore the numerical features during MILP solving from the perspective of the underlying LP solver. This is guided mainly by inspecting condition numbers of matrices that occur during the solving process. One goal of this thesis is to collect and share this information and to investigate some of the *folklore* around LP and MILP numerics.

In summary, this work puts a focal lens on linear programming in MILP solving—quite literally, even, as we demonstrate with our LP-based visualization package TreeD in Section 4.8. This interactive visualization technique projects the LP relaxations encountered while traversing the branch-and-cut tree into a 3-dimensional space.

1.1. Contributions and Publications

Our work is mainly based on the MINLP solver framework SCIP Optimization Suite, most notably including the LP solver SoPlex and the constraint integer programming solver SCIP. While writing this thesis, we developed PySCIPOpt[1] (Maher, Miltenberger, et al., 2016), providing an interface to SCIP from the Python programming language. This tool allows for rapid prototyping of new algorithmic ideas as well as analysis and manipulation of internal solver data using Python. We present details about the core concepts of the implementation and performance comparisons with SCIP's C API in Chapter 2.

Using PySCIPOpt, we developed an application named TreeD[2] (Miltenberger, 2021b), a novel 3D-visualization tool for branch-and-cut trees to provide a new perspective and further insight on how an individual instance is solved by SCIP. TreeD also allows for convenient data collection and is used to conduct many of the numerical experiments presented in this thesis. See Section 4.8 for a description of the approach and further details.

LP performance improvements like exploitation of sparse data structures and the *bound flipping ratio test* discussed in Section 3.5 and Section 3.4 have already been presented in Gamrath, Gleixner, et al. (2019) in the context of solving challenging real-world problem instances (both LP and MILP) originating from supply chain optimization models.

Cao, Gleixner, and Miltenberger (2016) discuss the benefits and disadvantages of different solvers and solver techniques for dealing with difficult LP models in the energy and electricity markets. Such benchmarks remain of high interest in the optimization

[1] https://github.com/scipopt/PySCIPOpt
[2] https://github.com/mattmilten/TreeD

community. Prof. Hans Mittelmann[3] dedicates a lot of time, effort, and computing resources to provide up-to-date results online[4] for a large variety of solvers covering a wide field of disciplines beyond LP and MILP. We created a web service[5] that takes these raw performance numbers and individual log files and presents them in an interactive and more intuitive way to make them more accessible to a wider range of interested users. This project is presented in Appendix A.

The results of Chapter 6 are partly published in Miltenberger, Ralphs, and Steffy (2018). We investigate numerical features during the MILP solving process and what we can learn from this.

Furthermore, many smaller features and performance and stability improvements have been implemented in both SCIP and SoPlex during the development of this thesis and have been published in the different SCIP Optimization Suite reports for version 3.2 (Gamrath, Fischer, et al., 2016), version 4.0 (Maher, Fischer, et al., 2017), version 5.0 (Gleixner, Eifler, et al., 2017), version 6.0 (Gleixner, Bastubbe, et al., 2018), and version 7.0 (Gamrath, Anderson, et al., 2020). We discuss these in Chapter 3.

Two other technical contributions are the transition to the modern version control system `git`[6] from the aging CVS and the introduction of the cross-platform build system `CMake`[7] that facilitates the creation of user-friendly installer packages and has further improved the usability and distribution of the SCIP Optimization Suite. We want to mention this here, because the accessibility of community-driven and open source software development is an often overlooked aspect in academia.

Finally, we want to mention the development of a new LP interface between SCIP and SoPlex that replaces the legacy one and facilitates several new features and algorithmic developments such as persistent scaling (Section 3.7) and LP solution polishing (Chapter 5).

1.2. Linear and Mixed-Integer Linear Programming

In 2012, a feature story of the NewScientist magazine[8] called the Simplex method *"The algorithm that runs the world"*. This perfectly visualizes the impact on real-world applications Linear and Mixed-Integer Linear Programming and especially the simplex method have.

The simplex algorithm is among the top 10 algorithms of the 20th century (Cipra, 2000) and it's very likely to stay among the most used algorithms for the foreseeable future.

In mathematical notation a mixed-integer linear programming problem (MILP) can be formalized in the following way:

[3] School of Mathematical and Statistical Sciences, Arizona State University
[4] `http://plato.asu.edu/bench.html`
[5] `https://github.com/mattmilten/mittelmann-plots`
[6] `https://git-scm.com/`
[7] `https://cmake.org/`
[8] `http://www.newscientist.com/article/mg21528771.100-the-algorithm-that-runs-the-world`

$$\begin{aligned} \min \quad & c^\top x \\ \text{s.t.} \quad & Ax \leq b \\ & l \leq x \leq u \\ & x_j \in \mathbb{Z} \qquad \forall\, j \in \mathcal{I}, \end{aligned} \tag{1.1}$$

If the set $\mathcal{I}$ is empty, that is, if there are only continuous variables, problem (1.1) reduces to a linear programming problem (LP):

$$\begin{aligned} \min \quad & c^\top x \\ \text{s.t.} \quad & Ax \leq b \\ & l \leq x \leq u \end{aligned} \tag{1.2}$$

There are also several other problem classes that can be distinguished: Binary optimization problems only allow variables to be either 0 or 1, while mixed-binary problems may also contain continuous variables. A problem that only consists of integer variables is called an integer programming problem or IP. For most aspects of this thesis, though, the general distinction between LP and MILP is sufficient. It should be noted that these abbreviations are synonymous for both the problem class and the specific problem instance.

With the growing popularity and applicability of mixed-integer non-linear programming, or MINLP, people are more frequently using the term MIP or mixed-integer optimization (MIO) also to encompass non-linear models. We will explicitly mention it whenever we are referring to non-linear problems and will otherwise restrict ourselves to linear optimization, that is, MILP problems.

1.2.1. History and Impact

At the turn of the 21st century, Bixby, Fenelon, et al. (2000) reported how solving techniques for MILPs have evolved over the last decades. Solvers have become reliable and performant enough to deal with all kinds of general problems in the field of operations research and combinatorial optimization. The paper provides computational results using the CPLEX solver and gives an overview of the implementational techniques—many of which still being state-of-the-art today.

About a decade later, Achterberg and Wunderling (2013) analyzed the progress since then and we recommend this paper for readers interested in the evolution of algorithmic and mathematical ideas developed for high-performance MILP solvers.

Cao, Gleixner, and Miltenberger (2016) showed how we can utilize existing methods better via parameter tuning to achieve improved solving times on selected model instances from the energy and power sector.

Pushing performance to solve harder and larger models both for general black-box and specialized applications has always been the driving force of LP and MILP development. It is worth noting that the most advanced codes available today are actually able to handle general purpose models and the range of model types is ever

increasing. Still, there are also specialized implementations, for example Concorde[9] to tackle the Traveling Salesman Problem (TSP) better than any other solver. We refer to the excellent book by Cook (2011) for further information about this particular subclass of mathematical optimization problems.

Most recently, new exciting advancements using quantum computing (Nannicini, 2021) and machine learning (Nair et al., 2020) open new doors for further progress.

The area of mathematical optimization is fast-moving and interesting because it is useful for practical applications, can be challenging and demanding from a computational perspective, and still provides lots of open academic research questions.

1.3. LP Solving Approaches and Duality Theory

The early history of linear programming starts with Fourier (1827), who introduces a technique later called the *Fourier-Motzkin elimination* after its rediscovery by Motzkin (1936). While this algorithm was designed to solve the feasibility problem for a set of inequalities, it can also integrate an objective function that is to be minimized or maximized.

In 1947, George Dantzig invented the simplex algorithm (see Dantzig (1987) for a retrospective view). This method is now called *primal simplex* and has since been refined and improved multiple times. One of the most notable advancements are arguably the *dual simplex method* by Beale (1954) and Lemke (1954) and the first ideas to avoid explicit matrix inversion (Dantzig and Orchard-Hays, 1954) in 1954.

From a theoretical perspective the simplex method is not an efficient method as it has exponential run time, that is, the number of necessary iterations with respect to the input size is not bounded by a polynomial term. Typically, such algorithms are frowned upon because they are deemed impractical for solving real-world problems. Khachiyan (1979) introduces the *ellipsoid method* which stands as the first method to solve linear programming problems in polynomial time. Despite this theoretical advantage, it could not outperform the simplex method on practical problems. This changed when Karmarkar (1984) proposed the *interior point method*—a polynomial-time algorithm that improved the theoretical run time of the ellipsoid method and could also solve real-world LPs in a competitive time. Soon after, the *primal-dual interior point method* by Kojima, Mizuno, and Yoshise (1989) became the basis for state-of-the-art interior point implementations and is often faster than simplex-type methods.

Bixby (2012) provides a comprehensive coverage of the history of linear and integer optimization including corresponding implementations and is highly recommended for further reading.

Since the simplex method and the interior point algorithm are the two fundamental procedures to solve real-world linear optimization problems, we want to explain their ideas and highlight their differences. We will only provide a short introduction to

[9] https://www.math.uwaterloo.ca/tsp/concorde/

these methods and would like to refer to Schrijver (1986) and Vanderbei (1996) for an in-depth description and details about the underlying mathematical ideas.

The notion of *duality* is too important and too substantial to omit, so we will first introduce this concept before describing the solving methods.

1.3.1. Duality

We want to illustrate duality using the famous *diet problem* (see Dantzig (1990) for an interesting story about the beginnings of the simplex method). Recall the standard LP from above:

$$\begin{aligned} \min \quad & c^\top x \\ \text{s.t.} \quad & Ax \geq b \\ & x \geq 0 \end{aligned} \tag{1.3}$$

Now, let the x variables symbolize amounts of various foods (measured in some suitable units), while the constraints model the nutrient requirements of a specific diet plan:

$$\begin{aligned} x &= \left\{x_{\text{apple}}, x_{\text{potato}}, x_{\text{cereal}}, \dots\right\} \\ b &= \left\{b_{\text{protein}}, b_{\text{fat}}, b_{\text{sugar}}, \dots\right\} \end{aligned}$$

Each constraint's coefficients represent the amount of a specific nutrient like carbohydrates, fat, sugar, etc., in one unit of that food item:

$$0.3x_{\text{apple}} + 1.9x_{\text{potato}} + 4.5x_{\text{cereal}} + \dots \geq b_{\text{protein}}$$

Finally, the objective is to minimize the cost of the diet, hence, the c values specify the price of a single unit of each food item. Such diet problems have actually been among the first linear programming models to be investigated. Their practicality was likely very limited due to the lack of variety and they are mostly interesting from a theoretical point of view.

We can form the corresponding *dual LP* as follows:

$$\begin{aligned} \max \quad & b^\top y \\ \text{s.t.} \quad & A^\top y \leq c \\ & y \geq 0 \end{aligned} \tag{1.4}$$

In this dual version, the variables y are used to model prices for nutrient supplements:

$$y = \left\{y_{\text{protein}}, y_{\text{fat}}, y_{\text{sugar}}, \dots\right\}$$

These supplements must be cheaper than the actual food items, so the right hand side of the constraints is c and the constraint matrix is transposed to model the nutrient distribution of a single food item in each row:

primal: $\min c^\top x$	dual: $\max b^\top y$
ith constraint $A_{i.}x \leq b_i$	ith variable $y_i \leq 0$
ith constraint $A_{i.}x \geq b_i$	ith variable $y_i \geq 0$
ith constraint $A_{i.}x = b_i$	ith variable y_i free
jth variable $x_j \geq 0$	jth constraint $y^\top A_{.j} \leq c_j$
jth variable $x_j \leq 0$	jth constraint $y^\top A_{.j} \geq c_j$
jth variable x_j free	jth constraint $y^\top A_{.j} = c_j$

Table 1.1.: LP dualization formulas

$$0.3y_{\text{protein}} + 0.2y_{\text{fat}} + 10.4y_{\text{sugar}} + \cdots \leq c_{\text{apple}}$$

The objective is now to maximize the profit of selling these supplements according to the nutrient demand of the chosen diet.

In this example, the optimum represents a cost equilibrium between consuming regular food and relying only on food supplements. The dual variables y are also referred to as *shadow prices*.

With this illustrative example, we hope to make the concept of duality more tangible. Table 1.1 states how to transform an LP into its dual form considering different variations of constraint senses and variable bounds.

There are several important aspects of this duality that come in handy to work with LPs.

Lemma 1 (Weak duality). *If $\check{x}$ is a feasible solution to the LP $\max_{x \geq 0} \{c^\top x \mid Ax \leq b\}$ and $\check{y}$ is a feasible solution to the dual LP $\min_{y \geq 0} \{b^\top y \mid A^\top y \geq c\}$, then*

$$c^\top \check{x} \leq b^\top \check{y}.$$

Proof. This lemma follows directly from the feasibility conditions for $\check{x}$ and $\check{y}$:

$$\begin{aligned} A\check{x} \leq b \;&\Rightarrow\; \check{y}^\top A\check{x} \leq \check{y}^\top b \\ A^\top \check{y} \geq c \;\Leftrightarrow\; \check{y}^\top A \geq c^\top \;&\Rightarrow\; \check{y}^\top A\check{x} \geq c^\top \check{x} \end{aligned} \tag{1.5}$$

The two inequalities are preserved after multiplication with $\check{x}$ and $\check{y}$, respectively, because of the nonnegativity conditions $\check{x} \geq 0$ and $\check{y} \geq 0$, resulting in the desired statement $c^\top \check{x} \leq \check{y}^\top A\check{x} \leq b^\top \check{y}$. □

Please note that we interchanged the terms *primal* and *dual* for the diet example above because the *primary* formulation is usually referred to as *primal problem*. Most textbooks use the definition from Lemma 1. Both variants are equivalent since the dualizing the dual LP again yields the original primal problem.

An immediate implication from weak duality is that as soon as equality is attained, that is, $c^\top \check{x} = b^\top \check{y}$, both solutions $\check{x}$ and $\check{y}$ are optimal for their respective problems—any further improvement according to their objective functions would violate the weak duality condition.

The fundamental theorem of linear programming is called the *duality theorem*:

Theorem 1 (Duality theorem)**.**

1. *If both primal and dual LPs have feasible solutions, then they also have an optimal solution and the objective values of both optimal solutions coincide.*
2. *If one of the problems is unbounded, then the other has no feasible solution.*
3. *If one of the problems has no feasible solution, then the other problem is either unbounded or infeasible.*

Here is a small example to demonstrate the existence of an LP that is infeasible in its primal form as well as in its dual:

$$\begin{aligned} \max \quad & 2x_1 - x_2 \\ \text{s.t.} \quad & x_1 - x_2 \le 1 \\ & -x_1 + x_2 \le -2 \\ & x \ge 0 \end{aligned} \qquad \begin{aligned} \min \quad & y_1 - 2y_2 \\ \text{s.t.} \quad & y_1 - y_2 \ge 2 \\ & -y_1 + y_2 \ge -1 \\ & y \ge 0 \end{aligned}$$

The infeasibility can be seen by adding up both constraints: The primal system on the left then reduces to $0 \le -1$ while the dual reveals $0 \ge 1$.

For the proof of Theorem 1, we refer to the literature on LP theory, for example Vanderbei (1996) or Schrijver (1986).

Another important result from duality theory is *Farkas' lemma* (Farkas, 1902):

Lemma 2 (Farkas' lemma)**.** *For given $A \in \mathbb{R}^{n,m}$ and $b \in \mathbb{R}^m$ exactly one of the following statements is true:*

1. *There is $x \in \mathbb{R}^n$ with $Ax = b$ and $x \ge 0$.*
2. *There is $y \in \mathbb{R}^m$ with $A^\top y \ge 0$ and $b^\top y < 0$.*

It is fairly easy to see that both statements cannot be true at the same time:

$$A^\top y \ge 0 \overset{x \ge 0}{\Rightarrow} x^\top A^\top y \ge 0 \Rightarrow b^\top y \ge 0 \text{ contradicting } b^\top y < 0.$$

Figure 1.2 provides an intuitive geometrical interpretation of Lemma 2.

Lemma 2 actually provides a proof of infeasibility for LPs in form of the vector y that is also commonly referred to as *dual ray*. This ray is used to derive further implications when solving MILPs and coming across an infeasible node LP. This process is called *conflict analysis* and can help to speed up the solution process, see Witzig, Berthold, and Heinz, 2019 for a recent overview.

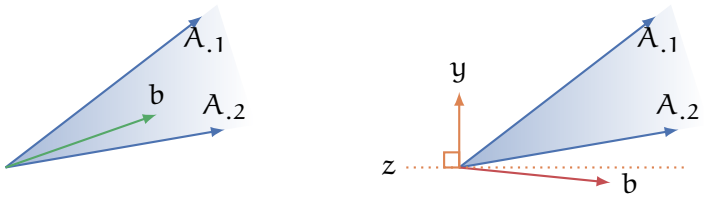


Figure 1.2.: The condition $Ax = b$ translates to $b = \sum_{i=1}^{n} x_i A_{.i}$ for $x \geq 0$, that is, on the left, b is within the cone defined by the columns of A, hence, feasible. In the right illustration, b is outside the feasible region and $b^\top y < 0$ while $y^\top A_{.i} \geq 0$. Recall that $b^\top y = \|b\|\|y\| \cos\alpha$, so for non-zero vectors b and y the angle α between them has to be obtuse to satisfy $b^\top y < 0$. The dotted hyperplane $y^\top z = 0$ separates b from the cone.

1.3.2. Interior Point Method

Unlike simplex-type methods that proceed along the extreme points on the boundary of the feasible region, the *interior point method* or *barrier algorithm* strictly moves within the feasible set and converges towards the optimum.

For completeness and because of its mathematical elegance, we want to sketch the general idea of an interior point method.

Let $\min_{x \geq 0} \{c^\top x \mid Ax = b\}$ be the LP to be solved. The constraints and variable bounds can be combined with the objective function to get the unconstrained *Lagrangian* problem

$$\mathcal{L}_p(x, y) = c^\top x - \mu \sum_{j=1}^{n} \log(x_j) - y^\top (Ax - b)$$

Here, $\mu > 0$, the so-called *barrier parameter* controls the influence of the logarithmic term. The smaller μ gets, the more we allow the x variables to approach zero when minimizing $\mathcal{L}_p(x, y)$. We can apply the same transformation on the dual LP $\max_{s \geq 0} \{b^\top y \mid A^\top y + s = c\}$:

$$\mathcal{L}_d(x, y, s) = b^\top y + \mu \sum_{j=1}^{n} \log(s_j) - x^\top (A^\top y + s - c).$$

Minimizing or maximizing these $\mathcal{L}$ functions requires finding a point where their derivatives reach zero. This is equivalent to these conditions:

$$\begin{aligned} Ax &= b, \quad x \geq 0 \\ A^\top y + s &= c, \quad s \geq 0 \\ x_j s_j &= \mu, \quad \forall j \in \{1, \ldots, n\} \end{aligned} \tag{1.6}$$

These are relaxed Karush-Kuhn-Tucker or KKT conditions (Kuhn and Tucker, 1951), which require μ to be zero in their original form.

The main idea of IPM is now to find a starting point and then successively reduce μ until the iterates are sufficiently close to optimality. This is carried out via solving a sequence of systems of linear equations of this type:

$$\begin{bmatrix} A & 0 & 0 \\ 0 & A^T & I \\ S_k & 0 & X_k \end{bmatrix} \begin{bmatrix} \Delta x_k \\ \Delta y_k \\ \Delta s_k \end{bmatrix} = \begin{bmatrix} r_x \\ r_y \\ r_s \end{bmatrix}$$

Here, X_k and S_k are just diagonal matrices consisting of the values of x_k and s_k respectively, which correspond to the current iterates at step k. The actual right-hand sides depend on which variant of IPM is used—see Mehrotra (1992) for a description of the popular and well-established predictor-corrector method. The most important aspect of these equations is that they can be reduced to this so-called *normal equation*:

$$AS_k^{-1}X_kA^T\Delta y_k = r_x - AS_k^{-1}r_s + AS_k^{-1}X_kr_x \tag{1.7}$$

Since both S_k^{-1} and X_k are diagonal, the constraint matrix of equation 1.7 is a normal matrix, that is, a symmetric positive semi-definite matrix that is suitable to be decomposed using the Cholesky factorization (Mészáros, 2005). The structure of this matrix remains unchanged throughout the solving process with only X_k and S_k varying so an initial symbolic factorization and the sparsity structure can be preserved. This is a major difference to simplex methods as we are going to see later.

Barrier methods are generally more susceptible to numerical issues and have a higher tendency of breaking down and not converging to optimality compared to simplex methods. On the other hand, they can also solve very large LP problems within a small number of iterations—for state-of-the-art solvers typically less than 200—and can make use of multi-threaded processor architectures to further improve the solving times. See Figure 4.3 for a comparison between the barrier and simplex methods when solving MILPs with SCIP. The memory requirement of the barrier method is a lot higher compared to simplex methods, especially when the instance has an unfavorable matrix structure leading to a very dense Cholesky factorization.

Another important aspect of IPM is its ability to also deal with non-linear continuous optimization problems. Convex problems can be solved to optimality while for non-convex problems they can find a local optimum. This feature makes IPMs an integral part of mixed-integer non-linear programming solvers because they can often provide better bounds than a linear relaxation could (Vigerske and Gleixner, 2018).

1.3.3. Simplex Algorithm

The simplex algorithm is built around the concept of a *basis*. This basis identifies one of the vertices of the polyhedron defined by the constraints $A_{i.}$ and the variable bounds of the LP. Since there is always an optimal vertex solution, there also exists an associated optimal basis. It goes without saying that this holds true only if there exists an optimal solution in the first place. Please also note that there may be more

than one optimal basis. See Definition 1 in Chapter 3 for a precise description and more details.

Geometrically speaking, from one iteration to the next, the simplex method modifies the basis in a way that resembles moving from the current vertex to a neighboring one. The direction of movement is chosen to improve the value of the objective function $c^\top x$ until no further progress can be made—we have arrived at the optimum.

There are two main variants of simplex methods: a primal and a dual one.

The primal simplex always preserves primal feasibility on its path to optimality. The dual simplex, on the other hand, moves along the dual feasible vertices that lie outside the primal feasible region. This can be seen as moving on the optimal side of the polyhedron until the boundary of the feasible region is reached.

The dual simplex method is equivalent to applying the primal simplex to the dual version of the original LP. There are some seemingly minor differences, though, that are responsible for the typically observed superiority of the dual simplex. We will highlight these in Chapter 3. See Algorithm 1.1 and Algorithm 1.2 for a side-by-side comparison of the two simplex methods. We want to point out that the two vectors $\hat{a}_q = B^{-1}A_{\cdot q}$ and $\hat{a}_p^\top = (B^{-1}N)_{p\cdot}$ are of different dimensions and refer to a column q and a row p of the simplex tableau matrix $B^{-1}A$. This slight abuse of notation allows for a more intuitive description of the algorithm and also stresses that the value $\hat{a}_{pq} = \hat{a}_{q_p} = \hat{a}_{p_q}^\top$ appears in both vectors.

To start the procedure we always need a primal feasible basis in case of the primal simplex and a dual feasible basis in case of the dual simplex. We highlight one way how to generate such a *starting basis* in Section 3.3.

The simplex algorithm is a myopic method in a way that it always only regards the local neighborhood for finding the next step direction towards optimality. There are different ways to decide where to go next from the current basis and this step of the algorithm is commonly referred to as *pricing*. We explain some of the well-known pricing alternatives in Section 3.5. One can say that this step is the most significant one and provides the greatest amount of freedom. Therefore, it is also the decisive factor in the total number of iterations required to reach optimality.

With its use of the basis the simplex algorithm also inhibits some elegant and convenient discrete features. In Section 3.9 we show how an exact solution can be computed in an iterative, yet efficient way. Koch (2004b) uses the basis as a fundamental tool to proof optimality of the NETLIB LP test set.

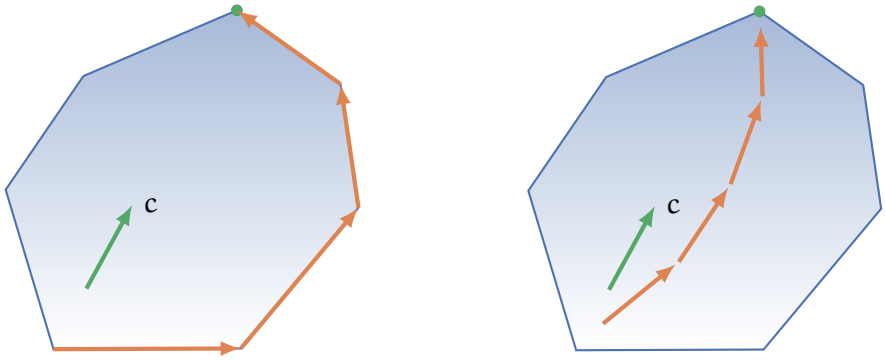


Figure 1.3.: Schematic of the solving processes of simplex and interior point methods. The left picture shows the simplex path along the border of the feasible region while the right image depicts the interior point or barrier method's central path through the interior of the feasible region.

Algorithm 1.1 Primal simplex algorithm

Input: LP (1.2) with $l = 0$, $u = \infty$, primal feasible basis $\mathcal{B}, B = A_{\mathcal{B}}, N = A_{\mathcal{N}}$

$x_{\mathcal{B}} \leftarrow B^{-1}b$
$d_{\mathcal{N}}^{\top} \leftarrow c_{\mathcal{N}}^{\top} - c_{\mathcal{B}}^{\top}B^{-1}N$
while not $d_{\mathcal{N}} \geq 0$ **do**
  $q \leftarrow \arg\min \{d_k \mid k \in \mathcal{N}\}$ (Pricing)
  $\hat{a}_q \leftarrow B^{-1}a_q$ (FTRAN)

  if $\hat{a}_q \leq 0$ **then**
    Return: LP is unbounded
  end if
  $p \leftarrow \arg\min \{x_{\mathcal{B}_k}/\hat{a}_{q_k} \mid \hat{a}_{q_k} > 0\}$
  $z^{\top} = e_p^{\top}B^{-1}$ (BTRAN)
  $\hat{a}_p^{\top} = z^{\top}N$
  $\alpha_P \leftarrow x_{\mathcal{B}_p}/\hat{a}_{pq}$ (primal steplength)
  $x_{\mathcal{B}} \leftarrow x_{\mathcal{B}} - \alpha_P \hat{a}_q$
  $\alpha_D \leftarrow -d_{\mathcal{N}_q}/\hat{a}_{pq}$ (dual steplength)
  $d_{\mathcal{N}}^{\top} \leftarrow d_{\mathcal{N}}^{\top} + \alpha_D \hat{a}_p^{\top}$
  $\mathcal{B}_p \leftarrow q$ (basis update)
end while
Return: optimal solution x and optimal basis $\mathcal{B}$

Algorithm 1.2 Dual simplex algorithm

Input: LP (1.2) with $l = 0$, $u = \infty$, dual feasible basis $\mathcal{B}, B = A_{\mathcal{B}}, N = A_{\mathcal{N}}$

$x_{\mathcal{B}} \leftarrow B^{-1}b$
$d_{\mathcal{N}}^{\top} \leftarrow c_{\mathcal{N}}^{\top} - c_{\mathcal{B}}^{\top}B^{-1}N$
while not $x_{\mathcal{B}} \geq 0$ **do**
  $p \leftarrow \arg\min \{x_{\mathcal{B}_k} \mid k \in \{1, \ldots, m\}\}$
  $z^{\top} = e_p^{\top}B^{-1}$ (BTRAN)
  $\hat{a}_p = z^{\top}N$
  if $\hat{a}_p \leq 0$ **then**
    Return: LP is unbounded
  end if
  $q \leftarrow \arg\min \{d_k/\hat{a}_{p_k} \mid \hat{a}_{p_k} > 0\}$
  $\hat{a}_q \leftarrow B^{-1}a_q$ (FTRAN)

  $\alpha_P \leftarrow x_{\mathcal{B}_p}/\hat{a}_{pq}$ (primal steplength)
  $x_{\mathcal{B}} \leftarrow x_{\mathcal{B}} - \alpha_P \hat{a}_q$
  $\alpha_D \leftarrow -d_{\mathcal{N}_q}/\hat{a}_{pq}$ (dual steplength)
  $d_{\mathcal{N}}^{\top} \leftarrow d_{\mathcal{N}}^{\top} + \alpha_D \hat{a}_p^{\top}$
  $\mathcal{B}_p \leftarrow q$ (basis update)
end while
Return: optimal solution x and optimal basis $\mathcal{B}$

1.3.4. SoPlex

SoPlex is an implementation of the revised primal and dual simplex algorithms and combines several defining features not present in other solvers. It is a very mature code base that has been in use in the academic world (Koch, 2004b) as well as in industrial projects for more than two decades. Thorsten Koch prepared the initial public version of the code and many other researchers mostly from the Zuse Institute Berlin (ZIB) have been passing the torch to keep the solver updated and maintained over the years until the present day.

The name of the code is an abbreviation for *sequential object-oriented simplex* hinting at the object-oriented programming paradigm of C++ that became very popular during the 1990s. The two major contributions in its first publication by Wunderling (1996), were shared and distributed memory parallelization schemes (called SMoPlex and DoPlex, respectively) as well as the use of either a column or a row representation of the simplex method. While the former is unfortunately lost in time, leaving SoPlex to be a sequential code, the twofold basis representation is still actively maintained and put to good use to gain performance improvements (see Maher, Fischer, et al., 2017). This feature is described in more detail in Section 3.2. Another important aspect that distinguished SoPlex from most other LP solvers is its exact solving capabilities using rational arithmetic and iterative refinement as laid out by Gleixner (2015). We will present two other features that are especially useful within the branch-and-bound approach to solve MILPs, namely LP solution polishing (see Chapter 5) and the preservation of scaling factors throughout the solving process as described in Section 3.7.

SoPlex is discussed in detail in Chapter 3.

1.3.5. Choosing the Method

We will further discuss which LP solving method—simplex or interior point—to use in Chapter 4 for solving MILPs. Choosing the fastest method even for solving just an isolated LP is a difficult question and it is still unclear how to answer it analytically. From a computational perspective, whenever enough cores are available on the computing hardware, the pragmatic way of solving or rather avoiding this question is to simply run all the different methods in parallel. This is typically conducted in a concurrent fashion, that is, the method that finishes first, concludes the optimization and is deemed the winner.

Unfortunately, this strategy will always be slower than selecting the fastest method to run independently. This is mostly due to side effects on cache and memory bandwidth and simply more work to be done by the processor and the operating system. So, ideally, we would like to avoid this overhead and be able to figure out the fastest method depending on certain problem data characteristics. There are a number of heuristic methods that try to make an educated guess but are not always successful. The most commonly used deciding metric is the sheer size of the model instance, because with growing dimensions, the interior point method is increasingly likely to

beat the simplex method. Recalling the way the two approaches work, this also appears apparent when considering the growing number of potential vertices the simplex method may have to traverse. The interior point method on the other hand is less affected by this increasingly complex geometry.

Still, to avoid choosing the slower method, either the concurrent approach or manual tuning for similar models is utilized in practice. When a basic solution is needed, the choice becomes even harder: Then, the interior point method's solution requires the application of a simplex-like procedure called *crossover* (Megiddo, 1991), effectively employing both methods. Such a basic solution also has the added benefit of a typically smaller support, that is, fewer variables have a non-zero value in the solution. This makes it also attractive for use cases that do not utilize the basis information.

Again, this demonstrates that despite its age, there are still unsolved problems and open questions in the research area of linear programming.

1.4. MILP Solving Approaches

Since MILP solving is $\mathcal{NP}$-hard there are naturally several more or less efficient and suitable solving approaches. In case the optimal solution or a proof of optimality are not necessarily required, one may choose a heuristic-based method to produce a feasible and hopefully acceptable solution as quickly as possible. We do not further discuss such *inexact* methods because they are fundamentally different from the *exact* approaches that can provide a proof of optimality together with the solution. If the problem class is known beforehand, for example traveling salesman problems (Applegate et al., 2006) or max-flow problems (Dantzig and Fulkerson, 2003), certain specialized techniques from the field of combinatorial optimization can be employed with very good results.

For many application areas, though, it is required to provide a proven optimal solution or at the very least a reliable measure for how much a feasible solution might still be improved upon. Furthermore, most applications do not revolve around the solution of a pure combinatorial problem but are modeled as general mixed-integer linear optimization problems. This is also the scenario we are regarding in this thesis.

The most popular and probably most investigated solving approach is the LP-based branch-and-bound method. It is extended by various cutting plane techniques, primal heuristics, presolving reductions and a variety of other methods—commonly referred to as *"bag of tricks"*. This culminates into a powerful solver that is able to tackle many problems from both academic as well as industrial background efficiently, despite the daunting complexity. Several such solvers are currently developed both by commercial vendors (for example CPLEX[10], Gurobi[11], Xpress[12]) and with an academic

[10] IBM. *ILOG CPLEX: High-performance software for mathematical programming and optimization*. https://www.ibm.com/products/ilog-cplex-optimization-studio. 2021.

[11] Gurobi Optimization, LLC. *Gurobi Optimizer Reference Manual*. http://www.gurobi.com. 2021.

[12] FICO. *FICO Xpress Optimization Suite*. http://www.fico.com/en/products/fico-xpress-optimization-suite. 2021.

background (like CBC[13] or SCIP[14]). The focus of our work is on SCIP and especially the SCIP Optimization Suite, with its MILP and MINLP solver SCIP and the LP solver SoPlex.

The branch-and-bound technique was first introduced by Land and Doig (1960). Similar to the concept of *divide and conquer* a problem is solved recursively by repeated divisions into subproblems. The high complexity of MILP solving stems from the fact that these subproblems are again MIPs—and often not significantly easier to solve—and that the tree that is generated in the process can grow exponentially large.

LP solutions provide a so-called *dual bound* that limits the possible primal bound, that is the value of an integer feasible solution, so entire subtrees can be safely excluded from the search. This *pruning* helps to keep the tree at a manageable size and also hints at the importance of good branching decisions, that is which variables define new bounds in the child nodes. There is an abundance of research items that are concerned with the design of branching rules: See Achterberg, Koch, and Martin, 2004; Achterberg and Berthold, 2009; Berthold and Salvagnin, 2013; Gamrath, Berthold, and Salvagnin, 2020 for details, especially regarding the branching rules of SCIP. The two main reasons for making MILP problems tractable in practice even for larger instances are powerful heuristics to find incumbent solutions and the clever exploitation of linear relaxations throughout the solving process. We refer to Berthold (2014) for a comprehensive analysis of primal heuristics.

Please note that the terms *primal* and *dual* appear both in the context of LP and MILP but they are not the same. For instance, there is no straightforward notion of a *dual of an MILP*. Instead, we typically only refer to primal and dual *solutions* with the latter one still being rooted in the primal space of variables ignoring the integrality restrictions. One can imagine the dual solution as the solution to the problem in an ideal or fantasy world that does not need to adhere to integrality rules and allows, for example, to send $1/3$ of a truck to half a warehouse. Consequently, interpreting the quality of the dual bound and the gap to the incumbent solution is strongly dependent on the chosen formulation to model the real-world problem. Often, there are multiple alternatives with various advantages and disadvantages and it is the task of an OR practitioner or model expert to choose a suitable one; this work is sometimes referred to as the *"Art of Modeling"*.

In addition to that, the term duality also comes into play when computing presolving reduction to produce a more compact formulation. Here, *dual reductions* are based on the objective as opposed to primal reductions that are based on the feasible domain. For more information on duality of MILPs, we refer to Wolsey (1981).

Whenever we are speaking of a *tree*, we usually refer to the search tree that is generated by the branch-and-bound technique. In a similar manner, *root*—if not specified otherwise—refers to the first node, that is, the root of said tree and includes all further processing up until the first branching occurs.

To give this notion a more tangible representation we demonstrate what this tree

[13] J. J. Forrest, S. Vigerske, H. G. Santos, et al. *coin-or/Cbc: Version 2.10.5*. 2020. DOI: 10.5281/zenodo.3700700.

[14] *SCIP: Solving Constraint Integer Programs*. http://scip.zib.de. 2021.

Algorithm 1.3 Concept of LP-based Branch-and-Bound for MILP

Input: $A, b, c, l, u, \mathcal{I}$ as defined in MILP (1.1)

$L_{\text{open}} \leftarrow \{\text{MILP (1.1)}\}$ // initialize set of open (sub-) problems
$\hat{z} \leftarrow \infty$ // initialize primal bound
while $L_{\text{open}} \neq \emptyset$ **do**
  select $L \in L_{\text{open}}$ // node selection
  $L_{\text{LP}} \leftarrow$ linear relaxation of L
  if L_{LP} infeasible **then**
    $L_{\text{open}} \leftarrow L_{\text{open}} \setminus \{L\}$ // prune infeasible node
  else
    $x^\star \leftarrow$ optimal solution of L_{LP}
    if $c^\top x^\star \geq \hat{z}$ **then**
      $L_{\text{open}} \leftarrow L_{\text{open}} \setminus \{L\}$ // prune non-improving subtree
    else
      $x_{\text{frac}} \leftarrow \{x_j^\star \notin \mathbb{Z}, j \in \mathcal{I}\}$ // determine fractional variables
      if $x_{\text{frac}} = \emptyset$ **then**
        $\hat{x} \leftarrow x^\star$ // found new incumbent solution
        $\hat{z} \leftarrow c^\top x^\star$ // store new primal bound
      else
        select $x_k \in x_{\text{frac}}$ // choose branching candidate
        $L_\triangle = \{L \mid x_k \geq \lceil x_k^\star \rceil\}$ // new lower bound for x_k in subproblem $L_\triangle$
        $L_\triangledown = \{L \mid x_k \leq \lfloor x_k^\star \rfloor\}$ // new upper bound for x_k in subproblem $L_\triangledown$
        $L_{\text{open}} \leftarrow L_{\text{open}} \setminus \{L\}$ // remove L from open problems
        $L_{\text{open}} \leftarrow L_{\text{open}} \cup \{L_\triangle, L_\triangledown\}$ // add new child nodes to open problems
      end if
    end if
  end if
end while
if $\hat{z} < \infty$ **then**
  Return: optimal solution $\hat{x}$ with solution value $\hat{z}$
else
  Return: MILP (1.1) is infeasible
end if

looks like in various stages of the solving process in Figure 1.4. We can see how cutting planes, bound propagation, and conflict graphs can already improve the quality of the root relaxation by pushing the dual bound upwards. We explain the details of the used visualization tool TreeD (Miltenberger, 2021b) in Section 4.8.

1.4.1. SCIP Optimization Suite

The SCIP Optimization Suite is a one-of-a-kind software project that has been used by several generations of bachelor, master, and PhD students for their research. It is also used by commercial software vendors to enable fast and reliable linear and integer optimization applications. The SCIP Optimization Suite in version 7 comprises several partly independent software packages and tools:

SCIP The constraint integer solving framework is the heart of the SCIP Optimization Suite and ties all other packages together. SCIP provides code to handle a large range of problem classes from LP over MILP to (mixed-integer) non-linear optimization. It is written in C and started as PhD project of Achterberg (2009). SCIP has a modular plug-in structure that allows for straightforward code extensions.

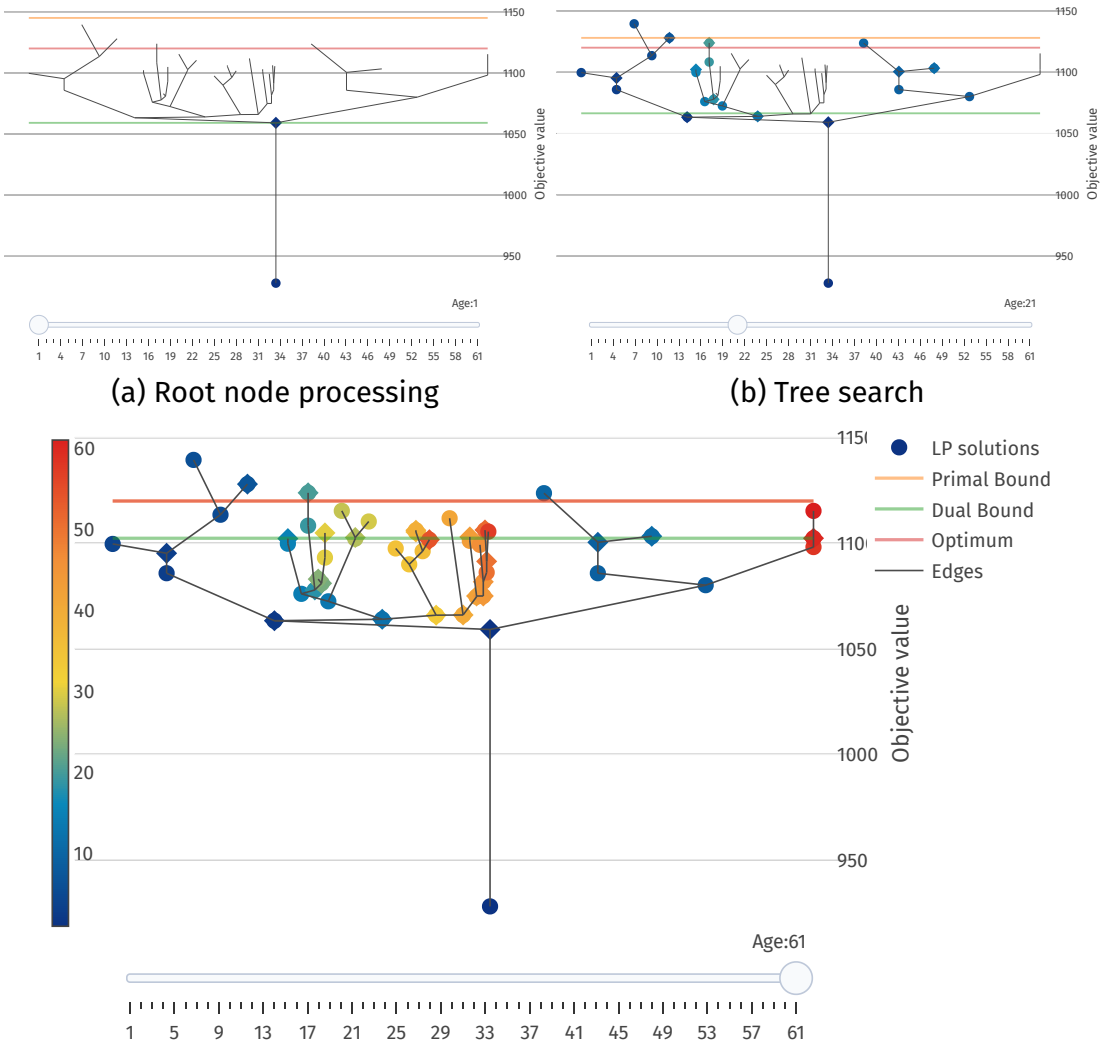
SoPlex The C++ implementation of the simplex method is the oldest component of the project and provides LP solving capabilities to SCIP. We describe SoPlex in more detail in Chapter 3 as it plays a major role in this thesis.

ZIMPL The Zuse Institute Mathematical Programming Language was developed by Koch (2004a) to provide an open-source modeling tool to efficiently formulate various kinds of optimization problems in an intuitive syntax.

GCG The Generic Column Generation framework extends SCIP to provide support for solving extremely large instances that do not require to handle all variables upfront but generates them during the solving process. This *branch-and-price* project was started by Gamrath and Lübbecke (2010).

UG The Ubiquity Generator is another extension that facilitates running SCIP (and other software that generate some kind of tree) on massively parallel hardware. Shinano et al. (2012) describe the approach that has been successfully used to find optimal solutions to several problem instances for the very first time.

There are several aspects that make SCIP unique: Firstly, and often taken for granted, is the high-quality code with lots of documentation and several extensive release reports that meticulously describe new features and improvements. The SCIP Optimization Suite supports a wide range of APIs and is continuously tested for stability and correctness. There is an active community all around the world engaging with the software and its developers and also contributing to the project. The SCIP Optimization Suite is actively developed at different research institutes and universities and used for teaching in Operations Research and Discrete Mathematics. Numerous previous developers are now working on commercial solvers like CPLEX, Gurobi, and Xpress



(a) Root node processing

(b) Tree search

(c) Finished optimization

Figure 1.4.: Visualization of the MILP solving process using TREED and SCIP 7.0 of MIPLIB3 instance `lseu`. In Figure 1.4a, the dual bound (green line) is at root node level before the first branching. Figure 1.4b shows the primal and dual bounds moving towards each other while in Figure 1.4c, SCIP already found the optimal solution and no open nodes are left. Note that the optimum is only known at the final stage and has been added to the previous pictures for a better understanding. The same applies for the already present tree paths. The color of the individual nodes shows the age of those nodes, that is, the order in which they are processed.

while still keeping in touch with the SCIP community. It is also rare that an academic software project is developed, maintained, and extended over several decades—and to date, SCIP shows no sign of slowing down in the future.

1.4.2. Other Solvers

In the world of commercial integer optimization, there are the three dominant solvers CPLEX (IBM, 2021), Gurobi (Gurobi Optimization, LLC, 2021), and Xpress (FICO, 2021). All of them have a mature code base with Gurobi being the youngest. Most notably, CPLEX has been very popular with academic research projects and to date is probably the most prevalent also in the commercial sector—although there are barely any reliable numbers about this. These solvers exhibit state-of-the-art performance and stability and have been competing for the title of *"fastest solver"* for many years. Although they provide solving capabilities beyond LP and MILP, we will not discuss this here. Instead, we focus on their performance impact as LP solvers inside SCIP in Chapter 4.

There are also smaller commercial solver vendors like MOSEK (MOSEK ApS, 2016) that specialize in certain problem classes or solving techniques. MOSEK is also included in some of our experiments with SCIP.

In regards to academic or open-source MILP solvers, we want to mention Cbc (Forrest, Vigerske, Santos, et al., 2020), supported by the COIN-OR Foundation[15]. Performance-wise, this solver is lagging behind SCIP and often cannot handle larger or harder models. The LP solver Clp (Forrest, Vigerske, Ralphs, et al., 2020) that is used in Cbc, though, has a quite impressive performance and often beats SoPlex. Other well-known open source solvers are GLPK[16] and lpsolve[17]. Despite their nominal emphasis on linear programming, they can actually deal with integer problems as well. They are still used fairly wide-spread due to their accessibility but they fall behind in terms of performance and, unfortunately, development efforts for most of these codes appear to have slowed down in recent years.

A relatively new solver is HiGHS[18], headed by Dr. Julian Hall. HiGHS can handle both LPs and MILPs and already surpasses Clp and Cbc in terms of performance and stability. Huangfu and Hall (2018) describe the main features of the solver, most notably the parallel simplex implementation. We did not include HiGHS as an LP solver for SCIP in our experiments in Chapter 4 because the corresponding interface is not yet stable and reliable enough. We are confident that HiGHS will play a major role in the academic and open-source linear and integer optimization community in the years to come.

[15] http://www.coin-or.org
[16] https://www.gnu.org/software/glpk/
[17] https://web.mit.edu/lpsolve/doc/
[18] https://www.highs.dev

1.5. Testing Methodology

This work contains several computational experiments to test and verify the different implementations and algorithmic ideas. To facilitate fair and unbiased comparisons we run the experiments on identical hardware and, if necessary, use aggregated results using different random seeds to limit white noise influence and performance variability. Every job is executed exclusively on a single machine to avoid any side-effects that usually occur when running multiple optimizations in parallel. The runs were conducted on a cluster with Intel® Xeon® Gold 5122 processors running with a clock speed of 3.6 GHz and 96 GB of memory.

We use the evaluation tool *IPET* (Hendel, 2021) and various common Python plotting libraries like *matplotlib*, *seaborn*, and *plotly*. Unless mentioned otherwise, we use the MIPLIB 2017 benchmark set (Gleixner, Hendel, et al., 2021) as the foundation for MILP-based experiments. For most LP-based experiments, we use a modified SCIP version that reads all input files as linear programming problems, effectively ignoring any integer restrictions. This allows for LP solver comparisons in a controlled environment with identical error checks and a unified output.

Our LP instance set consists of a culmination of several widely used and popular test sets from the LP and MILP communities:

- Netlib LP test set[19]
- Csaba Mészáros's LP collection[20]
- COR@L[21]
- MIPLIB (Bixby, Boyd, and Indovina, 1992), MIPLIB 3 (Bixby, Ceria, et al., 1998), MIPLIB 2003 (Achterberg, Koch, and Martin, 2006), MIPLIB 2010 (Koch, Achterberg, et al., 2011), and MIPLIB 2017 (Gleixner, Hendel, et al., 2021)

The *shifted geometric mean* has been established as the standard measure for comparing performance across a set of instances. It is defined for a set of data points $t_1, \ldots, t_k$ as

$$\left(\prod_{i=1}^{k} (t_i + s)\right)^{1/k} - s.$$

These data points are typically solving times or node counts of the MILP solver. The corresponding shift value has to be chosen accordingly and serves to dampen the effects of large relative differences in small numbers of t. We will note the shift values for the single experiments whenever it is not 1 second for time measurements or 100 for node counts.

[19] University of Tennessee Knoxville and Oak Ridge National Laboratory. Netlib LP Library. `http://www.netlib.org/lp/`, accessed December 2021.

[20] Csaba Mészáros. LP Test Set. `http://www.sztaki.hu/~meszaros/public_ftp/lptestset/`, accessed December, 2021.

[21] Computational Optimization Research At Lehigh. MIP Instances. `http://coral.ie.lehigh.edu/data-sets/mixed-integer-instances/`, accessed December, 2021.

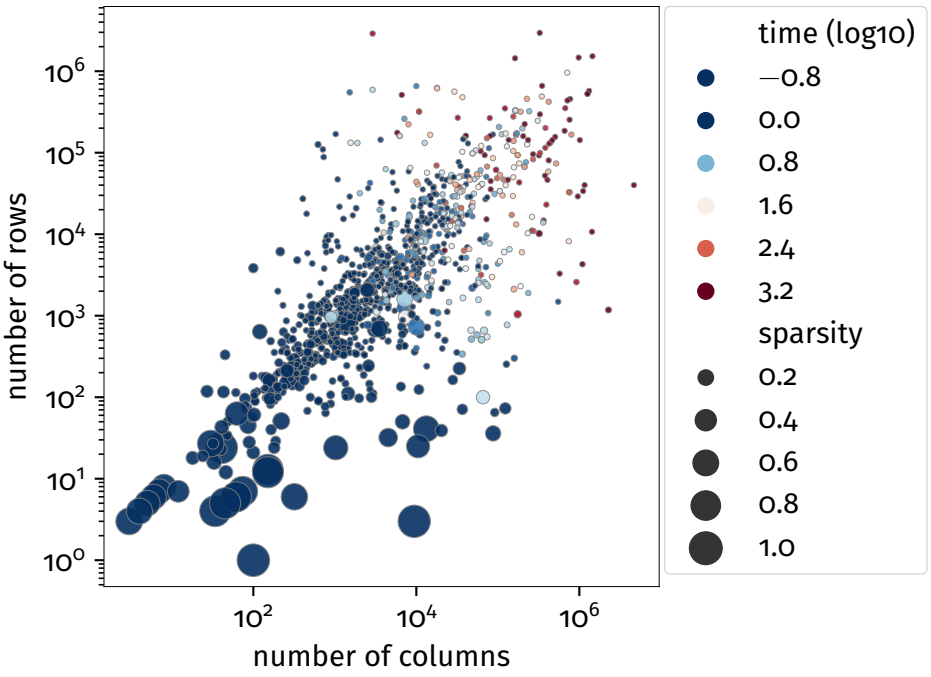


Figure 1.5.: Size distribution of LP test set, including solving time colorization for SoPlex with default settings. Sparsity is computed as non-zeros/ $(n * m)$

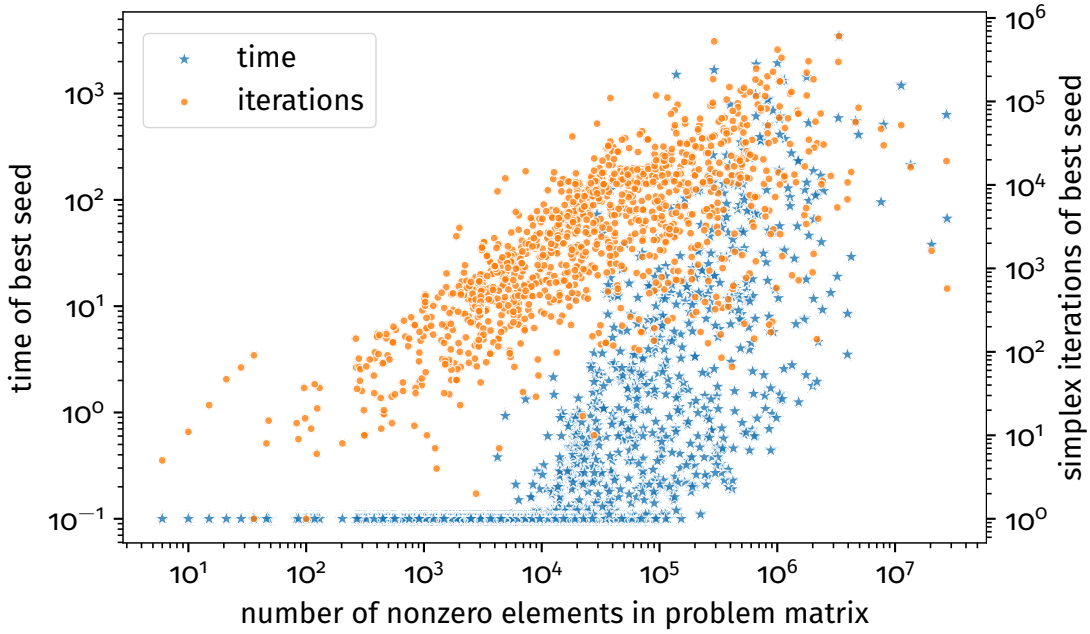


Figure 1.6.: Number of iterations and time to optimality (using a time limit of one hour) with respect to the number of non-zero elements in the problem matrices

Chapter 2

PySCIPOpt

High-performance optimization solvers rely on an implementation that is close to the machine, so all state-of-the-art codes—from commercial to free and open-source—are written in a programming language that is considered *low-level* by today's standards. The most popular programming languages for this area are C/C++.

C/C++ enable the creation of very efficient solvers that can put the available hardware resources to best use. This comes at the cost of a significantly higher development cost, that is, a longer time from idea to prototype to production code than in other higher-level programming languages. Python[1], on the other hand, is one of the languages that tries to minimize this development cost at the price of an often notably worse performance when executing the code.

Bjarne Stroustrup, the author and designer of the C++ language once said: *"There are only two kinds of programming languages: those people always complain about, and those nobody uses"*[2] – concisely making the point that there will always be certain trade-offs in programming.

For prototyping new algorithmic concepts and for less performance-critical features, Python is very well suited. One of its main charms is that Python is an interpreted language and can be executed without a dedicated compilation step: Users can write new code and immediately run it.

Using clever language extensions like Cython[3] (Dalcin et al., 2011), we are able to combine the best features of both programming language aspirations: PySCIPOpt (Maher, Miltenberger, et al., 2016) leverages Python's fast prototyping while relying on SCIP's C engine to avoid sacrificing too much performance. Coincidentally, Cython or more precisely its predecessor Pyrex, is just as old as SCIP, celebrating its 20th anniversary in 2022[4].

[1] https://www.python.org/
[2] Stroustrup, B. (2000). The C++ programming language. Addison-Wesley Professional.
[3] https://cython.org/
[4] https://mail.python.org/pipermail/python-list/2002-April/126661.html

2.1. Concept

We were inspired by the popular and highly regarded design of Gurobi's Python interface when creating the concept for PySCIPOpt. Furthermore, Prof. João Pedro Pedroso[5] encouraged this decision with his intent to publish a text book about mathematical optimization with PySCIPOpt[6] as a translation of the Japanese edition by Kubo et al. (2012) that is built around Gurobi.

The core concept of PySCIPOpt consists of two components—modeling and feature development—that we want to explain in the following paragraphs.

2.1.1. Modeling

When designing a tool for the creation of mathematical optimization models, one important aspect is to have a close connection between the computer code and the mathematical formulation. Ideally, one should be able to comprehend the meaning of the individual sets of variables and constraints just as easy from reading the code as from reading the mathematical formulation. To achieve this goal in PySCIPOpt, we use operator overloading and intuitive object-oriented classes to represent the entities of a model instance, like variables, constraints, and operators. The following example taken from Maher, Miltenberger, et al. (2016) illustrates this:

```
from pyscipopt import Model

m = Model()
x = m.addVar("x")
y = m.addVar("y", vtype="I")
m.setObjective(x + 3*y)
m.addCons(2*x - y*y >= 10)
```

$$\begin{aligned} \text{minimize} \quad & x + 3y \\ \text{subject to} \quad & 2x - y^2 \geq 10 \\ & x, y \geq 0 \\ & x \in \mathbb{R} \\ & y \in \mathbb{Z} \end{aligned}$$

We also see from this example that sensible default values are used to fill in method parameters that are not set explicitly by the user. This allows the code to stay minimal and readable while still providing all the advanced functionality from the underlying SCIP methods. Additionally, Python also allows passing these parameters in any order as long as the name of the parameter is specified. For completeness, this is the full specification of the `addVar()` method:

[5] https://www.dcc.fc.up.pt/~jpp/, Faculdade de Ciências da Universidade do Porto Departamento de Ciência de Computadores
[6] https://scipbook.readthedocs.io/

```
def addVar(
    self,              # the PySCIPOpt model
    name="",           # variable name
    vtype="C",         # variable type ("C", "B", or "I")
    lb=0.0,            # lower bound
    ub=None,           # upper bound
    obj=0.0,           # objective coefficient
    pricedVar=False)   # whether this is a pricing candidate
```

This paradigm of staying close to SCIP while offering more convenience is consistent throughout the package to appeal to newcomers and SCIP experts alike.

2.1.2. Feature Development

Employing PySCIPOpt purely as a framework to formulate and solve a model instance programmatically and then querying the resulting solution values is a common use case. However, many users would like to extend the available feature set in SCIP with their own algorithmic ideas. To make this possible, we support the majority of SCIP's comprehensive API to allow fine-grained control of how the solver handles a certain model instance.

SCIP's modular structure provides callback functions for almost every component of the branch-and-cut-and-price solving process. These callbacks can be implemented in pure Python code and are then called through PySCIPOpt during runtime of the solver. There is no compilation step necessary to make changes in the Python code, which facilitates rapid prototyping.

The most popular callbacks include custom cutting plane separation techniques, primal heuristics, and also constraint handlers that provide everything necessary to define entirely new user-specific constraint types.

2.2. Technical Details

The methods implemented in PySCIPOpt can loosely be categorized into three groups:

convenience functions These are functions that have no direct counterpart in SCIP but make feature development and model building more intuitive and straightforward. A good example is the `addCons()` method. It encapsulates multiple SCIP functions to add linear, quadratic, and general nonlinear constraints in a single function call. Convenience functions hide some of SCIP's complexity and make PySCIPOpt more appealing and user-friendly.

SCIP wrappers These are relatively straightforward wrappers of SCIP functions that often do not consist of more than two or three lines of code and just replicate

repetitions	number of variables n	PySCIPOpt	SCIP
5	1000	0.09 sec	0.03 sec
5	10 000	0.65 sec	0.09 sec
5	100 000	6.34 sec	0.72 sec
5	1 000 000	64.41 sec	8.48 sec

Table 2.1.: Performance comparison of building model 2.1 in PySCIPOpt using Python and in SCIP using C

a certain SCIP functionality. Wrapper functions account for the majority of functions implemented in PySCIPOpt and can be used whenever there are no complex data structures involved in the specific call, for example when changing the bounds of a variable.

internal helper functions These are needed to organize the code better, to provide improved structure, and to outsource some frequently needed functionality that should be hidden from the user.

The individual plug-in types are implemented in an object-oriented fashion. Every new user plug-in needs to be extended from the given base class, that provides the function stubs. Some of these functions are mandatory for a working implementation and some are optional.

To give an example: the heuristics plug-in needs to implement the `heurexec()` function to tell SCIP what this custom heuristic is supposed to do. It is not necessary, though, to also implement `heurinit()` if no additional initialization step is required to run the heuristic.

We refer the interested reader to the many different examples and test cases available in PySCIPOpt that demonstrate typical use cases and should provide a good starting point for further code development.

2.3. Performance

We compare the performance of building a simple LP model like this:

$$\begin{aligned} \min \quad & \mathbf{1}^\top x \\ \text{s.t.} \quad & x_i + x_{i+1} \geq 1 \quad \forall i \in \{1, \ldots, n\} \\ & x \geq 0. \end{aligned} \tag{2.1}$$

This model is built five times for different values of n to get a reliable performance measure. As we can see from Table 2.1, building a model from formulation 2.1 takes about eight times longer with PySCIPOpt than using SCIP's C API directly.

While this undoubtedly is a significant disadvantage for PySCIPOpt, the amount of code required to formulate this model in the two programming languages also differs dramatically:

PySCIPOpt:

```
def buildmodel(nvars):
    m = Model("bench")
    x = {}

    for i in range(nvars):
        x[i] = m.addVar(name=f"x_{i}", obj=1)

    for i in range(nvars - 1):
        m.addCons(x[i] + x[i + 1] >= 1, name=f"c_{i}")
```

C API of SCIP:

```
static
SCIP_RETCODE buildmodel(int nvars)
{
  char name[SCIP_MAXSTRLEN];
  SCIP* scip;
  SCIP_VAR** x;
  SCIP_CONS* c;
  int i;

  SCIP_CALL(SCIPcreate(&scip));
  SCIP_CALL(SCIPincludeDefaultPlugins(scip));
  SCIP_CALL(SCIPcreateProbBasic(scip, "bench"));

  SCIP_CALL(SCIPallocMemoryArray(scip, &x, nvars));

  for(i = 0; i < nvars; ++i)
  {
    (void) SCIPsnprintf(name, SCIP_MAXSTRLEN, "x_%d", i);
    SCIP_CALL(SCIPcreateVarBasic(scip, &x[i], name, 0,
      SCIPinfinity(scip), 1.0, SCIP_VARTYPE_CONTINUOUS));
    SCIP_CALL(SCIPaddVar(scip, x[i]));
  }

  for(i = 0; i < nvars-1; ++i)
  {
    (void) SCIPsnprintf(name, SCIP_MAXSTRLEN, "c_%d", i, i);
    SCIP_CALL(SCIPcreateConsBasicLinear(scip, &c, name, 0,
      NULL, NULL, 1, SCIPinfinity(scip)));
    SCIP_CALL(SCIPaddCoefLinear(scip, c, x[i], 1.0));
```

```
    SCIP_CALL(SCIPaddCoefLinear(scip, c, x[i+1], 1.0));
    SCIP_CALL(SCIPaddCons(scip, c));
    SCIP_CALL(SCIPreleaseCons(scip, &c));
  }

  for(i = 0; i < nvars; ++i)
    SCIP_CALL(SCIPreleaseVar(scip, &x[i]));

  SCIPfreeMemoryArray(scip, &x);
  SCIP_CALL(SCIPfree(&scip));
  return SCIP_OKAY;
}
```

After the model is created and the `optimize` call is executed, both approaches will perform the actual optimization in the same time because PySCIPOpt is then just calling SCIP's solving routines without any further detours.

PySCIPOpt is not the only software package with the aim to make mathematical programming more accessible but, to date, it is arguably the best solution to work with the deeper data structures and algorithmic components of SCIP without requiring C/C++ proficiency. For plain modeling or basic callback interfaces from Python, we would also like to mention the frameworks Pyomo (Hart, Watson, and Woodruff, 2011; Bynum et al., 2021) and PuLP[7]. These tools enable the user to run different commercial and academic or open-source LP, MILP, or general solvers with the same Python code to formulate the models. Because of this flexibility, they lack the functionality and deeper integration of PySCIPOpt and other custom-built or proprietary tools.

JuMP (Dunning, Huchette, and Lubin, 2017) is a software package for the programming language Julia (Bezanson et al., 2017) and has potential to rival PySCIPOpt and other Python-based tools in its usability and performance. Until now, Julia is still a niche programming language that does not yet have a large community and ecosystem of libraries compared to other languages.

2.4. Licensing and Impact

PySCIPOpt is licensed under the permissive free software MIT license making it easy for contributors to extend the feature set or provide bug fixes or other code improvements. There are automatic builds and a large test suite configured to verify every single commit that is pushed to the public GitHub repository. This *Continuous Integration* approach allows for better maintainability and consistent code quality. There is a growing number of researchers that prefer using PySCIPOpt for their work to avoid technical complications with C/C++. PySCIPOpt is also increasingly popular with academic lectures and courses teaching applied mathematical programming. A famous

[7] https://coin-or.github.io/pulp

example is the `ecole`[8] framework for combining machine learning with combinatorial optimization.

It is safe to say that PySCIPOpt has become the main entry point for users of the SCIP Optimization Suite with more than 500 stars on GitHub, 200 repository forks, and multiple dependent projects and packages as of October 2022.

[8] `https://www.ecole.ai/`

Chapter 3

Implementational Aspects of the Simplex Algorithm

The original publication of SoPlex by Wunderling (1996) is written in German and therefore inaccessible for large parts of the world of operations research. We want to remedy this by providing a description of the algorithm and explaining its advantages and unique features compared to other implementations.

SoPlex uses the revised primal and dual simplex algorithm to solve linear optimization problems. It is implemented in C++ and a core component of the SCIP Optimization Suite. SoPlex embraces object-oriented programming leading to a modular structure of the different algorithmic components, like presolving, pricing, or scaling. A variety of presolving techniques help to reduce the problem size and speed up the solving process.

The name SoPlex is an abbreviation for *sequential object-oriented simplex*, with its two parallel variants DoPlex and SMoPlex, referring to *distributed* and *shared memory* implementations respectively. While those parallel versions did show performance improvements in the original work of Wunderling (1996), they have never been made publicly available.

Until early 2018, SoPlex was used in the SPEC benchmark suite for CPU intensive codes *SPEC CPU2006*[1].

3.1. Stable Summation

There are ways to achieve summation results with double precision arithmetic that have smaller errors than a straightforward implementation (see Rump (2005) for further reference). This can be especially useful for summations of many values and for summations that are prone to numerical cancellation. Elimination happens when subtracting values of almost equal size. In this case most of the significant digits of both numbers vanish, leaving mainly numerical noise as the result.

[1] http://www.spec.org

We identified several places in both SCIP and SoPlex that might result in inaccurate summation results and applied a more sophisticated approach to compute the sum. For SoPlex this is done in all scalar products of any type of vector as well as in the different summations necessary for solving linear systems of equations using the LU factorization of the basis matrix.

The method splits the summation values into two double precision numbers and performs basis arithmetic on those pairs of high and low nominal values. While such methods can replace all basic arithmetic operations (sum, difference, product, and quotient), they are most important for summation due to its inherent numerical instability. There is a negligible performance impact compared to the usual computation because the additional internal additions can be performed efficiently, that is, in parallel, on modern CPUs.

It is difficult to assess the benefit of such a feature because in most cases those tiny numerical inaccuracies do not impede the computation of a correct solution within the usual tolerances. Still, we do see a positive effect of the stable summation when analyzing SCIP's performance solving MILPs as shown in Table 4.2. Note that although SCIP itself also implements the stable sum technique in various places, we only consider the performance impact of this feature within SoPlex.

3.2. Row Representation

The feature that makes SoPlex stand out from most other simplex implementations is its two-fold basis representation. While typical solvers—as well as the majority of textbooks—define the basis to be a subset of columns, one can also use a subset of rows. This can be seen as dualizing the perspective. There are several advantages of working with a so-called *row basis*, like easier addition and removal of rows, or a reduced basis size for certain problems.

Before we go into detail regarding the consequences, let us first formally define the row basis using this extended version of a linear programming problem:

$$\begin{aligned} \min \quad & c^\top x \\ \text{s.t.} \quad & \begin{bmatrix} L \\ l \end{bmatrix} \leq \begin{bmatrix} Ax \\ x \end{bmatrix} \leq \begin{bmatrix} U \\ u \end{bmatrix} \\ A \in \mathbb{R}^{m,n}, \quad & c, l, u \in \mathbb{R}^n, \quad L, U \in \mathbb{R}^m \end{aligned} \tag{3.1}$$

Effectively, we remove the variable bounds and extend the constraint matrix from A to $\begin{bmatrix} A^\top & I \end{bmatrix}^\top$.

The dual of (3.1) reads

$$\begin{aligned} \max \quad & l^\top d^+ & -u^\top d^- & & +L^\top y^+ & & -U^\top y^- & \\ \text{s.t.} \quad & d^+ & -d^- & & +A^\top y^+ & & -A^\top y^- & = c \\ & d^+, & d^-, & & y^+, & & y^- & \geq 0. \end{aligned} \tag{3.2}$$

The value $d_j = d_j^+ - d_j^-$ is commonly called the *reduced cost* of variable x_j. Note that these values are uniquely determined by the dual solution $y = y^+ - y^-$.

Usually, bounds can also take values ∞ or $-\infty$, for ease of notation we are assuming them to be finite.

Definition 1 (basis, basic solution)**.** *Given a linear programming problem in form* (3.1)*. Let $\mathcal{C} \subseteq \{1,\dots,n\}$ and $\mathcal{R} \subseteq \{1,\dots,m\}$ be index sets of variables and constraints of* (3.1)*, respectively.*

1. *We call $\mathcal{B} := (\mathcal{C},\mathcal{R})$ a* basis *of* (3.1) *if $|\mathcal{C}| + |\mathcal{R}| = m$. Variables and constraints with index in $\bar{\mathcal{C}} := \{1,\dots,n\} \setminus \mathcal{C}$ and $\bar{\mathcal{R}} := \{1,\dots,m\} \setminus \mathcal{R}$, respectively, are called* non-basic*.*
2. *We call a basis $(\mathcal{C},\mathcal{R})$* regular *if the vectors $A_{\cdot j}$, $j \in \mathcal{C}$, and e_i, $i \in \mathcal{R}$, are linearly independent.*
3. *We call a primal-dual pair $(x,y) \in \mathbb{R}^n \times \mathbb{R}^m$ a* basic solution *of* (3.1) *if there exists a regular basis $(\mathcal{C},\mathcal{R})$ such that*

$$\begin{aligned} x_j = l_j \text{ or } x_j = u_j, &\quad j \notin \mathcal{C}, \\ A_{i\cdot}x = L_i \text{ or } A_{i\cdot}x = U_i, &\quad i \notin \mathcal{R}, \\ y^\top A_{\cdot j} = c_j^\top, &\quad j \in \mathcal{C}, \\ y_i = 0, &\quad i \in \mathcal{R}. \end{aligned} \tag{3.3}$$

4. *A solution (x,y) is called* primal feasible *if $L \le Ax \le U$, $l \le x \le u$. It is called* dual feasible *if*

$$y_i = 0 \vee (y_i \ge 0 \wedge A_{i\cdot}x = L_i) \vee (y_i \le 0 \wedge A_{i\cdot}x = U_i) \quad \forall i \in \{1,\dots,m\} \tag{3.4}$$

and with $d_j = c_j - y^\top A_{\cdot j}$

$$d_j = 0 \vee (d_j \ge 0 \wedge x_j = l_j) \vee (d_j \le 0 \wedge x_j = u_j) \quad \forall j \in \{1,\dots,n\} \tag{3.5}$$

holds.

$\mathcal{B}$ is the well known definition of a basis and in our work is referred to as the *column basis*.

Let us now define the *row basis*.

Definition 2 (row basis)**.** *The* row basis $\mathcal{N} = \left(\bar{\mathcal{C}},\bar{\mathcal{R}}\right)$ *is an n-dimensional subset of the (extended) rows of this LP.*

As we can see, the general concept of partitioning variable and constraint indices into the four sets $\mathcal{C}$,$\mathcal{R}$,$\bar{\mathcal{C}}$, and $\bar{\mathcal{R}}$ allows us to define both a column basis $\mathcal{B}$ and a row basis $\mathcal{N}$. In Definition 1 we require the basis matrix B, corresponding to the column basis $\mathcal{B}$, to be regular. This is necessary for properly defining e.g., the solution values $x_{\mathcal{C}}$ that are not on their bounds, as we will see later. It is not immediately apparent that this regularity also holds for the row basis matrix N.

Lemma 3. *Let $\mathcal{B} = (\mathcal{C}, \mathcal{R})$ be a basis of* (3.1) *according to Definition 1. The vectors $A_{\cdot j}, j \in \mathcal{C}$ and $e_i \in \mathbb{R}^m, i \in \mathcal{R}$ are linearly independent if and only if the (row) vectors $A_{i\cdot}, i \in \bar{\mathcal{R}}$ and $e_j^\top \in \mathbb{R}^n, j \in \bar{\mathcal{C}}$ are linearly independent.*

Proof. Partition constraint matrix A according to basis $(\mathcal{C}, \mathcal{R})$:

$$A = \begin{bmatrix} A_{\mathcal{RC}} & A_{\mathcal{R}\bar{\mathcal{C}}} \\ A_{\bar{\mathcal{R}}\mathcal{C}} & A_{\bar{\mathcal{R}}\bar{\mathcal{C}}} \end{bmatrix}$$

This leads to the following partitioning of the basis matrices B and N:

$$B = \begin{bmatrix} A_{\mathcal{RC}} & I_{\mathcal{R}} \\ A_{\bar{\mathcal{R}}\mathcal{C}} & 0 \end{bmatrix} \qquad N = \begin{bmatrix} 0 & I_{\bar{\mathcal{C}}} \\ A_{\bar{\mathcal{R}}\mathcal{C}} & A_{\bar{\mathcal{R}}\bar{\mathcal{C}}} \end{bmatrix}$$

For their determinants the following holds.

$$\det(B) = -\det\left(A_{\bar{\mathcal{R}}\mathcal{C}}\right) \cdot \det(I_{\mathcal{R}}) = -\det\left(A_{\bar{\mathcal{R}}\mathcal{C}}\right) \cdot \det\left(I_{\bar{\mathcal{C}}}\right) = \det(N).$$

Here, we are using the fact that the determinant of a block-triangular matrix is equal to the product of the determinants of the blocks on the diagonal:

$$\det\left(\begin{bmatrix} A & B \\ 0 & C \end{bmatrix}\right) = \det(A) \cdot \det(C)$$

Furthermore, the two block matrices B and N can be transformed into the above block-triangular form by multiplication of a (block-) permutation matrix with determinant -1 that exchanges the two block rows. □

Corollary 1. *Forming the inverse matrices of the basis matrices B and N shows that both rely only on the inversion of sub-matrix $A_{\bar{\mathcal{R}}\mathcal{C}}$:*

$$B = \begin{bmatrix} A_{\mathcal{RC}} & I_{\mathcal{R}} \\ A_{\bar{\mathcal{R}}\mathcal{C}} & 0 \end{bmatrix} \qquad N = \begin{bmatrix} 0 & I_{\bar{\mathcal{C}}} \\ A_{\bar{\mathcal{R}}\mathcal{C}} & A_{\bar{\mathcal{R}}\bar{\mathcal{C}}} \end{bmatrix}$$

$$B^{-1} = \begin{bmatrix} 0 & A_{\bar{\mathcal{R}}\mathcal{C}}^{-1} \\ I_{\mathcal{R}} & -A_{\mathcal{RC}}A_{\bar{\mathcal{R}}\mathcal{C}}^{-1} \end{bmatrix} \qquad N^{-1} = \begin{bmatrix} -A_{\bar{\mathcal{R}}\mathcal{C}}^{-1}A_{\bar{\mathcal{R}}\bar{\mathcal{C}}} & A_{\bar{\mathcal{R}}\mathcal{C}}^{-1} \\ I_{\bar{\mathcal{C}}} & 0 \end{bmatrix}$$

From Corollary 1 we see that it is possible to compute the inverse matrix of one representation using only the inverse of the other representation and some matrix-vector products. We can transform data between representations as is required e.g., when using the row basis internally but need to communicate results in column representation, as is done in tableau-based cut generation.

The potential of this connection has already been explored by Gleixner (2012), where it is demonstrated that the inversion of $A_{\bar{\mathcal{R}}\mathcal{C}}$ is enough to formulate the entire simplex method in both primal and dual form. This technique has been called *kernel simplex* by Wunderling (2012) and is available for the dual simplex algorithm in CPLEX. From

personal communications with the author it seems that the proposed symmetrical simplex algorithm unifying primal and dual methods could not outperform the existing code base in CPLEX. We are not aware of any competitive implementation of this idea.

Note that the term *kernel* has also been used by various authors in the linear programming community, see e.g., Maros, 2003; Luce et al., 2009, to describe the non-trivial block during the LU factorization after performing some permutations.

The row representation seems not very intuitive when one is already familiar with the classical approach. On second thought, though, it can help to understand or visualize the concept of a simplex basis in the first place:

The intersection of all basic rows, be it a proper problem constraint or just a variable bound, defines the current vertex that corresponds to this basis. The connection to active set methods may also become more obvious as the row basis is precisely the active set in the simplex algorithm, as opposed to the set of non-basic variables in the traditional column representation.

Essentially, basic rows are tight, whereas the activity of a non-basic row is determined by the basic ones. This is the direct opposite of the definition of the column basis and explains the above notion of duality between the two representations.

From an algorithmic point of view, the primal simplex in column representation can be seen as an *entering* simplex since the arguably most significant part is selecting an entering index in the pricing step to improve the reduced costs. Equivalently, for the dual simplex in the same representation, we are searching for a *leaving* candidate in the pricing step to become primal feasible. Using a row basis, this notion of entering/leaving and primal/dual is interchanged and Table 3.1 captures this duality in the two representations quite nicely:

	column representation	row representation
entering type	primal simplex	dual simplex
leaving type	dual simplex	primal simplex

Table 3.1.: Overview of the different basis representations and simplex variants.

Let us now discuss conceptional and especially computational advantages of the two representations.

3.2.1. Addition of Cutting Planes

When we add valid inequalities to an existing LP and a corresponding optimal basis, we lose primal feasibility but retain dual feasibility. The following re-optimization with the dual simplex can then be started at the previous basis and usually takes few iterations. As we also modify the dimensions of the problem, the basis size will increase when using the column representation making it necessary to compute a

new LU factorization or adapt the existing one—SoPlex recomputes the factorization in most cases.

In the row representation, though, this is not necessary as the basis size is not linked to the number of constraints in the problem, so the factorization remains valid. Multiple new valid inequalities can be added to the set of LP rows without invalidating the factorization. This provides a computational advantage for problems that rely on a cut-focused solution approach.

See Section 4.5 for more details on the concept of cutting planes for solving MILPs.

3.2.2. Column Generation

In the column generation approach as described for example by Desaulniers, Desrosiers, and Solomon (2006), new variables or columns are generated during the optimization process and re-optimizations based on the previous LP basis are required. Analogous to Section 3.2.1, the existing LU factorization can be reused when working with the column representation but not with the row representation. In the latter, the basis size depends on the number of variables and changes throughout the solving process. Hence, we can expect a computational advantage of using the conventional column representation of the simplex method for this type of approach.

3.2.3. Different Problem Dimensions

Following the arguments in the preceding paragraphs, it should come as no surprise that there are advantages and disadvantages between the two representations just by regarding the dimensions of the problem in question as we can also see in Figure 3.2. Whenever there are more rows than columns, the row representation should be used to be able to work with a smaller basis matrix and profit from less memory consumption and shorter data arrays in the factorization. SoPlex automatically switches to the row representation when there are more than 1.2 times as many rows than columns in the problem. This threshold is not set to 1.0 because of some additional overhead with operations like computing the solution of a linear system with the basis matrix that require a transformation to the standard column representation. Such operations happen frequently when running SoPlex within a MILP solver.

Figure 3.1 visualizes how the choice of representation affects the solving times for the `allLP` test set. We only compare instances that can be solved with both settings and choose the best solving time over all seeds. Every instance has a data point either in the upper or lower part of the image depending on whether the row or the column representation yields better performance.

We can observe a trend of the column representation dominating the results for instances with more columns than rows. On the other hand, the winning instances for the row representation are more evenly distributed across the horizontal axis.

We use the instance dimensions after applying presolve reductions. The color intensity is calculated in the same manner as the row-column ratio and represents how strongly the other representation is dominated:

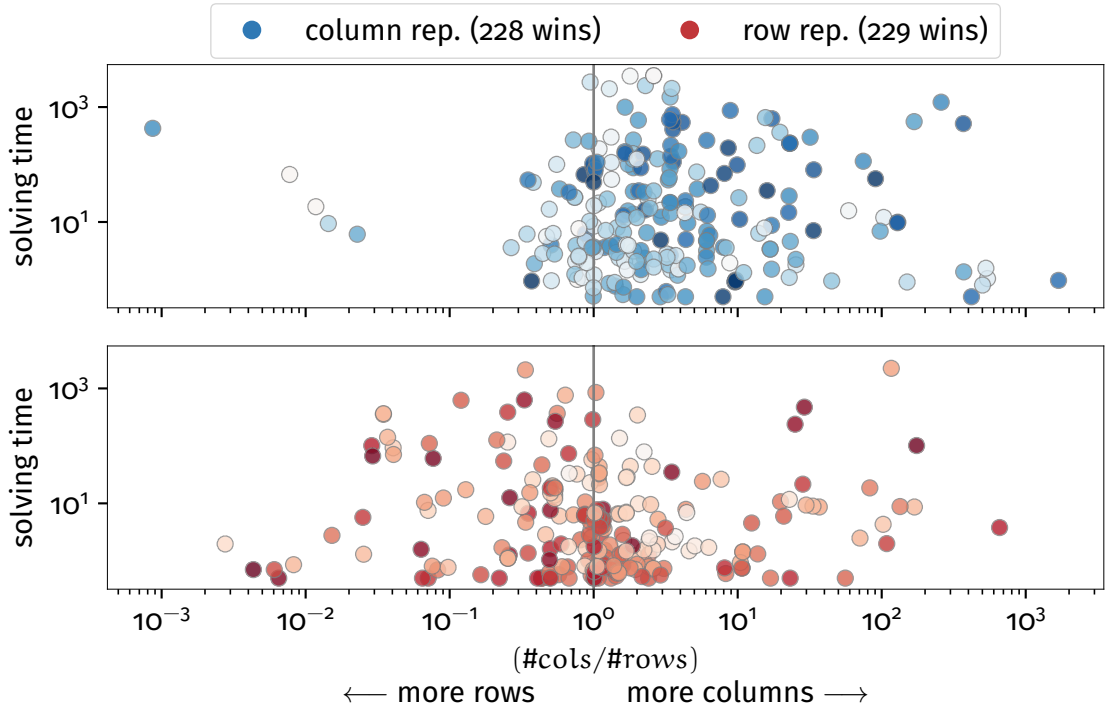


Figure 3.1.: Minimum solving times for row and column representation depending on row to column ratio on the `allLP` test set.

$$\frac{t_{row} - t_{col}}{\max(t_{row}, t_{col})}$$

Of the 489 instances, 229 were won by the row representation and 228 were won by the column representation. For the remaining instances both settings yield similar performances. Please note that we only considered instances where the slower setting took more than one second to solve. An interesting result from this comparison is that many instances are solved faster despite a technically disadvantageous basis representation. The more suited representation usually pays off, though, when dealing with longer-running problem instances.

It is important to note that both basis matrix representations carry the same amount of meaningful information—the remaining entries of the basis matrices are merely filled up with the corresponding parts of an identity matrix. This also means that despite the greater matrix dimensions, the amount of work required to factorize the matrix into LU form is not growing in the same ratio and can be carried out by simple permutations. Luce et al. (2009) showed that for most problems in practice, this is even true in general, that is, also for the smaller of the two variants. The so-called *Markowitz pivoting* technique (Markowitz, 1957) is used in most simplex implementations to perform the LU factorization and is able to handle these identity parts as

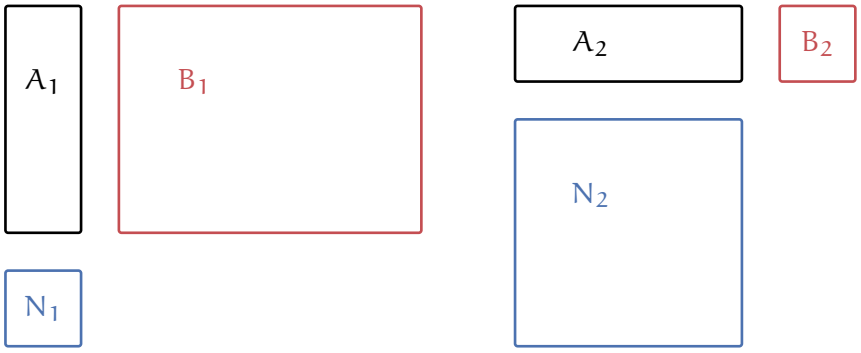


Figure 3.2.: Row (N) and column (B) basis matrix sizes for different problem dimensions A_1 and A_2

well. The main idea here is to construct a permutation that favors elements of maximal sparsity while preserving numerical stability—a row and column singleton with a 1 as only entry is always going to be preferred over any other rows and pushes all rows associated to the identity matrix to the front. We refer to the paper of Luce et al. (2009) for a comprehensive coverage of this part in the simplex algorithm.

3.3. Shifting to Generate a Feasible Basis

The primal and dual simplex algorithms require either a respectively primal or dual feasible starting basis. For a general LP instance it is unlikely that the typical starting basis consisting of all slack variables is feasible right away. In that case we need to perform some kind of transformation or solve an auxiliary problem to satisfy the necessary working conditions of the simplex algorithm. In a very recent paper, Huang et al. (2021) list several different simplex initialization variants and provide an overview as well as an outlook on the possibilities of using machine learning techniques to further improve upon the existing ideas. Their paper mainly discusses methods to create advanced starting bases that go beyond mere feasibility and also aim to achieve better numerical properties and a shorter iteration count to reach optimality. See also Galabova and Hall (2020) for a comprehensive description of one such method. One drawback of these initialization heuristics is that they are more expensive and do not provide a guarantee of leading to fewer iterations. Furthermore, the computation of accurate steepest edge weights is more time consuming compared to using the identity matrix as starting basis. Although SoPlex also implements code to compute an advanced starting basis, in our experience this did not result in improved overall performance.

Commonly, finding a feasible basis involves solving the so-called *phase-1 problem*. A rather straightforward way is the following: every constraint is extended by a new artificial variable rendering the trivial starting basis feasible so the primal simplex algorithm can be applied. Additionally, the objective function is modified to minimize

the value of these new artificial variables:

$$\begin{aligned} \min\ & 1^T z \\ \text{s.t. } & Ax + z = b \\ & x \geq 0 \\ & z \geq 0 \end{aligned} \tag{3.6}$$

We can use a primal feasible starting basis that sets all artificial variables z to be basic and all structural variables x onto their lower bounds. Optimizing this auxiliary LP should end up with a basis that is primal feasible for the original problem if and only if the objective function value is 0, that is, all artificial variables z are 0.

In case where the optimal basis does have a positive objective function value, the original problem instance is infeasible: There is no variable setting for x that satisfies all constraints without the help of adding z variables.

SoPlex on the other hand performs a so-called *shifting* technique to set up a feasible starting basis for either the primal or dual simplex method. In the primal case we require a basis that satisfies all bound and side constraints. Instead of modifying the basis, we simply relax all those violated constraints until they are satisfied. The resulting modified LP will be primal feasible for the trivial slack basis that sets all structural variables to either of their bounds and consists of only row variables.

After the primal simplex has found an optimal solution, that is, an optimal basis is available, we can tighten the previously relaxed constraints again. While this operation is breaking primal feasibility it is not going to impact the dual feasibility of the basis, so the dual simplex method can be started from this basis to eliminate all primal violations.

In a similar fashion we can also relax dual violations and start the optimization process with the dual simplex method and clean up using the primal method.

3.4. Long Steps in the Dual Simplex

This section is about a modification of the ratio test in the dual simplex that allows for a reduction of iterations by essentially performing multiple iterations combined. We implemented this technique in SoPlex version 1.6.0 and it has since become the default ratio test when using the dual simplex.

3.4.1. Mathematical Background

The idea of long steps in the dual simplex method has been known for a long time. It was introduced in a Russian article (Kirillova, Gabasov, and Kostyukova, 1979) and later translated into English by Kostina (2002) making it available to wider audience. Even before, the method is mentioned in an unpublished draft report by Fourer (1994). Maros (2003) and Koberstein (2005) give a detailed description of the procedure and explain how it can be implemented. Still, there are a few aspects of the method also

known as *bound flipping ratio test* that have not yet been mentioned in the literature. We need to sketch the algorithmic idea to be able to formulate them.

The bound flipping ratio test is only applicable in the dual simplex method and for problem instances that have variables or constraints with both upper and lower bounds. Its main idea is that dual feasibility can be maintained by switching or flipping the bound a non-basic variable is set to. This happens during the ratio test, that is, when determining the next non-basic index that needs to become basic to not violate feasibility. From Definition 1 we know that dual feasibility depends on the sign of the dual variable y_i or d_j in conjunction with the upper or lower bound being tight for the respective primal variable x_j or constraint $A_{i\cdot}x$. Hence, when the dual step length exceeds beyond the first break point, the corresponding dual variable changes its sign, rendering it infeasible for the current bound setting. Flipping the bound allows to further enlarge the step length. This can be repeated until no progress in the dual objective function is possible anymore.

3.4.2. Dual Pivot

In the following we want to explain the steps involved in a dual pivot, that is, a basis change that conserves dual feasibility and increases the dual objective function value while trying to reduce primal infeasibility. This example is done by using the column representation and follows the detailed description found in the PhD thesis of Koberstein (2005).

Beginning with a dual feasible basis $\mathcal{B}$ with $p \in \mathcal{B}$ being a violated primal variable, say $x_p > u_p$, that was chosen during the pricing step. This variable will be set to u_p and therefore leaves the basis. The corresponding reduced cost value d_p moves from 0 to some non-positive value. To compensate this change, also the dual variables y need to be modified to preserve dual feasibility.

Summarizing, the dual pivot can be described by the change $t \in \mathbb{R}$ of the pth dual basic constraint $y^\top A_{\cdot p} = y^\top B_{\cdot p}$:

$$t = y^\top B_{\cdot p} - \tilde{y}^\top B_{\cdot p} \tag{3.7}$$

$$\tilde{y}(t)^\top = y^\top - t \cdot e_p^\top B^{-1} \tag{3.8}$$

$$\tilde{d_{\mathcal{N}}}(t)^\top = d_{\mathcal{N}}^\top + t \cdot e_p^\top B^{-1} A_{\mathcal{N}} \tag{3.9}$$

$$\tilde{Z}(t) = Z + t \cdot (x_p - u_p) \tag{3.10}$$

Here, Z represents the current dual objective value. For this example, we require $t \geq 0$ to ensure $\tilde{Z} \geq Z$ because we need to maximize the dual objective when solving LP 1.2 and $x_p > u_p$. The dual ratio test determines the correct size of t and therefore also the non-basic index that should become basic. The final *dual step length* $t = \alpha_D$ (see also Algorithm 1.2) is computed according to (3.8) and (3.9) to satisfy $\tilde{y}_q = 0$ or $\tilde{d}_q = 0$ for one $q \in \mathcal{N}$.

In the conventional ratio test, the first entry $d_q, q \in \mathcal{N}$ in the reduced cost vector to become 0 while enlarging the absolute value of t, will define the entering pivot index. As soon as the step length is increased further, dual feasibility will be violated.

The *bound flipping ratio test*, on the other hand, allows us to choose such a larger step length to improve the objective function value even further. Since the dual simplex algorithm needs to preserve dual feasibility, it is necessary to repair the introduced violations. This can be done by flipping the corresponding primal non-basic variables from their lower to their upper bound or vice versa according to the feasibility conditions (3.5) and (3.4). This technique can only work for variables or slacks that are (finitely) bounded from both sides, of course. Another point to consider is that with every performed bound flip the following objective function improvement step is decreased. It is important to keep track of this behavior and to stop enlarging the step length before the dual objective function value is reduced again and bound flips become detrimental.

From (3.10) we can see that the initial slope of $\tilde{Z}(t)$ is the primal violation of the leaving variable. This variable x_p depends on the values of the non-basic variables. If, for instance $x_{q_k}, q_k \in \mathcal{N}$ is flipped from l_{q_k} to u_{q_k} the new value $\hat{x}_p$ of x_p is computed as follows:

$$\hat{x}_p = e_p^\top B^{-1} \left(s_{\bar{\mathcal{R}}} - A_{\cdot\bar{\mathcal{C}}} x_{\bar{\mathcal{C}}} + A_{\cdot q_k} (u_{q_k} - \ell_{q_k}) \right)$$

Therefore, the update formula for the dual objective function value needs to incorporate the new slope after the bound flip (note that we use $\dot{x}_p$ to denote a single step update):

$$\begin{aligned} \tilde{Z}(t) &= Z - t_1 (x_p - u_p) - (t_2 - t_1)(\dot{x}_p - u_p) \\ &= Z - t_1 (x_p - u_p) - (t_2 - t_1)(x_p + e_p^\top B^{-1} A_{\cdot q_1} (u_{q_1} - \ell_{q_1}) - u_p) \end{aligned}$$

The dual objective function increase is a piecewise linear function as depicted in Figure 3.3. The number of bound flips that can be performed in one iteration depends on how quickly the slope flattens. In case there is a pivot candidate x_{q_k} with an infinite bound the slope would be $-\infty$ and the procedure needs to be stopped.

After a number of bound flips have been performed and the entering index is chosen, it is necessary to update the basic solution as well to correspond to the changed non-basic values. This requires one additional solve with the basis matrix that can be done together with the mandatory FTRAN operation at the end of every simplex pivot. The change in the basic solution is determined by the difference between two flipped bounds:

$$\begin{bmatrix} \hat{x}_{\mathcal{C}} \\ \hat{s}_{\mathcal{R}} \end{bmatrix} = \begin{bmatrix} x_{\mathcal{C}} \\ s_{\mathcal{R}} \end{bmatrix} + B^{-1} \left(\hat{s}_{\bar{\mathcal{R}}} - A_{\cdot\bar{\mathcal{C}}} \hat{x}_{\bar{\mathcal{C}}} \right)$$

with

$$\hat{s}_{\bar{\mathcal{R}}} = \sum_{i \in \{L_i \nearrow U_i\}} (U_i - L_i)\, e_i + \sum_{i \in \{U_i \searrow L_i\}} (L_i - U_i)\, e_i$$

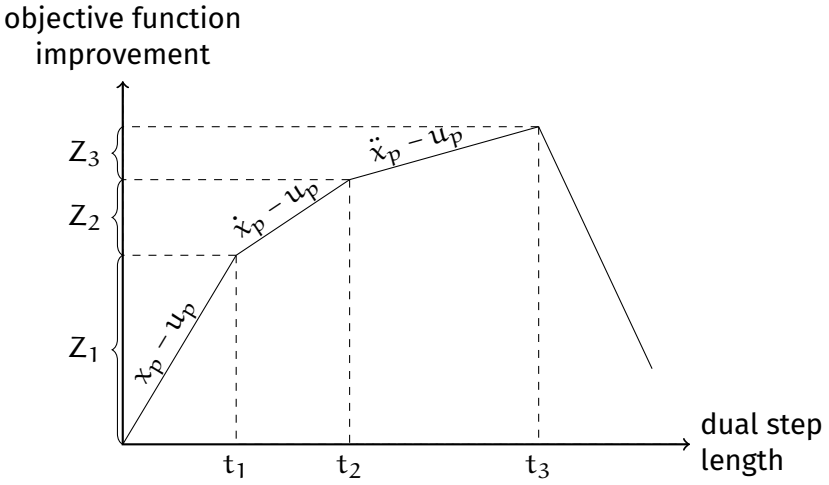


Figure 3.3.: Development of the dual objective function value with bound flips: t_1 refers to the step length of the conventional ratio test, Z_2 and Z_3 are further improvements on the objective function, $\dot{x}_p$ and $\ddot{x}_p$ are new values for x_p after the first and the second bound flip, respectively.

and

$$\hat{x}_{\bar{\mathcal{R}}} = \sum_{j \in \{l_j \nearrow u_j\}} (u_j - l_j)\, e_j + \sum_{j \in \{u_j \searrow l_j\}} (l_j - u_j)\, e_j .$$

3.4.3. Technical Improvements

We want to mention two technical improvements that can be implemented with the bound flipping ratio test and that to the best of our knowledge have not been described in the literature before.

Early Termination The textbook ratio test is usually performing two passes over the pivotal row. This is commonly known as *Harris ratio test* after Harris (1973) and tries to make a more stable pivot selection by allowing a small violation of feasibility. This violation can later be repaired by switching the simplex type from primal to dual or vice-versa. In the bound flipping ratio test the same approach can be applied while also storing the individual pivot candidates for subsequent sorting. We can save the best candidate during this collection phase and immediately return the selection in case no bound flips are possible. This avoids having to sort the candidates first and is a rather straightforward technique to avoid unnecessary operations.

Quick Selection When traversing the collected break points, we require them to be sorted in ascending order. This can be done by means of algorithms like quick-sort. In SoPlex, we use a partial quick-sort to avoid having to arrange the entire list beyond

the final flip that still improves the objective. The sorted part of the list is iteratively extended whenever more bound flips can be performed, providing at least the next k smallest elements in every new call—in SoPlex, k is set to four. This ensures that all predecessors of the selected element can be processed accordingly and dual feasibility is maintained.

In fact, the order of the flipped candidates is irrelevant, as long as they are not larger than the selected one. For such a scenario an implementation of the weighted-median-selection algorithm can be applied. This method selects the maximal number of smallest elements—according to their weights—to not exceed a certain capacity. This capacity is represented by the dual objective improvement. As this precisely meets the requirements of the bound flipping ratio test, we can save some computational overhead by not having to sort the flipped candidates.

While appearing computationally inferior, we still use the partial quick-sort implementation in SoPlex because a prototype version of the weighted-median-selection approach did not result in a positive performance impact.

3.4.4. Performance Impact

To demonstrate the effect of the bound flipping ratio test we compare the performance on the Netlib LP instance `fit2d` (see Table 3.2). This instance is well-suited for the bound flipping ratio test with its 10 500 variables, all of which are boxed, that is, have lower and upper bounds. The problem instance is also very dense with just 25 constraints and exhibits a 10-fold improvement in solving time and a 20 to 30-fold reduction of iterations compared to the simpler ratio test without bound flips.

	steepest edge		devex	
	time	iterations	time	iterations
no bound flips	0.8	6168	1.37	12 841
long step rule	0.08	283	0.13	432

Table 3.2.: Performance impact of the bound flipping ratio test on Netlib instance `fit2d` with SoPlex 5.0.2 and two different pricing variants.

Of course, we cannot evaluate the performance of any feature by inspecting just a single instance, so please refer to Table 3.4 for a more comprehensive LP performance impact analysis. We see an increased iteration count of 17% when disabling long steps and an increase of 7% with regards to the overall solving time. This benchmark compares the pure LP performance across the `allLP` instance set.

Still, we want to emphasize that this technique is very much dependent on the problem instance to be solved. We would also like to note that, keeping true to the object-oriented design of SoPlex, the bound flipping ratio test is implemented as sub-class of the previous default method *fast ratio test* and is able to fall back easily if no bound flips are possible for a specific instance. Every 100$^{\text{th}}$ iteration, another try with the

bound flipping ratio test is performed when otherwise no successful long steps could have been performed.

Finally, we need to add that despite our best efforts the bound flipping ratio test does not have a significant performance impact on SCIP when running the MIPLIB 2017 benchmark as shown in Table 4.2. There is a considerable reduction in the number of simplex iterations but this does not carry over to an improved overall solving time.

3.5. Pricing Variants

The *pricing* step in the simplex algorithm determines which edge of the polyhedron to cross in order to reach a neighboring vertex. Often, there are multiple options to go from one basis to an improving one. The choice of the pricing method can drastically impact the number of iterations, as it is defining this path to optimality.

There have been numerous studies about simplex pricing rules and the two most popular and successful methods have proven to be *devex pricing* and *steepest edge pricing*. The pricing rule chooses the next step direction based on the currently not fulfilled optimality conditions—dual infeasible variables in the primal simplex and primal infeasible variables in the dual simplex—a natural approach is to always take the most violated candidate. This pricing rule is known as *Dantzig pricing*, named after the simplex inventor George Dantzig, but just like the most infeasible branching rule in MILP solving it turns out to be a very impractical technique as shown by Achterberg, Koch, and Martin (2004). Its bad performance with respect to the total number of iterations is due to its lack of direction optimality. This is exactly where other pricing rules step in and try to improve upon by computing a weighted pricing score.

3.5.1. Steepest Edge Pricing

Steepest edge pricing is a rather expensive pricing rule that requires the solution of an additional system of linear equations involving the current basis matrix in every single iteration. The benefit is that the selected edge directions are typically leading to a smaller iteration count than other methods. The technique was introduced by Forrest and Goldfarb (1992) with an efficient update technique for the pricing weights. This paper also thoroughly describes different steepest edge variants and is highly recommended for further information on the topic.

In a nutshell, steepest edge pricing chooses the direction that has the most obtuse angle with the objective function and hence is expected to result in a short path to optimality.

3.5.2. Devex Pricing

At the most general level, *devex pricing* can be seen as an approximation of steepest edge pricing, saving computational cost per iteration when calculating the edge weights in exchange for a usually higher iteration count. This method was introduced

by Harris (1973). Interestingly, *devex* derives from the Latin term *devexus* meaning *steep*.

3.5.3. Shadow Pricing

This pricing rule is mostly relevant for theoretical purposes with respect to investigating computational complexity of the simplex algorithm as performed by Dadush and Huiberts (2019). SoPlex does not implement a shadow pricing rule, we still want to mention this technique for completeness.

3.5.4. Parallel Pricing Rules

There are also some variations of the above rules that can be employed for parallel implementations. In the original SoPlex publication (Wunderling, 1996), the *parallel multiple pricing rule* is described which inspects several pricing directions at the same time.

In Huangfu and Hall (2018) the authors also describe an elaborate pricing scheme that works well when executed in parallel.

The current version of SoPlex is entirely sequential and does not implement any parallel pricing rules.

3.5.5. Automatic Pricing Rule Selection

There have also been attempts to decide automatically which pricing rule might be the best for the current problem instance. We investigated three different applications for automatic algorithm selection for mixed-integer optimization (Hendel, Miltenberger, and Witzig, 2018): A multi-armed bandit implementation has been used to choose between devex pricing, steepest edge pricing and steepest edge pricing using non-exact initial weights. The results have been partly successful, demonstrating that SCIP using CPLEX as LP solver could benefit from slight performance improvements, while for SoPlex as LP solver this was not the case. Here, the criteria for a successful pricing rule was defined as the number of LP solves or processed branch-and-bound nodes in a fixed amount of time—the LP throughput. This setup is flawed with respect to determinism because the actual running time influences the solver's behavior. Implementing a deterministic clock is a major effort and usually involves tracking every single memory access as an alternative means of measuring the amount of performed work. Without this, the practicality of the approach is very limited.

Other applications for the multi-armed bandit approach inside SCIP's repertoire of algorithms have been proven to be more suitable to improve the performance. Especially the diving and large neighborhood search heuristics can benefit from this adaptive algorithm selection.

It is important to note that SoPlex implements a static heuristic pricing rule by default, that switches to steepest edge after 10 000 iterations of devex have been performed. The rationale of this pricing scheme is that steepest edge is more expensive,

so we try to solve the model instance with the cheaper method and only switch over for harder problems. This is clearly a compromise that will rarely result in a minimal number of iterations when compared to the fixed methods. Still, experimental studies over very diverse test sets and parameter combinations have revealed that this a viable approach—especially when SoPlex is being used as LP solver within SCIP. We will go into more detail in Section 4.9.

3.6. Exploiting Sparsity

The majority of real-world problems from all kinds of applications and backgrounds rely on sparse data: The actual information is diminishing in comparison to the dimensions. Take a realistic supply chain management problem: To produce a certain product in a machine or factory, only a very small number of different resources are required, while the entire supply chain may comprise many more resources that are used in other contexts. This culminates in problem instances that have few non-zero variable coefficients in their constraints despite a high number of total variables.

We then speak of *sparse data* and the solvers need to implement specialized data structures to store and manipulate these values. It is not affordable to store all the zero values so SoPlex—just like any other modern solver—keeps index lists to store the mathematical position of the values while the values themselves are packed one after another to allow for fast access.

There are variations of this storage technique, like scattering the values into a larger array to facilitate adding new non-zero entries at the correct place.

Huangfu and Hall (2018) give a detailed description of a high performance simplex solver and also explain how to exploit the sparsity effects. The term *hyper-sparsity* was introduced for the linear programming context by Hall and McKinnon (2005) and has been extensively studied by the authors. In contrast to sparse input data, hyper-sparse is referring to sparse output data in certain computations in the simplex method, like the solutions of linear systems with the basis matrix or its transpose. Exploiting hyper-sparsity requires to assess where in the output data non-zero values may appear. This can significantly reduce the computational time for the affected operations and leads to an average speed-up of more than 5 for instances where the techniques can be applied.

In SoPlex version 2.2 (Gamrath, Fischer, et al., 2016), we implemented such a sparsity-exploiting technique to accelerate the pricing process. This has been especially helpful for solving large-scale linear programs originating from supply-chain management optimization as demonstrated by Gamrath, Gleixner, et al. (2019). Figure 3.4 shows the number of necessary comparisons to determine the best pricing candidate for the next pivot. We need to scan the vector of basic variables for primal violations. Since only a subset of these violations have been modified during a simplex iteration, we can keep a dynamically updated list of candidates that can be much smaller than the basis size.

Despite all three methods shown in Figure 3.4 being mathematical equivalent, we

can see that a majority of comparisons can be avoided when only taking the updated violations into account—note that the the number of comparisons is depicted in logarithmic scale.

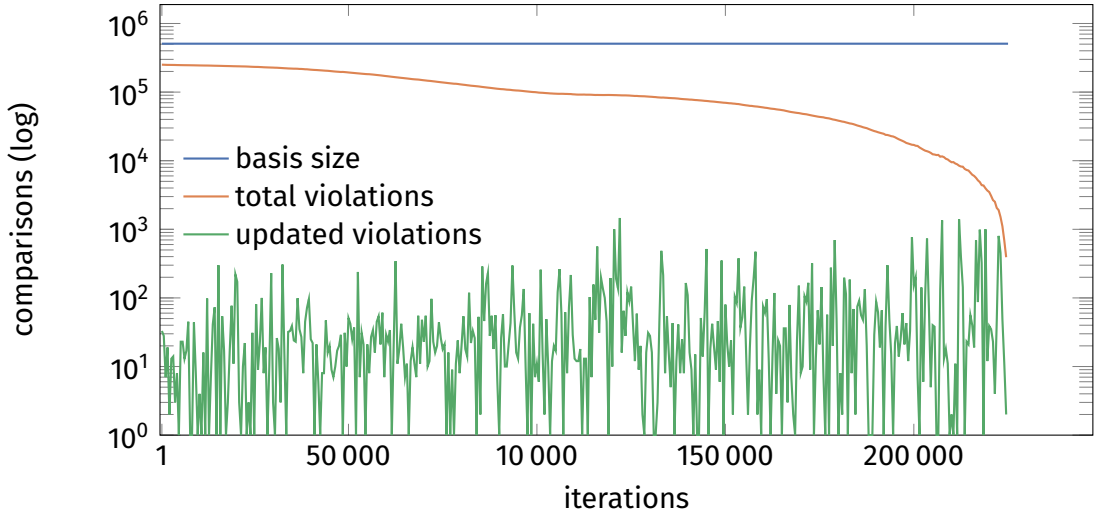


Figure 3.4.: Pricing statistics: The total number of current violations is several orders of magnitude larger than the number of updated violations per iteration.

Another application for exploiting sparsity is located in the PRICE step. Here, a matrix-vector multiplication needs to be performed. Depending on the sparsity of the multiplied vector, it can be beneficial to perform a row-wise rather than column-wise operation. This requires a row-wise storage of the constraint matrix A, which can be made available once without significant additional cost. What is more difficult to exploit is the fact that only the non-basic columns of $A_{\mathcal{N}}$ are required because the multiplying vector $\pi^T = e_j^T B^{-1}$ is the result of solving a linear system with $B = A_{\mathcal{B}}$ that involves the basic columns. We know that the resulting values will be 0 or 1 for those indices corresponding to basic variables. Hence, we can skip these operations and perform the multiplication only with $A_{\mathcal{N}}$ like it is presented in most text book descriptions. In the row-wise storage of A we need to update the partitions of basic and non-basic columns after every iteration to have all non-basic values in a consecutive block in memory:

$$A^{\text{ROW}} = [A_{\mathcal{N}} A_{\mathcal{B}}]$$

From personal communications with Julian Hall we have learnt that this additional overhead is usually worth it when weighed against the computational benefit of a shorter matrix-vector product.

Unfortunately, we were not able to speed up SoPlex with an implementation of this modified PRICE multiplication. Koberstein (2005) also explains the different ways of computing the pivotal row as a result of either a matrix-vector-multiplication using the row-wise or the column-wise storage of A. They, too, admit that they were not able

to implement an efficient method of skipping and updating the unnecessary basic indices in the row-wise multiplication.

There are two different data structures implemented in SoPlex to support sparse vectors: `SVector` to store only the value and index pairs including some additional slot for the number of non-zeros. The other is a semi-sparse vector called `SSVector` with length equal to the actual dimension of the vector it represents and all non-zero elements at their corresponding position. Additionally, it also holds an index list of all non-zero entries to allow faster access to them and facilitate easy inclusion of new entries. What makes these data structures different from conventional sparse vectors is the fact that they use a tuple of value and index to store the data. Typically, there are two separate vectors for the values and for the indices, respectively. All underlying linear algebra sub-routines are built on these data structures and we suspect that an extensive rewrite is necessary to implement sparsity-exploiting algorithms more efficiently.

Finally, we would like to mention a rather straight-forward performance improvement technique that is especially useful for an LP solver within an MILP solver. It is common to have an incumbent integer feasible solution at some point during the solving process without the proof of optimality, so further branch-and-bound nodes need to be explored. For the corresponding LP relaxations, we can set an *objective cutoff limit*, because we are not interested in nodes that exhibit a worse dual bound than the objective value of the current incumbent. Those nodes can be pruned and the LP relaxation can be aborted at an early stage. To determine when the objective limit has been reached, we need to compute the current dual objective value at every iteration, which can be costly if not implemented carefully. Intuitively, one would just compute the new objective value based on the current values of the variables. We added an update method for the non-basic part of the objective function value to reduce the computation to only the basic part. This prevents repeated multiplications of the objective coefficients c_j with those variables x_j that did not change from the previous iteration. As typically only one non-basic variable changes its value during a simplex iteration, the complexity of computing the non-basic part of the objective value reduces from $O(n - m)$ operations to a constant factor.

This update is of course not necessary if only lower bounds of 0 and no upper bounds are present—in this case, it suffices to compute the basic part because $c_{\mathcal{B}}^\top x_{\mathcal{B}} = c^\top x$. In virtually all cases, though, the LP solver is faced with non-zero lower and finite upper bounds coming from domain propagation or branching decisions of the MILP solver or simply from upper bounds on integer and binary variables of relaxations of combinatorial optimization problems.

The above technique has a greater performance impact on larger instances, like the ones investigated by Gamrath, Koch, et al. (2015) and Gamrath, Gleixner, et al. (2019).

3.7. Persistent Scaling

The main goals of scaling are different for LP and MILP solvers. When dealing with MILPs, one typically scales to improve the integrality, that is, a constraint like the following can be easier handled after applying a row scaling factor of 3:

$$1/3 \cdot x + 2/3 \cdot y \leq 13/6 \quad \xrightarrow{\cdot 3} \quad x + 2 \cdot y \leq 6.5.$$

For LPs, on the other side, we employ scaling to improve the numerical features of the basis matrix to reduce the chance of introducing errors that might invalidate the correctness of our solutions.

An example in Gurobi's documentation demonstrates[2] the effect of scaling by explicitly increasing the ranges of coefficients for a specific model instance. With further increasing detrimental scaling factors, the performance deteriorates until eventually the problem is incorrectly detected to be infeasible. This example shows that scaling affects not only the numerical conditions and the quality of the solutions but also the time and number of iterations required to solve a problem instance.

For an LP relaxation within an MILP solver, the simple way of using scaling—as well as presolving for that matter—is to apply it before the root node solving when there is no other information to be taken care of. After the root LP has been solved, all the data structures in the LP solver are transformed back into the original space to allow the MILP solver to easily apply modifications like bound changes or the addition of new LP rows.

Preserving the LP presolving transformations and translating all changes coming from the MILP solver into this transformed space is quite involved and likely too expensive to be useful. This is different for the less involved scaling, though, and our persistent scaling implementation proves that we can keep using scaled data even for later LP solves during the MILP solving process.

After the initial scaling has been computed, every change to the LP data and almost every information coming back from the LP solver needs to be scaled accordingly, to avoid having to scale the LP itself back and forth. Newly introduced columns or rows need to be assigned corresponding scaling factors, accordingly, extending the initial set of scaling factors stemming from the root LP. Additionally, results from computationally more involved operations like computing a certain row of the current basis matrix' inverse to generate cutting planes need to be un-scaled as well.

This extension to the API of SoPlex has been implemented since version 3.0 and allows preserving the scaling factors throughout the MILP solving process without negatively impacting the overall performance. To be able to use this in SCIP, we also had to rewrite the LP interface entirely as it was previously accessing internal data structures directly, circumventing the dedicated API.

Persistent scaling for SoPlex has been enabled by default since SCIP version 4.0. Table 4.2 shows that about five more instances can be solved to optimality, five less

[2] https://www.gurobi.com/documentation/9.5/refman/why_scaling_and_geometry_i.html

instances run into the time limit of one hour, and the average solving time is reduced by around 9%. The number of simplex iterations is also positively affected by persistent scaling and goes down by about 15% while the average number of nodes is largely the same. We used SCIP version 6.0.2 for this experiment and manually disabled persistent scaling in SoPlex as there is no parameter in SCIP to control this directly.

In the following, we discuss how scaling is actually implemented for SoPlex and other simplex solvers.

When we talk about LP scaling we think of two scaling matrices R and C. Those are diagonal square matrices of size m and n respectively. For all scaling factors we do not store and use the actual floating point values computed by the scaling methods but the closest base-2 exponents. This enables us to perform scaling operations without introducing any numerical noise because only the exponent of the value to scale is modified. The mantissa holding the digits remains untouched. Even though this procedure slightly modifies the actual scaling factors that are computed by the different methods, error-free computations are more crucial. Other benefits of this setup are reduced storage size for the factors (`int` instead of `double` or `rational`) and faster scaling operations since we only need to shift the exponent.

Besides the constraint matrix, also the upper and lower bounds of both columns and rows need to be scaled accordingly:

$$\begin{aligned} RACx' &= Rb \\ C^{-1}l \leq x' &\leq C^{-1}u \\ x &= Cx' \end{aligned}$$

Now, consider a scaled constraint matrix A' that is extended by artificial slack variables resulting in the matrix (A', I). A basis matrix $B' = [(A', I)\, P]_{1:m,1:m}$ (with P being a permutation matrix) for the scaled problem corresponds to the basis

$$B = R^{-1} \left[(A', I)\, P \right]_{1:m,1:m} \left[P^{\top} \tilde{C}^{-1} P \right]_{1:m,1:m}.$$

In this equation, $\tilde{C}$ is of the form

$$\begin{bmatrix} C & 0 \\ 0 & R^{-1} \end{bmatrix}.$$

We can see from those formulas that changes to the LP—as they frequently happen during MILP solving—need to be transformed carefully between the scaled and the original space. In case of the row representation it is additionally necessary to translate the values between the representations because the interface to SoPlex needs to provide the common column representation data of the basis matrix. This turns the seemingly simple task of storing scaling factors into a considerable implementational effort.

3.7.1. Scaling Methods

There are different ways to scale an LP and it cannot be easily determined which will lead to the best performance. Generally, we aim to move all coefficients of the problem as close as possible towards 1 or -1. This is to reduce the numerical noise that is introduced when working with floating point numbers. Whenever we are performing operations using floating point arithmetic, there is a certain inexactness, mostly because the number of precisely representable values is rather small and we usually end up with the closest approximation to the real value. We trust the numbers only until the first nine or ten digits, although a double precision floating point number consists of 15 to 17 significant decimal digits (see also Table 6.1 for a comparison of different floating point precisions). Some operations like subtraction of equally large numbers can then push those not trust worthy digits to the front and cause the aforementioned numerical noise.

A related issue is introduced by numerical tolerances: Take a commonly used feasibility tolerance of 10^{-6} for example. Then, a simple constraint of the form

$$200 \cdot x - y = 0$$

is feasible for values $x = 1$ and $y = 200$ but infeasible for $\tilde{x} = 1 + 10^{-8}$ and $y = 200$ with

$$200 \cdot \tilde{x} - y = 200.000002 - 200 = 2 \cdot 10^{-6} > 10^{-6}.$$

This is despite the value of $\tilde{x}$ being even within the integrality tolerances of most solvers, so being treated as an integer value.

SoPlex supports different scaling methods that we will describe briefly:

Equilibrium Scaling. This scaling method simply divides all coefficients in a row or column by the largest one. This trivially guarantees that the largest coefficient is of absolute value 1. There is still a degree of freedom in the order of the scaling, that is whether row or column scaling is applied first. Since this seemingly minor detail can lead to different outcomes as Example 1 shows, we employ a simple heuristic based on the row and column ratios to determine the order: The method that is applied first is the one that has the smaller maximal ratio of largest divided by smallest entry.

Two variants of this scaling method are implemented in SoPlex: Uni-equilibrium does only one pass, either over columns or rows, depending on the ratio, while bi-equilibrium always does both with the order being chosen by the ratio.

Example 1.

$$A = \begin{bmatrix} 0.1 & 1 \\ 10 & 1 \end{bmatrix}$$

Matrix A has a maximum row ratio of 100 and a maximum column ratio of 10, when dividing the largest coefficient by the smallest in every row and column respectively.

$$\textit{row scaling first: } A' = \begin{bmatrix} 0.1 & 1 \\ 1 & 0.1 \end{bmatrix} \qquad \textit{column scaling first: } A' = \begin{bmatrix} 0.01 & 1 \\ 1 & 1 \end{bmatrix}$$

Geometric Scaling. This method computes scaling factors based on the square root of the product of the largest and the smallest value in each respective row or column. There is also an extension called *iterated geometric scaling* that performs several of such sweeps over the constraint matrix to achieve a more balanced result. A maximum iteration count of 8 has proven to be enough—subsequent sweeps do not improve the result considerably.

Least Squares. This is the most sophisticated and also most expensive scaling variant. It has been implemented for SoPlex version 3.1 (Gleixner, Eifler, et al., 2017). The goal of *least squares scaling* is to achieve the best possible numerical stability of A with respect to Gaussian elimination or LU factorization. This method is also known as *Curtis-Reid scaling* (Curtis and Reid, 1972). Similar to equilibrium scaling it tries to even out outliers and smooth all entries of the matrix. Unlike the simpler methods this is carried out by solving a least squares problem:

$$\min \sum_{A_{ij} \neq 0} (\log_2 A_{ij} - \rho_i - \gamma_j)^2 \quad , i \in \{1, \dots, m\}, j \in \{1, \dots, n\} \tag{3.11}$$

The resulting ρ and γ values are rounded and define the scaling factors as $R_{ii} = 2^{-[\rho_i]}$ and $C_{jj} = 2^{-[\gamma_j]}$ respectively.

We can solve the least squares problem 3.11 using the conjugate gradients method (Hestenes and Stiefel, 1952) applied to this system of linear equations:

$$M\rho + E_A\gamma = \left[\sum_j \log_2 A_{1j}, \dots, \sum_j \log_2 A_{mj} \right]^T$$
$$E_A^T\rho + N\gamma = \left[\sum_i \log_2 A_{i1}, \dots, \sum_i \log_2 A_{in} \right]^T$$

Here, E_A is a 0-1-matrix with the same non-zero pattern as A whereas M and N are diagonal matrices that contain the numbers of non-zero entries for each row and column of A respectively.

Computing these least squares scaling factors is computationally more expensive than the other methods but still does not impair the performance on most instances.

Elble and Sahinidis (2012b) compare different scaling methods specifically for scaling linear programming problems. Their main result is that *equilibrium scaling* provides the best results under the assumption that computational costs cannot be neglected. SoPlex uses equilibrium scaling by default. When applied inside SCIP, users

can enable *aggressive scaling* to switch to *least squares* for numerically more challenging problem instances.

Note that interior point solvers are less susceptible to scaling as explained by Anderson et al. (1996).

3.8. LU Factorization and Update

We will not go into detail on how to implement an efficient and numerically robust LU factorization including the necessary update methods for the simplex algorithm. As this is a crucial ingredient in any simplex code, we give an overview of the standard methods and some fairly recent developments in this field.

In a nutshell, our task is to solve two systems of linear equations in every iteration of the simplex algorithm. Both involve the current basis matrix B while one uses the transpose $B^\top$:

$$\begin{aligned} &\text{BTRAN:} && B^\top z = e_p \\ &\text{FTRAN:} && B\hat{a}_q = a_q \end{aligned}$$

These two operations are historically called *FTRAN* and *BTRAN*—short for forward transformation and backward transformation—and hark back to the very first computer implementations using punch cards to encode the machine instructions: As BTRAN uses the transpose of the basis matrix, its implementation is equivalent to inserting the punch card backwards. It is interesting to see this naming convention still being present and used even today in state-of-the-art high-performance simplex implementations.

For solving these two systems of linear equations, a prior LU factorization of $\mathcal{B}$ has been proven highly useful and efficient:

$$\begin{aligned} &\text{factorize:} && B = LU \\ &\text{solve:} && LUx = b \Rightarrow Ly = b \text{ and } Ux = y \end{aligned}$$

Note that, while this factorization is less numerically stable than for example a QR factorization, that is, a factorization into an orthogonal matrix Q ($Q^\top = Q^{-1}$) and an upper triangular matrix R, the sparsity preservation and update functionalities of the LU factorization are more important for application in the simplex method. The two resulting systems involve lower and upper triangular matrices L and U that facilitate a straightforward solution process as shown in Algorithm 3.1.

The procedure for U can be carried out analogously, starting from the bottom and moving upwards. Due to the non-unit diagonal of U we need to add a multiplication with $1/u_{jj}$ to achieve the correct result, see Algorithm 3.2.

As we can see from those algorithms, sparsity in the input data, that is, in the respective right-hand sides and in the triangular matrices, can already be used to skip certain operations. This is only scratching the surface of what is possible to speed up these computations as demonstrated by Hall and McKinnon (2005). One important

Algorithm 3.1 lower triangular solve: FTRAN-L

Input: $b \in \mathbb{R}^m$ and $L \in \mathbb{R}^{m,m}$ lower triangular and unit diagonal
  $y \leftarrow b$
  for $j \in \{1, \ldots, m\}$ **do**
    if $y_j \neq 0$ **then**
      for $i \in \{j+1, \ldots, m\}$ with $L_{ij} \neq 0$ **do**
        $y_i \leftarrow y_i - L_{ij} y_j$
      end for
    end if
  end for
  Return: y with $Ly = b$

Algorithm 3.2 upper triangular solve: FTRAN-U

Input: $x \in \mathbb{R}^m$ and $U \in \mathbb{R}^{m,m}$ upper triangular
  $x \leftarrow y$
  for $j \in \{m, \ldots, 1\}$ **do**
    if $x_j \neq 0$ **then**
      $x_j \leftarrow {}^1/_{u_{jj}} x_j$
      for $i \in \{j-1, \ldots, 1\}$ with $U_{ij} \neq 0$ **do**
        $x_i \leftarrow x_i - U_{ij} x_j$
      end for
    end if
  end for
  Return: x with $Ux = y$

aspect is the accurate prediction of where the so-called *fill-in* happens. These are non-zero elements in the solution vector that do not already have a corresponding non-zero entry at the same position in the right-hand side vector. Graph algorithms applied to the non-zero structures of the triangular matrix and the right-hand side can be used to determine these positions. Of course, this introduces some significant overhead and needs to be adaptively activated depending on the sparsity of the current system. SoPlex only implements a subset of these techniques resulting in a worse performance than other solvers when dealing with highly sparse model instances.

In every iteration of the simplex method, we update the basis matrix by exchanging a column vector with one of the non-basic columns a_q of the constraint matrix A or a unit column. This can be formalized as

$$\tilde{B} = B + (a_q - Be_p)\, e_p^\top$$

with the incoming column a_q replacing the pth basic column in B and the basis itself is updated to $\tilde{\mathcal{B}} = \mathcal{B} \cup \{q\} \setminus \{p\}$. Every such update needs to be integrated into the factorization to properly represent the current basis matrix and in regular intervals a fresh factorization is computed to clean up the accumulated updates. Huangfu and Hall (2015) provide a survey of the standard methods and several variants further improving speed and efficiency. SoPlex implements both the rather straightforward method of representing the updated LU factorization as series of rank-one matrices called the *product-form update* developed by Dantzig and Orchard-Hays (1954) as well as the more technical *Forrest-Tomlin update* as described by Forrest and Tomlin (1972) and Suhl and Suhl (1993).

For a comprehensive overview of all the different basis inverse update techniques we refer to Elble and Sahinidis (2012a).

3.9. Iterative Refinement

The term *iterative refinement* was coined by Wilkinson (1963) for a more accurate solution of systems of linear equations. The main idea is to solve multiple auxiliary systems of linear equations that iteratively minimize the error in the solution of the original problem.

This technique can be applied to linear programming using the simplex method in an elegant way as demonstrated by Gleixner, Steffy, and Wolter (2012), Gleixner (2015), and Gleixner, Steffy, and Wolter (2016). The authors exploit the discrete nature of the simplex basis to perform subsequent optimizations of scaled versions of the original problem that aim for minimizing the error of the solution. This is achieved by using higher-precision arithmetic like in other simplex implementations, see for instance QSopt_ex (Espinoza, 2006). The main advantage, though, is that only very few iterations using expensive computations in rational arithmetic are necessary to improve upon the initial solution provided by the standard solver in double precision arithmetic.

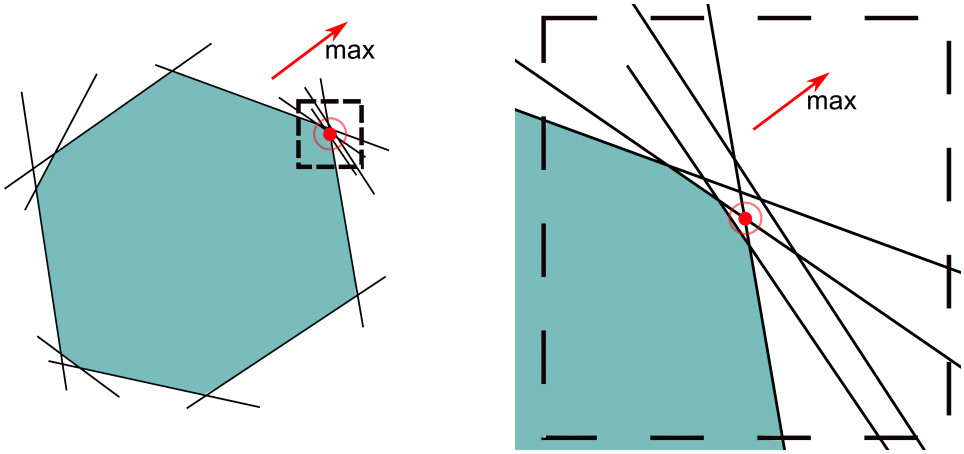


Figure 3.5.: LP iterative refinement to compute more accurate solutions

The authors came up with a very nice visualization of the technique as shown in Figure 3.5, where the refinement LPs can be seen as *zoomed-in* versions of the original formulation. While a potential bound or optimality violation is not visible in the first depiction, we can spot the infeasibility in the scaled version on the right.

3.10. Decomposition Based Dual Simplex

The *decomposition based dual simplex* (DBDS) is a method to exploit and avoid dual degeneracy in a given linear programming problem to reduce the number of iterations until optimality (Maher, Fischer, et al., 2017). Its core idea is based on the work of Elhallaoui et al. (2011), that introduces a column generation approach to deal with primal degeneracy, called *Improved Primal Simplex Algorithm*. This is especially helpful for solving instances of set partitioning problems. The method implemented in SoPlex can be seen as the dual version of this approach. It decomposes a problem that shows dual degeneracy at the current basis into a *reduced* and a *complementary* problem, that are non-degenerate. This reduces the total number of iterations, since fewer degenerate steps are taken.

DBDS relies on the row representation of the basis and is an iterative algorithm that keeps adding rows to the reduced problem until all primal violations are resolved.

In its current state, DBDS is an experimental algorithm that does not outperform the default simplex method in SoPlex in terms of time or iterations on general problems. There have been significant reductions in iterations on certain problem instances which shows that the method has potential as a problem specific implementation. It is unclear, though, what characteristics of a problem make it suitable for DBDS. The amount of degeneracy alone is not enough as experiments by Maher, Fischer, et al. (2017) have shown.

3.11. Performance Variability

The term *performance variability* was coined by Danna (2008) and further investigated by Koch, Achterberg, et al. (2011). Generally speaking, it refers to how much a seemingly insignificant change in the solver or the model data can introduce a drastic performance fluctuation. It is a stability measure that often helps assessing whether the results obtained on one machine will be reproducible on a different machine. Integer programming has been the focus of the mentioned works, while the class of linear programming problems is believed to be less impacted by this effect. We show how much the solving times and iteration counts vary after choosing a different random seed in SoPlex. The random seed determines the sequence of generated (pseudo) random numbers that are used throughout the code. This can be seen as a minimal change to impact the algorithmic behavior and is frequently used to stabilize performance benchmarks carried out by SCIP.

To measure performance variability, Koch, Achterberg, et al. (2011) use the *variability score* VS for every single instance in the different experiments:

Definition 3 (Variability score)**.**

$$VS(t,k) = \frac{1}{\sum_{i=1}^{k} t_i} \cdot \sqrt{\sum_{i=1}^{k} \left(t_i - \frac{\sum_{i=1}^{k} t_i}{k} \right)^2}$$

$VS(t,k)$ *measures how strongly a set of observations* $t_i, i = 1, \ldots, k$ *differs from one another.*

From Figure 3.6 we see that there is actually only a small number of instances that are sensitive with respect to the random seed choice. For those extreme cases, the solving times can vary by a factor of almost 10, as listed in Table 3.3. The vast majority of instances, though, does not show a strong performance variability.

3.12. Performance Impact of Selected Features

To assess how the mentioned features impact the performance of SoPlex we analyze computational experiments on the `allLP` test set. Table 3.4 compares default settings and runs with one of those features disabled and aggregate the results over three different random seeds. The iteration ratio column covers only those 1336 instances that have been solved by all settings.

We see that for all features except quick start steepest edge pricing, it is worthwhile to use default settings when aiming for the best solving time. There is a strong impact on the number of iterations when using a different pricing method that also leads to different solving times. On this test set, that includes many pure LP problems besides relaxations from MILP instances, we might as well use quick start steepest edge pricing for a slight performance increase.

In Chapter 4, Table 4.2, we compare how these LP improvements carry over to SCIP's performance on the MIPLIB 2017 benchmark set.

	iterations				time			
instance	max	min	ratio	VS	max	min	ratio	VS
cdma	260 271	56 336	4.6	0.3	158.2	16.2	9.3	0.4
cont1	107 902	41 395	2.6	0.3	3600.0	1570.2	2.3	0.2
dbic1	217 952	64 009	3.4	0.2	1132.7	189.3	6.0	0.3
mzzv42z	35 020	16 664	2.1	0.2	15.9	5.4	2.6	0.3
neos-2075418	116 602	46 961	2.5	0.3	698.4	95.3	7.3	0.6
neos-3402454	57 951	16 023	3.6	0.4	3600.0	609.0	5.9	0.5
neos-777800	11 398	2398	4.8	0.3	1.3	0.3	1.8	0.3
rlfdual	12 298	8317	1.5	0.1	5.7	1.3	2.9	0.3

Table 3.3.: allLP instances with a time or iterations ratio larger than 2.5 when solved with SoPlex 4.0.2 and a different random seed value. The test set contains 1382 instances, 1336 of which can be solved to optimality.

setting	wins	losses	iter ratio (all opt)	time ratio
no presolving	398	390	1.49	1.25
column representation	277	375	1.08	1.08
row representation	336	437	1.06	1.43
devex pricing	281	431	2.30	1.90
qsteep pricing	502	299	0.78	0.96
no bound flips	425	308	1.17	1.07
Curtis-Reid scaling	303	458	1.12	1.09
solution polishing	247	424	1.05	1.03

Table 3.4.: Performance impact of selected features on allLP

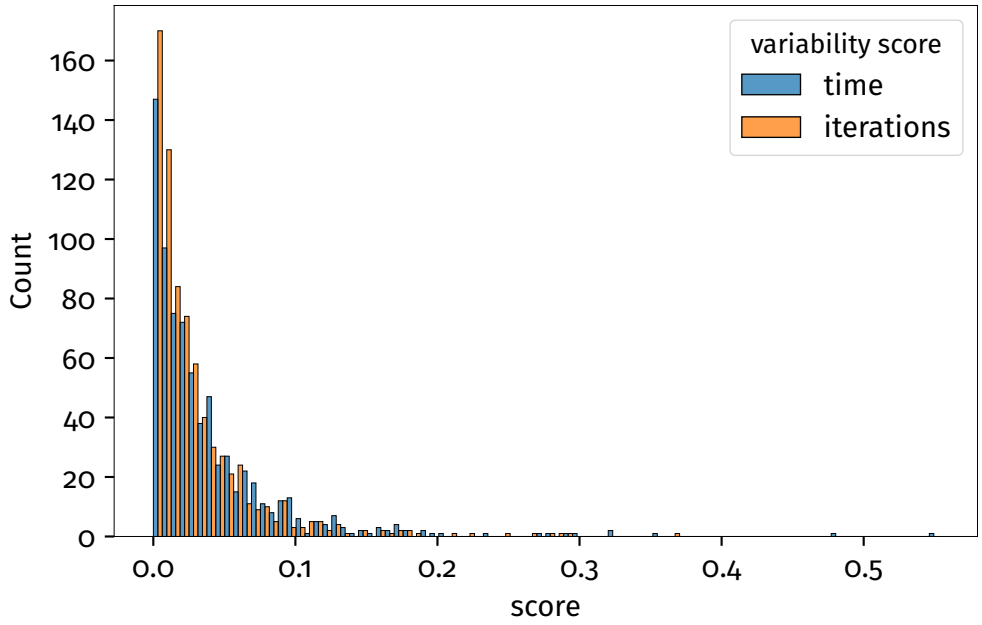


Figure 3.6.: Distribution of variability scores $> 10^{-6}$ for solving time and iterations of SoPlex 4.0.2 across three different seeds for the `allLP` test set

Chapter 4

Impact of Linear Programming in MILP

There are many situations in the MILP solving process where the solution of an LP is required. The most prominent one is arguably the root LP. This initial relaxation provides a first so-called dual bound for the MILP and determines the first branching decision—given that integer variables with fractional solution values are present. Often, it takes up a significant part of the overall solving time. During the tree search, on the other side, LP solutions of the branch-and-bound node relaxations can be obtained a lot faster with the help of previously processed nodes. This is mainly due to the warm-starting capabilities of the simplex algorithm. Here, an existing basis, for example from the parent node, can be used as a dual feasible starting basis for the next sub-problem and very few iterations suffice to solve it to optimality.

Dual feasibility is maintained throughout the tree search if only variable bounds or constraint sides are tightened. Table 4.1 lists the different problem modifications and whether they maintain primal or dual feasibility of the current basic (optimal) solution.

problem modification	primal feasibility	dual feasibility
additional variables	✓	-
additional constraints	-	✓
modified objective coefficients	✓	-
modified right-hand sides	-	✓
modified matrix coeffs (basic variables)	-	-
modified matrix coeffs (non-basic vars)	✓	-

Table 4.1.: Feasibility implications of different types of LP problem data modifications

While these two types of LPs—root LP and node relaxations—already cover the general branch-and-bound scheme, there are many sophisticated techniques that also rely on LP solutions, like heuristics, cutting plane generation or conflict analysis.

All this suggests that LP solving performance has got a major influence on the overall MILP solving performance. In this chapter we will investigate this impact and also show the fundamental differences of the two scenarios LP and MILP solving.

Koch, Martin, and Pfetsch (2013) mention the peculiar fact that the pure LP or simplex performance does not translate to a comparable MILP solving performance. This effect can also be observed for commercial solvers like Gurobi: A ~20% LP performance gain of version 9.1 compared to the previous version only resulted in a ~5% speed-up for solving MILPs. We want to investigate this unintuitive discrepancy using SCIP and its capability to plug-in various LP solvers.

Beside the widely used approach of using either a variant of the simplex method or the barrier algorithm internally, there are also other ways of computing feasible solutions to MILPs that we want to mention briefly: The commercial solver LocalSolver[1], for example, is using fast heuristics and a variety of combinatorial methods to find feasible solutions to MILPs without necessarily proving optimality. This is especially relevant and useful for model instances that have a very difficult and time consuming LP relaxation to solve or for very specific problem types. We will not go into detail for such approaches because they are fundamentally different from the LP-based branch-and-cut approach and usually do not provide a proof of optimality. Additionally, there are heuristic methods implemented in conventional MILP solvers that do not require an LP solution or even a dual bound in the first place.

We are only aware of few publications that are concerned with LP and MILP performance comparisons of different solvers. Meindl and Templ (2012) discuss the use of different LP solvers within SCIP and benchmark this against other implementations—it does not go into much detail, though, and mainly provides an overview.

In the following sections, we want to point out the different areas where LP solutions are most prominently used during the MILP solving process. To give a rough idea of the individual components that directly depend on the availability of LP solutions, we took the infamous *"SCIP flower"* and colored all buds red that actively solve new LPs (Figure 4.1).

4.1. Implementational Details

SCIP has an open interface to connect to many different LP solvers that are treated as black box. This is implemented through two main layers in the SCIP code: One is the internal LP object that represents the current relaxation and keeps information on pending bound changes and several other pieces of information and statistics. This layer is oblivious of the actual LP solver that is connected to the second layer—the LP interface or LPI. This interface implements methods to pass data to and from the connected LP solver and controls the LP solving process. Since the LP solver relies on its own data structures it cannot use the problem data that is stored in the LP object. Instead it has to construct a new copy of the relaxation and every modification to SCIP's

[1] http://www.localsolver.com/

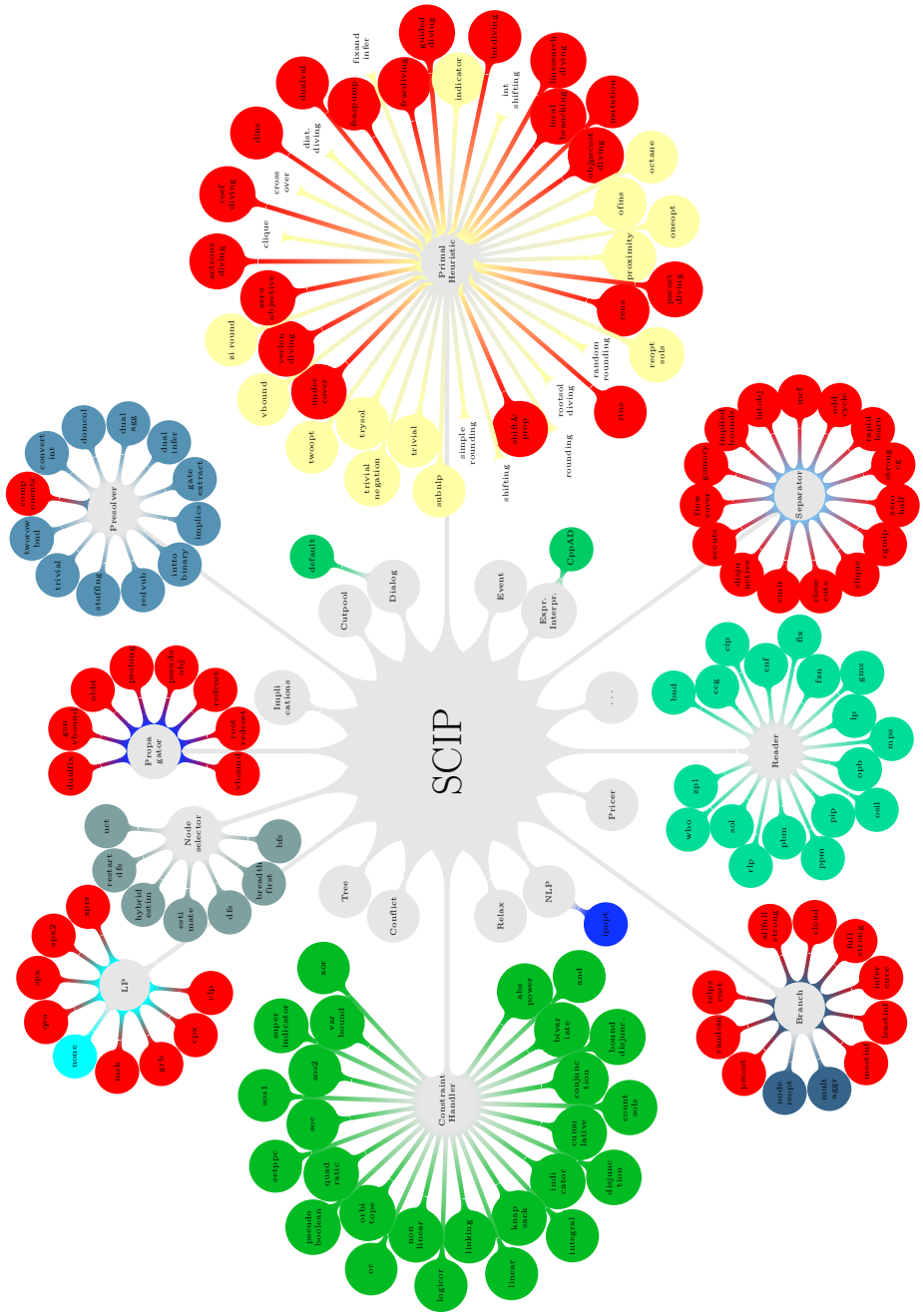


Figure 4.1.: Abstract visualization of SCIP's components with those colored in red that heavily rely on LP solutions or trigger new LP solves. The image depicts version 3.2 of SCIP and is only intended for the purpose of illustration and makes no claims of being complete.

internal LP needs to be transferred before initiating a new solve or re-optimization.

Naturally this setup leads to some overhead both memory- and performance-wise when compared to an implementation that uses an integrated LP solver as is the case for most other especially commercial MILP solvers. This is also one reason why SCIP will never reach the same level of performance and efficiency.

On the other hand, we can use this versatile interface to investigate how different LP solvers impact the overall optimization process.

We are aware that SCIP is not the only option to compare different LP solvers within a branch-and-bound based MILP code. Cbc of the *COIN-OR Foundation*[2] is arguably the most popular alternative. Still, we do not include this solver framework into our experiments because of its considerable weaker performance on general MILP problems when compared to SCIP and especially when competing against commercial MILP solvers.

4.2. Root and Node LPs

We have already mentioned in Chapter 1 that the initial LP relaxation is one of the most time-consuming aspects of many MILP instances. On the other hand, there are also instances with easy-to-solve LP relaxations that still require a lot of time to find the optimal solution because of excessively large branch-and-bound trees.

From personal experience, we can attest that most practical MILPs fall into one of the three categories:

- Easy instances that can be solved quickly to optimality. The definition of *quick* is of course subject to the actual application at hand and may differ quite a lot from case to case.
- Instances that require a lot of time to process the root LP but can be solved to optimality with a small search tree or even just in the root node.
- Instances that do not pose any difficulties to the LP solver but make it very hard to find good feasible solutions or prove optimality because of a very large branch-and-bound tree. Often, such instances have a weak formulation, that is, the linear relaxation is not helping much in guiding the search towards optimality and therefore results in a large number of nodes.

Of course, there are also instances that have a very difficult LP relaxation *and* a very large branch-and-bound tree; due to this inconvenience, they are usually transferred back to the modeling stage to find alternative formulations or to split up the problem into smaller, more manageable pieces.

In Figure 4.2 we show the amount of time that is spent solving LPs during the MILP solving process. This scatter plot shows how different the LP time fractions are distributed among the instances and that they do not correlate with the overall solving

[2] http://www.coin-or.org/

time—independent of the LP solver used. The widely scattered fractions for those instances that are exhausting the time limit, visible as a vertical *wall* at the one hour mark, are a good indicator for that. Only with CLP as LP solver, a number of instances break this barrier. This is because of a less responsive termination behavior after the time limit has already been exceeded. For a more detailed breakdown of the different types of LPs that are solved, please refer to Figure 4.5.

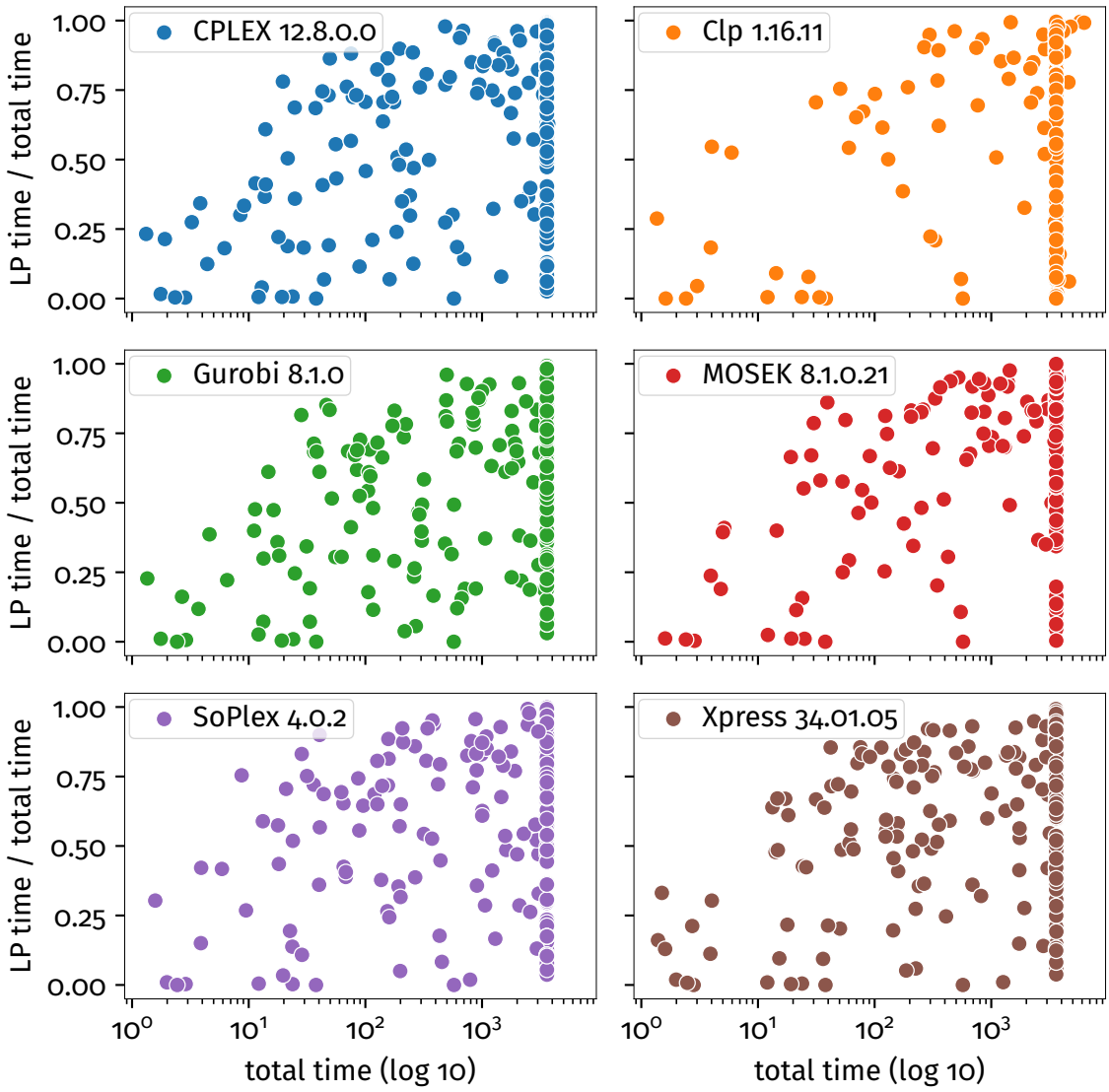


Figure 4.2.: Fraction of time spent solving LPs during MILP solving. We compare different LP solvers in SCIP on on the MIPLIB 2017 benchmark set, including instances that are not solved within one hour. We can see that with CLP, the time limit is not always respected.

Typically, we solve MILPs without any starting information, so the root LP needs to be solved from scratch. This is in stark contrast to practically every other LP solve that needs to be performed afterwards. Here, we can draw on useful data to warm- or hot-start the optimization of the relaxations.

Warm-starting, usually refers to providing the simplex algorithm with a feasible basis that is ideally already close to being optimal, that is, only a small number of iterations are necessary to reach it. This is the case during the cutting phase: whenever a new cut is added, the current basis remains dual feasible, but the dimension of the basis matrix is different and often a new LU factorization needs to be computed. See Section 4.5 for additional information.

Hot-starting on the other hand, denotes a similar process that leaves the factorization intact and allows an even faster re-optimization of the new LP. The simplest use-case is the actual branching process as detailed in Section 4.3: fixing a binary variable to either 0 or 1 or restricting the domain of an integer variable by including a disjunction to cut off the current node's LP solution. Since this is only further restricting the feasible domain of the active LP just like in the cutting step, the dual feasibility of the basis is preserved. Additionally, the basis matrix is not modified, the factorization remains also valid.

Here, we silently assume that the simplex method is used to solve those node LPs. While this is the most practical approach due to the mentioned warm- and hot-starting effects, there are also instances that are solved significantly faster using the barrier method and even the node LPs profit from using this method over the simplex algorithm. Berthold, Perregaard, and Mészáros (2018) give an overview of the other lesser known applications for an interior point or barrier solver when dealing with an MILP, including primal heuristics and presolving techniques that would not be possible with the simplex method.

Unfortunately, if both approaches are viable, it is extremely difficult to predict which method will be better suited for a specific LP or MILP instance. Usually, some heuristic guess is made based on the dimensions and sparsity of the problem matrix—larger problems generally being solved faster with the barrier. In case enough computing resources are available in form of parallel threads, most solvers actually run several methods concurrently when solving the root LP or a similar problem without useful warm-starting information to kick-start the simplex. This inability to reliably predict a certain method's performance even extends into whether to use the primal or dual variant of the simplex, so these are also often run in parallel. Typically, commercial MILP solvers use the available computing threads for other tasks like parallel node processing or running heuristics. So this concurrent approach is only feasible for the root node when the progress is largely dependent on the initial LP solution and not enough other work can be parallelized, yet.

Figure 4.3 shows how using either the simplex or barrier method affects the fractionality, iterations, and solving time of the root LPs of the MIPLIB 2017 benchmark set. We have been performing this experiment with MOSEK as LP solver because the other applicable solvers (CPLEX, Gurobi, and Xpress) have been reporting too many spurious iteration counts in SCIP. The qualitative results for the time and fractional-

ity comparisons have been similar, though. Please refer to Table B.5 for the detailed results.

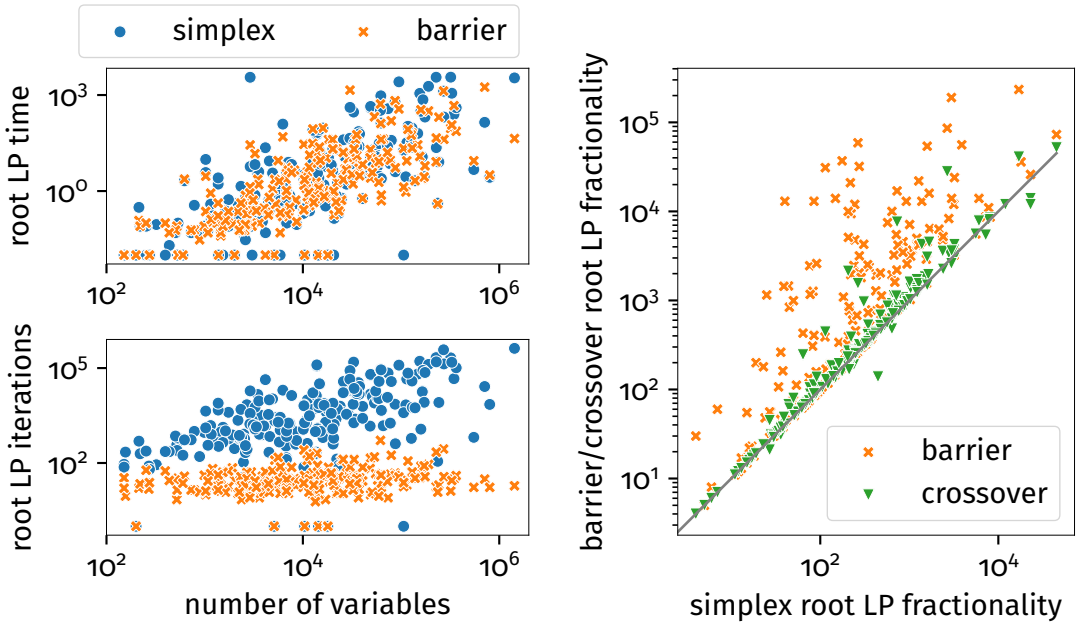


Figure 4.3.: Comparison between different LP solving methods in SCIP 6.0.2 using MOSEK 8.1.0.21 as LP solver (single-threaded). Barrier iteration counts are largely independent of the problem sizes, whereas the simplex iterations increase with growing dimensions. The respective solving times remain very comparable across the MIPLIB 2017 benchmark set. The right plot shows the difference in root LP fractionality when using either the simplex or the barrier method. Performing the additional crossover step reduces the fractionality dramatically.

We have also been running various experiments with the different LP solving methods in SCIP but decided that the results were too unreliable and unstable to make further conclusions. SCIP—at least in version 6.0.2—should mainly be used with the simplex method to solve the occurring LP relaxations.

In Figure 4.4 we can see that with CLP there is a large number of instances whose solving time is spent entirely or almost entirely by the LP solver. With SOPLEX and MOSEK, there is still a strong tendency towards the right-most third of the histogram, meaning that a significant portion of the instances' solving time is spent by the respective LP solver. CPLEX, GUROBI, and to a lesser extent also XPRESS display a more balanced time distribution and consequently, allow more time to be spent in SCIP itself instead of computing an LP solution.

The bad performance of CLP is likely due to the extremely high number of unstable resolves, as depicted in Figure 4.5. This instability detection is based on violations

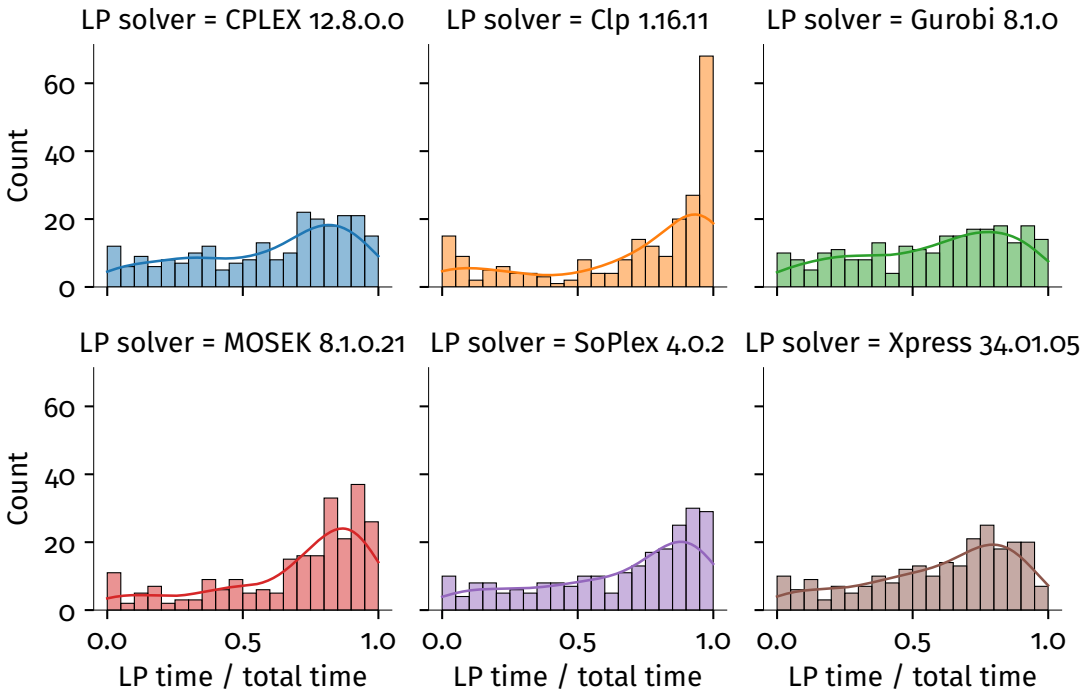


Figure 4.4.: Histogram depicting the fraction of time spent by the LP solver during MILP optimization. The bars depict how often a certain fraction of the total solving time is consumed by the respective LP solver. This also includes instances from the MIPLIB 2017 benchmark set that exhausted the one hour time limit.

in the LP solutions. Such solutions are then rejected by SCIP and another try with different parameters or tolerances is initiated. If necessary, the warm-start basis is even thrown away in hopes of avoiding numerical difficulties when performing a fresh start. This also explains the poor performance of CLP in SCIP and we should take those results with a grain of salt. Interestingly, with XPRESS there is not a single LP that is rejected by SCIP, indicating a very stable implementation of the solver.

4.3. Branching Rules

Branching Rules are the different strategies of choosing the variable to branch on in the branch-and-bound process. Usually, we select one of the integer or binary variables, say x_i, that has a fractional value $\check{x}_i$ in the LP relaxation's solution of the current node. Forcing this variable x_i to be either $\leq \lfloor \check{x}_i \rfloor$ (the *down branch*) or $\geq \lceil \check{x}_i \rceil$ (the *up branch*) invalidates the current LP solution in both child nodes and reduces the search space in the new subproblems. Figure 4.6 provides a visual aid. There are many different strategies for selecting the next variable, often strongly affecting

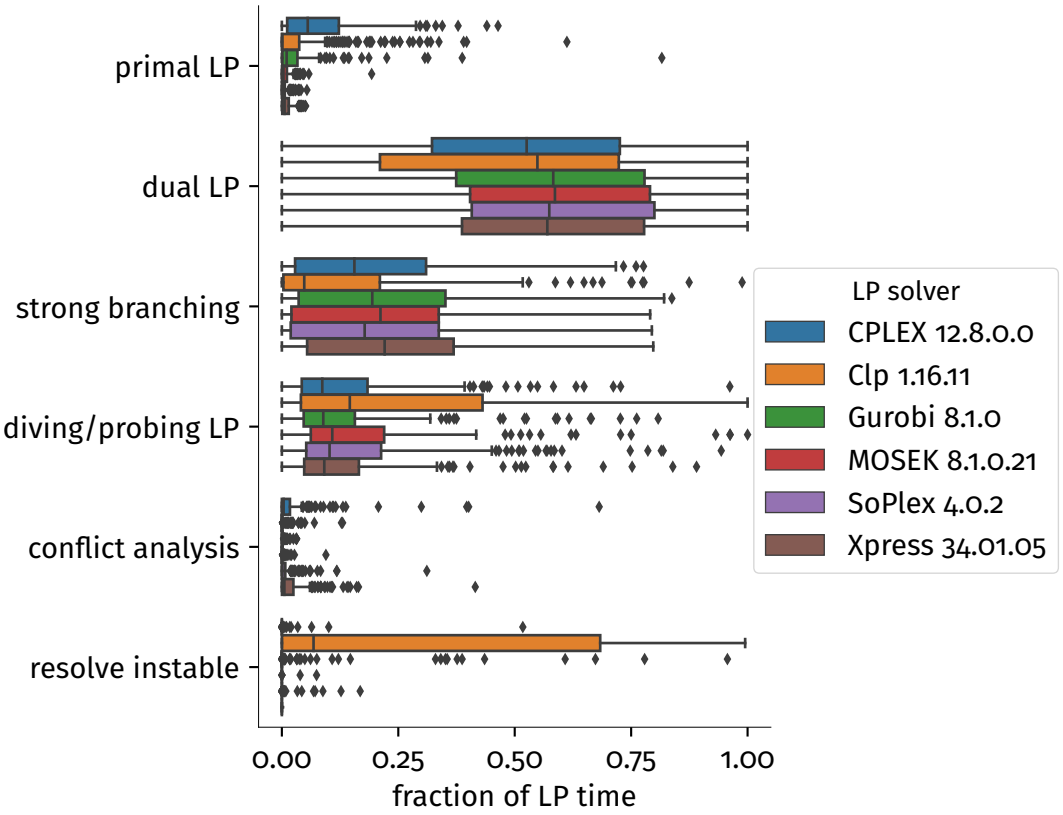


Figure 4.5.: Distribution of the different types of LPs that need to be solved during an MILP optimization on MIPLIB 2017 benchmark instances with SCIP in one hour. Only optimally solved instances are included. Noteworthy is the very high number of unstable LP solves with CLP and the relatively large number of primal LP solves with CPLEX. Gray markers represent outliers with respect to the interquartile range.

the overall number of nodes visited in the tree and the total time to optimality. The various branching rules can also involve very time-intensive computations to reduce the number of nodes so we need to be careful when comparing their efficiency as explained by Gamrath and Schubert (2018). As an example, imagine a branching rule that solves multiple LPs in an internal subroutine, essentially *hiding* these processed nodes from the global node counter.

In the following paragraphs, we give a short overview of the most popular and well-known branching rules that are available in typical branch-and-bound solvers for MILPs.

Most infeasible branching This branching rule is a very simple strategy that chooses the variable that violates its bounds the most to branch on, that is, the variable that

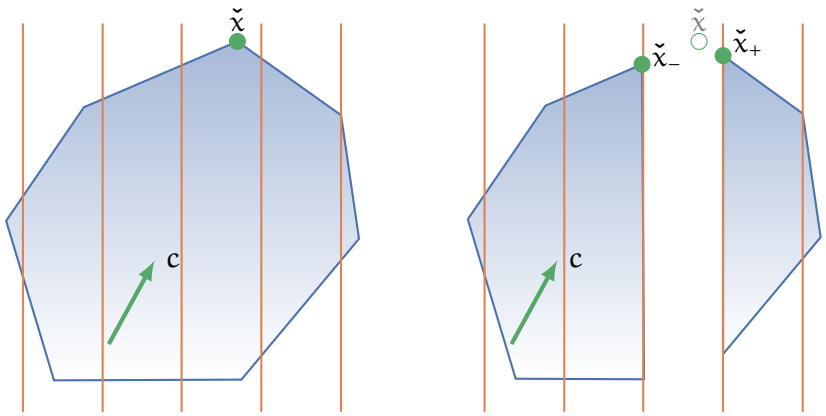


Figure 4.6.: Visualization of the branching procedure with the MILP imposed by intersection of the blue region and the orange integrality conditions. LP solution $\check{x}$ on the left is removed from the two subproblems on the right that include restrictions in horizontal direction: $x_i \leq \lfloor \check{x}_i \rfloor$ and $x_i \geq \lceil \check{x}_i \rceil$, respectively. The resulting LP solutions $\check{x}_-$ and $\check{x}_+$ are integer feasible and $\check{x}_+$is the optimal solution.

is closest to 0.5 in the LP relaxation's solution. This sounds like a good idea on paper because it is very easy to compute and one might think that the less violated variables may become feasible on their own when forcing the other ones onto one of their bounds. Additionally, this should have a larger impact on the LP relaxations in the new child nodes. In practice, though, Achterberg, Koch, and Martin (2004) showed that this branching strategy turns out to be as good as *random branching*, showing once more how intuition can fail us in mathematical programming.

Pseudo-cost branching This is a more involved branching rule that was first described by Bénichou et al. (1971). It basically tracks the objective gain of every up and down branch each variable has already taken part of. This provides a history of efficacy for each variable that is used to compute a score to determine the next variable to branch on. There are multiple variants of how exactly to compute the score and weigh the past results and we refer to Achterberg, Koch, and Martin (2004) for further details.

Strong branching This branching scheme is very expensive but often results in a significantly reduced tree size. The main idea here is to solve multiple LPs corresponding to some or even all available branching candidates. This provides a good overview of the actual effect of branching on those variables and we can choose the variable based on some score that is determined by the solution of the respective LP solutions. To save time, most implementations do not inspect all available fractional candidates and also limit the number of simplex iterations in each LP solve—the re-

sulting scores are usually still very informative. To push this to an extreme, there are even approaches that merely run the first simplex iteration with the newly modified variable until the ratio test is done. The resulting step length is then used to get some idea about the corresponding pivot or basis change without actually executing it. That way, practically all data structures are left unchanged and many variables can be probed without investing too much time.

Running the branching rule without those limits is commonly referred to as *Full Strong Branching*. For more information, we refer to the original authors Applegate et al. (1995) and to Gamrath (2014) for an extension that also includes domain propagation within the branching candidates' LP solves.

One trait that is present in all strong branching implementations is their dependency on quickly available LP solutions. In SCIP this is realized by backing up the current basis information before starting the strong branching process so the dual simplex can start from that base and can be reset just as quickly to the initial state right before strong branching.

Inference branching Especially when dealing with a feasibility problem, that is, a model instance with a constant objective function, we cannot rely on improvements or changes in the dual values of the branching candidates. Instead, we can use a technique from the field of Constraint Programming and satisfiability problems: *inference branching* tracks the amount of deductions and implications that a branching candidate entails. Hence, branching on a variable with a high inference rate is supposed to achieve a greater reduction of the remaining solution space to explore.

Hybrid branching To avoid the expensive cost of *strong branching* and to get around the missing historic information of *pseudo-cost branching* in the early stages of the tree search, we can also combine both ideas and initialize the pseudo-cost values by running *strong branching* for the first few levels of the tree search. Further down in the tree, *pseudo-cost branching* is used to speed up the branching decisions. In addition to that, we can also include the inference score in case the information gained from the objective is not meaningful enough.

Reliability branching This branching rule is a further extension of *hybrid branching* and uses *strong branching* not only to initialize the pseudo-costs but also whenever these pseudo-costs are not *reliable*, that is, when not enough previous branching data exists to make an informed decision. Achterberg, Koch, and Martin (2004) show how these last four branching rules are related to each other and provide extensive computational data that shows how modifying the parameters of these rules impact the solving time as well as the tree size. A version of *reliability branching* is still the default branching rule in SCIP.

Neural branching This is a very recent development extending the classical branching schemes. In Nair et al. (2020), the authors show how the branching decision can be improved significantly by collecting branching scores for possible candidates in a full strong branching manner. As explained above, this requires solving a lot of LPs—one for each candidate—so the benefit of better branching decisions will be overshadowed by the excessive computational overhead of the LP solves. Instead, the authors opted for approximated LP solutions and employed a GPU-accelerated ADMM method (*Alternating Direction Method of Multipliers*, Boyd et al., 2011) to solve these LPs in parallel to generate the learning data. This *Neural Branching* approach works because determining a good branching candidate does not rely on exact LP solutions as has been already discovered by Achterberg, Koch, and Martin (2004). So, machine learning appears to be a good fit to estimate the branching scores and the paper provides computational evidence.

In summary, we see that (most) branching rules heavily rely on LP solutions to determine the next variable to branch on. While we can expect the smallest tree using the *full strong branching* approach, the additional cost of computing all these LP solution need to be accounted for to still achieve an overall speedup. This balancing act is what makes branching rules an interesting field of MILP solving that is still in the focus of active research and development and further advances can be expected in the years to come.

Dey et al. (2021) provide a recent analysis regarding the performance of (full) strong branching for a variety of combinatorial problem classes using randomly generated data. One of the most interesting findings is that full strong branching keeps the search tree within at most twice the number of nodes of the optimal, that is, the smallest search tree for any of the analyzed instances. The authors also note that the underlying LP solver plays an important part, especially regarding whether an integer or fractional LP solution is returned (given that there is an integer LP solution at the current node). One important aspect that is largely ignored in this paper is the amount of work necessary to actually carry out a full strong branching approach throughout the entire MILP solving process—they focus on the size of the tree instead of the solving time.

4.4. Node Selection

Every branching decision typically creates two new subproblems. Naturally, the question arises which node to process next. In addition to the two child nodes at the current node, we may also decide to switch to another open node somewhere else in the tree. There are different strategies in choosing the next node to continue the search and they generally need to satisfy the following two goals:

1. Improve the primal bound by quickly finding new incumbent solutions.

2. Improve the dual bound by further exploring nodes with a very small LP objective value.

The first strategy tends to favor deep dives into the tree because we expect to find integer feasible solutions easier after many branching decisions and possibly variable fixings have been already performed. The second strategy on the other hand will explore the tree in a breadth-first manner to work on those nodes that have a very small dual bound and are typically found close to the root node where less restrictions have been imposed on the LP.

Intuitively, we want to combine both approaches to improve the dual bound while also finding new primal solutions as soon as possible. SCIP provides several different node selection rules and uses a method called *best estimate* that assesses how likely new solutions will be found in the corresponding subtree by calculating a measure based on pseudo-cost values and the current LP objective value. We refer to Achterberg (2007) for detailed descriptions.

The chosen node selection variant has a smaller impact on the MILP solving performance than other parameters, like for example the branching rule, and just using the straightforward *best bound* approach is still a viable choice.

Note that in Chapter 6, we are using *breadth-first search* as node selection rule to collect more balanced statistics across the entire tree. Otherwise, we might have to deal with deeper plunges down the tree without visiting other nodes of the same depth.

4.5. Cutting Planes

The idea of *cutting* in MILP solving is based on the idea of separating the current LP solution from the convex hull of the integer feasible points of the MILP as visualized in Figure 4.7: Let $\check{x}$ be a solution to the LP relaxation of MILP (1.1). Then, a valid cutting plane can be defined as a pair $(\alpha, \alpha_0) \in \mathbb{R}^{n+1}$ such that for all $x = (x_1, \ldots, x_n)$ in the convex hull of all integer feasible points of MILP (1.1) the following hold.

$$\sum_{j=1}^{n} \alpha_j x_j \leq \alpha_0 \text{ and } \sum_{j=1}^{n} \alpha_j \check{x}_j > \alpha_0.$$

Finding such a pair (α, α_0) is also known as the *separation problem* for $\check{x}$, as the corresponding hyperplane separates the point $\check{x}$ from the convex hull.

Typically, this requires a valid LP solution in the first place to have a point to separate from the convex hull. Additionally, many cutting plane methods also directly use the current simplex basis to compute valid inequalities that cut off the corresponding LP solution.

Such cuts are known as *Gomory cuts* or *Gomory fractional cuts*.

Assuming an IP problem $\{\min c^\top x \mid Ax = b, x \in \mathbb{N}\}$ and applying the fractionality operator $f(a) := a - \lfloor a \rfloor$, we can use the following transformation to generate a valid

Gomory inequality from the ith row:

$$\begin{aligned} \sum_{j=1}^{n} a_{ij} x_j &= b_i \\ \sum_{j=1}^{n} f(a_{ij}) x_j + \lfloor a_{ij} \rfloor x_j &= f(b_i) + \lfloor b_i \rfloor \\ \sum_{j=1}^{n} \underbrace{f(a_{ij}) x_j}_{\geq 0} &= \underbrace{f(b_i)}_{<1} + \lfloor b_i \rfloor - \sum_{j=1}^{n} \lfloor a_{ij} \rfloor x_j \\ \sum_{j=1}^{n} f(a_{ij}) x_j &\geq f(b_i) \end{aligned} \tag{4.1}$$

The last inequality follows from $\lfloor b_i \rfloor - \sum_{j=1}^{n} \lfloor a_{ij} \rfloor x_j \geq 0$. This approach can not only be applied to all rows but to all weighted combinations of rows, especially using any row of the inverse of the current optimal basis matrix B as weights:

$$\begin{aligned} B^{-1} A x &= B^{-1} b \\ x_{\mathcal{B}} + B^{-1} A_{\mathcal{N}} x_{\mathcal{N}} &= B^{-1} b \end{aligned}$$

Hence, for an optimal solution $\check{x}$ to any LP relaxation with fractional integer variable x_i, we can construct a Gomory cut based on the corresponding basic row:

$$\sum_{j \in \mathcal{N}} f(B^{-1} A_{\mathcal{N}})_{i.} x_j \geq f(B^{-1} b_i) = f(\check{x}_i)$$

With $0 < f(\check{x}_i) < 1$ and the non-basic variables $\check{x}_{\mathcal{N}} = 0$, we can see that the LP solution $\check{x}$ violates the cut since the left-hand side of the inequality is 0.

These cuts are also called *basis-dependent* and a similar approach is also possible for MILPs. The resulting inequalities are then known as *Gomory mixed-integer cuts*. Consequently, SCIP's LP interface needs to provide functionality to get these vectors based on the current optimal basis and every newly generated cutting plane requires the solution of one linear system of equations and the corresponding multiplication with the constraint matrix. This observation also implies that an accurate solution to this system of equations is necessary to avoid computing incorrect cutting planes that may even cut off integer feasible solutions.

Interestingly, we can also use sub-optimal bases to generate such cutting planes as shown by Conforti, Cornuéjols, and Zambelli (2014). During the optimization procedure of the dual simplex algorithm, we compute one row of the form $\hat{a}_p^\top = e_p^\top B^{-1} A_{\mathcal{N}}$ in every single iteration so it just needs to be stored for later use. After completing the simplex optimization, we can then generate Gomory cuts from those rows without having to compute additional weights. There are even cases where such inequalities from dual feasible bases lead to a tighter LP relaxation than with regular Gomory cuts generated from the optimal basis.

Unfortunately, we have not been able to develop this idea beyond a prototype implementation and could not integrate this approach into the full cut generation and

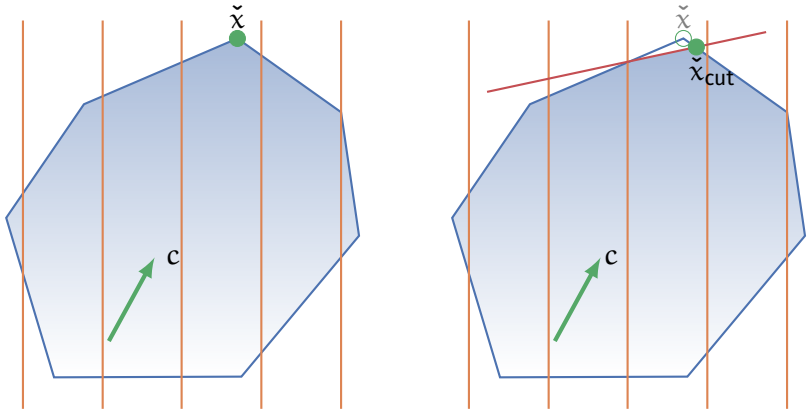


Figure 4.7.: Visualization of the cutting procedure with the MILP imposed by intersection of the blue region and the orange integrality conditions. LP solution $\check{x}$ on the left is cut off by adding a new valid inequality (red) to better approach the convex hull of feasible points and thereby provide a new relaxation solution $\check{x}_{cut}$.

filtering system of SCIP. It was unclear how to collect only the interesting data from the LP solver without incurring a prohibitively high computational cost.

There are also combinatorial or other structure-dependent cutting plane methods that do not rely on the simplex basis when generating new valid inequalities. The *subtour elimination constraints* for solving the traveling salesman problem (TSP) are a popular example. These cuts invalidate the current solution of the relaxed TSP by explicitly forbidding a specific subtour in that solution to force the solver in the subsequent iterations to find a different solution. As there are exponentially many possible subtours, it is not feasible to include them into the formulation directly. Maher, Miltenberger, et al. (2016) demonstrate how such a TSP-solving algorithm can be implemented using PySCIPOpt.

Interestingly, cutting plane methods can be used exclusively to solve MILPs as discussed by Zanette, Fischetti, and Balas (2011). Manfred Padberg (see Grötschel, 2004 for a portrait of his work) is even credited with stating *"branching is a sign of mathematical defeat"* (Koch, Martin, and Pfetsch, 2013), essentially elevating cutting over branching. But cutting planes work best when used in combination with the aforementioned branch-and-bound scheme, because they complement each other, typically providing better performance than each approach individually. The details of this synergy are well explained by Basu et al. (2022).

The application of several cutting planes in one round instead of separating them one-by-one was a ground-breaking finding concerning the practical impact of cuts in general.

One obvious disadvantage of adding cutting planes to a problem is that the size of rows increases, making each new LP relaxation slightly more difficult to solve. Furthermore, adding lots of cutting planes also introduces the risk of getting more numerical

instabilities or even losing integer feasible solutions to inaccurate cutting.

SCIP implements various techniques to filter cutting planes before adding them to the LP. For example, sparse cuts, that is, cuts with few non-zero coefficients are preferred. Then, subsequent cutting planes are required to not be too parallel to the already added rows to avoid numerical issues like singular or near-singular basis matrices. Another filtering step limits the allowed range of cut coefficients.

Finally, SCIP also implements a *row aging* mechanism that tracks for how long a certain cutting plane has been included in the LP without being active or tight at the optimal basis. As soon as a certain age limit has been reached, the corresponding cut is removed from the LP again as it appears to be redundant and does not help tighten the LP relaxation further.

Please note that this is just a rough and incomplete description of the cutting and separation techniques used in SCIP and meant to provide a minimal understanding of the general procedure.

4.6. Conflict Analysis

Conflict analysis is the tool to learn from infeasible nodes in the branch-and-bound tree. Infeasibility can result from a branching decision that led to an infeasible LP relaxation in one of the child nodes or from a not improving one with respect to the current best incumbent—a so-called objective cutoff. This objective cutoff can be thought of as additional constraint, hence rendering the LP relaxation infeasible once more.

These infeasible LPs generate a proof of infeasibility in the dual space. This *dual ray* can be used to derive useful information on the variable bounds to further shrink the feasible domain and reduce the solution space.

There are multiple ways to learn from infeasibility and a good introduction on the topic is presented in Witzig, Berthold, and Heinz, 2019.

Arguably the most involved technique is to add so-called *conflict constraints* to the LP relaxation. These are additional constraints that encode a specific root cause for infeasibility and learning a good set of conflicts often provides an effective way of improving the quality of the LP relaxations—similar to how cutting planes work.

From the experimental data shown in Figure 4.5, we can see that the time spent in conflict analysis LPs is negligible for almost all instances in our test set.

4.7. Primal Heuristics

One of the key components that allow MILP solvers to find optimal or at least very good solutions to $\mathcal{NP}$-hard problem instances is the clever use of heuristics. We established in Algorithm 1.3 that primal solutions can be found by branching on a variable and discovering that the resulting LP relaxation has an integer feasible solution. In reality, many solutions are found by heuristics to push the primal bound down in a shorter amount of time. The most helpful ingredients in that process are the LP

solutions that can provide a starting point or target for the heuristic. We typically distinguish heuristics into two parts: those that require an integer feasible initial solution and those that do not. Even the former group of improvement heuristics draw on available LP solutions to guide their search.

We are not discussing primal heuristics in detail and rather refer to Berthold (2014) for a comprehensive overview of the topic. We feel that we cannot completely omit these techniques as they usually make up a significant portion of any competitive MILP solver's code base. This is also visible in the chart of SCIP's components in Figure 4.1.

4.8. Visualization of MILP Search Trees

Despite the frequent application of MILP models for real world problems, visualizing the actual solving procedure is not straight forward. This can be useful for understanding the process and detecting possible bottlenecks or peculiarities of the solver or of a certain problem instance. A common visualization technique for depicting the solving progress is to plot the advancement in dual and primal solution quality over time. For a minimization problem, the dual solution values are going to be continuously increasing while the primal ones often behave in a piecewise constant fashion, decreasing towards the optimal solution value as depicted in Figure 4.8:

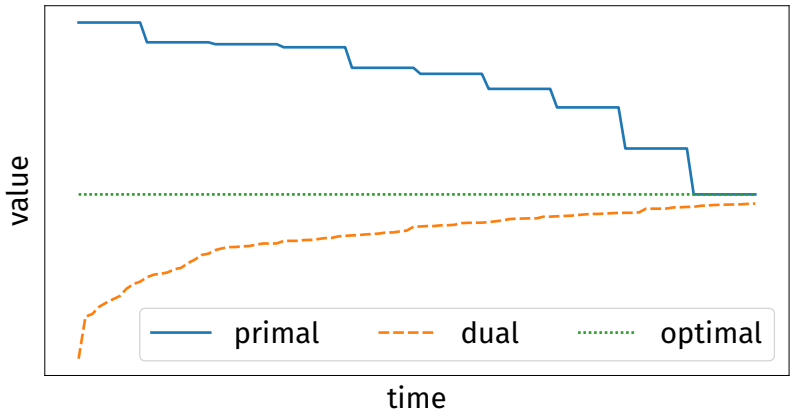


Figure 4.8.: Visualization of the MILP solving progress.

Such charts hide the complexity present in the form of the actual branch-and-bound tree that is constructed and traversed during the optimization.

A natural visualization of the branch-and-bound tree is to draw this exact tree, that is, every branching decision leads to two new child nodes until all open nodes are processed. This is a very detailed way to present the solution process and is well suited to show among other things how balanced a search tree is.

Still, there is room for improvement as the distance between the nodes does not represent the actual distance with respect to the model data. One of the defining

aspects of a node is its LP relaxation and the corresponding LP solution. This is the foundation for a visualization technique that tries to preserve the spatial dimension when drawing the tree. Every LP solution has to be mapped to a two-dimensional point, while maintaining distance in the original space as well as possible. Such a projection can be achieved in several ways. In the following, we use the two techniques *multi-dimensional scaling* (MDS) and *t-distributed stochastic neighborhood embedding* (t-SNE) to create a new perspective on MILP search trees. We refer to Kruskal (1964) and Borg and Groenen (2005) for more information on MDS and to Van der Maaten and Hinton (2008) for details on the t-SNE method.

In Figure 4.9 we demonstrate how this transformation works by projecting a simple two-dimensional point cloud onto a line—a one-dimensional space. While it is impossible even for such a simple and low-dimensional example to keep all distances intact, the overall structure—like the two clusters—can be preserved by the transformation.

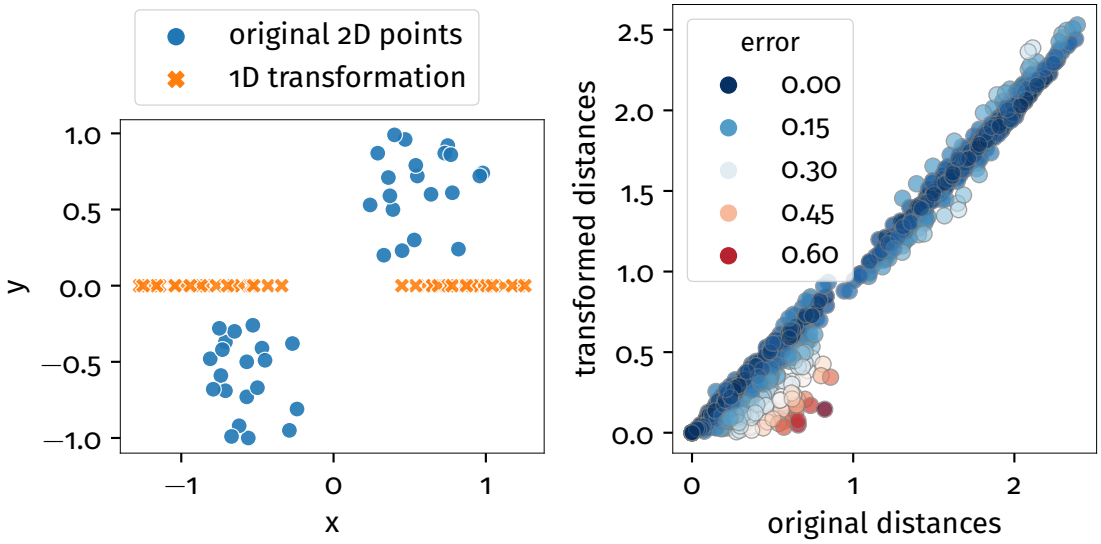


Figure 4.9.: MDS transformation of a cloud of random points from the two-dimensional plane onto a one-dimensional space. The right image shows the introduced error between pairwise distances of points in the original and the transformed space.

To get a feeling for the quality of distance preservation, a so-called *Shepard plot* (Shepard, 1962) can be used. This is a simple scatter plot comparing the original with the projected distances for every single pair of points—in our case the individual node LP solutions.

In Figure 4.10 we see a visualization of the MDS-transformed LP solutions of the branch-and-cut tree generated while solving the MIPLIB3 instance `lseu`. Such three-dimensional plots have the tendency to be quite hard to read in print, so we encourage

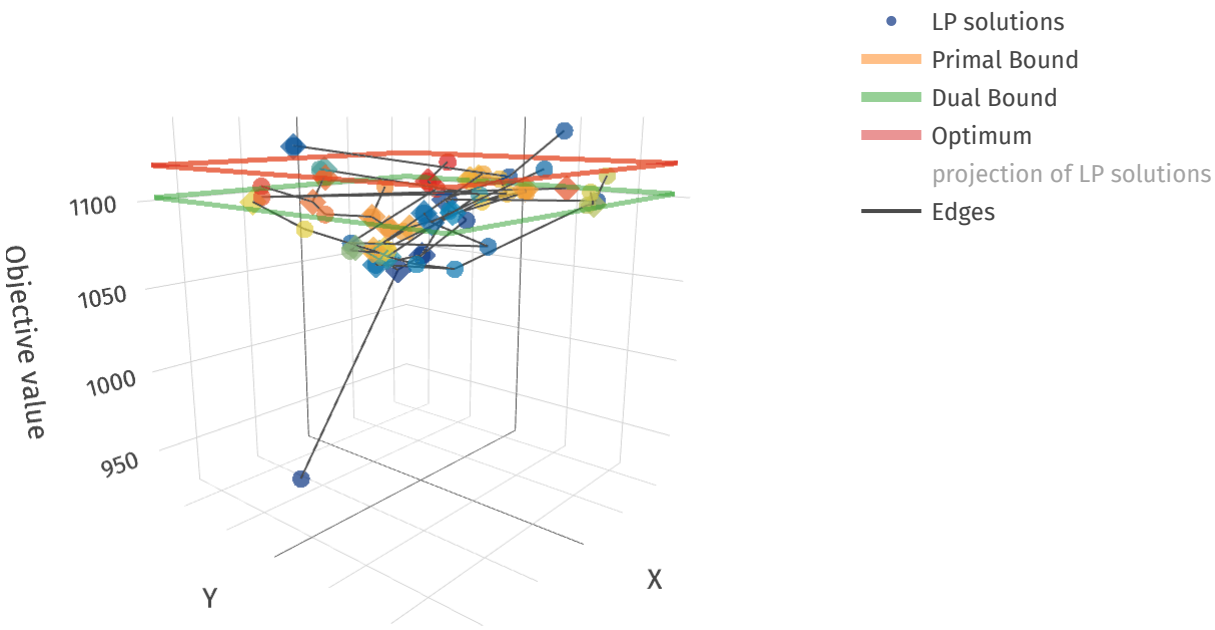


Figure 4.10.: TreeD visualization of MIPLIB3 instance `lseu` (89 binary variables, 28 constraints)

the reader to run their own experiments with the TreeD[3] code and inspect the interactive visualizations themselves. The module can be installed via the standard Python package repository PyPI using the following command:

```
$> pip install treed
```

The main dependencies are PySCIPOpt and hence also the SCIP Optimization Suite to provide all the LP data during the solving process. The transformation of LP solutions is carried out with the help of the Python module `Scikit-learn` (Pedregosa et al., 2011). In Chapter 6, we show a different use of TreeD that does not produce instance-wise visualizations but collects various LP solving data in a convenient form as `pandas DataFrame` for subsequent analysis.

The corresponding Shepard plot is shown in Figure 4.11 together with a histogram depicting the distribution of absolute errors in the transformed distances. We can see that despite the stronger reduction in dimensionality, most distances are still preserved reasonably well. Note that instance `lseu` has 89 binary variables and is solved in 61 nodes.

Apart from the data exploration functionality of TreeD, we believe that the generated visualizations also provide a certain artistic value to the field of MILP solving.

We refrain from listing the exact tabular data that was used to generate these plots because of the incurred randomization during the MDS transformation performed in TreeD and the very limited additional value.

[3] `https://github.com/mattmilten/TreeD`

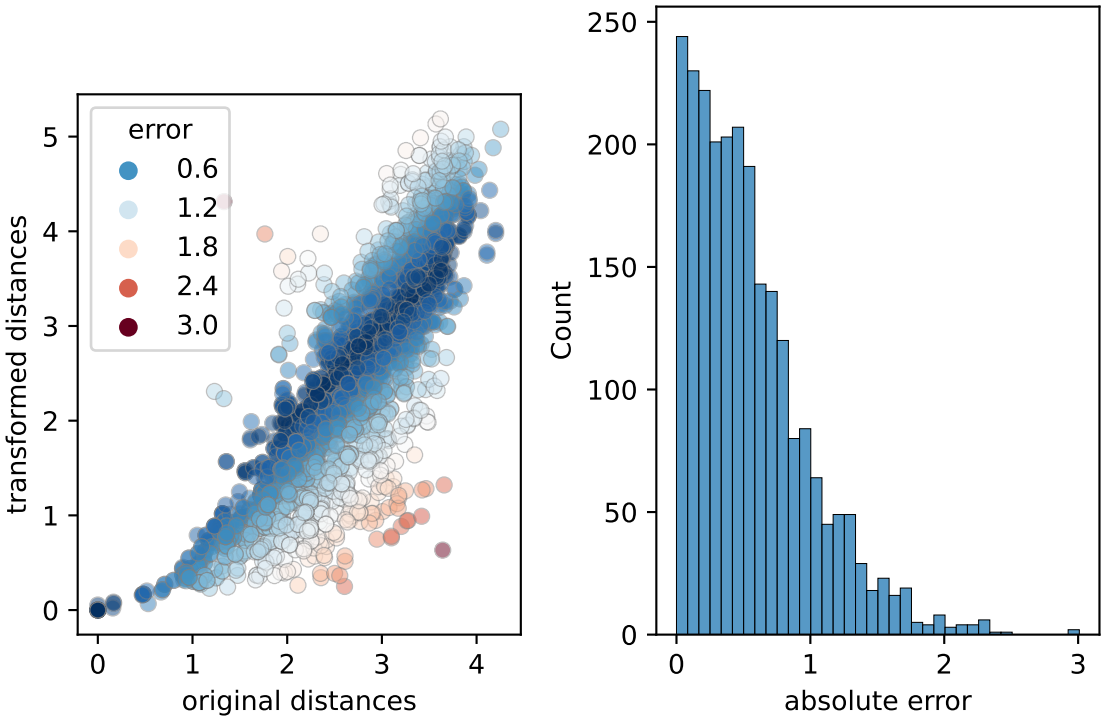


Figure 4.11.: Shepard plot of original and transformed pairwise distances of all encountered node LP solutions after applying the MDS transformation as seen in Figure 4.10. The absolute errors between the distances are color-coded as seen in the legend. The histogram on the right-hand side provides another view on the same data, revealing that most transformation errors are well below 1.

settings	iterations	nodes	time	timeout	optimal
default	126 613.7	1686.6	124.9	51.75	123.00
no bound flips	134 477.1	1668.2	122.4	51.25	120.75
no persistent scaling	147 041.8	1792.2	137.1	55.75	114.50
no sparse pricing	126 936.9	1685.8	126.0	51.00	122.50
no stable sum	131 192.6	1736.3	125.3	53.00	120.00

Table 4.2.: Impact of selected SoPlex features on the MIPLIB 2017 benchmark instances. The numbers for iterations, nodes, and time refer to those 96 instances that have been solved to optimality by all solver variants. Columns timeout and optimal show the average number of instances that exceeded the time limit of one hour or could be solved to optimality, respectively. Experiments have been repeated with four different random seeds.

4.9. Computational Study

Now that we have established an understanding of the fundamental influence of LP solutions and their solution values throughout the solving process, we want to compare and investigate how changing the specific LP solver impacts the performance of the MILP solver and alters the path to optimality.

First, to see how SCIP performs when deactivating certain features in SoPlex, we use the MIPLIB 2017 benchmark of 240 instances. Every experiment has been performed with four different random seeds to get reliable numbers. The results for SCIP 6.0.2 are shown in Table 4.2. Just as expected from the LP experiments in Chapter 3, we see that without the bound flipping ratio test, more iterations are necessary to solve the instances and we even fail to solve two to three instances within the time limit of one hour. Disabling sparse pricing has barely any impact on SCIP's performance, while persistent scaling clearly helps to solve more instances in less time, fewer nodes, and considerably fewer iterations. The stable sum implementation in SoPlex also has a measurable effect by reducing the number of nodes and simplex iterations and helps to increase the number of solved instances. Note that we are using the shifted geometric mean to compute the average values with a shift of 100 for iterations and nodes and a shift of 1 for the solving time. Timeouts and instances solved to optimality are compared using the arithmetic mean over the four seeded runs.

There is no reasonable way of swapping out the used LP solver without impacting the behavior of the MILP solver. Even if we kept a single, fixed search tree with all the associated node relaxations and starting bases, we could not get a sound comparison of LP solving performance because we would neglect hot-starting effects: restarting the LP solver from just the current basis without a ready-to-use factorization and other internal data structures. This is the reason for not pursuing this approach and rather comparing LP solvers within their respective SCIP LP interfaces. This is a more realistic

take and enables our results to be reproduced without excessive code modifications.

Maher, Ralphs, and Shinano (2019) suggest an interesting visualization technique to compare performance across problem instances that are solved within the given time limit as well as instances that can only be solved to some non-zero optimality gap. Usually, these two sets of instances are separated or the analysis only focuses on those instances that could be solved to optimality by every solver or setting.

This *cumulative* performance comparison has first been proposed by Dinh, Fukasawa, and Luedtke (2018). In Figure 4.12, we see an example of such a plot comparing the performance of different LP solvers in SCIP on the MIPLIB 2017 benchmark.

We also prefer this combined way of visualizing benchmark results over the traditional performance profiles because the results are presented in a very intuitive way without neglecting suboptimal results. Furthermore, performance profiles can be misleading and are generally harder to interpret as demonstrated by Gould and Scott (2016).

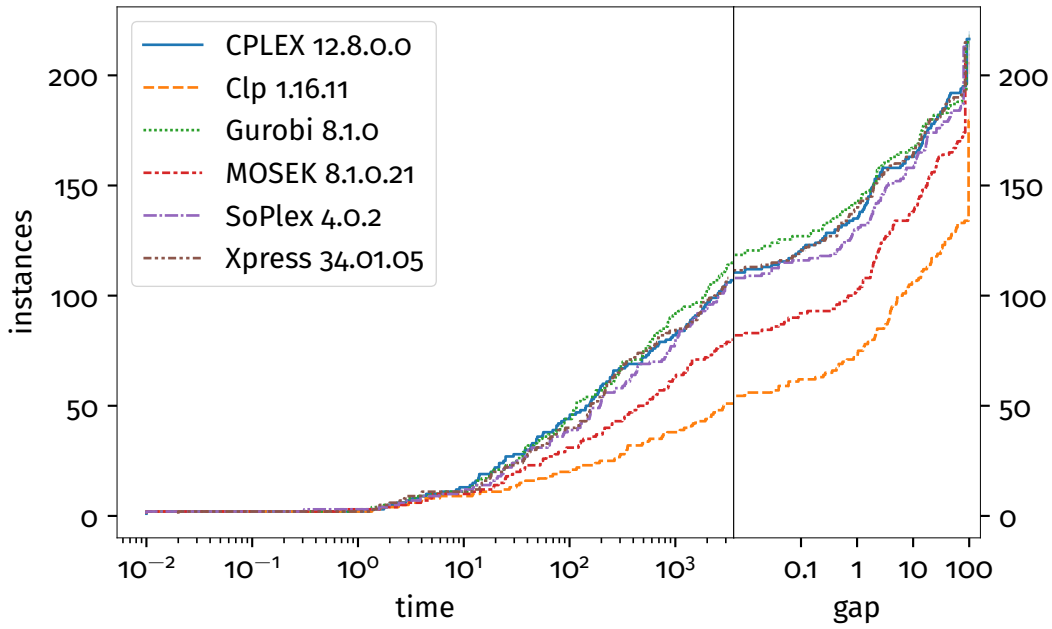


Figure 4.12.: Overall performance comparison of different LP solvers used in SCIP 6.0.2 with default settings solving the MIPLIB 2017 benchmark test set. The horizontal axis is split up into two parts to depict the number of solved instances within the time limit of one hour and the final gap of the remaining instances.

We can see a clear trend of CLP and MOSEK being outperformed by all other solvers, while GUROBI solves the most instances within the time limit and SOPLEX being a close runner-up to the three commercial solvers CPLEX, GUROBI, and XPRESS.

On the other hand, when comparing the pure LP performance of those solvers, we

	shifted geometric mean solving time			
	1365 instances, all optimal		128 instances, min 5 sec	
SCIP settings	default	presoloff	default	presoloff
LP solver				
Gurobi 8.1.0	0.980	0.800	49.100	39.372
CPLEX 12.8.0.0	1.026	0.866	49.814	41.954
Xpress 33.01.09	1.155	1.111	68.037	70.770
MOSEK 8.1.0.21	1.327	1.172	96.868	85.126
Clp 1.16.11	1.411	1.460	101.645	112.102
SoPlex 4.0.2	1.564	1.691	123.616	150.762

Table 4.3.: Comparison of different LP solvers in SCIP 6.0.2 on the `allLP` test set for two settings: default and without SCIP's presolving, each with two random seeds. Times are computed as shifted geometric means (shift of one second) over two sets of instances: those solved to optimality and those that need at least five seconds to be solved.

can observe a different ranking as shown in Table 4.3

These results are also aligned with the LP and MILP benchmark results of Hans Mittelmann[4] that we will discuss further in Appendix A. Note that we used a modified SCIP code to ignore all integrality information and treat all models as LPs. Please refer to Table B.1 for the detailed results.

In the following, we want to investigate why SoPlex performs so well when solving MILPs, despite its significantly worse pure LP performance.

Iteration count. When comparing the iteration counts of SoPlex with those of the other solvers, we often observe significantly higher numbers. Take for example problem instance `10teams`: SCIP with SoPlex needs between 7019 and 8033 iterations while all other LP interfaces show an iteration count between 911 and 1730.

Further experiments reveal that—at least for this specific model instance—this is due to the pricing method being used. By default, SCIP uses the LP solver's preferred pricing rule and for most solvers that is a variant of steepest edge pricing. SoPlex, on the other hand, uses its *automatic pricing* scheme described in Chapter 3 that starts off with devex and then switches over to steepest edge after 10 000 iterations. When using the steepest edge pricing rule to solve this instance, SoPlex takes between 1260 and 1550 iterations to find the optimal LP solution—so this is well within the range of the other solvers.

Naturally, the question arises, why this is not the default pricing scheme and computing the shifted geometric mean over all solving times and seeds exposes the rea-

[4] `http://plato.asu.edu/bench.html`

son: For the test set of all instances solved to optimality the time increases to 1.814 whereas for the harder instances that take at least five seconds for any solver we get an aggregated result of 117.815 seconds. So, while on the second set, there is a slight speed-up, we can observe a slow-down on the first benchmark. This experiment shows that it is not sufficient to judge the LP performance based on the number of simplex iterations alone.

In Figure 4.13, we can see that the times for the different LP solvers to solve the root LPs of the MIPLIB 2017 benchmark instances is less varied than the number of simplex iterations necessary.

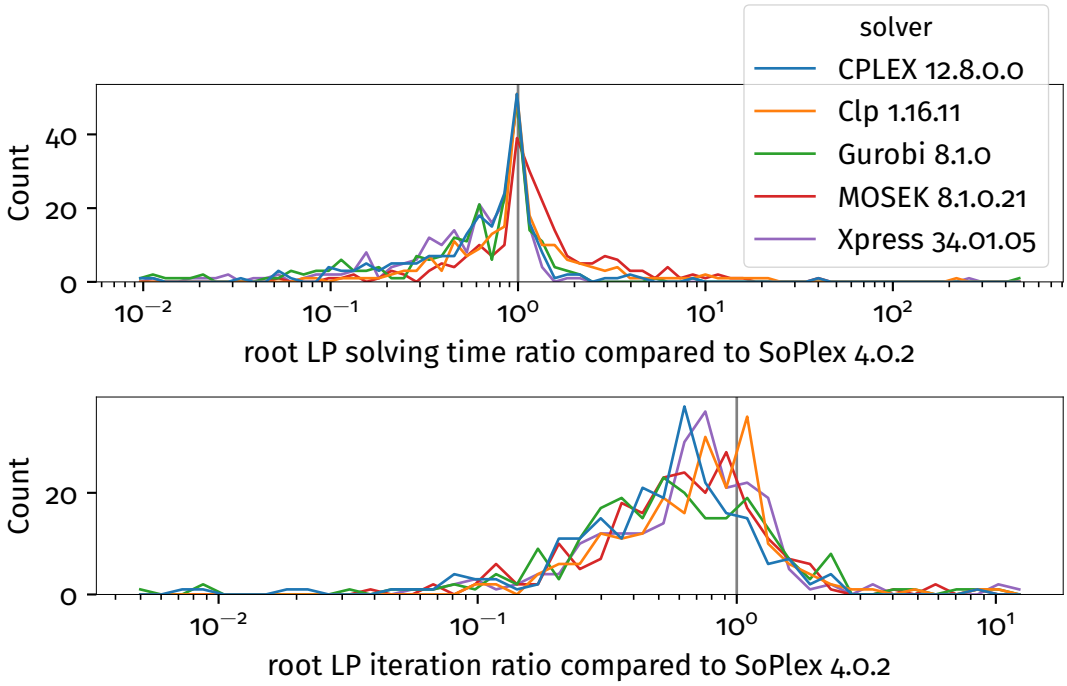


Figure 4.13.: Distribution of solving time and iteration ratios for different LP solvers in SCIP to solve the root LPs of the MIPLIB 2017 benchmark instances

Model size and difficulty. Table 4.3 also reveals that SoPlex struggles more with harder instances. There is a factor of about 1.6 between Gurobi and SoPlex regarding the shifted geometric mean time of all instances solved within the time limit. That factor grows to 2.5 for those instances that take more than five seconds for any solver. Furthermore, we can also see that SoPlex (and Clp for that matter) is affected differently by SCIP's presolving: while the faster solvers Gurobi, CPLEX, and MOSEK (Xpress is barely affected here) are actually able to achieve better times without SCIP presolving the instances, the performance of SoPlex and also Clp deteriorates. Since the primary goal of presolving is to make an instance more compact, we can accredit this

slowdown to the larger model sizes that the LP solver has to handle when using the `presoloff` setting.

In practice, hard LPs are often larger than hard MILPs, so it is more likely to see SoPlex struggle on its own than within SCIP when confronted with such a hard problem instance.

LP iterations and solving times in the tree. As we have established and discussed before, the node LP relaxations are significantly easier to solve, due to to the available warm-start information from a previously processed node. From Figure 4.14, we can deduce that both SoPlex and Clp require more time and iterations to solve the node LPs relative to the root. What is arguably most surprising here, is that at about depth 30, the numbers are increasing for those two solvers. Concerning the other four commercial solvers, we see that those relative numbers are clearly smaller leading to a better node throughput as we also confirm in Figure 4.16. And for LPs further down the tree, we can even observe a slight decrease, especially regarding the iteration counts.

Gap closed after the root node. The gap between current dual bound x_{dual} and incumbent value x_{primal} is a good measure for progress in an MILP solver. Hence, we want to compare how the choice of a specific LP solver affects the gap closed after the root node. It is important to note that we are using the gap definition of SCIP that differs from those of other solvers like Gurobi and CPLEX. While the latter ones compute the gap as

$$\frac{\left|x_{\text{primal}} - x_{\text{dual}}\right|}{\left|x_{\text{primal}}\right|},$$

SCIP uses the formula

$$\frac{\left|x_{\text{primal}} - x_{\text{dual}}\right|}{\min\{\left|x_{\text{primal}}\right|, \left|x_{\text{dual}}\right|\}}.$$

When both bounds have the same sign, the first gap variant moves between zero and one or 0% and 100%, whereas the one used by SCIP often exceeds the value of 100%.

Figure 4.15 shows how the different solvers frequently achieve a worse gap than possible with another solver. The gap is displayed on a logarithmic scale so all those results with a zero gap after the root node had to be slightly increased to appear in the chart. This explains the gap in the beginning of the otherwise rather smooth distribution of the best gap per instance.

Table 4.4 lists how often which LP solver reached the best possible gap using the default settings of SCIP and a node limit of one. We see that SCIP with CPLEX is able to provide the best results with regards to this performance measure.

From this experiment we conclude that the choice of LP solver within SCIP strongly affects already the first stage of the solving process before any branching decisions have to be made. Please refer to Table B.3 for the full results.

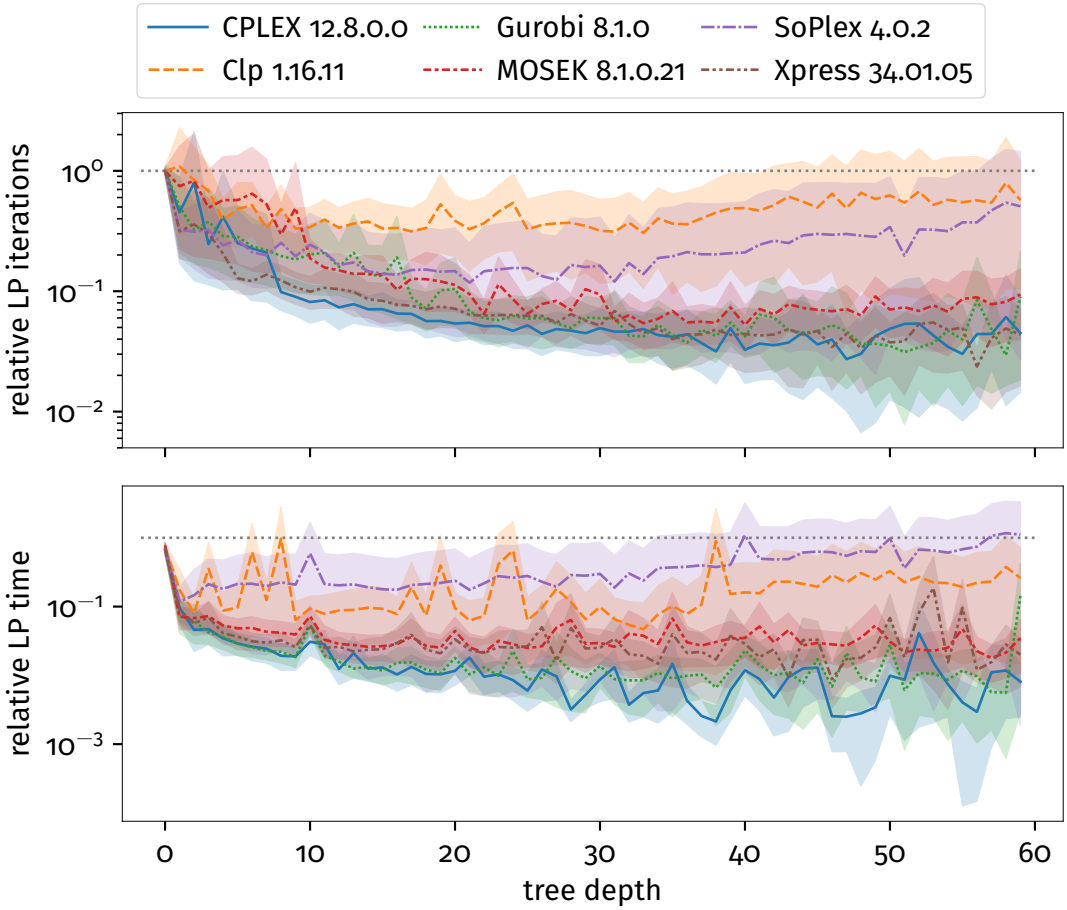


Figure 4.14.: Development of node LP solving times and simplex iterations when going down the tree. The metrics are relative to the respective number in the root LP. We only compare MIPLIB 2017 instances that have been solved within the time limit of one hour by all solvers (54 instances).

Node throughput. Another interesting measure for LP performance when solving an MILP is the node throughput. Here, we compute how many nodes can be processed in a given time. We restrict the results to instances that had at least a tree size of 100 with the respective solver. For this experiment we also deactivated both separation and heuristics to generate larger tree sizes—without this modification the results would look similar but less pronounced and would be based on a smaller data set. The enhanced box plot or *letter value plot* in Figure 4.16 is trying to convey more information about the distribution of the results than a conventional box plot. The central box is identical and represents the median with the central line and the two 25% quartiles next to it. Additional boxes subdivide the values outside of the typical quartiles to provide a more intuitive understanding of the distribution. For more information

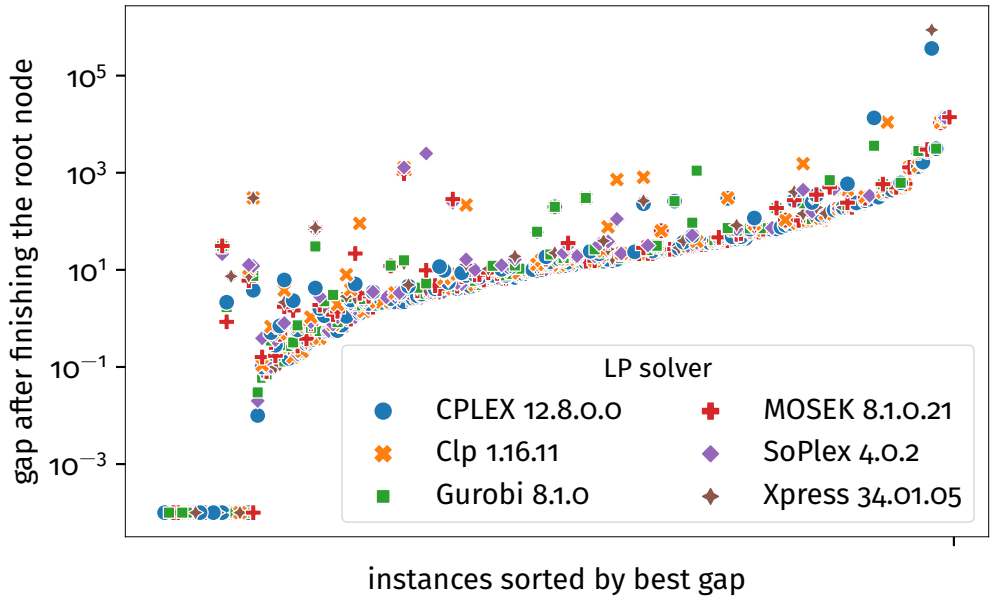


Figure 4.15.: Comparison of the gap after the root node processing for the different LP solvers. A node limit of one and a time limit of one hour was used on the MIPLIB 2017 benchmark set. The instances are sorted by the smallest gap of all solvers.

Table 4.4.: Number of times the respective LP solver reached the best gap after the root node on the MIPLIB 2017 benchmark instances. The root nodes of 177 instances could be finished by at least one solver within the one hour time limit.

CPLEX	Clp	Gurobi	MOSEK	SoPlex	Xpress
79	48	53	55	59	51

about this visualization, please refer to Hofmann, Wickham, and Kafadar (2017).

We can see from this chart that SCIP's node throughput is significantly reduced when using CLP as the LP solver. There is only little difference between the throughput averages of CPLEX, GUROBI, SOPLEX, and XPRESS, with CPLEX taking the lead. MOSEK takes the second last place in this ranking which confirm our previous findings about their respective performance within SCIP.

The full data set for this experiment can be found in Table B.6.

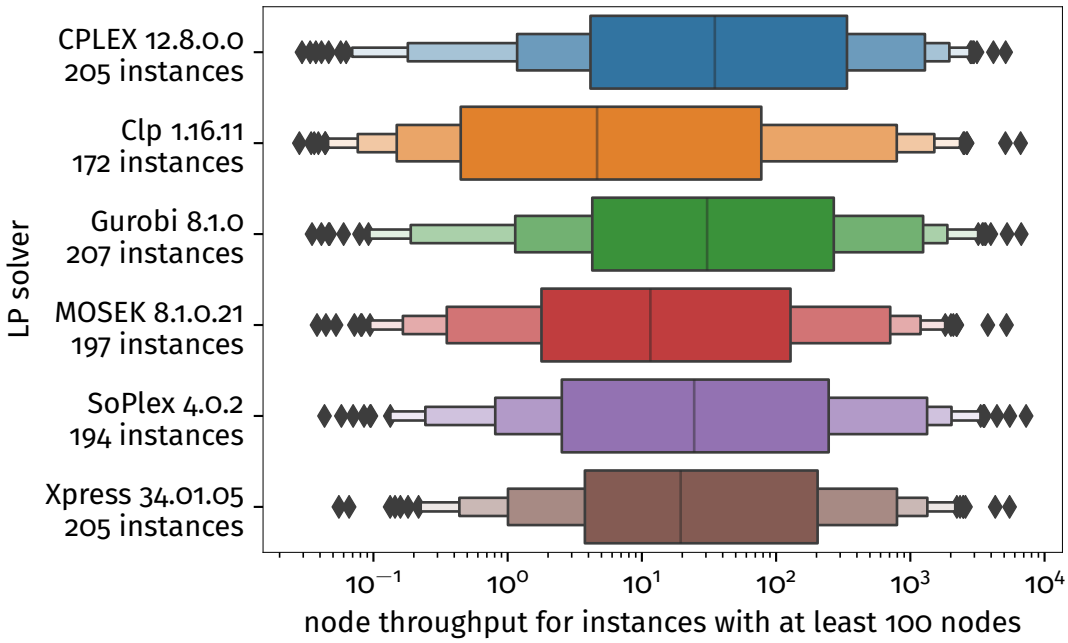


Figure 4.16.: Comparison of the node throughput for the different LP solvers with a time limit of one hour on the MIPLIB 2017 benchmark set. Cutting plane generation and primal heuristics are deactivated. Results are restricted to instances with a tree size of at least 100. The number of such instances is given underneath the LP solver name.

Chapter 5

LP Solution Polishing

Solutions to linear programs are rarely unique. Instead, due to the presence of dual degeneracy and the use of numerical tolerances, multiple distinct solutions may fulfill the optimality and feasibility conditions. Whenever there is a truly unique LP solution, we can often assume the problem to be of artificial nature or a miniature model.

This is especially true for LP relaxations in MILP solvers and one of the main reasons for their performance variability as observed by Koch, Achterberg, et al. (2011). A logical implication of this is trying to use the additional degree of freedom to improve the stability or the performance of the MILP solver. In this chapter, we describe our approach to realize this idea and a computational evaluation of our implementation.

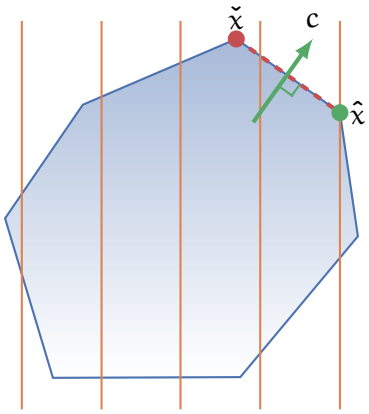


Figure 5.1.: Visualization of two distinct optimal LP solutions $\check{x}$ and $\hat{x}$, with only $\hat{x}$ being integer feasible. Here, the highlighted constraint is parallel to the objective, resulting in degenerate solutions on this optimal facet. LP solution polishing tries to find solutions with fewer fractional integer variables.

5.1. Related Work

Searching for another alternative LP solution on the optimal facet is not a new idea and has been investigated in different ways. For a good introduction to degeneracy in MILP problems, we refer to the works of Gamrath, Berthold, and Salvagnin (2020) and Berthold, Gamrath, and Salvagnin (2019) that focus on how branching decisions can be improved by exploiting multiple LP optima. Zanette, Fischetti, and Balas (2011) show how to exploit dual degeneracy by using the lexicographic simplex algorithm to find an optimal basis that is better suited to compute numerically stable cutting planes. A related approach for mixed-integer programming is *k-sample* (Fischetti et al., 2016). This approach runs the initial root LP and the cut loop several times on multiple cores using different random seeds, effectively exploiting the inherent variability. The aim is to collect different LP optima that provide richer cuts for the MILP solver. Alternatively, CPLEX implements an algorithm called *pump-reduce* (Achterberg, 2013) that fixes several variables and modifies the objective function value to explore different optimal LP solutions to improve cut selection and to reduce the fractionality of the solution.

Please note that there is also a technique called *solution polishing* by Rothberg (2007) that uses an evolutionary approach to improve upon a given integer feasible solution for MILPs. This is not related to our work—we coincidentally chose a similar name.

5.2. Description of the Approach

LP solution polishing tries to improve the quality of an existing LP optimum. The measure of solution quality in this case is the number of integer variables contained in the optimal basis. Since a non-basic variable is always at one of its bounds, it is also integral in the current LP solution because of the integrality of those bounds. Therefore, the fewer basic integer variables are present in an LP solution, the less fractional it can be—typically, a desired feature of solutions within an MILP solver. Of course, basic variables can have integral values as well, but this is less common and happens just by chance, due to their values being determined by the solution of a linear system with the optimal basis matrix, see Algorithm 1.2.

LP solution polishing, as described in Algorithm 5.1, starts after an optimal LP solution is found. The primal simplex method is employed to preserve feasibility of the solution while the pricing step is modified to only look for non-basic continuous variables or slack variables with zero reduced costs or dual multipliers to not deteriorate the objective function value. Then, a modified ratio test only accepts the pivot candidate to leave the basis if it is an integer variable. Otherwise, no basis change is performed. That way, in every successful iteration, the number of integer variables in the basis can be reduced by one.

Initially, the integrality information needs to be communicated from SCIP to SoPlex to avoid pivoting continuous variables out of the basis and thereby not reducing the

fractionality.

In contrast to the method described in Achterberg (2013), the presented algorithm does not modify the problem data internally. Furthermore, it is also possible to polish the solution of a pure linear program by treating all variables as integer variables. In general, we consider a polished solution to be superior, because more variables are precisely on their bounds rather than on some arbitrary value within the respective feasibility range.

Algorithm 5.1 LP solution polishing in SoPlex

Input: Optimal solution x, y, d with basis $\mathcal{B}$ of LP (1.2)

$\mathcal{C} = \{1, \ldots, n\}$ // set of problem variable indices
$\mathcal{R} = \{1, \ldots, m\}$ // set of slack variable indices
$\mathcal{N} = (\mathcal{R} \cup \mathcal{C}) \setminus \mathcal{B}$ // set of non-basic indices
$\mathcal{I} \subseteq \mathcal{C}$ // set of integer variable indices
$\mathcal{P} \leftarrow \mathcal{N} \cap (\mathcal{R} \cup (\mathcal{C} \setminus \mathcal{I}))$ // list of non-basic slack or continuous variables
while $\mathcal{P} \neq \varnothing$ **do**
  for $i \in \mathcal{P}$ **do** // find entering candidate among non-basic indices
    if $(c - A^\top y)_i = 0$ **then** // reduced cost or dual multiplier of 0
      select $j \in \mathcal{B}$ // primal ratio test
      if $j \in \mathcal{I}$ **then** // found integer variable x_j to leave the basis
        $\mathcal{B} \leftarrow \mathcal{B} \setminus \{j\} \cup \{i\}$ // perform regular basis update
        $\mathcal{N} \leftarrow \mathcal{N} \setminus \{i\} \cup \{j\}$
        update x and y
        $\mathcal{P} \leftarrow \mathcal{P} \setminus \{i\}$ // remove i from candidate list
      else
        no suitable index found to leave the basis, reject candidate i
      end if
    end if
  end for
  if no pivot performed or iteration limit reached **then**
    break
  end if
end while
Return: polished solution x, y, d and basis $\mathcal{B}$

It has to be considered that the algorithm as it is presented here, does not compute the least fractional LP solution but instead represents a *greedy* approach. Furthermore, as outlined in Algorithm 5.1, we keep a candidate list of non-basic slack and non-basic continuous variables to speed up the outer selection loop that replaces the pricing step of the simplex algorithm. Degenerate candidates that had to be skipped in a previous round are then tried again for a successful pivot with the updated basis. We employ a small tolerance when checking the reduced costs for degeneracy since these values are rarely exactly zero due to inaccuracies in their computation. The pro-

cedure is terminated when no more successful pivots could be made, the candidate list is exhausted, or the maximum number of iterations has been reached.

5.3. Impact on Numerical Stability

We want to investigate how our method influences the numerical features of the final optimal basis. LP solution polishing has the potential to improve the condition numbers of these bases: Pivoting more slack variables into the basis increases the number of unit vectors in the matrix and should result in a smaller condition number (see Chapter 6 for more details on the condition number and numerics in LPs and MILPs).

On the other hand, we can also construct matrices with orthogonal columns—the numerically most stable form with the smallest possible condition number of 1—and have some of them be almost parallel to a unit vector:

$$A_1 = \begin{bmatrix} -\epsilon/\alpha & 1/\alpha & 0 \\ 1/\alpha & \epsilon/\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix}, \text{ with } \alpha = \sqrt{1+\epsilon^2}, \quad \kappa(A_1) = 1$$

Swapping the first column of A_1 for such a unit vector deteriorates the condition number as the columns are not orthogonal anymore and the new matrix A_2 can be arbitrarily close to singularity:

$$A_2 = \begin{bmatrix} 1 & 1/\alpha & 0 \\ 0 & \epsilon/\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix}, \text{ with } \alpha = \sqrt{1+\epsilon^2}, \quad \kappa(A_2) \approx 1/\epsilon$$

Our experiments on pure LPs from the `allLP` test set do not show a trend towards improved final condition numbers. This contradicts our intuition towards lower condition numbers with more unit columns in the basis matrix.

Furthermore, the number of new slack variables introduced into the basis is rather insignificant compared to the matrix dimensions of most instances. We also want to stress that due to the utilization of integrality information, we do not necessarily increase the number of slacks in the basis but may also pivot in some continuous variables.

Besides, we know from our experimental results in Figure 6.3 that the condition numbers in the simplex algorithm are very stable as the method iterates toward optimality so we should not expect the additional LP solution polishing to have a strong effect.

Still, we want to present our findings in this area:

Figure 5.2 displays the differences of (logarithms of) the condition numbers of the optimal basis matrices with and without polishing applied to the first LP relaxation. The difference of logarithms relates to the original values a and b as follows:

$$\log a - \log b = c \Leftrightarrow a = 10^c \cdot b$$

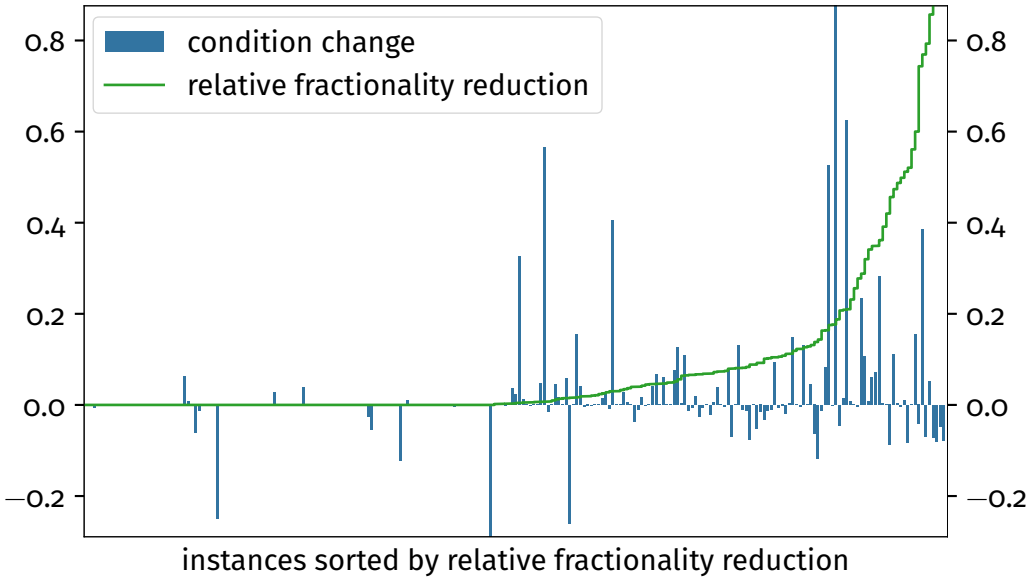


Figure 5.2.: Effect of LP solution polishing on the condition numbers of the optimal basis matrix of first LP relaxations of the MIPLIB 2017 benchmark instances. Condition number changes are differences in log10 between a run with and one without enabling polishing.

There is not a single instance exhibiting a change that exceeds more than one order of magnitude and hence we shall not draw any conclusions about the numerical effect of LP solution polishing here. At first glance, there is a trend of increasing condition numbers but this is too small to have any significance. Especially, if we recall that the condition number represents just an upper bound on the input error amplification.

Another metric concerning numerics is the following: SCIP counts the number of unstable LP solves, that is, the number of LP solutions that did not pass the feasibility or optimality check and triggered a fresh LP solve with different parameters. It turned out, though, that in the MIPLIB 2017 benchmark set, there are too few instances that exhibit these unstable solves (only nine across the different polishing settings and seeds) to make any sound conclusion about whether LP solution polishing can help in reducing this number. There were also no noticeable improvements when spot testing on instances that are numerically more challenging.

5.4. Reduced Costs on the Optimal Facet

Here, we want to mention another use-case for LP solution polishing with an entirely different goal. Instead of trying to reduce the integrality of a relaxation solution, we want to learn more diverse reduced cost values. In line with the definition of a basis (see Definition 1), only non-basic (slack or structural) variables can have non-zero

dual multipliers or reduced costs. By performing additional simplex steps along the optimal facet we are able to collect a wider range of reduced costs as more variables can be made non-basic. This is only possible in the state of degeneracy, that is when further iterations can be performed without losing optimality of the solution.

The reduced cost values of the current LP solution are used in various places in SCIP. A popular technique is *reduced cost strengthening* (Nemhauser and Wolsey, 1988; Achterberg, 2009).

5.4.1. Reduced Cost Strengthening

Let $\mathcal{B}$ be a basis of an LP in equality form:

$$\begin{aligned} \min \quad & c^\top x \\ \text{s.t.} \quad & Ax = b \\ & x \geq 0 \end{aligned} \tag{5.1}$$

Then, using $B = A_{\mathcal{B}}$ and $N = A_{\mathcal{N}}$, this LP is equivalent to

$$\begin{aligned} \min \quad & c_{\mathcal{B}} B^{-1} b + (c_{\mathcal{N}} - c_{\mathcal{B}} B^{-1} N) x_{\mathcal{N}} \\ \text{s.t.} \quad & x_{\mathcal{B}} = B^{-1} b - B^{-1} N x_{\mathcal{N}} \\ & x_{\mathcal{B}}, x_{\mathcal{N}} \geq 0. \end{aligned} \tag{5.2}$$

As we have already established earlier, when the basic variables $x_{\mathcal{B}} = B^{-1} b \geq 0$ and the reduced costs $d_{\mathcal{N}} = c_{\mathcal{N}} - c_{\mathcal{B}} B^{-1} N \geq 0$, then $z_{LP} = c_{\mathcal{B}} B^{-1} b$ is the optimal value of LP 5.1 and $\mathcal{B}$ an optimal basis—recall that both $d_{\mathcal{B}} = 0$ and $x_{\mathcal{N}} = 0$.

Now, assume that LP 5.1 is the relaxation of an IP, that is, all variables are required to be integer ($x \in \mathbb{N}^n$). Given an upper bound $\hat{z}$ of this IP, we know that an optimal IP solution $\hat{x}$ must satisfy

$$z_{LP} + d_{\mathcal{N}} \hat{x}_{\mathcal{N}} \leq \hat{z} \tag{5.3}$$

because of the equality

$$\{\min c^\top x \mid Ax = b, x \in \mathbb{N}^n\} = \{\min z_{LP} + d_{\mathcal{N}} x_{\mathcal{N}} \mid Ax = b, x \in \mathbb{N}^n\}.$$

Due to the lower bounds of $x_{\mathcal{N}} \geq 0$ and $d_{\mathcal{N}} \geq 0$, when propagating the previous inequality 5.3 we obtain

$$x_j \leq \frac{\hat{z} - z_{LP}}{d_j}$$

for $j \in \mathcal{N}$ such that $d_j > 0$. This is known as reduced cost strengthening and can provide tighter variable bounds.

Note that if we obtain another optimal basis $\mathcal{B}'$, with different reduced costs d', the propagation of variable x_j if $d'_j > 0$ is

$$x_j \leq \frac{\hat{z} - z_{LP}}{d'_j}.$$

In fact, we just need to store the largest reduced costs for each variable and use that value to apply the bound strengthening technique.

We implemented a prototype to collect the largest reduced costs during the solution polishing process and use them in SCIP but our experiments have been inconclusive and did not show a significant enough performance improvement to justify the additional overhead of maintaining the modified reduced cost storage. Still, we believe that there is some hidden potential here that should be investigated further.

5.4.2. Cutting Plane Evaluation

Another application area for reduced costs in an MILP solver is during the cut filtering process. A cutting plane that has non-zero coefficients for variables with positive reduced costs is expected to result in an improved dual bound in the next LP solution. Hence, more diverse reduced costs may enable a more reasonable cut selection. This idea has first been presented by Achterberg[1] and is described in the patent by Achterberg (2013). We did not test this extension in our implementation.

5.5. Computational Study

Since the choice of a different LP basis can drastically alter the solution path of SCIP, we need to carefully separate random effects from actual influences of our algorithm.

While the procedure is not supposed to change the objective function value, it may still happen in rare cases—especially when dealing with numerically sensitive models.

In Figure 5.3, we demonstrate the effect of LP solution polishing on the very first LP that is solved for each of the MIPLIB 2017 benchmark instances. We can see that for about half of the instances, there is a reduction in the number of fractional integer variables in that first LP solution and there are several instances with a drastic fractionality reduction of up to 80%. The additional simplex iterations incurred by the polishing procedure correspond to that reduction: For those instances that do not benefit from our technique, there is also barely any overhead—with a few exceptions. The iteration increase is at most about 40% and often a lot less. The extra time required to perform those polishing iterations displays a more erratic behavior: The largest increase is almost 80% but only nine instances require 20% or more additional solving time. The majority of samples remains well below 10% time increase. Note that we used averages from three different random seeds running with and without polishing in the root node. For the full data behind this chart, please refer to Table B.2.

Evaluating the effect of LP solution polishing on the overall performance of SCIP reveals a positive impact. Here, we compare four different settings: polishing for all LPs, polishing only for the root LP, polishing disabled, and the default mode. This default mode activates LP solution polishing during the probing and diving stages of

[1] LP basis selection and cutting planes. Mixed Integer Programming Workshop, Atlanta. 2010
https://www.isye.gatech.edu/news-events/events/past-conferences/2010-mixed-integer-programming-workshop

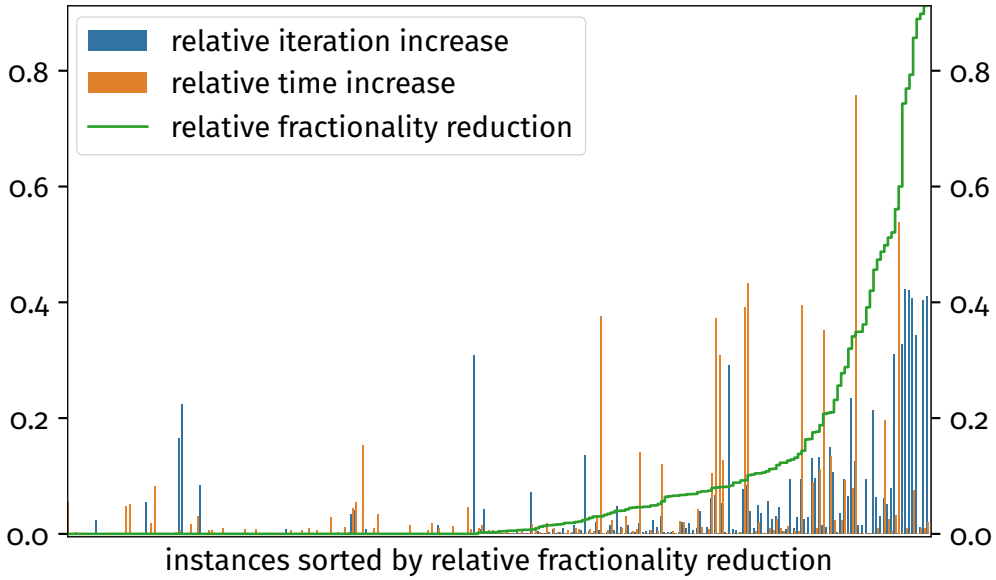


Figure 5.3.: Effect of solution polishing on the fractionality of the first LP solution for the MIPLIB 2017 benchmark instances. The green line depicts the relative fractionality reduction while the blue and orange bars show the relative increase in iterations and solving time for that specific instance. We used the averages of three random seeds running with SCIP version 7.

SCIP. These are used for various purposes in SCIP to initiate a quick LP exploration path where fewer fractional variables are deemed to be very useful. Experiments have revealed that this slightly reduced application of polishing leads to the best performance as also Table 5.1 demonstrates.

We can see from these results that our technique helps to solve three more instances to optimality while reducing the shifted geometric time to optimality by more than 6%. The number of nodes can be reduced by almost 8% and the primal-dual-integral (Berthold, 2014) for the solved instances is reduced by 3%. The latter measure indicates how quickly the primal and dual bounds move towards each other by computing the integral of the difference of those two numbers over the solving time.

Figure 5.4 overlays the root node fractionality reduction with the speedup factors of the total solving times. We see that instances with a larger speedup due to polishing, that is, instances with a downward-pointing bar, are more often showing a reduced root node fractionality. We can also observe that for instances with no root node fractionality reduction there is a higher chance of a slow-down when using polishing.

		count	solved	limit	time		nodes		PDI	
					sgm	Q	sgm	Q	am	Q
group	settings									
all	default	240	103	137	809.09	1.000	4489	1.000	97 716.1	1.000
	nopolish	240	100	139	816.89	1.010	4598	1.024	98 661.6	1.010
	rootpolish	240	105	135	792.32	0.979	4561	1.016	97 344.9	0.996
	alwayspolish	240	104	136	789.83	0.976	4294	0.957	98 724.2	1.010
affected	default	101	94	7	159.49	1.000	3103	1.000	21 059.6	1.000
	nopolish	101	92	9	163.56	1.026	3242	1.045	21 123.4	1.003
	rootpolish	101	96	5	158.30	0.993	3070	0.989	21 292.3	1.011
	alwayspolish	101	95	6	152.11	0.954	2753	0.887	20 056.9	0.952
all-optimal	default	92	92	0	90.12	1.000	1717	1.000	9015.7	1.000
	nopolish	92	92	0	95.87	1.064	1851	1.078	9297.9	1.031
	rootpolish	92	92	0	93.80	1.041	1815	1.057	8764.8	0.972
	alwayspolish	92	92	0	92.84	1.030	1692	0.985	9196.5	1.020

Table 5.1.: Different LP solution polishing settings on the MIPLIB 2017 benchmark instances with SCIP 7, a time limit of one hour, and three random seeds for every setting. Shifted geometric means are computed with a shift of 1 for the time values and a shift of 100 for the node counts. The quotients for respective numbers is always using the default settings as comparison.

5.6. Conclusion

This chapter demonstrates how the simplex method can be tweaked to exploit degeneracy in MILP relaxations to provide a better solution with fewer fractional variables. LP solution polishing can strongly reduce the LP fractionality on certain instances, often without considerably increasing the solving time. This has also a positive effect on the overall MILP performance on SCIP helping to solve more instances in a shorter amount of time.

LP solution polishing is not affecting numerical stability in a significant way, neither by observing condition number changes nor by counting the number of unstable LP solves within SCIP. Note that we did not test this on numerically challenging instances but on the MIPLIB 2017 benchmark instances that have rather reasonable numerical features. The following chapter deals with the topic of numerics and explains the concept of stability and condition numbers.

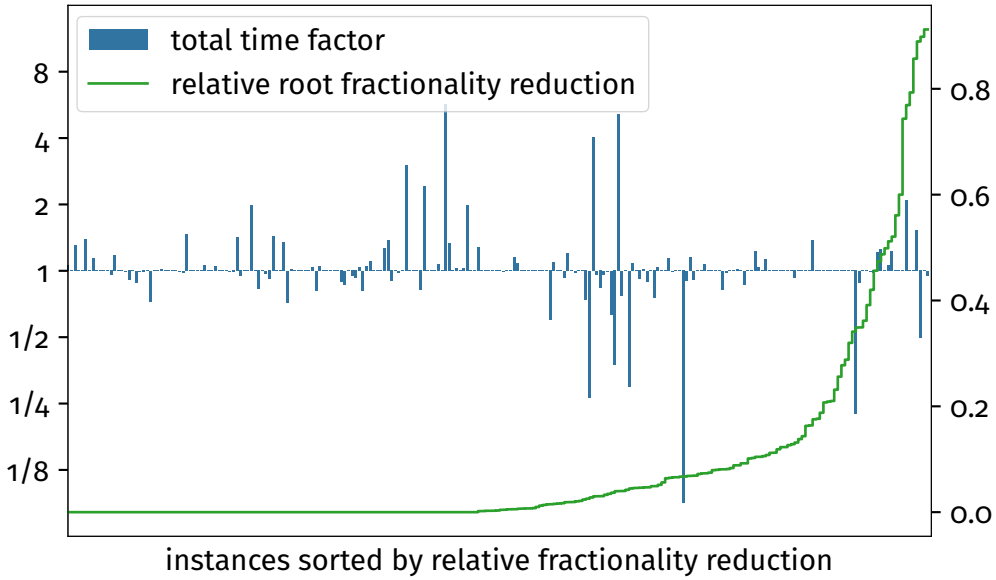


Figure 5.4.: Effect of solution polishing on the total solving time for the MIPLIB 2017 benchmark instances. The green line depicts the relative fractionality reduction while the blue bars show the speedup of using the default polishing mode compared to disabled polishing. The numbers are in log2 scale, so factors below zero indicate a speedup. We used the averages of three random seeds running with SCIP version 7.

Chapter 6

Numerics in Branch & Bound & Cut

Whenever we are dealing with numerical computations we need to be aware of inaccuracies that can be related to the problem to solve, the data that comprise the problem, and the specific algorithm employed.

Let us open the discussion with an extreme example: Due to the necessary use of numerical tolerances, it may happen that an ill-posed problem can be correctly regarded as *both* feasible and infeasible. An analogy outside of the mathematical context is the image presented in Figure 6.1 that can be read as two very different words:

Figure 6.1.: Calligraphic illusion *"Laurel & Yanny"*. Image credit: `https://twitter.com/AriadneRem/status/996609946228703232`

A corresponding LP example is the following feasibility problem:

$$\begin{aligned} x + 10^{-8} \cdot y &= 10^{-7} \\ x, y &= 0. \end{aligned} \tag{6.1}$$

Using the commonly used tolerance of $\epsilon = 10^{-6}$, there clearly are feasible solutions to this mathematically infeasible problem: Consider x to be basic, the non-basic variable y is then set to zero. This results in the numerically tolerated solution of $(x, y) = (10^{-7}, 0)$. For y basic, though, the simplex method cannot find a solution and reports infeasibility because with non-basic variable x set to zero, y has to be 10, strongly exceeding its bound regardless of the tolerance ϵ. Both results are numerically correct. While this is an artificial example, such inaccuracies are a frequent issue and may lead to serious complications when the precise results are required.

Chip design verification—one of the earliest application areas for SCIP as presented by Achterberg (2007)—is a good example of such a problem class.

There have been various attempts to use a *"condition number"* to assess the numerical difficulty or categorize problem instances. We want to give an overview and comparison on a number of different conditions and measures suitable for MILP solvers. Usually, the field of numerical analysis focuses on continuous problems and we want to transfer these methods and techniques over to the discrete world of mixed-integer linear programming. Furthermore, we focus on the applied component rather than the theoretical part because the traditional ideas of stability and convergence do not fit the discrete nature of MILPs.

First, we need to give a short introduction to the general concepts used in this chapter.

6.1. Background on Numerical Analysis

Two terms are frequently used when discussing numerical properties of a problem or an implementation: *stability* and *condition*. Stability usually refers to *backward stability* and measures the deviation between the algorithmically calculated solution to a problem and the true solution to a slightly perturbed problem. This quantity is independent of the data of the problem and is instead a property of the algorithm used to find the solution. Hence, it is also often referred to as *stability of an algorithm*. *Condition* on the other hand, is independent of the algorithm and is bound to the input data. For a given problem f with input x it can be defined as

$$\kappa = \lim_{\epsilon \to 0} \sup_{\mathrm{err}_{\mathrm{rel}}(x) \leq \epsilon} \frac{\mathrm{err}_{\mathrm{rel}}\left(f(x)\right)}{\mathrm{err}_{\mathrm{rel}}\left(x\right)}.$$

The relative error is defined as

$$\mathrm{err}_{\mathrm{rel}}\left(x\right) = \frac{\|x - \tilde{x}\|}{\|x\|}.$$

Of course, these definitions are dependent on a certain norm $\|\cdot\|$. Whenever it is not explicitly mentioned, we assume the Euclidean norm $\|\cdot\|_2$ for vector spaces.

The *condition number* κ measures how much the input error magnifies the output error when using exact arithmetic. It is important to understand that the stability of an algorithm cannot mitigate a high condition number of the numerical problem statement.

We can formulate this rule of thumb for numerical computations: *A stable algorithm for a well-conditioned problem is expected to compute an accurate solution.*

Arguably the most important condition number is the one for inverting a square matrix A and is commonly known as *the condition number of a matrix*[1] that was intro-

[1] https://nickhigham.wordpress.com/2019/01/23/who-invented-the-matrix-condition-number/

duced by Turing (1948). We also stick to this convention and recall its commonly used computational form:

Definition 4 (Condition number of a matrix)**.** *The condition number* κ *of a square matrix* $A \in \mathbb{R}^{n,n}$ *is defined as*

$$\kappa(A) = \|A\|\|A^{-1}\|.$$

This is actually a result of a more fundamental and geometric way of thinking about the condition of a problem using the notion of *ill-posedness* attributed to Hadamard (1902):

Definition 5 (Ill-posedness)**.** *A problem is well-posed if it has a (unique) solution and the solution depends continuously on the input. A problem that is not well-posed is* ill-posed*.*

This definition is kept vague on purpose because the meaning of ill-posedness depends heavily on the actual problem in question.

For the problem of inverting a square matrix, though, the set Σ of ill-posed problems consists of all singular matrices. Then the distance to ill-posedness can be defined as

$$d(A, \Sigma) = \min\{\|A - S\|,\ S \in \Sigma\}.$$

Theorem 2. *For* $A \in \mathbb{R}^{n,n}$ *the following holds.*

$$d(A, \Sigma) = \frac{1}{\|A^{-1}\|}.$$

Using Definition 4, we can connect the condition number with the distance to ill-posedness:

$$\kappa(A) = \frac{\|A\|}{d(A, \Sigma)}.$$

This allows us to interpret the condition number of matrix A as the inverse of the relative minimum distance to an ill-posed or singular matrix. Depending on the used norm $\|\cdot\|$, there are also techniques for estimating the condition number as shown by (Hager, 1984) for the ℓ_1-norm. Bürgisser and Cucker (2013) provide both an excellent overview and an in-depth analysis of condition numbers and the stability of numerical algorithms.

An illustrative use of condition numbers comes in the form of the stability analysis of an algorithm that is sketched in Figure 6.2. Let $\hat{f} : X \mapsto Y$ be the algorithm to solve a given problem f. For any input x the result $\hat{f}(x)$ may differ from the true solution $f(x)$. Yet, there is a $\tilde{x} \in X$ that satisfies $f(\tilde{x}) = \hat{f}(x)$. The difference between x and $\tilde{x}$ is called *backward error*, while the *forward error* is the difference between $f(x)$ and $\tilde{f}(x)$. The

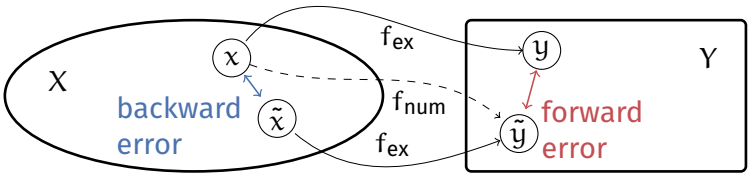


Figure 6.2.: Forward and backward error

algorithm $\tilde{f}$ is said to be *stable* or *backward stable* if the backward error is bounded by the forward error multiplied with the condition number of the problem:

$$\mathrm{err}_{\mathrm{rel}}(f) \leq \kappa_f(x) \cdot \mathrm{err}_{\mathrm{rel}}(x)$$

There are several studies on the condition number of convex optimization problems, which are generalizations of LP problems. Ordonez and Freund (2003) and Epelman and Freund (2002) measure the condition by looking at the entire problem, including right-hand side and objective function. The main focus of the work of Ordonez and Freund (2003) was to find a correlation between the condition number and the number of iterations an interior point algorithm would need to solve an LP instance.

Another result is that the majority of NETLIB LP instances (see Koch, 2004b for an overview) are actually ill-posed with respect to this condition metric. This is interesting in so far as those instances have been used for decades in benchmarks and publications and are nowadays considered mere toy problems that most solvers can easily deal with. So, we are inclined to find another condition metric that hopefully gains a better understanding of whether an instance is numerically difficult.

Important to note is that Ordonez and Freund (2003) succeeded using their LP condition number to estimate the necessary number of iterations an interior point method would need to solve a given instance. They achieved this by applying industry-standard presolving reductions to the models, before computing the condition number. The presolved NETLIB instances are also mostly well-defined with respect to their condition number. This presolved model then also corresponds to the actual data the solver uses to compute the solution, so there is a stronger relationship compared to using the instances in their original form.

We re-implemented this condition metric and will apply it to MILP instances later in this chapter.

LP condition number Bürgisser and Cucker (2013) extend the definition of condition as distance to ill-posedness to linear programs in the following way:

Definition 6 (LP condition number)**.** *For a feasible and well-posed LP* $\min\{c^T x \mid Ax = b,\ x \geq 0\}$ *with optimal basis* $\mathcal{B}$ *we define the condition number as*

$$\kappa_{LP}(\mathcal{A})_{rs} := \frac{\|\mathcal{A}\|_{rs}}{d_{LP}(\mathcal{A}, \Sigma_{LP})}$$

This requires the definition of the distance d_{LP} of an LP to ill-posedness and also a classification of ill-posed LPs. In line with the above notion we declare every LP ill-posed, that does not have a unique solution. This also includes all LPs with degenerate optimal solutions. Furthermore, infeasible or unbounded LPs are only ill-posed if they are sufficiently close to an LP that has an optimal solution.

We call this set of ill-posed LPs Σ_{LP}.

Definition 7 (LP distance to ill-posedness)**.** *For a feasible and well-posed LP with data $\mathcal{A}$ we define the distance to ill-posedness as*

$$d_{LP}(\mathcal{A}, \Sigma_{LP}) = \inf\{\|\Delta\mathcal{A}\|,\ \mathcal{A} + \Delta\mathcal{A} \in \Sigma_{LP}\}.$$

Naturally, this distance is zero for ill-posed LPs. Note that $\mathcal{A} := \begin{bmatrix} A & b \\ c^\top & 0 \end{bmatrix}$.

For infeasible or unbounded LPs that are not in Σ_{LP}, d_{LP} is undefined. Just like in Definition 4 for square matrices, we want to have an alternative formulation for the condition number of LPs that is more concrete and computationally tractable. A result from Bürgisser and Cucker (2013) about the distance to ill-posedness delivers the necessary tool:

Theorem 3. *For a feasible and well-posed LP with data $\mathcal{A}$ and optimal basis $\mathcal{B}$ the following holds for the distance to ill-posedness:*

$$d_{LP}(\mathcal{A}, \Sigma_{LP}) = \min_{S \in S_1 \cup S_2}\{\|\Delta S\|,\ S + \Delta S \textit{ is singular}\}$$

with

$$S_1 := \{m \times m \textit{ submatrix of } [A_{\mathcal{B}}\ b]\}$$
$$S_2 := \{m+1 \times m+1 \textit{ submatrix of } \begin{bmatrix} A \\ c^\top \end{bmatrix} \textit{ containing } A_{\mathcal{B}}\}.$$

Applying Theorem 2 allows us to formulate the LP condition number of Definition 6 with respect to those matrix sets:

$$\kappa_{LP}(\mathcal{A})_{rs} = \|\mathcal{A}\|_{rs} \cdot \max_{S \in S_1 \cup S_2} \|S^{-1}\|_{sr}$$

with $\mathcal{A}$, S_1, and S_2 as above.

Computing this number is possible but rather expensive: In the set $S_1 \cup S_2$ there are $n+1$ matrices of which we need to calculate the norm of the inverse. Depending on the norm, that is, the choice of r and s, it can be computationally tractable.

There are further metrics that can be studied to gain an understanding of the numerical properties of a problem.

Determinant The *determinant* $\det(A)$ of a square matrix $A \in \mathbb{R}^{n,n}$ is equal to the product of its eigenvalues. Geometrically, it is the volume of the convex hull spanned by the columns of A. There are some interesting algebraic properties of the determinant:

$$\begin{aligned} \det(I) &= 1 \\ \det(A^{-1}) &= 1/\det(A) \\ \det(cA) &= c^n \det(A) \\ \det(AB) &= \det(A)\det(B), \text{ for } B \in \mathbb{R}^{n,n}. \end{aligned}$$

Furthermore, the determinant of a triangular matrix is the product of its diagonal entries $\prod a_{ii}$. Together with the above multiplicativity, we can easily compute the determinant of a matrix factorized as LU with lower and upper triangular matrices L and U:

$$\det(A) = \det(LU) = \det(L)\det(U) = \prod_{i=1}^{n} l_{ii} \prod_{i=1}^{n} u_{ii}$$

Since the diagonal entries of L are usually set to one, the computation is further simplified to the product of the diagonal elements of U. Additionally, one needs to be aware that due to permutations in the factorization, $\det(A)$ might have a different sign than $\det(U)$, which can be avoided by just regarding the absolute values.

Trefethen and Bau (1997) comment on the determinant: *"The determinant, though a convenient notion theoretically, rarely finds a useful role in numerical algorithms"*. Take for instance the diagonal matrix $A = \mathrm{diag}(1^{-3}, \ldots, 1^{-3})$. Its determinant is $\det(A) = 1^{-5n}$ which can be arbitrarily close to zero, indicating closeness to singularity. The corresponding condition number on the other hand is $\kappa(A) = 1$, specifying that it is far away from singularity or ill-posedness.

Nevertheless, Zanette, Fischetti, and Balas (2011) investigate how cutting planes affect the determinant of optimal basis matrices. The determinant is also a measure of the size of the matrix coefficients: It is the (signed) volume of the parallelepiped constructed of the columns of the matrix. For their example with `stein15`—a pure $0/1$ problem—it is reasonable that the matrix coefficients grow if cuts are added with coefficients larger than 1. Also, when using the lexicographic simplex to perform the reoptimizations, the added cuts are less likely to be active and the following cut round is not going to base new cuts on top of them. While it might be worthwhile to investigate determinants in addition to condition numbers for certain problem classes, we could not find a good use of this metric for general LPs or MILPs.

Trace The *trace* $\mathrm{tr}(A)$ of a square matrix $A \in \mathbb{R}^{n,n}$ is defined as the sum of its diagonal entries $\sum_{i=1}^{n} a_{ii}$. Interestingly, the trace is also the sum of eigenvalues of the matrix:

$$\mathrm{tr}(A) = \sum \lambda_i.$$

Unfortunately, the trace is not multiplicative, that is $\mathrm{tr}(AB) \neq \mathrm{tr}(A) \cdot \mathrm{tr}(B)$, so we cannot directly compute this metric from the LU factorization. Still, we implemented

the computation of the sum of diagonal elements of U in SoPlex but could not find a useful application for this measure.

As the eigenvalues of a triangular matrix are the elements on its diagonal, we can easily determine the smallest and largest eigenvalues. We use this ratio of diagonal extreme values of U as estimate for the condition number κ of $A = LU$.

In total, SoPlex provides the following matrix metrics for the current basis matrix $A_{\mathcal{B}} = LU$ at every iteration of the simplex algorithm:

- determinant of the basis matrix $A_{\mathcal{B}}$
- trace of U from the current LU factorization $A_{\mathcal{B}} = LU$
- estimated condition number of $A_{\mathcal{B}}$ by computing the ratio of largest and smallest absolute values of the diagonal elements of U

Skeel's condition number Yet another metric one might consider is *Skeel's condition number*. This is defined as

$$\kappa_{\text{skeel}}(B) = \left\| \left|B^{-1}\right| \cdot |B| \right\|_\infty$$

and has been proposed by Skeel (1979), although the naming convention followed later by other authors. It differs from the classical condition number $\kappa(B)$ by taking the element-wise absolute values of all matrix coefficients. This results in a condition number that is always less or equal to κ and most importantly invariant to row scaling:

$$B = \begin{bmatrix} 1 & 0 \\ 0 & 10^9 \end{bmatrix}, \quad \kappa(A) = 10^9, \quad \kappa_{\text{skeel}}(B) = 1.$$

Unfortunately, this variant of the condition number is even more difficult to compute and this is likely the reason for it not being used more often in practice. We did not perform any experiments or analyses with Skeel's condition number but still feel that mentioning it here is justified and may inspire future research.

We investigate and analyze how the numerical features of an MILP instance evolve during the solving process.

6.2. Numerical Analysis for LP and MILP

In this section we are going to explore the numerical stability of the branch-and-cut algorithm. There is a strong connection between the geometry of a specific instance and its numerical properties. Unfortunately, it is not known how to visualize or grasp this geometry for any reasonable problem sizes, so various abstractions and approximations are required. One such measure is the condition number of the basis matrix that is used in the simplex algorithm. Recall the definition of the traditional condition number κ of a (regular square) matrix A:

$$\kappa(A) := \|A\| \cdot \|A^{-1}\|.$$

Although this number specifies the condition or distance to ill-posedness of solving a linear system of equations, it is still an appropriate measure for several other operations with this matrix or its inverse—especially as we are dealing with many linear systems when solving LPs and MILPs.

Let us inspect some examples to illustrate the expressiveness of the condition number. First, a matrix with a large range of coefficients:

$$A = \begin{bmatrix} 10^5 & 1 \\ 0 & 10^{-5} \end{bmatrix},\ A^{-1} = \begin{bmatrix} 10^{-5} & -1 \\ 0 & 10^5 \end{bmatrix} \text{ with } \|A\| \approx 10^5 \text{ and } \|A^{-1}\| \approx 10^5.$$

Hence, for the condition number $\kappa(A) \approx 10^{10}$. We see that mixing coefficients of vastly different magnitudes easily lead to large error amplifications and should be avoided in the modeling phase. Of course, scaling can be used to reduce the range but, as noted before, may then cause an increase of violations in the original space when there are (numerically tolerated) violations in the scaled space.

Next, let us consider a matrix with large coefficients and a reduced range:

$$A = \begin{bmatrix} 10^5 & 1 \\ 0 & 10^5 \end{bmatrix},\ A^{-1} = \begin{bmatrix} 10^{-5} & -10^{-10} \\ 0 & 10^{-5} \end{bmatrix} \text{ with } \|A\| \approx 10^5 \text{ and } \|A^{-1}\| \approx 10^{-5}.$$

Here, the condition number $\kappa(A) \approx 1$ is a lot smaller than the former case and is not going to cause numerical issues—it is very close to a diagonal matrix. Still, those large coefficients can deteriorate violations in the transformed or presolved space depending on the bounds of the variables and right-hand sides of the constraints. Again, despite this case being way less critical than the first example, we should still aim for reducing the overall magnitude of coefficients.

Finally, we want to demonstrate the case of a deceptively well-behaved matrix $A \in \mathbb{R}^{n,n}$:

$$A = \begin{bmatrix} 1 & -2 & 0 & \dots & 0 \\ 0 & 1 & -2 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & 0 \\ 0 & \dots & 0 & 1 & -2 \\ 0 & \dots & \dots & 0 & 1 \end{bmatrix},\quad A^{-1} = \begin{bmatrix} 1 & 2 & 4 & \dots & 2^{n-1} \\ 0 & 1 & 2 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & 4 \\ 0 & \dots & 0 & 1 & 2 \\ 0 & \dots & \dots & 0 & 1 \end{bmatrix}.$$

This matrix has a very small range of coefficients, the norm is also very small, but the coefficients of the inverse grow exponentially with the size of the matrix. Such a banded matrix appears in a benchmark set of quadratic programming instances collected by Maros and Mészáros (1999). This `laser` or `ilaser` model instance originates from the field of electrical engineering and has caused us some headaches when trying to avoid numerical resonance catastrophes in the solving process; eventually, we concluded that there is no cure for this instance and a warning message alerts the user that the model is likely ill-posed.

A common guideline and recommendation is to inspect the ranges of the coefficients of a problem instance. As the last example shows, this is not sufficient to guard

against all numerical issues. For a more informed analysis, we need to include further metrics like the condition number as proxy for the problem's geometry.

To estimate the condition number of the current basis matrix, we implemented the *power method* in SoPlex. After repeated multiplication with A followed by a normalization, a randomly chosen vector converges to the largest eigenvector that can be used to get the corresponding eigenvalue. In a similar fashion, we can repeatedly apply A^{-1} to get the smallest eigenvalue. Typically, a few iterations are enough to converge and the computational overhead is relatively small due to the readily available LU-factorization of the basis matrix. Given that the condition number is merely providing a rough upper bound on the expected error when solving a linear system with this matrix, we do not require high accuracy and focus on the magnitude by inspecting the logarithmic value, instead.

We use TreeD to collect the condition numbers for all the simplex bases in a SCIP optimization run. Since TreeD utilizes the LP event handler of SCIP, it allows easy access to various LP statistics that can then be stored for later analysis. The LP event handler is a callback that is executed whenever an LP is solved and allows capturing the relevant data for this specific LP. In the interactive version of TreeD the user is able to choose those metrics to be displayed in the nodes of the tree visualization.

Discussing the condition of a *discrete* problem is in fact quite complicated. Formally, the slightest deviation of an integer variable from its value in an optimal (or feasible) solution will render the new solution infeasible. This is due to the solution set being an entirely discrete and hence discontinuous set. The classical concept of conditioning on the other side is tailored to continuous or smooth neighborhoods around the data points. Accordingly, we would have to set the condition number to ∞ rendering every single integer programming problem ill-conditioned. We refer to the work of Jarck (2020) for a detailed discussion on the topic.

This discrepancy is a stark contrast to the practical applicability of integer programming, so we retreat to inspecting condition numbers originating from LP relaxations of those problems. As large parts of the computational MILP solving process rely on LP solving as we have seen in Chapter 4, we believe that it is reasonable to do so.

Recently, developers of the FICO Xpress solver presented a *"numerical attention"* prediction based on machine learning[2]. The *attention level* α, that is estimated using this approach, is defined as follows:

$$\begin{aligned} \alpha &:= 0.01 * p_{sus} + 0.3 * p_{unstab} + p_{ill} \\ &\text{with} \\ p_{sus} &:= \text{percent of suspicious LP bases,} && 10^7 \leq \kappa < 10^{10} \\ p_{unstab} &:= \text{percent of unstable LP bases,} && 10^{10} \leq \kappa < 10^{14} \\ p_{ill} &:= \text{percent of ill-posed LP bases,} && 10^{14} \leq \kappa. \end{aligned} \tag{6.2}$$

The reasoning behind those classification ranges is explained in Section 6.6 when discussing different data types and how those influence the accuracy of our results.

[2] https://fico.force.com/FICOCommunity/s/blog-post/a5Q2E000000wvf6UAA/fico2388

This definition has first been published in a patent from IBM for CPLEX[3]. The concept of collecting various κ values during the MILP solving process is also known as *MIP-Kappa* or *MIP-*κ.

Both CPLEX and Xpress use the same definition of the attention level α and recommend users to be careful as soon as $\alpha > 0$. Since computing α can be expensive because many condition numbers need to be computed, a cheaper way of getting an idea of the numerical stability is certainly welcome. Furthermore, α is only available *after* the optimization has already been finished. The prediction takes into account several readily-available characteristics of the model data like coefficient ranges and bound sizes. Its quality is reportedly good enough to serve as a warning indicator: Models with a problematic α are reliably detected, while false positives, that is, instances that do not turn out to be numerically challenging, might still provoke a warning. Essentially, when the prediction is incorrect, in most cases it provides an unnecessary warning. This is not an issue for real-world applications and operations research practitioners.

6.2.1. Conditioning of the Simplex Algorithm

There are elaborate and quite complex definitions of the stability of a linear programming problem—independent of the used solving technique—as described by Bürgisser and Cucker (2013). These take the entire problem instance—including right-hand side and objective function—into account to provide a more general view on its numerical features.

We choose a simpler approach that relies only on the current basis matrix and the corresponding conventional condition number. This provides both a computationally feasible implementation as well as a reliable and expressive assessment of the actual numerical stability.

Figure 6.3 shows the condition number of each basis matrix encountered during the solving process of the initial LP relaxation and during each iteration of the re-solve occurring after adding a new round of cuts for selected instances from our test set.

One can observe that during the early iterations—especially of the initial relaxation in the root node—the condition numbers of the basis matrices grow quickly. This is expected, as more structural variables are pivoted into the basis, while slack variables are pivoted out. As the initial basis is typically the identity matrix which has a condition number of one, the conditioning can only degrade at first. We can conclude that during the simplex algorithm, condition numbers of the basis matrices grow quickly until the characteristic condition number magnitude is reached.

We expect the condition numbers of the basis matrices to degrade further as a result of operations performed in the root node and our initial computations are aimed at confirming this. After the initial optimization, the MILP solver adds cutting planes to the LP that are computed using the current basis matrix itself as explained in Sec-

[3] https://patents.google.com/patent/US8712738B2/

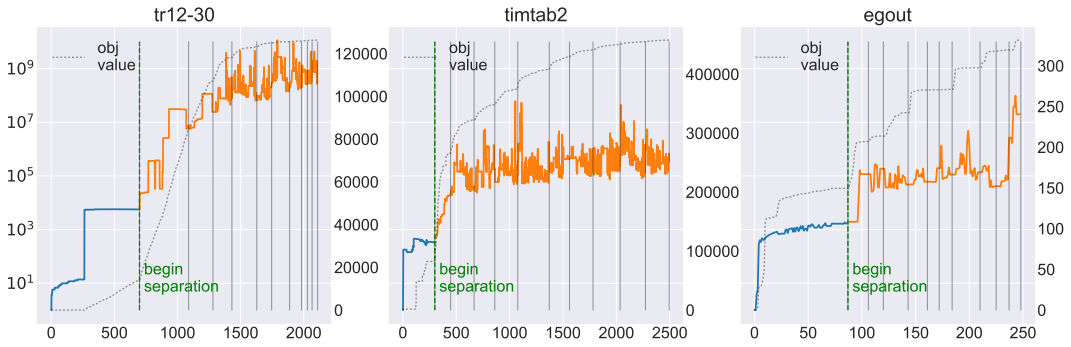


Figure 6.3.: Condition number development (vertical axis, in log scale) for every simplex iteration in the root node (horizontal axis) including re-optimizations after adding cutting planes in multiple rounds (vertical lines). A plot of objective values at each iteration is overlaid as a dashed gray line with the scales given to the right of each plot.

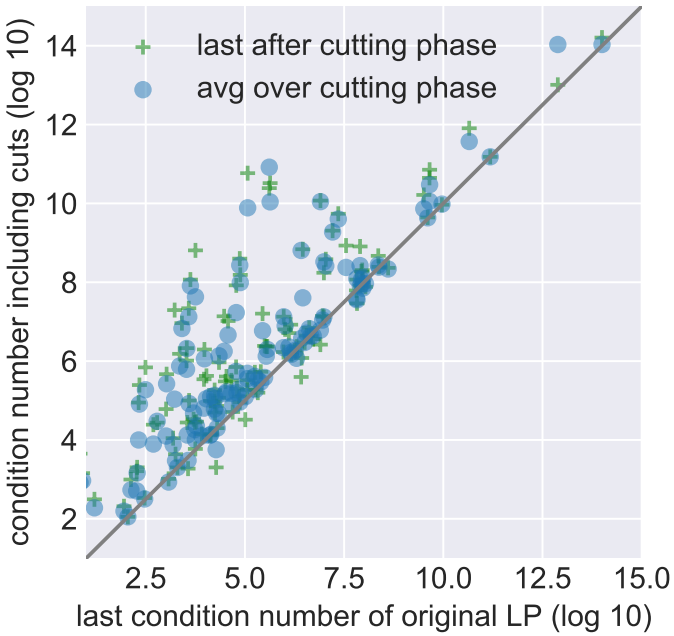


Figure 6.4.: Root node comparison of condition numbers of the original LP and after including cutting planes. Most instances show a higher condition number during the cutting stage and when this process is completed.

tion 4.5. Consequently, an ill-conditioned basis matrix may lead to an imprecise calculation of the coefficients of the new inequality.

Moreover, adding these new rows to the LP often further deteriorates its condition number as can be seen in Figure 6.3. This sample of MIPLIB instances clearly shows the expected behavior.

Figure 6.4 is a visualization of the difference between the condition number of the optimal basis of the original LP and two other metrics: (1) the average over all bases encountered during the cutting procedure and (2) the condition number of the final optimal basis. While for some instances there is a slight improvement after adding cuts, in most cases addition of cuts leads to the expected increased condition number.

Please note that these plots are taken from Miltenberger, Ralphs, and Steffy (2018) with an earlier version of the analysis code.

6.2.2. Condition Number Trend in the Tree

When inspecting condition numbers, there can be pretty large differences between individual instances and aggregating those numbers can easily be misleading. We demonstrate this in Figure 6.5 that shows the expected range of condition numbers for a specific tree depth and parameter setting. We use the breadth-first search node selection rule (see Section 4.4) to generate a more balanced tree.

Additionally, we adjust SCIP's separation emphasis parameter to compare runs with disabled cutting plane generation, runs with more aggressive separation, and the default cut setting. One might expect larger condition numbers when more cutting planes are added to the problem and smaller condition numbers when there are no additional cutting planes. However, this is not something we can deduce from this chart. It is apparent that cutting planes contribute to a smaller tree size and that condition numbers appear to decrease for LPs in deeper nodes of the tree.

The latter observation is flawed, though, because the number of instances that reach a certain tree depth is also diminishing accordingly. We are effectively comparing different sets of instances when descending the depth level in this chart.

An alternative might be to use a relative depth measure to normalize the different tree sizes. Unfortunately, this produces even more distorted results and prevents any sensible deduction or conclusions.

We actually need to inspect either individual instances to get a feeling for the condition number development or compute a trend per instance and then investigate similarities.

As we can see from Figures 6.6 and 6.7, the condition number trend in the branch-and-cut tree can be quite unpredictable and does not necessarily follow the intuitive expectation of more cutting planes implying larger condition numbers.

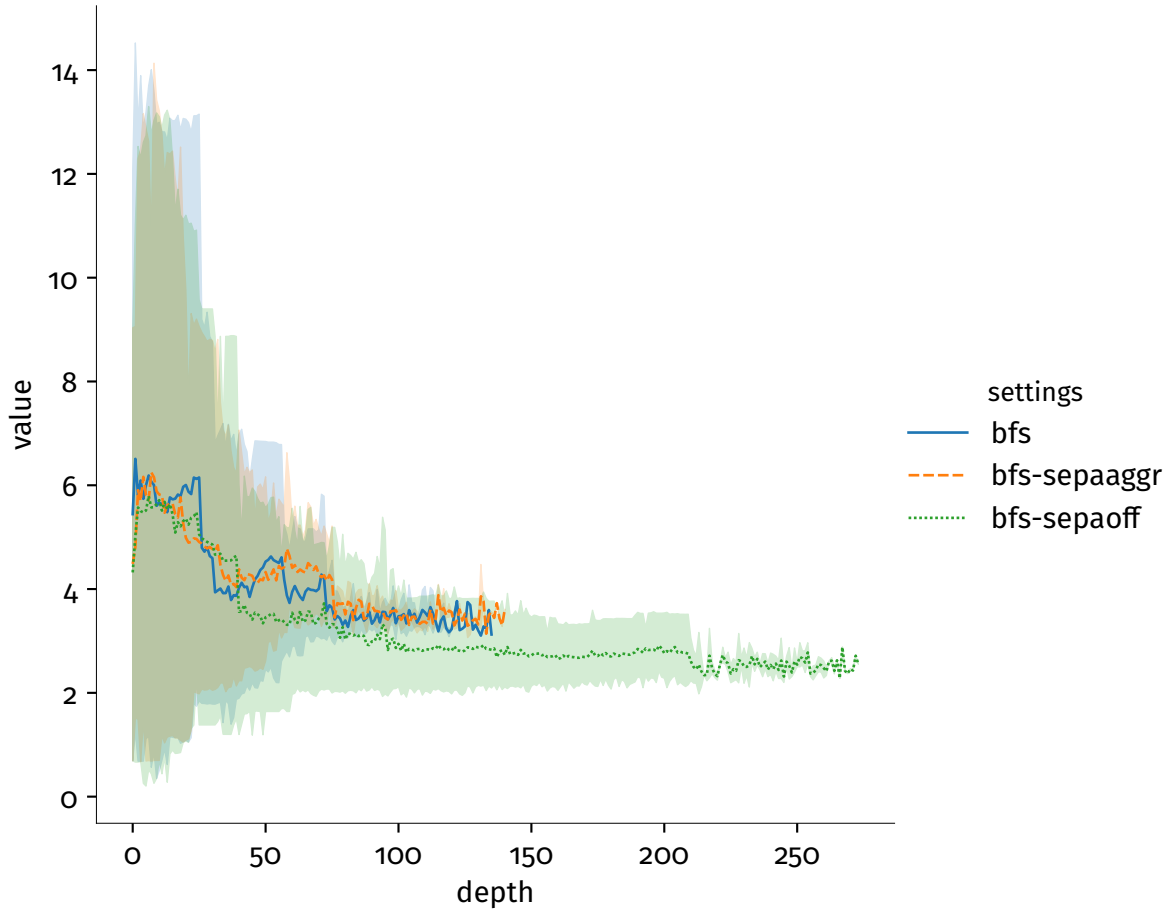


Figure 6.5.: Aggregated condition numbers (in log10 scale) for different tree depths. This chart contains data for all instances from the different MIPLIB benchmark sets that could be solved in one hour and that have at least reached a tree depth of 10. The line represents the mean value of condition numbers at this level while the shaded area shows the respective 95% confidence interval. Compared parameter sets are using breadth-first search with different cutting plane separation strategies (default, aggressive, and off).

6.3. Tailing-off Effect of Cutting Planes

The tailing-off effect of cutting planes describes the diminishing dual bound improvement of additional cutting planes during the separation procedure.

While it is possible to reach optimality with cutting planes alone, most solvers rely on further techniques like branching to reach optimality. Zanette, Fischetti, and Balas (2011) discuss this topic in more detail.

One of the main culprits for cuts having less impact over time is the fact that more

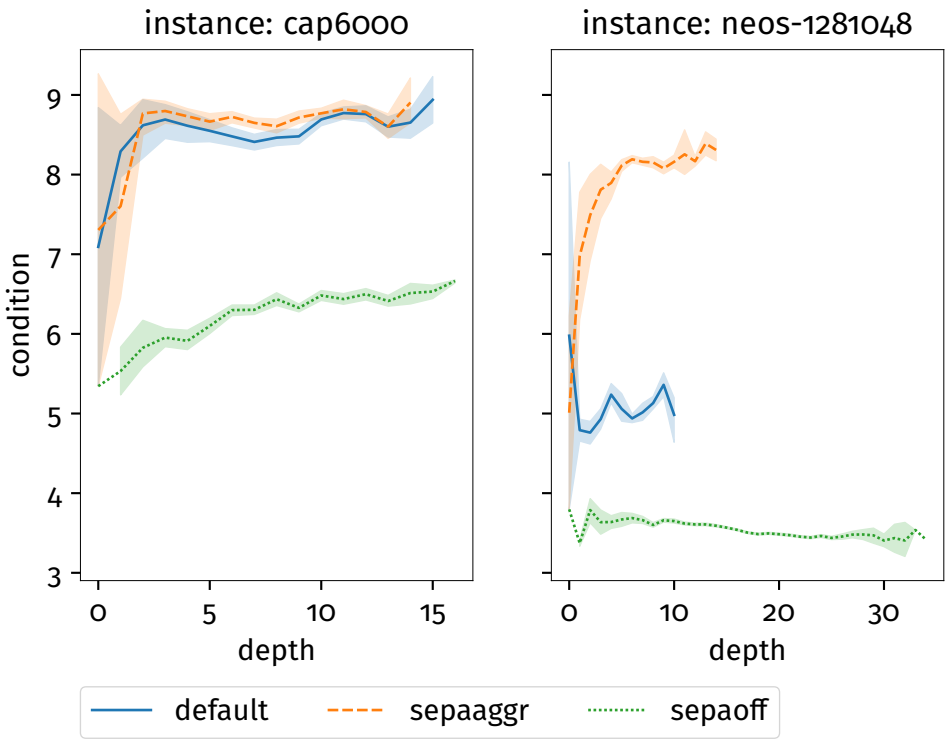


Figure 6.6.: Expected developments of condition numbers (in log10 scale): aggressive separation (sepaaggr) leads to an increased condition number across the entire tree while disabled cutting plane separation generates a larger tree but has smaller condition numbers.

cuts of higher rank have to be added and those tend to cause numerical difficulties. The rank of a cut describes whether the cut is coming directly from the LP formulation or is derived from a previous cutting round.

Intuitively, one can imagine this process as slowly shaving off the edges of the polyhedron making it harder for the simplex method to determine an optimal vertex among many very similar sub-optimal ones. Many almost parallel rows in the LP naturally lead to high condition numbers and may induce inaccurate results.

Following this train of thought, we seek to find a correlation between the stagnating bound improvement and an increased condition number of the corresponding basis matrices. In other words, we want to predict when the tailing-off phase starts by monitoring the geometry of the current LP polytope via the condition number. We hoped to find a holistic metric to be used as proxy for determining when no more cuts should be added.

Unfortunately, we have not been able to determine any kind of such a relationship in our experiments. Even disabling most of the advanced tricks of an MILP solver to get a clearer picture did not help to provide a better understanding of condition numbers

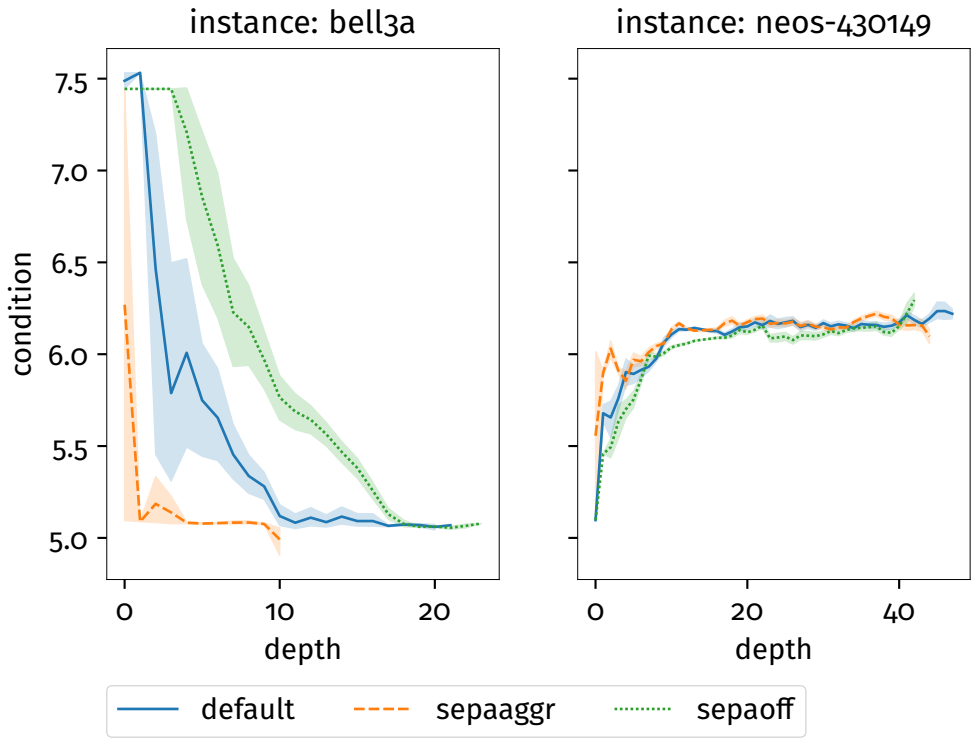


Figure 6.7.: Unexpected developments of condition numbers (in log10 scale): Disabling cutting plane generation (sepaoff) leads to worse condition numbers. The left instance exhibits decreasing condition numbers going down the tree while for the right instance, the numbers are growing.

for this context.

Furthermore, identifying the actual tailing-off effect can be difficult for many practical instances. Often, jumps in the bound improvement suggest that the tailing-off has not yet begun. This makes it hard to label the corresponding condition numbers to train a machine learning algorithm or even just to compute a regression model.

For general purpose MILP instances we are not optimistic that we can provide meaningful algorithmic control of the solver to detect tailing-off based on condition numbers. The currently used methods in SCIP of discontinuing the cutting phase, for example when detecting stalling, appear to be sufficient if not superior because of the reduced computational requirements: Inspecting the bound and gap improvement is significantly less expensive than computing condition numbers.

After investigating many different experiments, Prof. Ted Ralphs compared our attempts to predict and learn a pattern from condition numbers during the MILP solving process with *"doing a weather forecast for different planets"*, each planet resembling an individual problem instance.

In Figure 6.6 and Figure 6.7 we can see that the condition number trend in the

branch-and-cut tree can be very unexpected.

6.4. LP Condition Numbers for MILPs

The difficulty of defining a condition number for LPs mainly lies in the fact that there are so many different basis matrices that are encountered along the way and even neighboring bases may have significantly different condition numbers. We end up with even more possibilities when considering the additional degrees of freedom introduced by numerical tolerances.

In this section, we compute LP condition numbers κ_{LP} as defined in Ordonez and Freund (2003) for a range of MILPs and compare those numbers to other metrics like the attention level. We want to investigate if this *a priori* measurement can be used to provide additional information about the expected numerical features of the instances during the optimization process.

We follow the proposed approach of Ordonez and Freund (2003) in computing the this condition number measure $\kappa_{LP}(d)$

$$\kappa_{LP}(d) := \frac{\|d\|}{\min\{\rho_P(d), \rho_D(d)\}}$$

for a general LP d with

$$\begin{aligned} d := \quad \min \quad & c^\top x \\ \text{s.t.} \quad & A_{i.}x \le b_i, \text{ for } i \in L, \\ & A_{i.}x = b_i, \text{ for } i \in E, \\ & A_{i.}x \ge b_i, \text{ for } i \in G, \\ & x_j \ge 0 \text{ for } j \in L_B, \\ & x_j \le 0 \text{ for } j \in U_B. \end{aligned} \tag{6.3}$$

The components $\rho_P(d)$ and $\rho_D(d)$ are calculated as

$$\begin{aligned} \rho_P(d) = \min_{\substack{i\in\{1,\dots,m\} \\ j\in\{-1,1\}}} \min_{y,s^+,s^-,\nu} \max\left\{\left\|A^\top y + s^+ - s^-\right\|_1, \left|b^\top y - \nu\right|\right\} \\ \text{s.t. } y_i = j, \\ y_l \le 0 \text{ for } l \in L, \\ y_l \ge 0 \text{ for } l \in G, \\ s_k^- = 0 \text{ for } k \in N \setminus U_B, \\ s_k^+ = 0 \text{ for } k \in N \setminus L_B, \\ \nu + \sum_{k\in L_B} l_k s_k^+ - \sum_{k\in U_B} u_k s_k^- \ge 0, \\ s^+, s^- \ge 0 \end{aligned} \tag{6.4}$$

and

$$\begin{aligned}\rho_D(d) = \min_{\substack{i\in\{1,\dots,n\}\\ j\in\{-1,1\}}} \min_{x,p,g} \max\left\{\|Ax-p\|_1, \left|c^\top x+g\right|\right\}\\ \text{s.t. } x_i = j,\\ x_k \geq 0 \text{ for } k\in L_B,\\ x_k \leq 0 \text{ for } k\in U_B,\\ p_l \leq 0 \text{ for } l\in L,\\ p_l = 0 \text{ for } l\in E,\\ p_l \geq 0 \text{ for } l\in G,\\ g \geq 0.\end{aligned} \tag{6.5}$$

As we can see from the definitions of ρ_P and ρ_D, we need to solve $2n+2m$ LPs to compute these values: there are two LPs for every j and every i, spanning the number of rows and the number of columns. Since there is only a minor change between the individual LPs, we can warm-start the solving process from the previous iteration.

Furthermore, we need to compute the scaling value $\|d\|$ as:

$$\|d\| = \max\{\|A\|_{\infty,1}, \|b\|_1, \|c\|_1\}.$$

Here, the vector norm for b and c is chosen as the 1-norm $\|b\|_1 := \sum_{i=1}^{n} |b_i|$ while the matrix norm for A is

$$\|A\|_{\infty,1} := \max\{\|Ax\|_1 \text{ with } \|x\|_\infty = 1\}.$$

To avoid the expensive computation of this norm, we follow the approach of Ordonez and Freund (2003) and employ lower and upper bounds instead as

$$\max\{\|A\|_{1,1}, \|A\|_{2,2}, \|A\|_F, \|Ae\|_1, \|A\tilde{a}\|_1\} \leq \|A\|_{\infty,1} \leq \min\left\{\|A\|_{L_1}, \sqrt{nm}\|A\|_{2,2}\right\}$$

where e is the vector of ones and $\tilde{a}_j := \text{sign}(A_{i^*,j})$ with $i^* := \text{argmax}_{i=1,\dots,m}\|A_{i.}\|_1$.

The amount of work to compute κ_{LP} for any given LP or MILP is significant and we cannot hope to use this approach in practice for reasonably sized problems.

Nevertheless, we hoped to predict or estimate the numerical difficulties solvers might encounter—as the plots of the collected data in Figure 6.8 show, there is no identifiable trend or correlation between the different metrics. We include runs with both CPLEX and XPRESS to provide more solver-agnostic results.

It appears there is no clear connection between (the logarithm of) this *a priori* computed stability measure κ_{LP} and those that can be extracted during or after the solving process. Neither the attention level, that is, the aggregation of all κ values, nor the maximal κ itself relate to the κ_{LP} value. We can also see that there is a fairly large number of instances with an infinite κ_{LP} value, here represented by those values $\geq 10^{20}$ while the largest condition numbers in the tree are almost always smaller than 10^{15} and often a lot smaller. We can also spot this discrepancy in the comparison with the attention level that is also not signalling a numerically difficult model. This

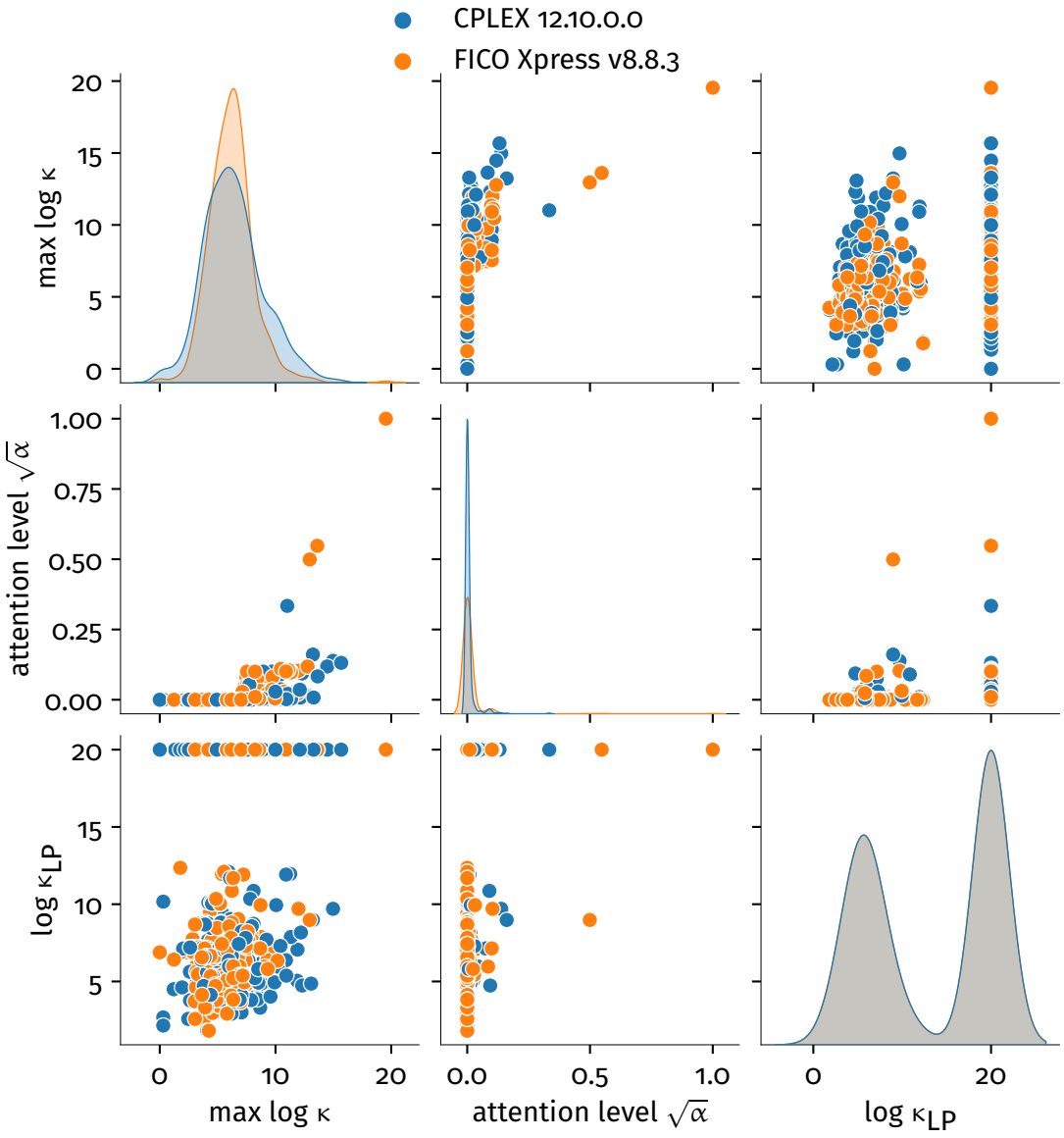


Figure 6.8.: Statistics of the κ_{LP} values and the tree condition numbers for various MILPs. Note that the κ_{LP} values are in each case the lower bounds from the estimation. The lower right subplot shows the κ_{LP} distribution which is independent of the solver.

once again demonstrates how hard it is to grasp and understand numerical issues that occur in MILP solvers. Table B.4 lists all our collected κ_{LP} values as well as the largest encountered condition numbers in the tree and the attention level as defined in Equation (6.2).

The code to reproduce these computations is available on GitHub[4], both as updated version of the original code from Ordonez and Freund (2003) using CPLEX and as Python version using Gurobi as LP solver.

6.5. Geometry of the Polyhedron and its Impact on Branching

The geometry of the polyhedron, that is, whether it is very thin in a few directions can make a difference in how many branch-and-bound nodes are necessary to find the optimal solution. Here, it can also be useful to know the thin directions and focus on the corresponding variable axis when deciding how to branch. This relation has been investigated by Derpich and Vera (2006) with somewhat promising results. By intuition it seems clear that one should incorporate the *thinness* of the polyhedron to avoid branching on directions that may generate a larger number of nodes. On the other hand, we also need to keep track of the remaining branching criteria so only a combined strategy can be successful.

Krishnamoorthy (2017) constructed counter examples to show that our intuition fails us when it comes to branching in thin directions and Mahajan and Ralphs (2010) proved that finding the best direction is an $\mathcal{NP}$-hard problem itself.

What might be interesting to investigate is whether problems with a rather thin feasible region are more difficult to solve than those with a more uniform shape.

6.6. Exact MILP Solving

There have also been multiple attempts to mitigate the numerical issues of traditional solvers based on double precision arithmetic and implement exact rational MILP solvers, instead.

Typically, computers use a floating point representation for numbers. Computer codes can make use of different data types that determine the accuracy of this digital representation and a short comparison of some of those is given in Table 6.1.

A consequence of this limited binary representation of an uncountably infinite set of numbers is that there are numbers, for example $1/3$, that cannot be represented accurately. Instead, the digital number may deviate from the exact value up to the epsilon value of the chosen data type.

This is also why computations with the prevalent `double` data type are using a tolerance value of about 10^{-6}: As long as our condition numbers remain well below a value of 10^{10} we can expect accurate results of up to a relative error of 10^{-6}. In other words, unavoidable double precision errors of magnitude 10^{-16} are amplified up to

[4] `https://github.com/mattmilten/condition`

data type	number of bits	mantissa	exponent	epsilon
`float`	32	23	8	$2^{-23} \approx 1 \times 10^{-7}$
`double`	64	52	11	$2^{-52} \approx 2 \times 10^{-16}$
`__float80`	80	63	15	$2^{-63} \approx 1 \times 10^{-19}$
`__float128`	128	112	15	$2^{-112} \approx 2 \times 10^{-34}$

Table 6.1.: Different accuracies of floating point representations and their respective number of bits in mantissa and exponent as defined in the IEEE standard 754.

10^{-6} for computations that involve a condition number of 10^{10}. That is the reason for the attention level classification of condition number ranges in Equation (6.2).

In addition to the floating point data types in Table 6.1, SoPlex has the ability to use rational arithmetic throughout the most important parts of the solver and can also make use of the much faster iterative refinement technique presented in Section 3.9. In combination with the rational extension of SCIP that was first developed by Cook et al. (2013) and then further improved and refined by Eifler and Gleixner (2021) very impressive speedups could be achieved. This makes exact rational MILP solving a viable alternative when faced with instances that cannot be solved to satisfactory results with conventional codes. Most commercial MILP solvers like Gurobi, or Xpress have options to switch to a higher precision arithmetic mode, often called *quad precision* using the `__float128` data type. They do not implement a rational arithmetic, though.

Unfortunately, the use of rational arithmetic comes with a significant computational overhead because most CPU chips are not optimized to perform these operations efficiently, in sharp contrast to their double precision counterparts. This makes it often impractical for real-world applications and a numerically careful and stable implementation remains unavoidable.

6.7. Outlook and Future Work

We want to stress the exploratory nature of the results of this chapter. We believe that the fascinating topic of numerics in LP and MILP solvers is far from covered and well-understood but our computational results and analysis provide another useful contribution to guide and inspire future research.

Currently, the best practice is to inspect conspicuous models individually and trying to understand how to treat or circumvent numerical issues. Firstly, the main question is whether the bounds, right-hand sides, and matrix and objective coefficients are all in some reasonable and confined range. Secondly, we can analyze more complicated features that for example take into account certain condition number measures. Still, careful modeling will remain key in avoiding numerically difficult instances yielding

questionable results.

From working with the large-scale supply network planning instances from our industry partner SAP (see Gamrath, Gleixner, et al., 2019), we realized a rather obscure numerical detail: The infinity value of SCIP (all larger values are treated as infinity) had to be increased from 10^{20} to 10^{30} to account for the huge solution values that otherwise triggered incorrect results. Furthermore, numerical tolerances frequently allowed small violations in the presolved space and got amplified in the original formulation, effectively rendering the solution to be infeasible. In those cases, expectations regarding the interpretability or correctness of the reported results need to be adjusted accordingly.

As mentioned earlier, also a very stable algorithm cannot prevent numerical issues stemming from a high condition number because of an unfavorable formulation. A solution that is applicable to all kinds of problems is not likely to be found without sacrificing a lot of performance by using exact arithmetic during the computations.

At the moment, the most promising direction is to implement reliable warning messages that alert users when a solution may not be fully trusted. Ideally, these warnings should be accompanied by hints and suggestions pointing to the root causes and how to avoid them in the future. Klotz (2014) provides an excellent practical guide about this topic. Machine learning techniques could be a suitable tool to help with these tasks as proposed by FICO Xpress blog post mentioned before and by Berthold and Hendel (2021).

In addition, we need to keep looking for even more stable and numerically robust algorithms and implementations as well as to develop new measures to analyze our model instances and to enhance the prediction of their behavior during the solving process.

Chapter 7

Conclusion

Linear programming is the central theme of this dissertation—from simplex techniques to numerical experiments to LP-based visualizations of MILP solving.

In this thesis we presented various ways how LP solving influences and impacts the MILP solver SCIP. We gave an overview of the implementational details of the simplex solver SoPlex and explained how the MILP performance can be improved by treating the LP solver less like a black box.

With LP solution polishing in Chapter 5, we demonstrated an efficient way to exploit degeneracy effects during the optimization of LP relaxations to return less fractional variable assignments in the computed solutions. Due to the abundance of degenerate LP relaxations in general MILPs, this technique could be applied frequently without introducing a detrimental overhead. The reduced fractionality of these polished LP solutions provided a decent performance benefit for SCIP. We also discussed how further insight from the additional simplex iterations might be useful for other components of an MILP solver.

In this regard, SCIP's restricting interface to the LP solver can be detrimental to similar algorithmic developments as it always requires a certain overhead and additional implementational effort. On the other hand, SCIP's modularity allowed for many of our investigations and experiments in the first place. Our experiments in Chapter 4 compared the performance of the available LP solvers in SCIP and how this can impact important metrics like the root gap and the node throughput.

We also learnt that persistent scaling is causing a significant positive performance impact—something quite unexpected but very welcome. The initial goal of implementing this advanced scaling feature was merely to improve the numerical stability and to reduce the number of rejected solutions containing too large violations. This also demonstrated that numerical features are important to consider when trying to improve the sheer performance of a solver.

Unfortunately, regardless of our best efforts to push the general MILP performance using some of the simplex additions and improvements presented in Chapter 3, like the bound flipping ratio test or sparsity exploitation techniques, we could only achieve a measurable effect on LP models or on selected MILP instances. In Chapter 4 we also

demonstrated the often unpredictable nature of MILP solving and how the LP solver influences the overall MILP performance of SCIP.

Going back to our initial conjecture of *"LP not mattering for MILP"*, we showed that it is not just about the speed of an LP solver but also, and maybe more importantly, how stable and reliable it is implemented and integrated into the MILP framework. Furthermore, our numerous experiments showed that the fraction of time spent in the LP solver is very similar across most solver implementations. This indicated that even a considerable performance difference between them did not carry over as pronounced to the MILP performance.

We are certain that the competitive landscape in the field of mathematical optimization is going to keep on changing and adapting to new technologies and new research advancements. We also tried to capture and honor this remarkable progress in our work presented in Appendix A.

We hope that our experiments concerning numerical features of LP and MILP solving shined a light on this topic and helped to gain a better understanding of the effects of branching and cutting. Despite the somewhat inconclusive results in Chapter 6, a very positive and successful outcome has been the TreeD project. TreeD provided interesting visualizations of MILP solving trees and can also conveniently be used as an analytics framework to collect various data during the solving process.

With TreeD, we also demonstrated the practicality of PySCIPOpt and showed how new tools can be built and distributed on the provided framework and foundation that are presented in Chapter 2. The PySCIPOpt project proved to be a reliable and welcome addition to the SCIP Optimization Suite and has since been used in numerous academic projects. This has been an important step in making the field of mathematical programming and optimization more accessible to a wider audience who may not be willing to dive into the technical details of SCIP.

Concerning the implications of the numerical experiments in Chapter 6, we are still confident that incorporating the condition number into algorithmic decisions in the MILP solver can be beneficial—we may just have to find the right questions to ask in this regard. We are expecting further developments in this field of interest and are convinced that our contributions can be useful to provide a better understanding of numerical features in MILP solvers.

Bibliography

Please note that the numbers in brackets refer to the page of this thesis that cites the respective work.

Achterberg, T. "Constraint Integer Programming". PhD thesis. Technische Universität Berlin, 2007 [75, 102].

– *LP relaxation modification and cut selection in a MIP solver*. US Patent 8,463,729. 2013 [92, 93, 97].

– "SCIP: Solving constraint integer programs". In: *Mathematical Programming Computation* 1.1 (2009), pp. 1–41 [19, 96].

Achterberg, T. and T. Berthold. "Hybrid Branching". In: *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, 6th International Conference, CPAIOR 2009*. Ed. by W. J. van Hoeve and J. N. Hooker. Vol. 5547. Lecture Notes in Computer Science. Springer, 2009, pp. 309–311 [17].

Achterberg, T., T. Koch, and A. Martin. "Branching rules revisited". In: *Operations Research Letters* 33.1 (2004), pp. 42–54 [17, 46, 72–74].

– "MIPLIB 2003". In: *Operations Research Letters* 34.4 (2006), pp. 1–12. DOI: 10.1016/j.orl.2005.07.009 [22].

Achterberg, T. and R. Wunderling. "Mixed Integer Programming: Analyzing 12 Years of Progress". In: *Facets of Combinatorial Optimization: Festschrift for Martin Grötschel*. Ed. by M. Jünger and G. Reinelt. Springer Berlin Heidelberg, 2013, pp. 449–481. DOI: 10.1007/978-3-642-38189-8_18 [6].

Anderson, E. D., J. Gondzio, C. Mészáros, and X. Xu. "Implementation of interior-point methods for large scale linear programs". In: *Interior Point Methods of Mathematical Programming*. Springer, 1996, pp. 189–252 [55].

Applegate, D. L., R. E. Bixby, V. Chvátal, and W. J. Cook. *Finding cuts in the TSP (A preliminary report)*. Tech. rep. 95-05. Center for Discrete Mathematics & Theoretical Computer Science (DIMACS), 1995 [73].

– *The traveling salesman problem: a computational study*. Princeton university press, 2006 [16].

Basu, A., M. Conforti, M. Di Summa, and H. Jiang. "Complexity of branch-and-bound and cutting planes in mixed-integer optimization". In: *Mathematical Programming* (2022), pp. 1–24. DOI: 10.1007/s10107-022-01789-5 [77].

Beale, E. "An alternative method for linear programming". In: *Mathematical Proceedings of the Cambridge Philosophical Society*. Vol. 50. 4. Cambridge University Press. 1954, pp. 513–523 [7].

Bénichou, M., J.-M. Gauthier, P. Girodet, G. Hentges, G. Ribière, and O. Vincent. "Experiments in mixed-integer programming". In: *Mathematical Programming* 1 (1971), pp. 76–94 [72].

Berthold, T. "Heuristic algorithms in global MINLP solvers". PhD thesis. TU Berlin, 2014, p. 366 [17, 79, 98].

Berthold, T., G. Gamrath, and D. Salvagnin. *Exploiting Dual Degeneracy in Branching*. Tech. rep. 19-17. Berlin: ZIB, 2019 [92].

Berthold, T. and G. Hendel. "Learning To Scale Mixed-Integer Programs". In: *Proceedings of the AAAI Conference on Artificial Intelligence* 35.5 (2021), pp. 3661–3668 [121].

Berthold, T., M. Perregaard, and C. Mészáros. "Four good reasons to use an interior point solver within a MIP solver". In: *Operations Research Proceedings 2017*. Springer, 2018, pp. 159–164 [68].

Berthold, T. and D. Salvagnin. *Cloud branching*. Tech. rep. 13-01. Berlin: ZIB, 2013 [17].

Bezanson, J., A. Edelman, S. Karpinski, and V. B. Shah. "Julia: A Fresh Approach to Numerical Computing". In: *SIAM Review* 59.1 (2017), pp. 65–98. DOI: 10.1137/141000671 [30].

Bixby, E. R., M. Fenelon, Z. Gu, E. Rothberg, and R. Wunderling. "MIP: Theory and Practice — Closing the Gap". In: *System Modelling and Optimization*. Ed. by M. J. D. Powell and S. Scholtes. Boston, MA: Springer US, 2000, pp. 19–49 [6].

Bixby, R. E. "A brief history of linear and mixed-integer programming computation". In: *Documenta Mathematica* (2012), pp. 107–121 [7].

Bixby, R. E., E. A. Boyd, and R. R. Indovina. "MIPLIB: A test set of mixed integer programming problems". In: *Siam News* 25.2 (1992), p. 16 [22].

Bixby, R. E., S. Ceria, C. M. McZeal, and M. W. P. Savelsbergh. "An Updated Mixed Integer Programming Library: MIPLIB 3.0". In: *Optima* 58 (1998), pp. 12–15 [22].

Borg, I. and P. J. Groenen. *Modern multidimensional scaling: Theory and applications*. Springer Science & Business Media, 2005 [80].

Boyd, S., N. Parikh, E. Chu, B. Peleato, and J. Eckstein. "Distributed Optimization and Statistical Learning via the Alternating Direction Method of Multipliers". In: *Foundations and Trends in Machine Learning* 3 (2011), pp. 1–122. DOI: 10.1561/2200000016 [74].

Bürgisser, P. and F. Cucker. *Condition - The Geometry of Numerical Algorithms*. Vol. 349. Grundlehren der math. Wissenschaften. Springer, 2013. DOI: 10.1007/978-3-642-38896-5 [103–105, 110].

Bynum, M. L., G. A. Hackebeil, W. E. Hart, C. D. Laird, B. L. Nicholson, J. D. Siirola, J.-P. Watson, and D. L. Woodruff. *Pyomo–optimization modeling in python*. Third. Vol. 67. Springer Science & Business Media, 2021 [30].

Cao, K.-K., A. M. Gleixner, and M. Miltenberger. "Methoden zur Reduktion der Rechenzeit linearer Optimierungsmodelle in der Energiewirtschaft - Eine Performance–Analyse". In: *EnInnov 2016: 14. Symposium Energieinnovation 2016*. 2016 [4, 6].

Cipra, B. A. "The best of the 20th century: Editors name top 10 algorithms". In: *SIAM news* 33.4 (2000), pp. 1–2 [5].

Conforti, M., G. Cornuéjols, and G. Zambelli. *Integer Programming*. Springer, 2014. DOI: 10.1007/978-3-319-11008-0 [76].

Cook, W., T. Koch, D. E. Steffy, and K. Wolter. "A hybrid branch-and-bound approach for exact rational mixed-integer programming". In: *Mathematical Programming Computation* 5.3 (2013), pp. 305–344 [120].

Cook, W. J. *In pursuit of the traveling salesman*. Princeton University Press, 2011 [7].

Curtis, A. R. and J. K. Reid. "On the automatic scaling of matrices for Gaussian elimination". In: *IMA Journal of Applied Mathematics* 10 (1972), pp. 118–124 [54].

Dadush, D. and S. Huiberts. "A friendly smoothed analysis of the simplex method". In: *SIAM Journal on Computing* 49.5 (2019), STOC18–449 [47].

Dalcin, L., R. Bradshaw, K. Smith, C. Citro, S. Behnel, and D. Seljebotn. "Cython: The Best of Both Worlds". In: *Computing in Science & Engineering* 13.02 (2011), pp. 31–39. DOI: 10.1109/MCSE.2010.118 [25].

Danna, E. "Performance variability in mixed integer programming." Talk at Workshop on Mixed Integer Programming 2008, Columbia University, New York, USA. 2008 [59].

Dantzig, G. and D. R. Fulkerson. "On the max flow min cut theorem of networks". In: *Linear inequalities and related systems* 38 (2003), pp. 225–231 [16].

Dantzig, G. B. *Origins of the simplex method*. Tech. rep. Stanford Univ CA Systems Optimization Lab, 1987 [7].

– "The diet problem". In: *Interfaces* 20.4 (1990), pp. 43–47 [8].

Dantzig, G. B. and W. Orchard-Hays. "The product form for the inverse in the simplex method". In: *Mathematical Tables and Other Aids to Computation* (1954), pp. 64–67 [7, 57].

Derpich, I. and J. R. Vera. “Improving the efficiency of the Branch and Bound algorithm for integer programming based on "flatness" information”. In: *European Journal of Operational Research* 174.1 (2006), pp. 92–101. DOI: 10.1016/j.ejor.2005.02.051 [119].

Desaulniers, G., J. Desrosiers, and M. M. Solomon. *Column generation*. Vol. 5. Springer Science & Business Media, 2006 [38].

Dey, S. S., Y. Dubey, M. Molinaro, and P. Shah. *A Theoretical and Computational Analysis of Full Strong-Branching*. Tech. rep. 8642. Optimization Online, 2021 [74].

Dinh, T., R. Fukasawa, and J. Luedtke. “Exact algorithms for the chance-constrained vehicle routing problem”. In: *Mathematical Programming* 172.1 (2018), pp. 105–138 [84].

Dunning, I., J. Huchette, and M. Lubin. “JuMP: A modeling language for mathematical optimization”. In: *SIAM Review* 59.2 (2017), pp. 295–320 [30].

Eifler, L. and A. M. Gleixner. “A Computational Status Update for Exact Rational Mixed Integer Programming”. In: *Integer Programming and Combinatorial Optimization*. Springer International Publishing, 2021, pp. 163–177 [120].

Elble, J. M. and N. V. Sahinidis. “A review of the LU update in the simplex algorithm”. In: *International Journal of Mathematics in Operational Research* 4.4 (2012), pp. 366–399 [57].

– “Scaling linear optimization problems prior to application of the simplex method”. In: *Computational Optimization and Applications* 52.2 (2012), pp. 345–371. DOI: 10.1007/s10589-011-9420-4 [54].

Elhallaoui, I., A. Metrane, G. Desaulniers, and F. Soumis. “An Improved Primal Simplex Algorithm for Degenerate Linear Programs”. In: *INFORMS Journal on Computing* 23.4 (2011), pp. 569–577. DOI: 10.1287/ijoc.1100.0425 [58].

Epelman, M. and R. Freund. “A New Condition Measure, Preconditioners, and Relations Between Different Measures of Conditioning for Conic Linear Systems”. In: *SIAM Journal on Optimization* 12.3 (2002), pp. 627–655. DOI: 10.1137/S1052623400373829 [104].

Espinoza, D. G. “On linear programming, integer programming and cutting planes”. PhD thesis. Georgia Institute of Technology, 2006 [57].

Farkas, J. “Theorie der einfachen Ungleichungen.” In: *Journal für die reine und angewandte Mathematik (Crelles Journal)* 1902.124 (1902), pp. 1–27. DOI: doi:10.1515/crll.1902.124.1 [10].

FICO. *FICO Xpress Optimization Suite*. http://www.fico.com/en/products/fico-xpress-optimization-suite. 2021 [16, 21].

Fischetti, M., A. Lodi, M. Monaci, D. Salvagnin, and A. Tramontani. “Improving branch-and-cut performance by random sampling”. In: *Mathematical Programming Computation* 8.1 (2016), pp. 113–132. DOI: 10.1007/s12532-015-0096-0 [92].

Forrest, J. J. and D. Goldfarb. “Steepest-edge simplex algorithms for linear programming”. In: *Mathematical Programming* 57.1 (1992), pp. 341–374. DOI: 10.1007/BF01581089 [46].

Forrest, J. J., S. Vigerske, T. Ralphs, L. Hafer, H. G. Santos, M. Saltzman, B. Kristjansson, and A. King. *coin-or/Clp: Version 1.17.6*. Version releases/1.17.6. 2020. DOI: 10.5281/zenodo.3748677 [21].

Forrest, J. J., S. Vigerske, H. G. Santos, T. Ralphs, L. Hafer, B. Kristjansson, jpfasano, EdwinStraver, M. Lubin, rlougee, jpgoncal1, h-i-gassmann, and M. Saltzman. *coin-or/Cbc: Version 2.10.5*. 2020. DOI: 10.5281/zenodo.3700700 [17, 21].

Forrest, J. J. and J. A. Tomlin. “Updated triangular factors of the basis to maintain sparsity in the product form simplex method”. In: *Mathematical programming* 2.1 (1972), pp. 263–278 [57].

Fourer, R. *Notes on the dual simplex method*. Draft report 9. Northwestern University, 1994 [41].

Fourier, J.-B. J. “Analyse des travaux de l’Académie Royale des Sciences, pendant l’année 1824, Partie mathématique, Histoire de l’Academie Royale des Sciences de l’Institute de France. 1827. Lal George and Andrew W. Appel. Iterated register coalescing”. In: *ACM Trans. Program. Lang. Syst* 18 (1827), pp. 300–324 [7].

Galabova, I. L. and J. Hall. “The "Idiot" crash quadratic penalty algorithm for linear programming and its application to linearizations of quadratic assignment problems”. In: *Optimization Methods and Software* 35.3 (2020), pp. 488–501 [40].

Gamrath, G. “Improving strong branching by domain propagation”. In: *EURO Journal on Computational Optimization* 2.3 (2014), pp. 99–122 [73].

Gamrath, G., D. Anderson, K. Bestuzheva, W.-K. Chen, L. Eifler, M. Gasse, P. Gemander, A. M. Gleixner, L. Gottwald, K. Halbig, G. Hendel, C. Hojny, T. Koch, P. L. Bodic, S. J. Maher, F. Matter, M. Miltenberger, E. Mühmer, B. Müller, M. Pfetsch, F. Schlösser, F. Serrano, Y. Shinano, C. Tawfik, S. Vigerske, F. Wegscheider, D. Weninger, and J. Witzig. *The SCIP Optimization Suite 7.0*. Tech. rep. 20-10. Berlin: ZIB, 2020 [5].

Gamrath, G., T. Berthold, and D. Salvagnin. “An exploratory computational analysis of dual degeneracy in mixed-integer programming”. In: *EURO Journal on Computational Optimization* 8 (2020), pp. 241–246. DOI: 10.1007/s13675-020-00130-z [17, 92].

Gamrath, G., T. Fischer, T. Gally, A. M. Gleixner, G. Hendel, T. Koch, S. J. Maher, M. Miltenberger, B. Müller, M. E. Pfetsch, C. Puchert, D. Rehfeldt, S. Schenker, R. Schwarz, F. Serrano, Y. Shinano, S. Vigerske, D. Weninger, M. Winkler, J. T. Witt, and J. Witzig. *The SCIP Optimization Suite 3.2*. Tech. rep. 15-60. Berlin: ZIB, 2016 [5, 48].

Gamrath, G., A. M. Gleixner, T. Koch, M. Miltenberger, D. Kniasew, D. Schlögel, A. Martin, and D. Weninger. "Tackling Industrial-Scale Supply Chain Problems by Mixed-Integer Programming". In: *Journal of Computational Mathematics* 37 (2019), pp. 866–888. DOI: 10.4208/jcm.1905-m2019-0055 [4, 48, 50, 121].

Gamrath, G., T. Koch, A. Martin, M. Miltenberger, and D. Weninger. "Progress in presolving for mixed integer programming". In: *Mathematical Programming Computation* 7.4 (2015), pp. 367–398. DOI: 10.1007/s12532-015-0083-5 [50].

Gamrath, G. and M. E. Lübbecke. "Experiments with a Generic Dantzig-Wolfe Decomposition for Integer Programs". In: *Experimental Algorithms*. Ed. by P. Festa. Vol. 6049. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2010, pp. 239–252. DOI: 10.1007/978-3-642-13193-6_21 [19].

Gamrath, G. and C. Schubert. "Measuring the impact of branching rules for mixed-integer programming". In: *Operations Research Proceedings 2017*. 2018, pp. 165–170. DOI: 10.1007/978-3-319-89920-6_23 [71].

Gleixner, A. M. "Exact and Fast Algorithms for Mixed-Integer Nonlinear Programming". PhD thesis. Technische Universität Berlin, 2015 [15, 57].

– *Factorization and update of a reduced basis matrix for the revised simplex method*. Tech. rep. 12-36. Berlin: ZIB, 2012 [36].

Gleixner, A. M., G. Hendel, G. Gamrath, T. Achterberg, M. Bastubbe, T. Berthold, P. M. Christophel, K. Jarck, T. Koch, J. Linderoth, M. Lübbecke, H. D. Mittelmann, D. Ozyurt, T. K. Ralphs, D. Salvagnin, and Y. Shinano. "MIPLIB 2017: Data-Driven Compilation of the 6th Mixed-Integer Programming Library". In: *Mathematical Programming Computation* (2021). DOI: 10.1007/s12532-020-00194-3 [22].

Gleixner, A. M., D. E. Steffy, and K. Wolter. "Improving the Accuracy of Linear Programming Solvers with Iterative Refinement". In: *ISSAC '12. Proceedings of the 37th International Symposium on Symbolic and Algebraic Computation*. ACM, 2012, pp. 187–194. DOI: 10.1145/2442829.2442858 [57].

– *Iterative Refinement for Linear Programming*. INFORMS Journal on Computing. 2016 [57].

Gleixner, A. M., M. Bastubbe, L. Eifler, T. Gally, G. Gamrath, R. L. Gottwald, G. Hendel, C. Hojny, T. Koch, M. Lübbecke, S. J. Maher, M. Miltenberger, B. Müller, M. E. Pfetsch, C. Puchert, D. Rehfeldt, F. Schlösser, C. Schubert, F. Serrano, Y. Shinano, J. M. Viernickel, M. Walter, F. Wegscheider, J. T. Witt, and J. Witzig. *The SCIP Optimization Suite 6.0*. Tech. rep. 18-26. Berlin: ZIB, 2018 [5].

Gleixner, A. M., L. Eifler, T. Gally, G. Gamrath, P. Gemander, R. L. Gottwald, G. Hendel, C. Hojny, T. Koch, M. Miltenberger, B. Müller, M. E. Pfetsch, C. Puchert, D. Rehfeldt, F. Schlösser, F. Serrano, Y. Shinano, J. M. Viernickel, S. Vigerske, D. Weninger, J. T. Witt, and J. Witzig. *The SCIP Optimization Suite 5.0*. Tech. rep. 17-61. Berlin: ZIB, 2017 [5, 54].

Gould, N. and J. Scott. “A note on performance profiles for benchmarking software”. In: *ACM Transactions on Mathematical Software (TOMS)* 43.2 (2016), pp. 1–5 [84].

Grötschel, M. *The sharpest cut: The impact of Manfred Padberg and his work*. SIAM, 2004 [77].

Gurobi Optimization, LLC. *Gurobi Optimizer Reference Manual*. `http://www.gurobi.com`. 2021 [16, 21].

Hadamard, J. “Sur les problèmes aux dérivées partielles et leur signification physique”. In: *Princeton university bulletin* (1902), pp. 49–52 [103].

Hager, W. W. “Condition estimates”. In: *SIAM Journal on scientific and statistical computing* 5.2 (1984), pp. 311–316 [103].

Hall, J. A. J. and K. I. M. McKinnon. “Hyper-Sparsity in the Revised Simplex Method and How to Exploit it”. In: *Computational Optimization and Applications* 32.3 (2005), pp. 259–283. DOI: `10.1007/s10589-005-4802-0` [48, 55].

Harris, P. M. “Pivot selection methods of the Devex LP code”. In: *Mathematical programming* 5.1 (1973), pp. 1–28 [44, 47].

Hart, W. E., J.-P. Watson, and D. L. Woodruff. “Pyomo: modeling and solving mathematical programs in Python”. In: *Mathematical Programming Computation* 3.3 (2011), pp. 219–260 [30].

Hendel, G. *IPET Interactive Performance Evaluation Tools*. `http://github.com/GregorCH/ipet`. 2021 [22].

Hendel, G., M. Miltenberger, and J. Witzig. “Adaptive Algorithmic Behavior for Solving Mixed Integer Programs Using Bandit Algorithms”. In: *OR 2018: International Conference on Operations Research*. 2018 [47].

Hestenes, M. R. and E. Stiefel. *Methods of conjugate gradients for solving linear systems*. Vol. 49. 1. NBS Washington, DC, 1952 [54].

Hofmann, H., H. Wickham, and K. Kafadar. “value plots: Boxplots for large data”. In: *Journal of Computational and Graphical Statistics* 26.3 (2017), pp. 469–477 [89].

Huang, M., Y. Zhong, H. Yang, J. Wang, F. Zhang, B. Bai, and L. Shi. *Simplex Initialization: A Survey of Techniques and Trends*. 2021 [40].

Huangfu, Q. and J. J. Hall. “Novel update techniques for the revised simplex method”. In: *Computational Optimization and Applications* 60.3 (2015), pp. 587–608 [57].

– “Parallelizing the dual revised simplex method”. In: *Mathematical Programming Computation* 10.1 (2018), pp. 119–142 [21, 47, 48].

IBM. *ILOG CPLEX: High-performance software for mathematical programming and optimization*. `https://www.ibm.com/products/ilog-cplex-optimization-studio`. 2021 [16, 21].

Jarck, K. "Exact mixed-integer programming". PhD thesis. Technische Universität Berlin, 2020 [109].

Karmarkar, N. "A new polynomial-time algorithm for linear programming". In: *Proceedings of the sixteenth annual ACM symposium on Theory of computing*. ACM. 1984, pp. 302–311 [7].

Khachiyan, L. G. "A polynomial algorithm in linear programming". In: *Doklady Academii Nauk SSSR*. Vol. 244. 1979, pp. 1093–1096 [7].

Kirillova, F., R. Gabasov, and O. Kostyukova. "A method of solving general linear programming problems". In: *Doklady AN BSSR (in Russian)* 23 (1979), pp. 197–200 [41].

Klotz, E. "Identification, Assessment, and Correction of Ill-Conditioning and Numerical Instability in Linear and Integer Programs". In: *Bridging Data and Decisions*. 2014. Chap. 3, pp. 54–108. DOI: 10.1287/educ.2014.0130 [121].

Koberstein, A. "The dual simplex method, techniques for a fast and stable implementation". PhD thesis. Universität Paderborn, 2005 [41, 42, 49].

Koch, T. "Rapid Mathematical Prototyping". PhD thesis. Technische Universität Berlin, 2004 [19].

– "The final NETLIB-LP results". In: *Operations Research Letters* 32.2 (2004), pp. 138–142. DOI: 10.1016/S0167-6377(03)00094-4 [13, 15, 104].

Koch, T., T. Achterberg, E. Andersen, O. Bastert, T. Berthold, R. Bixby, E. Danna, G. Gamrath, A. Gleixner, S. Heinz, A. Lodi, H. Mittelmann, T. Ralphs, D. Salvagnin, D. Steffy, and K. Wolter. "MIPLIB 2010: Mixed integer programming library version 5". In: *Mathematical Programming Computation* 3.2 (2011), pp. 103–163. DOI: 10.1007/s12532-011-0025-9 [22, 59, 91].

Koch, T., A. Martin, and M. E. Pfetsch. "Progress in Academic Computational Integer Programming". In: *Facets of Combinatorial Optimization: Festschrift for Martin Grötschel*. Ed. by M. Jünger and G. Reinelt. Springer Berlin Heidelberg, 2013, pp. 483–506. DOI: 10.1007/978-3-642-38189-8_19 [64, 77].

Kojima, M., S. Mizuno, and A. Yoshise. "A primal-dual interior point algorithm for linear programming". In: *Progress in mathematical programming*. Springer, 1989, pp. 29–47 [7].

Kostina, E. "The long step rule in the bounded-variable dual simplex method: numerical experiments". In: *Mathematical Methods of Operations Research* 55.3 (2002), pp. 413–429 [41].

Krishnamoorthy, B. "Thinner is not always better: Cascade knapsack problems". In: *Operations Research Letters* 45.1 (2017), pp. 77–83. DOI: 10.1016/j.orl.2016.12.005 [119].

Kruskal, J. B. "Multidimensional scaling by optimizing goodness of fit to a nonmetric hypothesis". In: *Psychometrika* 29.1 (1964), pp. 1–27 [80].

Kubo, M., J. Pedroso, M. Muramatsu, and A. Rais. *Mathematical Optimization: Solving Problems using Python and Gurobi*. Kindaikagakusha, Tokyo, 2012 [26].

Kuhn, H. W. and A. W. Tucker. *Nonlinear programming*. English. Proc. Berkeley Sympos. math. Statist. Probability, California July 31 - August 12, 1950, 481-492 (1951). 1951 [11].

Land, A. H. and A. G. Doig. "An Automatic Method of Solving Discrete Programming Problems". In: *Econometrica* 28.3 (1960), pp. 497–520 [17].

Lemke, C. E. "The dual method of solving the linear programming problem". In: *Naval Research Logistics Quarterly* 1.1 (1954), pp. 36–47 [7].

Luce, R., J. D. Tebbens, J. Liesen, R. Nabben, M. Grötschel, T. Koch, and O. Schenk. *On the Factorization of Simplex Basis Matrices*. Tech. rep. 09-24. Berlin: ZIB, 2009 [37, 39, 40].

Mahajan, A. and T. Ralphs. "On the Complexity of Selecting Disjunctions in Integer Programming". In: *SIAM Journal on Optimization* 20.5 (2010), pp. 2181–2198. DOI: 10.1137/080737587 [119].

Maher, S. J., M. Miltenberger, J. P. Pedroso, D. Rehfeldt, R. Schwarz, and F. Serrano. "PySCIPOpt: Mathematical Programming in Python with the SCIP Optimization Suite". In: *Mathematical Software – ICMS 2016*. Ed. by G.-M. Greuel, T. Koch, P. Paule, and A. Sommese. Cham: Springer International Publishing, 2016, pp. 301–307 [4, 25, 26, 77].

Maher, S. J., T. Fischer, T. Gally, G. Gamrath, A. M. Gleixner, R. L. Gottwald, G. Hendel, T. Koch, M. E. Lübbecke, M. Miltenberger, B. Müller, M. E. Pfetsch, C. Puchert, D. Rehfeldt, S. Schenker, R. Schwarz, F. Serrano, Y. Shinano, D. Weninger, J. T. Witt, and J. Witzig. *The SCIP Optimization Suite 4.0*. Tech. rep. 17-12. Berlin: ZIB, 2017 [5, 15, 58].

Maher, S. J., T. Ralphs, and Y. Shinano. *Assessing the Effectiveness of (Parallel) Branch-and-bound Algorithms*. Tech. rep. 19-03. Berlin: ZIB, 2019 [84].

Markowitz, H. M. "The elimination form of the inverse and its application to linear programming". In: *Management Science* 3.3 (1957), pp. 255–269 [39].

Maros, I. *Computational techniques of the simplex method*. International series in operations research & management science. Kluwer Academic Publ., 2003 [37, 41].

Maros, I. and C. Mészáros. "A repository of convex quadratic programming problems". In: *Optimization Methods and Software* 11.1-4 (1999), pp. 671–681 [108].

Megiddo, N. "On finding primal-and dual-optimal bases". In: *ORSA Journal on Computing* 3.1 (1991), pp. 63–65 [16].

Mehrotra, S. "On the Implementation of a Primal-Dual Interior Point Method". In: *SIAM Journal on Optimization* 2.4 (1992), pp. 575–601. DOI: 10.1137/0802028 [12].

Meindl, B. and M. Templ. "Analysis of commercial and free and open source solvers for linear optimization problems". In: *Eurostat and Statistics Netherlands within the project ESSnet on common tools and harmonised methodology for SDC in the ESS* 20 (2012) [64].

Mészáros, C. "The cholesky factorization in interior point methods". In: *Computers & Mathematics with Applications* 50.7 (2005). Numerical Methods and Computational Mechanics, pp. 1157–1166. DOI: 10.1016/j.camwa.2005.08.016 [12].

Miltenberger, M. *mittelmann-plots - Interactive Visualizations of Mittelmann benchmarks*. http://github.com/mattmilten/mittelmann-plots. 2021 [137].

– *TreeD*. http://github.com/mattmilten/TreeD. 2021 [4, 19].

Miltenberger, M., T. Ralphs, and D. E. Steffy. "Exploring the Numerics of Branch-and-Cut for Mixed Integer Linear Optimization". In: *Operations Research Proceedings 2017*. 2018, pp. 151–157. DOI: 10.1007/978-3-319-89920-6 [5, 112].

MOSEK ApS. *The MOSEK C optimizer API manual. Version 8.0 (Revision 45)*. 2016 [21].

Motzkin, T. S. *Beitrage zur Theorie der linearen Ungleichungen: Inaugural-Dissertation*. Azriel, 1936 [7].

Nair, V., S. Bartunov, F. Gimeno, I. von Glehn, P. Lichocki, I. Lobov, B. O'Donoghue, N. Sonnerat, C. Tjandraatmadja, P. Wang, R. Addanki, T. Hapuarachchi, T. Keck, J. Keeling, P. Kohli, I. Ktena, Y. Li, O. Vinyals, and Y. Zwols. "Solving Mixed Integer Programs Using Neural Networks". In: *arXiv* 2012.13349 (2020) [7, 74].

Nannicini, G. "Fast Quantum Subroutines for the Simplex Method". In: *Integer Programming and Combinatorial Optimization*. Ed. by M. Singh and D. P. Williamson. Cham: Springer International Publishing, 2021, pp. 311–325. DOI: 10.1007/978-3-030-73879-2_22 [7].

Nemhauser, G. L. and L. A. Wolsey. *Integer and combinatorial optimization*. New York, NY, USA: Wiley-Interscience, 1988. DOI: 10.1002/acs.4480040410 [96].

Ordonez, F. and R. Freund. "Computational Experience and the Explanatory Value of Condition Measures for Linear Optimization". In: *SIAM Journal on Optimization* 14.2 (2003), pp. 307–333. DOI: 10.1137/S1052623402401804 [104, 116, 117, 119].

Pedregosa, F., G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830 [81].

Rothberg, E. "An evolutionary algorithm for polishing mixed integer programming solutions". In: *INFORMS Journal on Computing* 19.4 (2007), pp. 534–541 [92].

Rump, S. M. "High Precision Evaluation of Nonlinear Functions". In: *Proceedings of 2005 International Symposium on Nonlinear Theory and its Applications, Bruge, Belgium*. 2005, pp. 733–736 [33].

Schrijver, A. *Theory of Linear and Integer Programming*. New York, NY, USA: John Wiley & Sons, Inc., 1986 [8, 10].

SCIP: Solving Constraint Integer Programs. http://scip.zib.de. 2021 [17].

Shepard, R. N. "The analysis of proximities: multidimensional scaling with an unknown distance function. I." In: *Psychometrika* 27.2 (1962), pp. 125–140 [80].

Shinano, Y., T. Achterberg, T. Berthold, S. Heinz, and T. Koch. "ParaSCIP: a parallel extension of SCIP". In: *Competence in High Performance Computing 2010*. Ed. by C. Bischof, H.-G. Hegering, W. Nagel, and G. Wittum. 2012, pp. 135–148. DOI: 10.1007/978-3-642-24025-6_12 [19].

Skeel, R. D. "Scaling for numerical stability in Gaussian elimination". In: *Journal of the ACM (JACM)* 26.3 (1979), pp. 494–526 [107].

Suhl, L. M. and U. H. Suhl. "A fast LU update for linear programming". In: *Annals of Operations Research* 43.1 (1993), pp. 33–47 [57].

Trefethen, L. and D. Bau. *Numerical Linear Algebra*. Other Titles in Applied Mathematics. Society for Industrial and Applied Mathematics (SIAM, 3600 Market Street, Floor 6, Philadelphia, PA 19104), 1997 [106].

Turing, A. M. "Rounding-off errors in matrix processes". In: *The Quarterly Journal of Mechanics and Applied Mathematics* 1.1 (1948), pp. 287–308. DOI: 10.1093/qjmam/1.1.287 [103].

Van der Maaten, L. and G. Hinton. "Visualizing data using t-SNE." In: *Journal of machine learning research* 9.11 (2008) [80].

Vanderbei, R. J. *Linear programming: foundations and extensions*. Vol. 4. International Series in Operations Research & Management Science. Kluwer Academic Publishers, Boston, MA, 1996, pp. xviii+418 [8, 10].

Vigerske, S. and A. Gleixner. "SCIP: global optimization of mixed-integer nonlinear programs in a branch-and-cut framework". In: *Optimization Methods and Software* 33.3 (2018), pp. 563–593. DOI: 10.1080/10556788.2017.1335312 [12].

Wilkinson, J. H. *Rounding errors in algebraic processes*. Prentice-Hall, Inc., Englewood Cliffs, N.J., 1963, pp. vi+161 [57].

Witzig, J., T. Berthold, and S. Heinz. "A Status Report on Conflict Analysis in Mixed Integer Nonlinear Programming". In: *Integration of AI and OR Techniques in Constraint Programming. CPAIOR 2019*. Vol. 11494. 2019, pp. 84–94. DOI: 10.1007/978-3-030-19212-9_6 [10, 78].

Wolsey, L. A. "Integer programming duality: Price functions and sensitivity analysis". In: *Mathematical Programming* 20.1 (1981), pp. 173–195. DOI: 10.1007/BF01589344 [17].

Wunderling, R. “Paralleler und objektorientierter Simplex-Algorithmus”. PhD thesis. Technische Universität Berlin, FB Math., 1996 [15, 33, 47].

– “The kernel simplex method.” Talk at the 21st International Symposium on Mathematical Programming, ISMP, Berlin, Germany. 2012 [36].

Zanette, A., M. Fischetti, and E. Balas. “Lexicography and degeneracy: can a pure cutting plane algorithm work?” In: *Mathematical Programming* 130.1 (2011), pp. 153–176. DOI: 10.1007/s10107-009-0335-0 [77, 92, 106, 113].

Appendix A

Mittelmann Benchmark Plots

Prof. Hans Mittelmann (Arizona State University) maintains the largest publicly available benchmark data set for linear, mixed-integer, nonlinear, and combinatorial optimization problems[1]. His benchmarks are unique in the way that they allow comparisons between both commercial and open-source solvers and are usually kept up-to-date with new versions of these solvers.

Solvers are compared on a set of instances for every class of problems, using a certain time limit. Runs that do not end in an optimal solution are counted as timeouts, without further penalization added. The number of correctly solved instances is also displayed.

The benchmarks themselves are performed without any stabilization techniques like different random seeds and some of the test sets are too small to provide a clear assertion of each solvers general performance.

Nevertheless, we are very happy that these benchmarks exist and are maintained with such passionate commitment. As the results are only available in plain text format and hence quite hard to process and analyze, we created an interactive web application called `mittelmann-plots` (Miltenberger, 2021a) that uses the Python graphing library Plotly[2] to illustrate the individual results.

For every single class of benchmarks there is a table that automatically sorts the solvers by their respective scores and also displays the percentages of solved instances. The scores are computed by taking the shifted geometric mean of all solving times or time outs. The solver with the smallest mean is declared the winner and all scores are divided by the best score to achieve a relative comparison amongst all solvers. In addition to that, we include a “virtual best” or “portfolio” solver that assumes the best results for every single instance. This also depicts how much variability is present in the benchmarks and how much potential is still attainable by the individual solvers.

Such aggregated numbers can easily be misleading and it is always a good idea to also inspect the individual instance-wise performance results. Our tool provides an

[1] `http://plato.asu.edu/bench.html`
[2] `https://plotly.com/`

interactive chart that displays both the absolute solving times for a selected solver as well as the relative speedup factors to all other solvers for every single instance as shown in Figure A.1. To account for huge differences in solving times ranging from fractions of a second to hours, we use a logarithmic scale for the relative numbers as well as the absolute solving times.

We believe that this visualization tool provides significantly added value to the raw benchmark results on Hans Mittelmann's webpage.

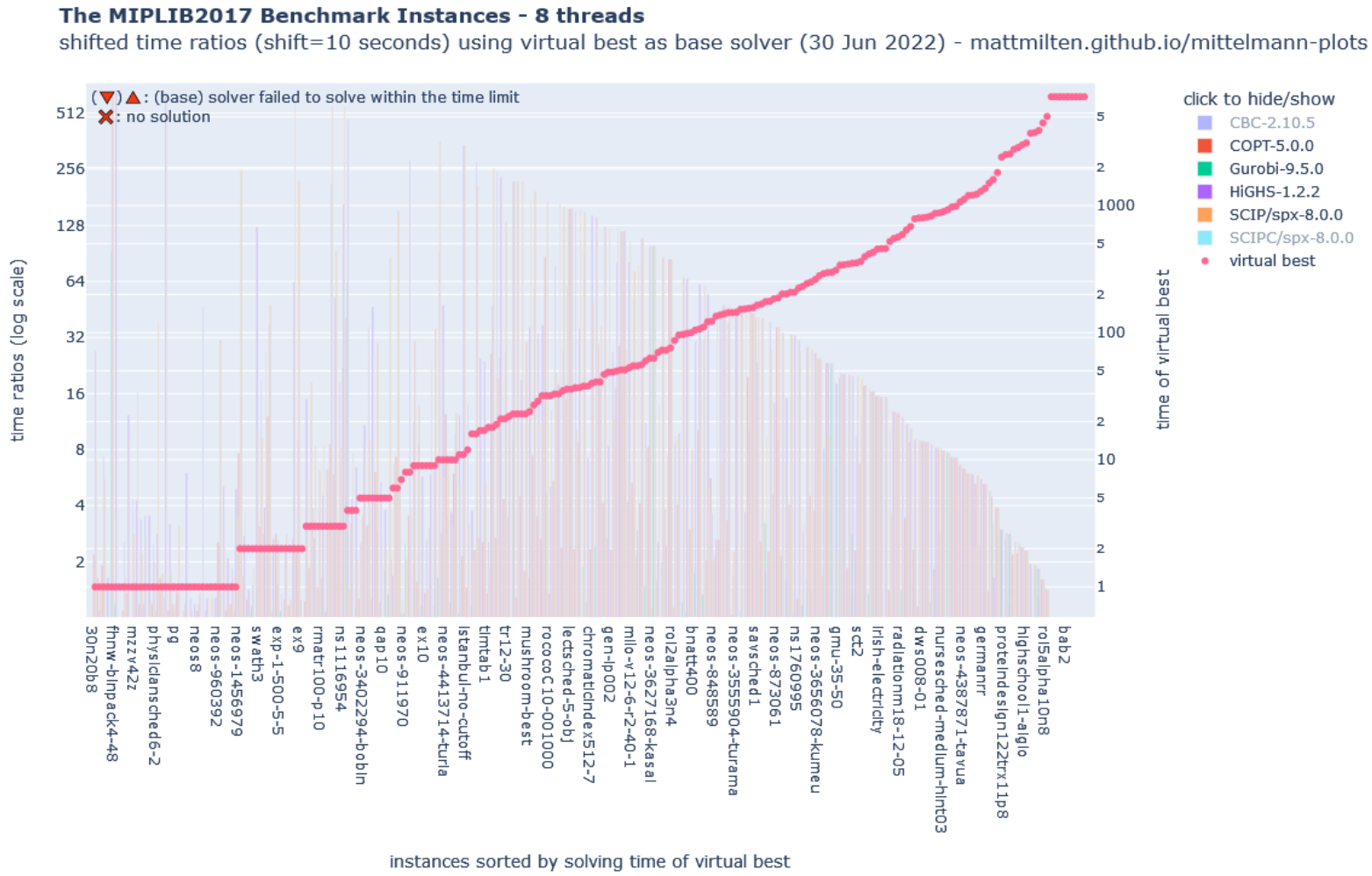


Figure A.1.: Screenshot of `mittelmann-plots` on the MIPLIB 2017 benchmark

Appendix B

Experimental Data and Results

This appendix provides full data tables of the experiments conducted for the thesis and presented in the respective chapters.

We mark unsuccessful optimizations with an asterisk “*” at the respective times for that instance and setting. Additionally, we use an exclamation mark “!” to highlight infeasible instances and a “+” when a node limit has been reached.

Table B.1.: LP solver comparison (`allLP` test set, SCIP 6.0.2)

	time						iters					
LP solver	CPLEX	Clp	Gurobi	MOSEK	SoPlex	Xpress	CPLEX	Clp	Gurobi	MOSEK	SoPlex	Xpress
instance	12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09	12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09
10teams	0.1	0.1	0.1	0.1	0.3	0.1	925	911	1283	1032	8033	1730
16_n14	23.1	30.5	23.6	41.8	36.3	16.9	134 425	110 844	158 748	88 309	121 054	47 553
22433	0.0	0.0	0.0	0.0	0.0	0.0	263	227	239	254	589	258
23588	0.0	0.0	0.0	0.0	0.0	0.0	185	145	191	149	337	276
25fv47	0.1	0.2	0.2	0.3	0.4	0.2	2347	3225	2876	2936	7839	2857
30_70_45_095_100	3.5	5.4	1.1	10.7	7.4	7.2	13 346	11 024	11 364	18 383	16 184	21 126
30n20b8	0.2	0.2	0.2	0.2	0.2	0.2	675	269	402	306	617	349
50v-10	0.0	0.0	0.0	0.0	0.0	0.0	305	275	469	238	220	339
80bau3b	0.1	0.1	0.2	0.2	0.4	0.1	4341	3160	2721	2817	6827	2761
CMS750_4	1.4	1.4	1.5	1.5	1.7	1.4	0	0	0	0	0	0
L1_d10_40	3600.0*	3600.0*	3593.0*	3600.0*	3600.0*	3591.9*	157 398*	136 217*	201 336*	129 000*	74 400*	159 334*
Linf_520c	386.7	2755.8	889.8	1589.7	3600.0*	1752.6	126 668	256 263	248 694	183 201	50 797*	180 035
Test3	0.9	0.9	0.9	0.9	1.2	0.9	1849	1351	1608	1482	4828	2068
a1c1s1	0.1	0.1	0.1	0.1	0.1	0.1	523	1060	618	306	1047	1053
aa01	1.0	1.2	1.1	1.6	2.8	1.6	3516	3823	3803	4447	12 277	5443
aa03	0.7	0.9	0.8	1.0	2.3	1.1	2249	2656	2378	2557	10 766	3980
aa3	0.7	0.9	0.8	1.0	2.3	1.1	2249	2656	2378	2557	10 766	3980
aa4	0.4	0.5	0.5	0.6	1.5	0.6	1374	1463	1323	1668	8923	2278
aa5	0.6	0.8	0.8	1.0	2.2	1.1	2123	2504	2543	2913	11 034	3975
aa6	0.5	0.6	0.5	0.6	1.6	0.7	1554	1711	1414	1685	8992	2185
acc-tight4	0.4	0.6	1.3	2.7	1.3	0.6	3693	2916	6881	9167	7626	4446
acc-tight5	0.3	0.5	0.4	2.3	0.8	0.6	2852	2345	2661	8263	5700	3968
acc-tight6	0.3	0.5	0.4	2.3	0.8	0.5	2730	2635	2471	8287	5438	3826
adlittle	0.0	0.0	0.0	0.0	0.0	0.0	97	74	62	100	135	70
afiro	0.0	0.0	0.0	0.0	0.0	0.0	7	15	5	8	11	13
aflow30a	0.0	0.0	0.0	0.0	0.0	0.0	1504	456	571	113	725	471
aflow40b	0.1	0.1	0.1	0.1	0.1	0.1	5143	1490	1673	182	2207	1560
agg	0.0	0.0	0.0	0.0	0.0	0.0	60	77	75	54	124	84
agg2	0.0	0.0	0.0	0.0	0.0	0.0	114	156	151	120	210	159
agg3	0.0	0.0	0.0	0.0	0.0	0.0	114	160	169	114	206	154
air02	0.2	0.2	0.2	0.2	0.2	0.2	153	171	149	117	362	204
air03	0.2	0.2	0.2	0.3	0.3	0.2	339	438	411	443	915	489
air04	1.0	1.2	1.1	1.6	2.8	1.6	3516	3823	3803	4447	12 277	5443
air05	0.4	0.5	0.5	0.6	1.5	0.6	1374	1463	1323	1668	8923	2278

Continued on next page

Table B.1.: LP solver comparison (`allLP` test set, SCIP 6.0.2)

	time						iters					
LP solver	CPLEX	Clp	Gurobi	MOSEK	SoPlex	Xpress	CPLEX	Clp	Gurobi	MOSEK	SoPlex	Xpress
instance	12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09	12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09
air06	0.7	0.8	0.8	1.0	2.3	1.1	2249	2656	2378	2557	10766	3980
aircraft	0.1	0.1	0.1	0.1	0.1	0.1	1986	1941	2267	3134	2039	2457
aligninq	0.0	0.0	0.0	0.0	0.1	0.0	341	354	381	424	755	547
app1-1	0.1	0.1	0.1	0.1	0.1	0.1	77	1386	67	75	1280	1280
app1-2	1.2	5.0	2.3	1.3	14.7	1.1	700	13979	5104	804	13887	15195
arki001	0.1	0.1	0.1	0.1	0.1	0.1	1025	779	700	463	1427	2538
ash608gpia-3col	0.8	0.8	0.6	3.1	0.3	2.2	4355	4408	5509	3684	3985	5926
assign1-5-8	0.0	0.0	0.0	0.0	0.0	0.0	227	281	349	231	1079	418
atlanta-ip	7.1	6.8	7.9	11.6	19.2	8.3	18624	15451	23278	17159	27339	27048
atm20-100	0.2	0.2	0.2	0.3	0.3	0.2	2043	3038	2852	2967	4286	3250
b1c1s1	0.1	0.1	0.1	0.1	0.1	0.1	761	1472	1050	655	1482	1503
b2c1s1	0.1	0.1	0.1	0.1	0.1	0.1	1253	1762	1480	901	1855	2022
bab1	1.2	1.2	1.2	1.4	1.3	1.2	1846	1739	3224	2962	3593	2438
bab2	66.3	162.0	16.3	52.5	130.2	40.1	110087	130756	96382	63874	137508	124249
bab3	76.7	627.1	69.0	172.3	464.7	245.1	137961	257810	181631	140050	269078	473464
bab5	1.5	1.9	1.1	2.2	4.7	2.1	19770	13721	18264	13532	27390	26681
bab6	25.0	44.3	14.2	38.8	80.0	24.8	73400	70563	74454	50513	101200	97300
bal8x12	0.0	0.0	0.0	0.0	0.0	0.0	22	26	12	26	26	25
bandm	0.0	0.0	0.0	0.0	0.0	0.0	250	268	252	301	642	343
bas1lp	5.7	2.2	1.8	20.9	1.8	1.7	7447	1845	1139	18017	1091	1330
baxter	0.7	0.8	0.6	1.2	2.1	2.4	7286	5945	6159	4899	11267	23137
bc	0.7	0.7	0.7	0.8	0.7	0.7	83	795	67	606	857	828
bc1	0.7	0.7	0.7	0.7	0.7	0.7	83	795	67	606	857	828
beaconfd	0.0	0.0	0.0	0.0	0.0	0.0	0	0	0	0	0	0
beasleyC3	0.0	0.0	0.0	0.0	0.0	0.0	269	268	262	355	277	271
bell3a	0.0	0.0	0.0	0.0	0.0	0.0	47	51	45	47	47	47
bell5	0.0	0.0	0.0	0.0	0.0	0.0	40	44	42	40	43	42
berlin_5_8_0	0.1	0.1	0.1	0.1	0.0	0.1	626	583	650	561	651	591
bg512142	0.1	0.1	0.1	0.1	0.1	0.1	818	1209	1297	1377	2388	1575
biella1	1.2	1.6	1.5	2.2	3.6	2.1	6239	6378	8180	8369	15881	10460
bienst1	0.0	0.0	0.0	0.0	0.0	0.0	114	354	315	459	393	356
bienst2	0.0	0.0	0.0	0.0	0.0	0.0	114	354	315	459	393	356
binkar10_1	0.0	0.0	0.0	0.0	0.0	0.0	426	553	441	550	664	566
bk4x3	0.0	0.0	0.0	0.0	0.0	0.0	6	5	3	7	5	6

Continued on next page

Table B.1.: LP solver comparison (`allLP` test set, SCIP 6.0.2)

	time						iters					
LP solver	CPLEX	Clp	Gurobi	MOSEK	SoPlex	Xpress	CPLEX	Clp	Gurobi	MOSEK	SoPlex	Xpress
instance	12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09	12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09
blend	0.0	0.0	0.0	0.0	0.0	0.0	60	76	50	74	88	50
blend2	0.0	0.0	0.0	0.0	0.0	0.0	8	122	5	10	132	155
bley_xl1	18.4	3600.0*	18.7	722.5	3600.0*	1404.5	16 439	418 850*	54 002	50 750	531 815*	465 809
blp-ar98	0.5	0.5	0.5	0.5	0.5	0.5	479	389	513	462	458	489
blp-ic97	0.2	0.2	0.2	0.2	0.3	0.2	390	234	309	245	655	278
blp-ic98	0.3	0.3	0.3	0.4	0.4	0.4	383	196	225	219	266	336
bnatt350	0.1	0.1	0.1	0.1	0.1	0.1	842	646	612	550	633	680
bnatt400	0.1	0.1	0.1	0.1	0.1	0.1	1154	753	711	655	675	725
bnatt500	0.1	0.1	0.1	0.1	0.1	0.1	1195	931	907	846	849	948
bnl1	0.0	0.0	0.1	0.1	0.1	0.1	1001	931	1174	1050	1762	1059
bnl2	0.1	0.1	0.1	0.1	0.1	0.1	1264	1228	1221	1053	2619	1459
boeing1	0.0	0.0	0.0	0.0	0.0	0.0	427	374	363	398	604	323
boeing2	0.0	0.0	0.0	0.0	0.0	0.0	133	159	83	132	180	126
bore3d	0.0	0.0	0.0	0.0	0.0	0.0	35	59	36	41	75	51
bppc4-08	0.0	0.0	0.1	0.1	0.1	0.1	384	239	732	486	2236	924
brandy	0.0	0.0	0.0	0.0	0.0	0.0	130	223	140	188	555	225
brazil3	11.8	19.4	6.8	11.6	67.1	53.4	24 328	25 429	15 829	17 036	63 235	113 451
buildingenergy	18.7	43.7	12.2	1045.5	388.1	20.4	132 178	128 439	121 255	183 122	122 495	263 741
cap6000	0.1	0.1	0.2	0.1	0.1	0.1	127	153	1661	155	186	421
capri	0.0	0.0	0.0	0.0	0.0	0.0	112	209	133	152	280	182
car4	0.4	0.4	0.3	0.4	1.4	0.5	1107	1185	1067	1360	10 313	2380
cari	0.1	0.1	0.1	0.1	0.1	0.1	423	440	547	475	675	501
cbs-cta	0.3	0.3	0.3	0.3	0.8	0.3	4468	3130	3142	368	8725	4955
cdma	3600.0*	3600.0*	3600.0*	3600.0*	3600.0*	3600.0*	0*	0*	0*	2 462 693*	0*	0*
cep1	0.0	0.0	0.1	0.1	0.0	0.0	1441	1505	1584	1390	1989	1380
ch	0.2	0.2	0.2	0.3	0.8	0.2	3019	3719	3509	3218	8752	4040
chromaticindex1024-7	74.8	72.7	228.5	259.1	220.7	129.7	60 214	106 360	100 585	69 924	91 005	116 791
chromaticindex512-7	17.0	17.0	38.9	52.3	49.8	34.7	30 366	51 224	46 134	33 860	51 605	63 938
circ10-3	3.2	9.5	1.2	6.2	4.1	2.7	7134	8345	7203	3595	6869	4299
cmflsp50-24-8-8	0.9	1.1	1.0	1.6	2.2	1.1	5588	7233	4554	7326	9518	8220
co-100	2.2	2.1	2.2	2.2	2.1	2.1	536	462	800	459	570	674
co5	0.7	1.1	0.6	1.1	2.0	1.0	6049	6109	5603	6294	11 618	8983
co9	3.6	4.7	2.3	4.5	5.6	4.2	16 955	13 207	11 805	12 965	15 769	18 578
cod105	4.2	2.8	1.1	12.0	6.3	5.2	5224	3452	1355	5698	12 409	8846

Continued on next page

Table B.1.: LP solver comparison (`allLP` test set, SCIP 6.0.2)

	time						iters					
LP solver	CPLEX	Clp	Gurobi	MOSEK	SoPlex	Xpress	CPLEX	Clp	Gurobi	MOSEK	SoPlex	Xpress
instance	12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09	12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09
comp07-2idx	3.1	1.9	1.1	2.9	5.4	1.5	6703	3790	4813	3764	9601	3941
comp21-2idx	0.6	0.8	0.4	1.1	1.3	0.4	2829	2315	2499	2375	4397	2076
complex	1.0	0.5	0.1	1.0	0.7	0.7	7638	2002	1515	2612	4242	3247
cont1	584.0	724.6	190.9	3015.8*	1377.9	241.6	35682	56507	47900	47689*	42543	40614
cont11	3580.9*	3600.0*	1395.4	3600.0*	3600.0*	2042.9	169542*	167983*	117518	76000*	146406*	145194
cont11_l	3594.1*	3600.0*	3571.6*	3600.0*	3600.0*	3592.0*	357077*	381704*	319737*	539000*	54000*	311943*
cont4	450.1	172.7	136.8	724.7	996.5	227.8	24190	54391	40558	41660	43853	41984
core2536-691	2.5	5.7	2.2	5.2	7.0	3.8	15464	20093	12524	17090	31685	26582
core4872-1529	15.1	27.1	20.4	39.7	36.6	28.5	0	0	0	0	0	0
cost266-UUE	0.1	0.1	0.1	0.1	0.1	0.1	1692	1360	1808	1508	2280	1827
cov1075	0.1	0.1	0.0	0.1	0.0	0.1	2314	389	223	458	423	1219
cq5	0.6	0.6	0.4	0.8	1.4	0.7	5640	4849	5644	5635	9844	7777
cq9	1.4	2.2	1.5	3.5	4.4	2.7	12349	10786	11290	12728	14882	17144
cr42	0.0	0.0	0.0	0.0	0.0	0.0	454	972	476	367	828	1030
cre-a	0.2	0.2	0.2	0.2	0.2	0.2	2605	2416	3091	2705	3935	3331
cre-b	1.3	2.5	1.4	2.3	2.5	1.6	10450	10207	11875	9595	14350	12468
cre-c	0.1	0.1	0.2	0.2	0.2	0.1	2208	2079	2575	2316	2764	2468
cre-d	1.0	1.5	1.0	1.4	1.8	1.0	7268	7010	7266	6470	12302	7161
crew1	0.1	0.2	0.2	0.2	0.8	0.2	721	353	787	730	6558	1144
cryptanalysiskb128n5obj14	3.4	3.8	4.9	20.4	4.2	3.3	11957	17034	20544	18784	15037	16402
cryptanalysiskb128n5obj16	3.5	4.5	5.1	21.5	4.1	4.0	12153	17162	20907	18800	15049	16867
csched007	0.0	0.0	0.0	0.1	0.1	0.1	590	765	692	976	1522	1031
csched008	0.0	0.0	0.0	0.0	0.1	0.1	1187	1014	385	637	1176	1270
csched010	0.0	0.1	0.1	0.1	0.1	0.1	993	1301	1345	1462	2533	1290
cvs16r128-89	0.6	0.8	0.7	1.5	2.7	1.5	5614	6550	6725	11335	12157	10006
cycle	0.1	0.1	0.1	0.1	0.1	0.1	618	387	315	576	720	463
czprob	0.0	0.0	0.1	0.1	0.1	0.0	424	807	447	809	1178	638
d10200	0.1	0.1	0.1	0.2	0.2	0.1	693	708	680	793	2621	1037
d20200	0.6	0.6	0.6	0.8	1.6	0.8	2104	1674	1736	1976	10128	3179
d2q06c	0.7	1.4	0.9	1.4	2.1	1.3	5220	7027	5840	6193	13165	8748
d6cube	0.2	0.2	0.2	0.2	0.4	0.3	454	407	609	553	2650	1068
dano3_3	5.3	36.0	14.5	13.5	13.8	9.6	22979	66441	57874	31182	39995	32080
dano3_4	5.3	36.0	14.5	13.4	13.8	9.6	22979	66441	57874	31182	39995	32080
dano3_5	5.3	35.9	14.5	13.2	13.9	9.6	22979	66441	57874	31182	39995	32080

Continued on next page

Table B.1.: LP solver comparison (`allLP` test set, SCIP 6.0.2)

		time						iters					
	LP solver	CPLEX	Clp	Gurobi	MOSEK	SoPlex	Xpress	CPLEX	Clp	Gurobi	MOSEK	SoPlex	Xpress
instance		12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09	12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09
dano3mip		5.3	36.0	14.6	13.4	13.8	9.5	22 979	66 441	57 874	31 182	39 995	32 080
danoint		0.0	0.1	0.1	0.1	0.1	0.1	636	1302	1069	1063	3377	1140
datt256		3600.0*	3600.0*	1386.1	3597.7*	3600.0*	1203.5	322 376*	246 331*	141 856	338 000*	139 110*	75 658
dbic1		20.6	286.8	19.4	92.5	163.1	17.3	50 008	217 600	40 215	89 072	122 847	41 829
dbir1		2.7	2.9	2.9	3.0	7.0	2.6	1524	1905	2092	2129	12 120	1699
dbir2		3.4	3.6	3.6	6.2	4.9	3.5	7801	7450	6138	12 876	11 925	9865
dc1c		2.0	2.8	2.3	3.5	4.7	3.8	8570	9343	10 461	10 399	17 672	15 073
dc1l		5.1	6.7	4.7	9.9	12.2	12.1	10 296	11 298	12 252	13 826	20 409	24 809
dcmulti		0.0	0.0	0.0	0.0	0.0	0.0	307	349	268	390	454	350
de063155		0.0	0.1	3600.0*	0.1	0.2	0.0	950	2688	0*	538	3696	859
de063157		0.1*	3600.0*	3600.0*	3600.0*	3600.0*	0.1*	770*	0*	0*	0*	326 035k*	971*
de080285		0.0	0.1	0.0	0.1	0.1	0.0	411	2402	372	431	758	659
decomp2		0.2	0.2	0.2	0.2	0.3	0.2	2008	1598	1438	2023	3235	2455
degen2		0.0	0.0	0.0	0.0	0.0	0.0	344	446	454	506	1119	604
degen3		0.1	0.2	0.2	0.3	0.8	0.2	1377	1776	1756	1992	8955	2458
degme		3600.0*	3600.0*	3600.0*	3600.0*	3600.0*	3600.0*	0*	0*	0*	0*	0*	0*
delf000		0.1	0.2	0.1	0.1	0.1	0.1	856	3321	977	1485	1837	1979
delf001		0.1	0.3	0.1	0.1	0.1	0.1	861	4413	933	1710	1828	2077
delf002		0.1	0.3	0.1	0.1	0.1	0.1	936	4061	908	1595	1938	2053
delf003		0.1	0.3	0.1	0.1	0.2	0.1	978	3567	1053	1504	2268	2068
delf004		0.1	0.3	0.1	0.1	0.2	0.1	1082	3526	1203	1626	2677	2154
delf005		0.1	0.4	0.1	0.1	0.2	0.1	1151	4114	1238	1662	2597	2227
delf006		0.1	0.3	0.1	0.1	0.3	0.1	1202	3590	1339	1590	3622	2127
delf007		0.1	0.5	0.1	0.1	0.3	0.1	1310	4458	1333	1588	3362	2329
delf008		0.1	0.7	0.1	0.1	0.2	0.1	1351	5243	1374	1655	3334	2253
delf009		0.1	0.5	0.1	0.2	0.2	0.1	1414	4444	1390	1740	3263	2265
delf010		0.1	0.4	0.1	0.1	0.3	0.1	1291	4231	1395	1673	3613	2347
delf011		0.1	0.5	0.1	0.1	0.2	0.1	1341	5463	1422	1746	3213	2444
delf012		0.1	0.4	0.1	0.2	0.3	0.1	1311	3664	1472	1701	3471	2489
delf013		0.1	0.5	0.1	0.1	0.2	0.1	1243	4593	1312	1687	2436	2270
delf014		0.1	0.3	0.1	0.1	0.2	0.1	1186	3640	1311	1598	2724	2330
delf015		0.1	0.4	0.1	0.1	0.2	0.1	1306	4122	1410	1687	2516	2345
delf017		0.1	0.5	0.1	0.1	0.2	0.1	1137	5013	1392	1713	2741	2451
delf018		0.1	0.6	0.1	0.1	0.2	0.1	1152	5670	1403	1858	2935	2473

Continued on next page

Table B.1.: LP solver comparison (`allLP` test set, SCIP 6.0.2)

	time						iters					
LP solver	CPLEX	Clp	Gurobi	MOSEK	SoPlex	Xpress	CPLEX	Clp	Gurobi	MOSEK	SoPlex	Xpress
instance	12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09	12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09
delf019	0.1	0.2	0.1	0.1	0.2	0.1	1222	3651	1387	1789	3150	2521
delf020	0.1	0.6	0.1	0.1	0.2	0.1	1377	4647	1444	1792	3635	2544
delf021	0.1	0.5	0.1	0.1	0.2	0.1	1411	5035	1421	1727	3633	2561
delf022	0.1	0.7	0.1	0.1	0.2	0.1	1264	5641	1178	1793	3105	2514
delf023	0.1	0.4	0.1	0.1	0.3	0.1	1299	4035	1243	1816	4292	2412
delf024	0.1	1.0	0.1	0.2	0.2	0.1	1267	8463	1172	1687	3529	2412
delf025	0.1	0.6	0.1	0.1	0.3	0.1	1383	5187	1308	1600	4121	2411
delf026	0.2	0.7	0.1	0.2	0.2	0.1	0	0	0	0	0	0
delf027	0.1	0.5	0.1	0.1	0.2	0.1	1308	5234	1148	1602	3353	2421
delf028	0.1	0.7	0.1	0.1	0.2	0.1	1180	5771	1099	1559	3232	2431
delf029	0.1	0.6	0.1	0.2	0.2	0.1	2116	5601	1102	1669	2643	2403
delf030	0.1	0.4	0.1	0.2	0.2	0.1	1230	4544	1137	1584	3127	2274
delf031	0.1	0.5	0.1	0.1	0.2	0.1	1171	5140	1065	1474	3287	2219
delf032	0.1	0.3	0.1	0.2	0.2	0.1	2091	4000	1078	1596	2555	2264
delf033	0.1	0.4	0.1	0.1	0.2	0.1	1107	4638	1190	1502	3051	2234
delf034	0.1	0.5	0.1	0.2	0.2	0.1	1282	5091	1168	1559	3193	2341
delf035	0.1	0.4	0.1	0.2	0.2	0.1	1121	4003	1138	1591	2792	2277
delf036	0.1	0.7	0.1	0.1	0.3	0.1	1789	6858	1264	1586	3871	2262
deter0	0.1	0.1	0.1	0.1	0.1	0.1	1752	2874	1590	2731	2315	3761
deter1	0.2	0.3	0.2	0.4	0.4	0.3	5312	7657	4941	7513	6710	10935
deter2	0.2	0.3	0.2	0.4	0.4	0.3	5814	9270	5914	7765	7531	12752
deter3	0.3	0.5	0.3	0.5	0.5	0.4	7268	10543	6460	10515	9134	15477
deter4	0.1	0.1	0.1	0.2	0.2	0.1	2842	5123	2445	3192	4169	6419
deter5	0.2	0.4	0.2	0.4	0.4	0.2	4897	7886	4425	7160	6303	10161
deter6	0.2	0.2	0.2	0.3	0.3	0.2	3800	5904	4092	5843	6118	8372
deter7	0.2	0.4	0.2	0.5	0.5	0.3	6460	8890	5886	8950	7317	12746
deter8	0.1	0.2	0.1	0.3	0.2	0.2	3615	5547	3153	5661	4606	7928
df2177	0.2	3600.0*	0.1	0.5	0.3	0.4	1734	1736389*	704	1304	1249	2251
dfl001	3.4	7.2	5.0	7.7	19.3	9.9	17276	21804	20445	21068	37871	36004
dfn-gwin-UUM	0.0	0.0	0.0	0.0	0.0	0.0	74	480	92	74	446	549
dg012142	0.1	0.6	0.6	0.9	1.0	0.5	1775	4442	4885	5091	10130	5213
disctom	0.3	1.8	1.4	0.8	5.2	0.5	1985	8009	10013	3928	20016	3406
disp3	0.0	0.0	0.0	0.1	0.0	0.0	232	274	201	242	350	251
dolom1	2.8	4.4	3.2	5.3	6.1	5.2	8946	11534	10620	10993	18130	15870

Continued on next page

Table B.1.: LP solver comparison (`allLP` test set, SCIP 6.0.2)

	time						iters					
LP solver	CPLEX	Clp	Gurobi	MOSEK	SoPlex	Xpress	CPLEX	Clp	Gurobi	MOSEK	SoPlex	Xpress
instance	12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09	12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09
drayage-100-23	0.2	0.2	0.2	0.4	0.3	0.2	691	1063	977	4877	2004	796
drayage-25-23	0.2	0.2	0.3	0.2	0.3	0.2	937	1013	955	703	2407	796
ds	11.0	13.4	18.2	34.6	25.6	19.6	6335	6194	60766	11684	13935	9520
ds-big	368.6	633.0	603.1	1672.2	809.4	592.1	74948	88922	621370	189087	81560	92984
dsbmip	0.1	0.1	0.1	0.1	0.1	0.1	1145	2223	1197	1502	3751	1589
dws008-01	0.2	0.2	0.2	0.3	0.2	0.2	522	217	355	279	261	279
e18	1.2	1.0	1.3	2.1	5.8	1.2	5729	4293	7378	4899	11802	6150
e226	0.0	0.0	0.0	0.0	0.0	0.0	298	249	298	315	654	346
egout	0.0	0.0	0.0	0.0	0.0	0.0	0	2	0	0	2	2
eil33-2	0.2	0.1	0.1	0.2	0.2	0.2	243	241	222	182	783	348
eilA101-2	5.2	6.4	4.1	6.0	26.1	8.0	1333	1536	4856	1289	10645	1984
eilB101	0.1	0.1	0.1	0.1	0.4	0.1	650	667	791	565	6026	987
enigma	0.0	0.0	0.0	0.0	0.0	0.0	42	51	34	58	62	45
enlight13	0.0	0.0	0.0	0.0	0.0	0.0	0	0	0	0	0	0
enlight14	0.0	0.0	0.0	0.0	0.0	0.0	0	0	0	0	0	0
enlight15	0.0	0.0	0.0	0.0	0.0	0.0	0	0	0	0	0	0
enlight16	0.0	0.0	0.0	0.0	0.0	0.0	0	0	0	0	0	0
enlight9	0.0	0.0	0.0	0.0	0.0	0.0	0	0	0	0	0	0
enlight_hard	0.0	0.0	0.0	0.0	0.0	0.0	0	0	0	0	0	0
etamacro	0.0	0.0	0.0	0.0	0.0	0.0	809	492	360	375	790	606
ex10	545.3	623.3	296.3	3599.2*	3600.0*	2601.0	304902	104456	84164	294000*	48073*	266523
ex1010-pi	3.2	4.8	3.8	10.1	6.3	6.1	10493	10390	11669	18478	18320	17766
ex3sta1	9.0	19.9	8.4	10.7	2.8	14.6	11846	21010	9160	9039	7616	28955
ex9	74.7	103.5	55.9	84.4	1438.2	394.5	80809	44825	33957	27513	217086	119887
exp-1-500-5-5	0.0	0.0	0.0	0.0	0.0	0.0	406	407	419	407	398	397
f2000	99.8	34.6	40.0	161.7	77.0	72.2	173901	41281	46600	92942	56756	88238
farm	0.0	0.0	0.0	0.0	0.0	0.0	0	0	0	0	0	0
fast0507	2.0	2.2	1.9	2.1	4.5	2.7	4136	3379	3462	2848	12041	5685
fastxgemm-n2r6s0t2	0.1	0.2	0.1	0.2	0.1	0.3	1091	1740	2650	623	1407	2457
fffff800	0.1	0.1	0.1	0.1	0.1	0.1	135	280	268	255	415	224
fhnw-binpack4-4	0.0	0.0	0.0	0.0	0.0	0.0	523	446	620	696	629	468
fhnw-binpack4-48	0.2	0.1	0.1	0.1	0.1	0.1	3867	3737	4737	3609	3598	3572
fiball	0.4	0.4	0.4	0.5	0.6	0.3	1228	2601	3188	1725	3763	2397
fiber	0.0	0.0	0.0	0.0	0.0	0.0	209	149	142	159	244	199

Continued on next page

Table B.1.: LP solver comparison (`allLP` test set, SCIP 6.0.2)

	time						iters					
LP solver	CPLEX	Clp	Gurobi	MOSEK	SoPlex	Xpress	CPLEX	Clp	Gurobi	MOSEK	SoPlex	Xpress
instance	12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09	12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09
finnis	0.0	0.0	0.0	0.0	0.0	0.0	207	208	184	176	507	231
fit1d	0.1	0.1	0.1	0.1	0.1	0.1	74	53	67	132	192	78
fit1p	0.1	0.1	0.1	0.1	0.1	0.1	528	796	532	767	2980	1166
fit2d	0.3	0.2	0.3	0.3	0.3	0.2	218	168	155	182	228	177
fit2p	0.8	1.7	1.4	1.4	2.6	1.7	5072	6369	7049	5262	14574	9593
fixnet6	0.0	0.0	0.0	0.0	0.0	0.0	52	72	60	85	82	66
flugpl	0.0	0.0	0.0	0.0	0.0	0.0	9	9	6	11	8	8
fome11	7.2	16.5	10.6	16.8	39.9	20.1	17877	25056	21509	24053	40353	33068
fome12	14.2	33.2	21.3	31.5	74.1	38.3	16935	23011	19793	22037	39193	36347
fome13	29.8	64.8	42.4	63.9	166.1	76.9	19420	23801	19735	21214	39692	35080
fome20	2.5	18.1	2.6	2.6	15.9	4.5	13237	40883	14353	9369	35613	30670
fome21	5.9	46.7	6.1	6.2	43.4	11.7	16186	45694	13637	9563	37296	32196
forplan	0.0	0.0	0.0	0.0	0.0	0.0	129	188	186	138	245	228
fxm2-16	0.2	0.2	0.2	0.3	0.4	0.2	3784	3671	3367	3039	8534	4423
fxm2-6	0.1	0.1	0.1	0.1	0.1	0.1	1507	1442	1268	1049	2814	1684
fxm3_16	2.5	3.3	2.4	6.7	15.8	1.8	43170	43688	34486	32527	47246	39503
fxm3_6	0.3	0.3	0.3	0.6	0.9	0.4	5799	5824	4887	5014	11466	7233
fxm4_6	1.2	1.4	1.4	3.8	5.3	1.5	27699	22403	19933	20382	26811	26012
g200x740i	0.0	0.0	0.0	0.0	0.0	0.0	309	315	352	350	337	312
gams10a	0.0	0.0	0.0	0.0	0.0	0.0	0	0	0	0	0	0
gams30a	0.0	0.0	0.0	0.0	0.0	0.0	0	0	0	0	0	0
ganges	0.0	0.1	0.1	0.1	0.0	0.1	162	226	215	232	230	224
ge	0.5	0.6	0.5	0.6	2.3	0.6	4979	5055	3708	4412	10771	6194
gen	0.0	0.0	0.0	0.0	0.0	0.0	218	155	150	166	186	250
gen-ip002	0.0	0.0	0.0	0.0	0.0	0.0	51	46	37	67	44	53
gen-ip054	0.0	0.0	0.0	0.0	0.0	0.0	29	29	37	47	29	32
gen1	0.4	0.4	0.4	2.1	0.4	0.4	815	314	332	3521	570	447
gen2	4.2	5.6	6.8	7.8	10.4	6.9	4581	6872	5496	4325	12596	6291
gen4	1.2	3.1	1.1	82.0	3.9	1.2	811	1962	733	29560	7613	1052
ger50_17_trans	0.9	1.2	1.0	0.6	2.3	1.0	3154	2860	3543	1679	8182	3835
germanrr	0.6	0.6	0.7	0.8	2.1	0.9	2133	1685	3389	1794	7966	4905
germany50-DBM	0.4	0.2	0.3	0.6	0.4	0.7	4200	2975	5283	3434	6329	8145
gesa2	0.0	0.0	0.0	0.0	0.0	0.0	468	439	560	466	705	507
gesa2-o	0.0	0.0	0.0	0.0	0.0	0.0	350	440	343	374	463	404

Continued on next page

Table B.1.: LP solver comparison (`allLP` test set, SCIP 6.0.2)

	time						iters					
LP solver	CPLEX	Clp	Gurobi	MOSEK	SoPlex	Xpress	CPLEX	Clp	Gurobi	MOSEK	SoPlex	Xpress
instance	12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09	12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09
gesa2_o	0.0	0.0	0.0	0.0	0.0	0.0	350	440	343	374	463	404
gesa3	0.0	0.0	0.0	0.0	0.0	0.0	533	563	590	508	731	552
gesa3_o	0.0	0.0	0.0	0.0	0.0	0.0	405	503	408	398	533	448
gfd-schedulen180f7d50m30k	68.7	45.1	37.2	3600.0*	169.7	6.3	61633	68172	49298	214000*	78410	79144
gfrd-pnc	0.0	0.0	0.0	0.0	0.0	0.0	308	352	173	275	411	348
glass-sc	0.2	0.2	0.1	0.3	0.2	0.6	1145	557	1715	527	880	2363
glass4	0.0	0.0	0.0	0.0	0.0	0.0	38	72	36	37	74	72
gmu-35-40	0.0	0.0	0.0	0.0	0.0	0.0	200	375	351	714	769	542
gmu-35-50	0.0	0.0	0.0	0.1	0.1	0.0	612	523	530	948	1505	714
gmut-75-50	1.3	1.9	1.5	4.2	5.8	2.0	2115	3512	3561	2412	11623	7092
gmut-77-40	0.5	0.6	0.5	0.8	2.2	0.6	1651	2459	2619	3727	11918	4121
go19	0.0	0.0	0.0	0.1	0.1	0.1	865	696	857	758	2379	1091
gr4x6	0.0	0.0	0.0	0.0	0.0	0.0	7	10	4	12	11	10
graph20-20-1rand	0.2	0.1	0.3	0.3	0.3	0.2	1823	817	3566	2217	3238	2365
graphdraw-domain	0.0	0.0	0.0	0.0	0.0	0.0	138	163	272	204	233	290
greenbea	0.3	0.4	0.3	0.3	2.3	0.3	2893	3253	2174	1840	17153	2880
greenbeb	0.3	0.7	0.3	0.5	2.4	0.4	2907	5839	2957	3540	17506	4971
grow15	0.1	0.0	0.1	0.1	0.1	0.1	1117	717	1193	482	1761	1349
grow22	0.1	0.1	0.1	0.1	0.1	0.1	1737	1631	2293	849	2990	2195
grow7	0.0	0.0	0.0	0.0	0.0	0.0	315	232	374	209	554	526
gt2	0.0	0.0	0.0	0.0	0.0	0.0	29	29	30	42	53	22
h80x6320d	0.2	0.2	0.2	0.2	0.2	0.1	169	204	471	187	207	223
hanoi5	0.2	0.6	0.2	0.6	0.4	0.5	5138	4211	4221	2987	4805	5866
haprp	0.0	0.0	0.0	0.0	0.0	0.0	589	590	618	252	722	562
harp2	0.0	0.0	0.0	0.0	0.0	0.0	384	252	353	415	733	324
highschool1-aigio	3600.0*	3600.0*	3594.6*	3600.0*	3600.0*	3598.1*	245613*	239314*	477333*	249000*	178439*	622473*
hypothyroid-k1	4.4	4.2	3.3	4.7	4.3	3.1	2641	2628	2740	2903	3889	3367
i_n13	40.1	371.7	30.3	51.1	65.5	31.1	96062	170755	98621	69751	77329	35001
ic97_potential	0.0	0.0	0.0	0.0	0.0	0.0	405	486	279	504	623	493
icir97_tension	0.1	0.1	0.1	0.1	0.1	0.0	4	12	5	256	11	19
iiasa	0.0	0.0	0.1	0.0	0.0	0.0	885	860	1240	1131	1977	974
iis-100-0-cov	0.1	0.1	0.0	0.1	0.1	0.2	1221	327	1051	388	523	914
iis-bupa-cov	0.2	0.2	0.1	0.4	0.2	0.5	1809	839	2378	972	1333	2616
iis-pima-cov	0.2	0.4	0.3	0.5	0.3	0.8	1420	813	2912	900	1105	2919

Continued on next page

Table B.1.: LP solver comparison (`allLP` test set, SCIP 6.0.2)

	time						iters					
LP solver	CPLEX	Clp	Gurobi	MOSEK	SoPlex	Xpress	CPLEX	Clp	Gurobi	MOSEK	SoPlex	Xpress
instance	12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09	12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09
in	3590.1*	3600.0*	3579.7*	3600.0*	3600.0*	3576.1*	316414*	432968*	433013*	53000*	123020*	252206*
irish-electricity	39.7*	110.3*	84.8*	199.0	149.5*	87.8*	37418*	66560*	87537*	64299	75269*	83059*
irp	0.5	0.5	0.5	0.5	0.6	0.5	207	250	205	260	511	335
israel	0.0	0.0	0.0	0.0	0.0	0.0	122	103	207	134	211	148
istanbul-no-cutoff	0.6	0.7	0.5	1.3	0.3	0.3	4058	3814	3551	2774	3725	3868
ivu06-big	2672.4	3098.5	3586.9*	3600.0*	3600.0*	3590.2*	60220	64639	172244*	47000*	29876*	95176*
ivu52	111.7	82.0	140.4	99.2	3600.0*	149.4	65376	44285	436929	47497	297760*	85471
janos-us-DDM	0.0	0.0	0.0	0.0	0.0	0.1	823	920	1078	844	1091	1629
jendrec1	0.7	1.3	1.0	1.2	1.8	0.8	3214	4270	4239	4885	10237	3789
k16x240	0.0	0.0	0.0	0.0	0.0	0.0	16	40	17	17	40	38
k1mushroom	27.9	28.8	22.4	26.8	26.0	17.2	4354	4461	4515	4439	4923	4390
karted	3600.0*	3600.0*	3586.2*	3600.0*	3600.0*	3598.0*	139305*	79452*	167585*	221000*	70300*	96769*
kb2	0.0	0.0	0.0	0.0	0.0	0.0	43	52	37	42	55	29
ken-07	0.0	0.0	0.0	0.1	0.0	0.0	808	1031	803	1053	1421	1004
ken-11	0.2	0.3	0.3	0.3	0.5	0.2	5428	6586	5343	6724	9185	6535
ken-13	0.6	0.7	0.7	1.0	2.1	0.6	12915	13441	13159	13809	16740	13314
ken-18	3.0	4.7	2.9	6.0	22.5	3.1	45728	48661	46037	49840	60948	49251
kent	0.3	0.3	0.3	0.3	0.3	0.3	0	0	0	0	0	0
khb05250	0.0	0.0	0.0	0.0	0.0	0.0	16	138	26	84	148	124
kl02	0.5	0.5	0.5	0.5	0.6	0.5	347	254	208	304	710	453
kleemin3	0.0	0.0	0.0	0.0	0.0	0.0	0	0	0	0	0	0
kleemin4	0.0	0.0	0.0	0.0	0.0	0.0	0	0	0	0	0	0
kleemin5	0.0	0.0	0.0	0.0	0.0	0.0	0	0	0	0	0	0
kleemin6	0.0	0.0	0.0	0.0	0.0	0.0	0	0	0	0	0	0
kleemin7	0.0	0.0	0.0	0.0	0.0	0.0	0	0	0	0	0	0
kleemin8	0.0	0.0	0.0	0.0	0.0	0.0	0	0	0	0	0	0
l152lav	0.1	0.1	0.1	0.1	0.1	0.1	387	402	367	377	1528	551
l30	4.5	3600.0*	2.6	907.2	45.0	4.2	17856	3704169*	6643	1151327	40199	11827
l9	0.0	3600.0*	0.0	0.0	0.0	0.0	82067	451577*	344	470	850	423
large000	0.1	0.3	0.1	0.2	0.2	0.1	1277	5046	1671	2624	3638	3324
large001	0.5	0.4	0.3	0.8	1.0	0.2	7042	5190	3657	5957	9664	4470
large002	0.1	1.4	0.2	0.3	0.4	0.2	1598	8524	1669	2708	3804	3475
large003	0.1	0.8	0.1	0.2	0.3	0.2	1427	6575	1673	2643	4362	3540
large004	0.1	0.9	0.2	0.2	0.5	0.1	1641	6119	1826	2658	5431	3416

Continued on next page

Table B.1.: LP solver comparison (`allLP` test set, SCIP 6.0.2)

		time						iters					
	LP solver	CPLEX	Clp	Gurobi	MOSEK	SoPlex	Xpress	CPLEX	Clp	Gurobi	MOSEK	SoPlex	Xpress
instance		12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09	12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09
large005		0.1	1.1	0.1	0.2	0.4	0.1	1540	7585	1898	2863	4768	3480
large006		0.1	0.8	0.2	0.2	0.4	0.1	1373	6127	1848	2775	5259	3511
large007		0.1	0.6	0.2	0.3	0.4	0.2	1591	5089	1999	2919	4773	3554
large008		0.1	0.6	0.2	0.2	0.4	0.2	1614	5506	1851	2683	5225	3577
large009		0.1	1.2	0.2	0.2	0.4	0.2	1573	7800	2101	2675	4842	3543
large010		0.2	0.7	0.2	0.2	0.5	0.1	1659	5866	1877	2671	5988	3615
large011		0.1	0.5	0.2	0.2	0.4	0.2	1509	4821	1869	2752	5291	3708
large012		0.1	0.8	0.2	0.2	0.5	0.2	1565	5996	1923	2861	5777	3755
large013		0.1	0.6	0.2	0.2	0.4	0.2	1520	5332	1990	2919	5085	3707
large014		0.1	0.9	0.2	0.2	0.4	0.2	1680	7339	1942	3087	4967	3730
large015		0.1	0.6	0.2	0.2	0.3	0.2	1459	6642	1883	2975	3787	3602
large016		0.1	0.5	0.1	0.2	0.3	0.2	1305	4438	1882	2976	4084	3697
large017		0.1	0.9	0.1	0.2	0.3	0.2	1333	8214	1962	2906	4427	3740
large018		0.1	0.5	0.1	0.2	0.2	0.1	1375	6271	1900	2998	4251	3672
large019		0.1	0.4	0.1	0.2	0.4	0.1	1676	5283	1839	2950	4845	3710
large020		0.1	0.7	0.1	0.2	0.3	0.2	1646	5434	2034	3014	4636	3927
large021		0.1	0.9	0.2	0.2	0.4	0.2	1779	7235	2088	3033	5502	3786
large022		0.1	0.6	0.2	0.2	0.3	0.1	1476	5895	1984	2611	4256	3606
large023		0.2	1.2	0.1	0.2	0.5	0.1	2177	7227	1597	2608	6467	3159
large024		0.2	1.1	0.2	0.3	0.4	0.2	1601	8734	1501	2319	4507	3379
large025		0.2	1.6	0.1	0.3	0.5	0.2	1625	8824	1452	2406	4978	3259
large026		0.2	1.3	0.2	0.3	0.4	0.2	1690	9823	1367	2339	4645	3151
large027		0.2	0.9	0.2	0.3	0.4	0.2	1789	6213	1515	2206	3939	3376
large028		0.2	0.8	0.2	0.3	0.6	0.3	1654	5478	1472	2224	5084	3659
large029		0.3	1.0	0.2	0.3	0.3	0.2	1815	6035	1432	2136	3416	3194
large030		0.2	1.2	0.2	0.3	0.3	0.2	1539	7303	1392	2188	3503	3382
large031		0.2	1.4	0.2	0.3	0.5	0.2	1508	8398	1336	2192	4567	3189
large032		0.2	2.4	0.2	0.3	0.4	0.2	1778	14305	1353	2169	3696	3288
large033		0.2	1.0	0.2	0.2	0.4	0.2	1873	6542	1461	2005	4311	3197
large034		0.2	0.9	0.2	0.3	0.5	0.2	1577	6519	1358	2020	4436	3348
large035		0.2	1.3	0.2	0.3	0.4	0.2	1841	7975	1597	2066	3749	3220
large036		0.2	1.1	0.2	0.2	0.3	0.2	2681	7458	1425	1956	3877	3162
lectsched-1		0.4	0.4	0.4	0.4	0.4	0.4	0	0	0	0	0	0
lectsched-1-obj		0.4	0.4	0.4	0.4	0.4	0.4	0	0	0	0	0	0

Continued on next page

Table B.1.: LP solver comparison (`allLP` test set, SCIP 6.0.2)

	time						iters					
LP solver	CPLEX	Clp	Gurobi	MOSEK	SoPlex	Xpress	CPLEX	Clp	Gurobi	MOSEK	SoPlex	Xpress
instance	12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09	12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09
lectsched-2	0.2	0.2	0.2	0.2	0.2	0.2	0	0	0	0	0	0
lectsched-3	0.3	0.3	0.3	0.4	0.4	0.3	0	0	0	0	0	0
lectsched-4-obj	0.1	0.1	0.1	0.1	0.1	0.1	0	0	0	0	0	0
lectsched-5-obj	0.3	0.3	0.3	0.3	0.3	0.3	0	0	0	0	0	0
leo1	0.1	0.1	0.1	0.2	0.2	0.1	342	246	256	316	1060	490
leo2	0.3	0.3	0.3	0.3	0.5	0.3	512	294	469	342	1240	650
liu	0.0	0.0	0.0	0.0	0.0	0.0	540	537	534	538	564	552
lo10	11.0	71.5	12.8	17.6	305.9	17.8	18177	55175	22512	37980	76680	54200
long15	23.2	48.3	24.3	56.2	127.8	32.2	44375	45741	44644	55761	64118	52140
lotfi	0.0	0.0	0.0	0.0	0.0	0.0	120	196	120	192	179	134
lotsize	0.0	0.0	0.0	0.1	0.1	0.0	895	1487	897	1293	1462	1463
lp22	7.2	11.0	9.9	23.4	13.4	14.0	21963	23312	25624	32507	30490	38065
lpl1	6.8	25.3	3600.0*	20.2	47.7	18.1	33885	24422	0*	28801	33879	48293
lpl2	0.1	0.1	0.1	0.1	0.1	0.1	1176	1268	1549	1176	1635	2010
lpl3	0.8	0.5	0.5	0.6	0.8	0.7	7130	4267	4363	4014	5546	6781
lrn	1.2	0.9	0.5	0.5	0.8	0.8	10619	6214	4081	2538	7190	7223
lrsa120	1.1	1.7	0.7	0.3	1.0	0.6	10982	6678	6554	1312	8713	5247
lseu	0.0	0.0	0.0	0.0	0.0	0.0	18	32	23	23	20	23
m100n500k4r1	0.0	0.0	0.0	0.0	0.1	0.0	381	419	476	395	3423	822
macrophage	0.0	0.0	0.0	0.0	0.0	0.0	677	682	691	684	660	691
mad	0.0	0.0	0.0	0.0	0.0	0.0	82	280	68	34	56	80
manna81	0.1	0.1	0.1	0.1	0.1	0.1	2877	2828	2638	2816	2882	2881
map06	8.0	5.2	4.0	7.3	9.9	5.7	29445	12963	16984	11990	19635	25638
map10	7.2	4.9	3.6	7.9	9.5	5.7	28337	12411	15979	12867	19244	25597
map14	6.8	4.4	3.5	7.3	9.2	5.9	25994	11496	15551	11837	18431	28390
map16715-04	9.2	5.3	4.0	8.1	9.8	5.6	33300	13007	17632	12889	19254	25481
map18	5.7	3.9	3.3	19.4	8.1	4.8	21979	10601	14871	25189	17278	25514
map20	5.0	3.5	2.9	5.3	7.6	4.5	19678	9744	12789	9569	16894	24207
markshare1	0.0	0.0	0.0	0.0	0.0	0.0	23	23	22	21	11	11
markshare2	0.0	0.0	0.0	0.0	0.0	0.0	27	29	26	23	12	13
markshare_4_0	0.0	0.0	0.0	0.0	0.0	0.0	14	12	14	7	5	8
markshare_5_0	0.0	0.0	0.0	0.0	0.0	0.0	18	19	19	13	9	10
maros	0.1	0.1	0.1	0.1	0.1	0.1	1026	969	675	901	2093	952
maros-r7	0.6	0.8	0.6	0.8	0.8	0.6	2570	2355	2405	2501	3936	2651

Continued on next page

Table B.1.: LP solver comparison (`allLP` test set, SCIP 6.0.2)

	time						iters					
LP solver	CPLEX	Clp	Gurobi	MOSEK	SoPlex	Xpress	CPLEX	Clp	Gurobi	MOSEK	SoPlex	Xpress
instance	12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09	12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09
mas74	0.0	0.0	0.0	0.0	0.0	0.0	30	78	53	92	47	32
mas76	0.0	0.0	0.0	0.0	0.0	0.0	25	32	38	44	39	31
maxgasflow	0.7	0.9	0.8	0.8	0.7	0.9	0	0	0	0	0	0
mc11	0.0	0.0	0.0	0.0	0.0	0.0	400	404	621	694	404	400
mcf2	0.0	0.1	0.1	0.1	0.1	0.1	636	1302	1069	1063	3377	1140
mcsched	0.1	0.2	0.3	0.3	0.4	0.3	3488	3257	3274	3486	5795	4434
methanosarcina	0.2	0.1	0.2	0.2	0.1	0.1	652	654	649	686	626	692
mik-250-1-100-1	0.0	0.0	0.0	0.0	0.0	0.0	103	102	100	105	103	103
mik-250-20-75-4	0.0	0.0	0.0	0.0	0.0	0.0	81	81	81	90	81	79
milo-v12-6-r2-40-1	0.2	0.4	0.1	0.1	0.2	0.1	2335	4949	2883	1712	4270	2902
mine-166-5	0.2	0.1	0.1	0.4	0.2	0.2	2391	907	1624	1074	1734	1693
mine-90-10	0.2	0.1	0.2	0.3	0.1	0.2	2484	1154	3196	1381	1984	2121
mining	858.0	1748.9	1398.0	1029.2	3600.0*	209.7	249 547	341 705	406 717	239 454	177 472*	240 868
misc03	0.0	0.0	0.0	0.0	0.0	0.0	82	54	85	72	63	60
misc06	0.0	0.0	0.0	0.0	0.0	0.0	578	583	746	534	761	694
misc07	0.0	0.0	0.0	0.0	0.0	0.0	165	90	126	135	196	109
mitre	0.1	0.3	0.1	0.2	0.2	0.1	3051	5909	3217	2952	4616	2941
mkc	0.1	0.2	0.1	0.1	0.1	0.1	535	4295	1005	179	459	372
mkc1	0.1	0.2	0.1	0.1	0.1	0.1	535	4295	1005	179	459	372
mod008	0.0	0.0	0.0	0.0	0.0	0.0	11	15	7	14	7	13
mod010	0.1	0.1	0.1	0.1	0.2	0.1	752	657	871	696	2403	698
mod011	0.1	0.1	0.1	0.1	0.2	0.1	1341	2795	667	1351	6023	3727
mod2	8.2	19.8	7.4	17.9	61.2	25.1	30 647	32 264	27 035	26 406	46 709	53 126
model1	0.0	0.0	0.0	0.0	0.0	0.0	61	152	61	110	117	98
model10	2.5	14.6	4.0	12.3	11.4	7.3	16 709	46 669	24 473	39 617	33 626	37 738
model11	4.0	14.7	2.1	6.1	18.5	2.8	36 192	50 661	21 257	29 587	57 114	25 653
model2	0.1	0.1	0.1	0.1	0.1	0.1	1007	1764	1077	1353	4595	1686
model3	0.2	0.6	0.2	0.4	0.8	0.2	2769	7819	3566	3917	11 579	3770
model4	0.3	1.7	0.5	1.1	1.4	0.6	4354	13 496	6667	9575	13 346	7688
model5	0.6	2.2	0.8	1.6	3.0	0.7	8233	23 057	12 639	13 395	22 188	9597
model6	0.5	1.2	0.6	0.8	1.7	0.7	5744	9102	6272	4479	13 639	7249
model7	0.6	1.7	0.9	1.7	2.2	1.1	6433	10 965	9565	10 308	15 641	10 412
model8	0.2*	0.3	0.2*	0.3	0.3	0.2	981*	2609	1459*	1924	2967	2245
model9	0.4	0.6	0.5	1.2	1.3	0.5	7257	6171	6946	10 265	13 610	6154

Continued on next page

Table B.1.: LP solver comparison (`allLP` test set, SCIP 6.0.2)

	time						iters					
LP solver	CPLEX	Clp	Gurobi	MOSEK	SoPlex	Xpress	CPLEX	Clp	Gurobi	MOSEK	SoPlex	Xpress
instance	12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09	12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09
modglob	0.0	0.0	0.0	0.0	0.0	0.0	177	175	218	273	236	164
modszk1	0.0	0.0	0.0	0.0	0.0	0.0	65	605	99	951	601	624
momentum1	3.1	1.9	1.8	2.5	1.7	1.5	3511	3382	4409	2895	3533	3304
momentum2	1.4	10.2	0.7	2.3*	3600.0*	0.7	8892	18106	4745	3796*	0*	4559
momentum3	18.7	78.6	17.9	49.7	44.5	35.4	37706	43230	24690	21667	39478	69129
msc98-ip	2.9	19.4	0.6	3600.0*	2.3	0.9	14028	43658	5658	0*	11387	9796
mspp16	37.9	25.6	47.8	45.5	24.6	28.0	13	44	237	25	430	44
multi	0.0	0.0	0.0	0.0	0.0	0.0	29	43	34	28	35	30
mushroom-best	0.3	0.4	0.3	0.5	0.5	0.3	505	384	450	506	670	711
mzzv11	16.2	45.8	1.1	3.9	33.6	0.9	37864	75083	6604	9527	52825	8025
mzzv42z	8.5	8.9	0.7	2.0	15.3	0.6	23050	23739	4579	7187	32423	6246
n15-3	15.2	102.6	7.2	26.2	102.0	51.3	48519	54096	19555	21694	42401	32655
n2seq36q	0.8	0.6	0.7	0.9	1.0	0.6	4352	2851	3771	4110	5348	3619
n3-3	0.3	0.3	0.3	0.3	0.5	0.5	2190	2317	1840	1340	3888	3198
n3700	0.1	0.1	0.1	0.1	0.1	0.1	237	657	102	226	1061	676
n3701	0.1	0.1	0.1	0.1	0.1	0.1	217	688	99	279	1086	772
n3702	0.1	0.1	0.1	0.1	0.1	0.1	211	629	103	282	883	704
n3703	0.1	0.1	0.1	0.1	0.1	0.1	230	643	97	282	1014	695
n3704	0.1	0.1	0.1	0.1	0.1	0.1	238	653	87	241	916	733
n3705	0.1	0.1	0.1	0.1	0.1	0.1	213	709	120	246	895	781
n3706	0.1	0.1	0.1	0.1	0.1	0.1	194	672	112	309	947	721
n3707	0.1	0.1	0.1	0.1	0.1	0.1	242	623	96	253	892	739
n3708	0.1	0.1	0.1	0.1	0.1	0.1	226	600	99	275	987	615
n3709	0.1	0.1	0.1	0.1	0.1	0.1	227	718	85	264	817	773
n370a	0.1	0.1	0.1	0.1	0.1	0.1	221	704	75	278	937	743
n370b	0.1	0.1	0.1	0.1	0.1	0.1	209	700	103	260	931	728
n370c	0.1	0.1	0.1	0.1	0.1	0.1	211	724	79	286	1084	786
n370d	0.1	0.1	0.1	0.1	0.1	0.1	211	724	79	286	1084	786
n370e	0.1	0.1	0.1	0.1	0.1	0.1	200	689	90	254	966	778
n3div36	0.6	0.4	0.5	0.5	0.6	0.5	987	111	169	131	499	337
n3seq24	9.8	7.4	8.6	8.8	38.8	8.7	3903	2226	3541	2980	10389	3418
n4-3	0.1	0.1	0.1	0.1	0.1	0.1	810	628	651	396	1097	730
n5-3	0.0	0.0	0.1	0.1	0.1	0.1	666	522	503	462	1017	640
n9-3	0.3	0.3	0.2	0.2	0.3	0.3	2335	2266	1450	1137	2387	2393

Continued on next page

Table B.1.: LP solver comparison (`allLP` test set, SCIP 6.0.2)

	time						iters					
LP solver	CPLEX	Clp	Gurobi	MOSEK	SoPlex	Xpress	CPLEX	Clp	Gurobi	MOSEK	SoPlex	Xpress
instance	12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09	12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09
nag	0.2	0.1	0.2	0.3	0.2	0.2	163	282	136	2162	253	256
nb10tb	3600.0*	3600.0*	3600.0*	3600.0*	3600.0*	3600.0*	0*	0*	0*	0*	2 761 053*	0*
nemsafm	0.0	0.0	0.0	0.0	0.0	0.0	192	226	183	234	390	208
nemscem	0.0	0.0	0.0	0.0	0.0	0.0	394	467	363	389	831	399
nemsemm1	1.6	1.7	1.7	1.9	2.0	1.7	5807	5595	6336	6311	11 207	6415
nemsemm2	0.6	0.6	0.6	0.7	0.8	0.6	5788	6230	6387	5770	10 574	5789
nemspmm1	1.2	1.6	1.0	1.6	4.3	1.5	9007	9390	7972	7815	21 007	10 744
nemspmm2	1.1	2.0	1.2	1.7	2.5	1.4	7672	11 270	7918	8154	14 727	9491
nemswrld	9.0	20.5	9.7	20.2	29.6	20.5	29 111	45 740	36 752	36 797	45 784	59 871
neos	41.1	333.4	18.2	591.4	45.0	32.9	114 497	83 212	123 304	77 256	81 963	181 295
neos-1053234	0.0	0.0	0.1	0.1	0.0	0.0	271	443	259	226	263	455
neos-1053591	0.0	0.0	0.0	0.0	0.0	0.0	224	623	355	526	643	451
neos-1056905	0.0	0.0	0.0	0.0	0.0	0.0	210	210	209	210	231	210
neos-1058477	0.0	0.0	0.0	0.0	0.0	0.0	52	50	73	67	60	47
neos-1061020	1.0	1.6	1.4	2.2	2.4	1.1	6955	7759	5740	8293	11 429	7512
neos-1062641	0.0	0.0	0.0	0.0	0.0	0.0	126	320	178	246	329	286
neos-1067731	0.6	1.4	0.8	1.2	2.2	0.7	6579	10 025	6795	9586	14 143	9044
neos-1096528	2.8	2.7	3.0	3.1	2.6	2.9	129	143	176	157	136	158
neos-1109824	0.2	0.1	0.1	0.1	0.1	0.1	129	143	166	148	154	163
neos-1112782	0.1	0.1	0.1	0.1	0.1	0.0	212	267	1805	212	622	223
neos-1112787	0.0	0.0	0.1	0.1	0.1	0.1	182	225	2096	223	503	213
neos-1120495	0.1	0.1	0.1	0.1	0.1	0.1	102	101	140	96	102	114
neos-1121679	0.0	0.0	0.0	0.0	0.0	0.0	23	23	22	21	11	11
neos-1122047	5.9	5.8	6.1	6.9	6.2	6.0	1400	1490	5793	1016	6076	1498
neos-1126860	5.8	5.9	6.1	8.0	6.0	5.9	2005	2051	4585	4200	6696	5844
neos-1140050	21.1	32.7	13.3	10.9	23.9	24.5	25 913	20 147	13 297	10 674	13 807	21 254
neos-1151496	0.1	0.1	0.2	0.1	0.8	0.1	1207	897	1645	527	10 941	844
neos-1171448	0.2	0.4	0.2	0.6	1.1	0.3	1030	3957	3506	3112	9195	1481
neos-1171692	0.1	0.2	0.1	1.9	0.1*	0.1	468	1466	1463	5828	1868*	692
neos-1171737	0.1	0.2	0.1	0.4	0.1	0.1	662	1930	2099	2239	2267	867
neos-1173026	0.0	0.0*	0.0	0.0	0.0	0.0	6	15*	98	101	33	28
neos-1200887	0.0	0.0	0.0	0.0	0.0	0.0	380	284	225	372	226	179
neos-1208069	0.1	0.1	0.1	0.2	0.7	0.2	1892	1153	1038	1566	10 730	1656
neos-1208135	0.1	0.2	0.1	0.2	0.7	0.1	1441	1494	1054	1480	10 775	1754

Continued on next page

Table B.1.: LP solver comparison (`allLP` test set, SCIP 6.0.2)

	time						iters					
LP solver	CPLEX	Clp	Gurobi	MOSEK	SoPlex	Xpress	CPLEX	Clp	Gurobi	MOSEK	SoPlex	Xpress
instance	12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09	12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09
neos-1211578	0.0	0.0	0.0	0.0	0.0	0.0	120	115	66	102	118	81
neos-1215259	0.1	0.1	0.1	0.3	0.4	0.2	1495	1109	1297	1965	6457	2266
neos-1215891	0.0	0.0	0.0	0.0	0.0	0.0	0	0	0	0	0	0
neos-1223462	0.6	5.3	0.2	0.4	2.1	0.4	3565	21770	2506	1617	11893	3312
neos-1224597	0.1	0.2	0.1	0.1	0.3	0.1	698	2495	1235	585	5007	1404
neos-1225589	0.0	0.0	0.0	0.0	0.0	0.0	101	115	74	101	425	195
neos-1228986	0.0	0.0	0.0	0.0	0.0	0.0	119	123	80	100	110	76
neos-1281048	0.0	0.1	0.1	0.1	0.2	0.1	563	1167	595	860	5810	1169
neos-1311124	0.0	0.0	0.0	0.0	0.0	0.0	132	501	132	147	269	179
neos-1324574	0.4	0.2	0.1	0.2	0.7	0.5	3058	2010	1383	1624	5828	3625
neos-1330346	0.2	0.1	0.1	0.1	0.1	0.2	1602	967	1086	770	1155	1665
neos-1330635	0.1	0.0	0.1	0.0	0.0	0.0	710	596	473	450	318	435
neos-1337307	0.3	0.6	0.1	0.4	0.4	0.2	4693	4737	4894	3486	5133	4016
neos-1337489	0.0	0.0	0.0	0.0	0.0	0.0	120	115	66	102	118	81
neos-1346382	0.0	0.0	0.0	0.2	0.0	0.0	189	209	112	4411	246	118
neos-1354092	1.8	69.0	1.7	6.3	65.8	2.5	5159	125484	7038	15554	97398	8766
neos-1367061	5.6	2.7	10.2	8.6	4.0	5.2	6358	6656	13272	7598	7387	11258
neos-1396125	0.1	0.1	0.1	0.2	0.1	0.1	1264	1164	812	2093	3167	1358
neos-1407044	5.2	80.3	6.3	13.1	239.5	10.0	12290	100481	19426	22439	212852	24653
neos-1413153	0.1	0.1	0.1	0.1	0.1	0.1	1134	730	861	821	1438	1160
neos-1415183	0.1	0.1	0.1	0.1	0.1	0.1	1973	837	1136	1064	2595	1420
neos-1417043	8.9	2111.4	12.2	11.2	1250.2	9.7	3876	451398	8402	5678	60450	3025
neos-1420205	0.0	0.0	0.0	0.0	3600.0*	0.0	76	129	123	127	0*	258
neos-1420546	1.6	1.6	0.8	3.9	28.2	11.7	5791	4063	2357	6564	44666	27181
neos-1420790	0.1	0.2	0.1	0.3	0.8	0.4	807	1290	1096	2591	10851	4318
neos-1423785	0.5	1.0	0.4	0.7	3.9	0.7	8270	12230	4732	10471	17012	13182
neos-1425699	0.0	0.0	0.0	0.0	0.0	0.0	18	24	18	18	28	24
neos-1426635	0.0	0.0	0.0	0.2	0.0	0.0	189	209	112	4411	246	118
neos-1426662	0.0	0.0	0.0	0.1	0.0	0.0	203	386	225	1629	352	246
neos-1427181	0.0	0.0	0.0	0.1	0.0	0.0	196	344	136	951	340	235
neos-1427261	0.0	0.0	0.0	0.1	0.0	0.0	228	511	241	1662	484	267
neos-1429185	0.0	0.0	0.0	0.1	0.0	0.0	182	267	111	964	273	190
neos-1429212	6.7	16.6	8.1	9.5	16.6	7.9	10415	12624	13004	8565	13945	9948
neos-1429461	0.0	0.0	0.0	0.1	0.0	0.0	175	235	112	967	219	258

Continued on next page

Table B.1.: LP solver comparison (`allLP` test set, SCIP 6.0.2)

	time						iters					
LP solver	CPLEX	Clp	Gurobi	MOSEK	SoPlex	Xpress	CPLEX	Clp	Gurobi	MOSEK	SoPlex	Xpress
instance	12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09	12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09
neos-1430701	0.0	0.0	0.0	0.0	0.0	0.0	165	139	106	688	178	135
neos-1430811	10.0	42.5	12.4	15.7	34.9	29.8	15895	32019	45110	12520	18922	61792
neos-1436709	0.0	0.0	0.0	0.0	0.0	0.0	224	330	116	285	365	260
neos-1436713	0.0	0.0	0.0	0.1	0.0	0.0	274	562	284	1630	661	275
neos-1437164	0.0	0.0	0.0	0.1	0.0	0.0	413	303	306	783	689	628
neos-1439395	0.0	0.0	0.0	0.1	0.0*	0.0	203	143	84	1647	209*	122
neos-1440225	0.1	0.1	0.1	0.1	0.1	0.1	818	355	509	768	430	827
neos-1440447	0.0	0.0	0.0	0.0	0.0	0.0	141	120	97	97	124	85
neos-1440457	0.0	0.0	0.0	0.1	0.0	0.0	324	440	216	1150	410	469
neos-1440460	0.0	0.0	0.0	0.0	0.0	0.0	179	203	103	221	207	185
neos-1441553	0.0	0.0	0.0	0.0	0.0	0.0	247	366	310	552	1089	403
neos-1442119	0.0	0.0	0.0	0.1	0.0	0.0	199	341	165	1092	279	309
neos-1442657	0.0	0.0	0.0	0.1	0.0	0.0	182	267	111	1310	273	190
neos-1445532	0.2	0.2	0.2	0.2	0.1	0.2	0	0	0	0	0	0
neos-1445738	0.2	0.2	0.2	0.1	0.1	0.2	0	0	0	0	0	0
neos-1445743	0.2	0.2	0.2	0.2	0.1	0.2	0	0	0	0	0	0
neos-1445755	0.2	0.2	0.1	0.1	0.2	0.2	0	0	0	0	0	0
neos-1445765	0.2	0.2	0.2	0.2	0.2	0.2	0	0	0	0	0	0
neos-1451294	0.3	0.2	0.3	0.5	1.2	0.3	2694	1343	1978	2162	11945	2574
neos-1456979	0.1	0.2	0.1	0.2	0.2	0.2	431	461	502	636	541	714
neos-1460246	0.0	0.0	0.0	0.0	0.0	0.0	325	263	497	656	1727	489
neos-1460265	0.0	0.0	0.1	0.1	0.1	0.0	608	643	1432	914	993	565
neos-1460543	0.2	0.2	0.1	0.6	0.9	0.3	4009	1662	2060	4272	10156	4003
neos-1460641	0.1	0.1	0.1	0.2	0.3	0.1	1114	827	2765	1968	4586	1724
neos-1461051	0.1	0.0	0.1	0.1	0.0	0.1	384	145	308	181	196	268
neos-1464762	0.1	0.1	0.1	0.2	0.4	0.2	974	998	1760	1875	6313	3811
neos-1467067	0.0	0.0	0.0	0.0	0.0	0.0	80	510	140	172	181	99
neos-1467371	0.1	0.1	0.1	0.2	0.6	0.1	1325	1181	1724	2111	10303	2060
neos-1467467	0.1	0.1	0.2	0.2	0.3	0.1	1069	2462	3269	2139	6527	2030
neos-1480121	0.0	0.0	0.0	0.0	0.0	0.0	20	15	21	36	49	8
neos-1489999	0.0	0.0	0.0	0.0	0.0	0.0	613	485	509	504	686	559
neos-1516309	0.1	0.1	0.1	0.1	0.0	0.1	31	31	35	16	47	49
neos-1582420	0.1	0.1	0.1	0.2	0.2	0.1	1143	1063	1265	1078	2878	1516
neos-1593097	0.5	1.5	0.4	0.9	2.2	0.6	2612	10234	4384	4540	11099	2583

Continued on next page

Table B.1.: LP solver comparison (`allLP` test set, SCIP 6.0.2)

	time						iters					
LP solver	CPLEX	Clp	Gurobi	MOSEK	SoPlex	Xpress	CPLEX	Clp	Gurobi	MOSEK	SoPlex	Xpress
instance	12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09	12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09
neos-1595230	0.0	0.0	0.0	0.0	0.0	0.0	477	497	478	521	683	567
neos-1597104	1.0	2.8	1.9	110.1	4.5	3.2	1186	1407	23748	16433	3676	60503
neos-1599274	0.1	0.1	0.1	0.1	0.1	0.1	182	252	2471	66	512	379
neos-1601936	2.5	5.2	1.1	3.0	4.6	1.2	11175	17324	5150	7775	17686	6857
neos-1603512	0.0	0.0	0.1	0.1	0.2	0.0	415	576	439	341	3983	631
neos-1603518	0.1	0.1	0.1	0.1	0.7	0.1	1485	922	765	643	10454	1233
neos-1603965	3.0	1.0	0.6	1.9	1.5	0.6	0	0	0	0	0	0
neos-1605061	32.0	13.8	3.2	7.6	55.9	2.6	128498	27556	10599	15904	131469	10297
neos-1605075	4.8	15.8	2.7	4.0	9.0	2.7	15140	39954	9192	8324	24046	12674
neos-1616732	0.0	0.0	0.0	0.0	0.0	0.0	387	209	306	212	204	226
neos-1620770	0.1	0.1	0.1	0.1	0.1	0.1	668	792	799	715	792	945
neos-1620807	0.0	0.0	0.0	0.0	0.0	0.0	183	222	242	202	131	110
neos-1622252	0.1	0.1	0.1	0.1	0.1	0.1	711	807	842	769	498	699
neos-2075418-temuka	33.2	34.9	175.9	82.7	465.9	27.6	102381	43492	115361	86276	107855	69197
neos-2657525-crna	0.0	0.0	0.0	0.0	0.0	0.0	88	117	77	123	123	107
neos-2746589-doon	2.2	4.6	1.6	3.3	6.4	2.3	9288	10214	10176	13985	14363	10198
neos-2978193-inde	0.2	0.3	0.2	0.2	0.2	0.2	415	2304	684	606	1617	629
neos-2987310-joes	0.8	0.9	0.9	0.9	1.7	0.8	3496	3644	3282	1688	10630	3247
neos-3004026-krka	0.2	0.2	0.1	0.2	0.2	0.1	1537	4160	4226	1916	3155	4160
neos-3024952-loue	0.3	0.5	0.1	0.1	0.9	0.2	5286	5554	2250	1982	10899	5794
neos-3046615-murg	0.0	0.0	0.0	0.0	0.0	0.0	0	76	0	41	2	76
neos-3083819-nubu	0.2	0.2	0.2	0.2	0.2	0.2	504	515	587	458	1228	522
neos-3216931-puriri	3.6	5.5	2.5	2.6	1.9	1.5	12502	16149	9131	5445	7407	7332
neos-3381206-awhea	0.1	0.1	0.1	0.1	0.1	0.1	1007	2101	947	585	1474	991
neos-3402294-bobin	5.1	8.7	5.5	5.3	9.5	38.5	1759	3625	5620	436	6367	30342
neos-3402454-bohle	29.8	333.3	195.9	2662.8	3600.0*	109.9	1843	3875	57160	17988	39606*	412044
neos-3555904-turama	3.1	3.6	7.6	3.6	6.2	3.9	1714	8843	28160	1367	14618	7508
neos-3627168-kasai	0.0	0.1	0.0	0.0	0.0	0.1	559	1337	1162	532	1505	1539
neos-3656078-kumeu	0.3	5.0	0.3	0.4	1.2	0.3	5232	21903	4403	5271	16306	8075
neos-3754480-nidda	0.0	0.0	0.0	0.0	0.0	0.0	277	197	198	203	602	265
neos-3988577-wolgan	142.9	187.1	1.4	20.9	99.7	15.0	124623	163691	19861	23343	65509	42444
neos-4300652-rahue	1.0	1.0	2.1	17.9	0.9	0.8	2676	3003	5547	8142	2390	2591
neos-430149	0.0	0.0	0.0	0.0	0.0	0.0	147	120	141	126	268	120
neos-4338804-snowy	0.0	0.0	0.0	0.0	0.0	0.0	0	0	0	0	0	0

Continued on next page

Table B.1.: LP solver comparison (`allLP` test set, SCIP 6.0.2)

	time						iters					
LP solver	CPLEX	Clp	Gurobi	MOSEK	SoPlex	Xpress	CPLEX	Clp	Gurobi	MOSEK	SoPlex	Xpress
instance	12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09	12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09
neos-4387871-tavua	0.1	0.1	0.1	0.1	0.1	0.1	209	205	235	204	305	204
neos-4413714-turia	3.1	3.1	3.1	3.1	3.1	3.1	1576	1717	1476	1227	2474	2225
neos-4532248-waihi	3.3	2.6	3.8	4.9	2.7	3.1	1667	590	3596	2426	710	4103
neos-4647030-tutaki	3.8	4.6	3.6	3.6	6.4	4.1	2865	4748	3866	2688	5936	5585
neos-4722843-widden	3.9	3.5	2.9	5.2	3.6	2.5	9246	9510	5029	5957	5901	6743
neos-4738912-atrato	0.1	0.1	0.1	0.1	0.1	0.1	889	1238	857	1657	2096	1381
neos-476283	14.9	7.6	9.9	11.9	13.2	7.9	5570	2519	3996	5461	14428	5603
neos-4763324-toguru	6.4	6.8	6.3	15.4	20.9	5.8	7082	8104	13915	7949	13001	11079
neos-480878	0.1	0.1	0.1	0.1	0.1	0.1	293	312	327	523	603	451
neos-494568	0.3	0.4	0.3	0.3	0.3	0.3	330	4706	484	755	1705	664
neos-495307	0.1	0.1	0.1	0.1	0.1	0.1	7	3	3	3	4	4
neos-4954672-berkel	0.0	0.0	0.0	0.0	0.0	0.0	361	314	194	337	377	326
neos-498623	0.4	1.2	0.4	0.7	1.0	0.4	917	6386	896	2276	4921	1267
neos-501453	0.0	0.0	0.0	0.0	0.0	0.0	3	4	1	1	2	2
neos-501474	0.0	0.0	0.0	0.0	0.0	0.0	213	189	215	206	222	233
neos-503737	0.1	0.1	0.1	0.1	0.8	0.1	887	488	2445	631	11221	1335
neos-504674	0.0	0.0	0.0	0.0	0.0	0.0	331	370	349	472	367	374
neos-504815	0.0	0.0	0.0	0.0	0.0	0.0	271	308	295	404	305	330
neos-5049753-cuanza	7.4	7.5	7.8	10.8	14.5	7.5	5676	7454	6797	9408	7894	9278
neos-5052403-cygnet	68.6	119.3	63.2	576.3	108.1	21.1	26685	42471	35166	50574	35997	12785
neos-506422	0.1	0.1	0.1	0.1	0.1	0.1	145	123	141	45	112	78
neos-506428	1.7	1.1	1.3	1.4	1.2	1.1	1451	2092	2003	31	1005	292
neos-5093327-huahum	2.3	1.3	1.9	3.1	1.8	1.4	11590	2219	6972	5750	6260	3524
neos-5104907-jarama	28.9	21.6	16.0	76.3	60.8	23.4	22889	20778	18180	26566	24380	35257
neos-5107597-kakapo	0.1	0.1	0.1	0.2	0.1	0.1	59	1584	61	1529	174	1584
neos-5114902-kasavu	24.0	24.1	25.6	31.3	81.1	23.8	11325	12281	11282	14630	13826	19370
neos-512201	0.0	0.0	0.0	0.0	0.0	0.0	324	365	350	489	363	363
neos-5188808-nattai	0.6	0.4	0.7	0.9	0.5	0.5	2402	2367	3394	2763	2506	2618
neos-5195221-niemur	0.9	0.8	1.0	2.3	0.9	0.7	2720	2384	3449	2729	3304	2248
neos-520729	1.7	23.0	1.8	1.2	14.8	6.5	14041	89302	28564	131	40268	60706
neos-522351	0.0	0.0	0.0	0.0	0.0	0.0	9	1012	12	142	205	578
neos-525149	1.0	1.0	1.0	1.0	1.1	1.1	386	449	1306	302	1942	642
neos-530627	0.0	0.0	0.0	0.0	0.0	0.0	48	55	36	60	54	69
neos-538867	0.0	0.0	0.0	0.0	0.0	0.0	94	69	143	66	129	115

Continued on next page

Table B.1.: LP solver comparison (`allLP` test set, SCIP 6.0.2)

	time						iters					
LP solver	CPLEX	Clp	Gurobi	MOSEK	SoPlex	Xpress	CPLEX	Clp	Gurobi	MOSEK	SoPlex	Xpress
instance	12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09	12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09
neos-538916	0.0	0.0	0.0	0.0	0.0	0.0	86	79	147	78	102	119
neos-544324	1.5	1.4	1.4	1.7	1.9	1.5	1227	692	702	1045	3410	1595
neos-547911	0.4	0.4	0.4	0.5	0.5	0.4	1039	660	629	870	2341	1499
neos-548047	0.4	0.5	0.3	0.8	1.1	0.3	4335	4291	3852	5190	11798	4280
neos-548251	0.0	0.0	0.0	0.0	0.0	0.0	677	994	533	573	1370	991
neos-551991	0.2	0.1	0.1	0.3	0.1	0.1	2666	1234	1540	3124	2949	1195
neos-555001	0.1	0.1	0.1	0.1	0.1	0.1	584	1848	2600	476	3043	1584
neos-555298	0.1	0.1	0.1	0.1	0.1	0.1	717	2328	4011	568	3075	2182
neos-555343	0.1	0.1	0.1	0.2	0.2	0.1	1830	1787	2600	1838	8963	2012
neos-555424	0.1	0.1	0.1	0.1	0.1	0.1	1744	2231	2252	2016	3999	1939
neos-555694	0.1	0.1	0.1	0.1	0.1	0.1	447	2118	465	466	1195	643
neos-555771	0.1	0.2	0.1	0.1	0.1	0.1	325	3103	477	540	1211	719
neos-555884	0.1	0.1	0.1	0.2	0.2	0.1	2246	2909	1607	2028	4354	2536
neos-555927	0.1	0.1	0.1	0.1	0.1	0.1	91	230	100	93	290	176
neos-565672	88.6	27.6	62.9	35.6	143.0	70.5	122266	76298	693615	78911	86328	136075
neos-565815	0.7	0.6	0.5	1.3	1.3	0.6	3594	1712	4372	1685	5885	8722
neos-570431	0.0	0.1	0.1	0.1	0.1	0.0	718	1046	876	1017	1552	1081
neos-574665	0.1	0.1	0.1	0.1	0.1	0.1	115	475	958	116	755	524
neos-578379	3.9	2.8	1.9	9.4	42.9	4.2	12847	4856	4829	17001	26510	10316
neos-582605	0.0	0.0	0.0	0.0	0.0	0.0	418	503	317	387	548	707
neos-583731	0.0	0.0	0.0	0.0	0.0	0.0	70	108	40	12	85	21
neos-584146	0.0	0.0	0.0	0.0	0.0	0.0	328	396	317	310	296	346
neos-584851	0.0	0.0	0.0	0.0	0.0	0.0	358	419	420	359	260	503
neos-584866	0.4	0.6	0.8	1.8	0.5	1.6	3077	3683	4322	5211	4534	7203
neos-585192	0.5	0.5	0.6	0.6	0.8	0.6	689	949	1043	925	4391	1265
neos-585467	0.5	0.5	0.5	0.5	0.5	0.5	555	815	700	730	1554	986
neos-593853	0.0	0.0	0.0	0.0	0.0	0.0	392	624	219	431	628	589
neos-595904	0.1	0.1	0.1	0.1	0.2	0.1	805	827	858	714	1933	785
neos-595905	0.0	0.0	0.0	0.0	0.0	0.0	298	322	366	269	575	356
neos-595925	0.0	0.0	0.0	0.0	0.0	0.0	424	357	455	309	825	438
neos-598183	0.0	0.0	0.0	0.0	0.1	0.0	615	438	575	433	1152	652
neos-603073	0.0	0.0	0.0	0.1	0.0	0.0	366	538	626	402	715	579
neos-611135	1.0	1.6	0.9	1.8	2.7	0.9	1821	6357	1715	3505	10245	2389
neos-611838	0.1	0.1	0.1	0.2	0.2	0.1	1367	1413	1058	1530	1861	1469

Continued on next page

Table B.1.: LP solver comparison (`allLP` test set, SCIP 6.0.2)

	time						iters					
LP solver	CPLEX	Clp	Gurobi	MOSEK	SoPlex	Xpress	CPLEX	Clp	Gurobi	MOSEK	SoPlex	Xpress
instance	12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09	12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09
neos-612125	0.1	0.1	0.2	0.2	0.2	0.1	1551	1496	1218	1502	2054	1660
neos-612143	0.1	0.1	0.1	0.2	0.2	0.1	1453	1465	1166	1504	2025	1601
neos-612162	0.1	0.1	0.2	0.2	0.2	0.1	1440	1424	1175	1470	1890	1574
neos-619167	0.3	4.8	0.2	3600.0*	0.3	0.2	3087	24060	3052	0*	3830	3323
neos-631164	0.0	0.0	0.0	0.0	0.0	0.0	684	709	503	508	755	591
neos-631517	0.0	0.0	0.0	0.0	0.0	0.0	650	565	522	512	858	587
neos-631694	0.1	0.1	0.2	0.1	0.4	0.1	506	868	4543	888	6036	1899
neos-631709	8.5	5.4	72.8	44.2	65.3	36.6	9197	6713	149227	31573	30672	80830
neos-631710	296.8	184.5	3593.8*	1403.1	848.7	236.2	30820	45096	463328*	176586	49469	174892
neos-631784	1.2	1.0	14.0	25.8	18.7	6.7	3247	2310	66753	38815	21453	26904
neos-632335	0.3	0.3	0.3	3600.0*	0.8	0.2	1792	2107	1788	0*	6419	2014
neos-633273	0.2	0.3	3600.0*	3600.0*	0.8	0.2	1613	1962	0*	0*	7031	1875
neos-641591	0.9	1.1	0.9	1.4	3.6	1.5	3843	4351	4941	4018	12641	6794
neos-655508	0.2	0.1	0.2	0.2	0.2	0.2	0	119	13692	695	119	542
neos-662469	0.9	1.0	0.9	1.4	3.6	1.5	3843	4351	4941	4018	12641	6794
neos-686190	0.1	0.1	0.1	0.1	0.1	0.1	181	263	396	262	904	321
neos-691058	0.3	0.2	0.2	0.6	1.5	0.3	2977	1364	1643	2388	11471	2185
neos-691073	0.4	0.2	0.2	0.5	1.5	0.3	3305	1318	1595	2014	8281	1988
neos-693347	1.3	0.7	0.5	0.9	2.8	0.5	4894	1974	1360	1756	8854	1680
neos-702280	7.8	7.5	6.5	11.7	11.5	6.7	3794	4072	3463	5139	11677	7364
neos-709469	0.0	0.0	0.0	0.0	0.0	0.0	159	216	177	169	410	226
neos-717614	0.0	0.1	0.1	0.1	0.1	0.1	758	1175	856	1068	1159	1081
neos-738098	2.6	7.0	2.0	8.8	11.9	2.6	6811	11066	6636	9342	15997	8294
neos-775946	0.2	0.2	0.2	0.3	0.2	0.2	576	1889	635	1043	1215	1171
neos-777800	0.1	0.2	0.2	0.3	1.6	0.2	714	1111	1260	1663	12105	1194
neos-780889	7.9	28.7	6.1	11.4	30.1	21.0	10063	22446	12226	20613	22543	40273
neos-785899	0.1	0.1	0.1	0.1	0.0	0.1	374	317	417	375	373	348
neos-785912	0.1	0.1	0.1	0.1	0.3	0.1	546	397	479	501	2255	731
neos-785914	0.0	0.0	0.1	0.1	0.1	0.0	178	201	338	258	754	282
neos-787933	0.9	0.9	0.9	0.9	0.9	0.9	0	0	0	0	0	0
neos-791021	0.1	1.0	0.1	0.1	1.7	0.4	844	5281	3219	906	11487	4561
neos-796608	0.0	0.0	0.0	0.0	0.0	0.0	40	54	49	50	62	54
neos-799711	0.8	1.4	0.8	3.0	6.3	1.0	10672	13057	3998	11425	21731	11628
neos-799838	0.3	0.7	0.3	0.3	2.6	0.9	1978	2978	1235	527	9719	4696

Continued on next page

Table B.1.: LP solver comparison (`allLP` test set, SCIP 6.0.2)

	time						iters					
LP solver	CPLEX	Clp	Gurobi	MOSEK	SoPlex	Xpress	CPLEX	Clp	Gurobi	MOSEK	SoPlex	Xpress
instance	12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09	12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09
neos-801834	0.1	0.1	0.1	0.2	0.2	0.1	1007	1048	1013	1061	2036	1425
neos-803219	0.0	0.0	0.0	0.0	0.0	0.0	315	584	239	296	1026	585
neos-803220	0.0	0.0	0.0	0.0	0.0	0.0	339	587	230	126	1412	631
neos-806323	0.0	0.1	0.1	0.0	0.1	0.0	386	711	362	267	1086	781
neos-807454	0.2	0.4	0.2	0.3	0.8	0.2	2451	3320	1894	1965	10 442	2348
neos-807456	0.2	0.2	0.3	0.4	1.1	0.5	3323	2341	3514	2994	12 514	5018
neos-807639	0.0	0.0	0.1	0.0	0.1	0.1	443	759	378	285	1247	854
neos-807705	0.0	0.1	0.1	0.0	0.1	0.1	387	739	383	269	1727	925
neos-808072	0.2	0.3	0.2	0.3	0.7	0.2	1839	2718	1018	1260	9165	1788
neos-808214	0.1	0.1	0.1	0.1	0.6	0.1	943	478	924	556	10 573	845
neos-810286	0.8	1.4	0.6	1.2	1.9	0.8	5663	6940	3101	4241	13 305	5547
neos-810326	0.1	0.2	0.1	0.4	0.4	0.2	1563	1481	1049	2854	4800	2080
neos-820146	0.0	0.0	0.0	0.0	0.0	0.0	212	246	168	359	443	533
neos-820157	0.0	0.0	0.0	0.1	0.0	0.0	310	402	475	836	558	534
neos-820879	0.2	0.2	0.2	0.3	0.4	0.3	892	796	950	1027	2098	1289
neos-824661	0.7	5.5	0.7	3.1	3.7	0.6	3919	13 498	4689	9964	11 255	3125
neos-824695	0.3	1.5	0.3	0.7	1.6	0.3	2737	6887	2726	4384	8235	2068
neos-825075	0.0	0.0	0.0	0.1	0.0	0.0	320	233	362	301	1049	535
neos-826224	0.6	5.4	0.8	0.8	1.0	0.6	3585	13 362	24 483	3794	6411	3029
neos-826250	0.1	0.7	0.2	0.4	0.3	0.2	1559	4360	7096	2612	3993	1426
neos-826650	0.1	0.2	0.1	0.2	1.6	0.2	1645	2161	2915	1825	11 236	2077
neos-826694	0.3	1.1	0.3	0.8	2.3	0.3	2871	5523	9553	5281	11 100	3257
neos-826812	0.4	1.2	0.3	0.7	2.1	0.5	4453	5268	8903	4304	10 820	5856
neos-826841	0.1	0.2	0.1	0.2	0.8	0.1	1537	2284	2915	1766	8118	2428
neos-827015	5.6	8.0	7.6	10.6	11.9	18.6	15 376	17 533	15 479	14 190	19 016	33 774
neos-827175	0.6	3.2	0.7	0.9	3.8	0.7	4840	10 114	6039	4355	12 151	6169
neos-829552	1.8	2.4	1.9	2.5	4.2	3.8	5947	8404	5666	5189	12 881	13 940
neos-830439	0.0	0.0	0.0	0.0	0.0	0.0	76	140	106	85	74	176
neos-831188	0.1	0.2	0.1	0.3	0.7	0.4	2607	3274	1854	2542	8615	5641
neos-839838	0.5	0.3	0.6	0.8	1.1	0.4	5604	3698	2993	1871	7218	4992
neos-839859	0.1	0.0	0.1	0.1	0.1	0.1	1369	971	642	965	1879	1364
neos-839894	8.1	7.0	5.6	17.2	23.8	14.6	13 393	11 816	11 729	14 345	20 810	26 886
neos-841664	0.3	0.6	0.1	0.5	0.6	1.5	5252	6749	3506	5084	9017	18 427
neos-847051	0.1	0.1	0.1	0.1	0.2	0.1	1966	2474	1451	1707	5279	2273

Continued on next page

Table B.1.: LP solver comparison (`allLP` test set, SCIP 6.0.2)

	time						iters					
LP solver	CPLEX	Clp	Gurobi	MOSEK	SoPlex	Xpress	CPLEX	Clp	Gurobi	MOSEK	SoPlex	Xpress
instance	12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09	12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09
neos-847302	0.1	0.1	0.1	0.1	0.7	0.1	1105	840	1210	1194	10306	1333
neos-848150	0.1	0.1	0.1	0.1	0.5	0.1	993	424	1192	1020	10346	856
neos-848198	0.0	0.0	0.1	0.0	0.0	0.1	0	0	0	0	0	0
neos-848589	8.7	8.5	9.4	9.1	9.0	8.5	638	1401	664	640	1401	1388
neos-848845	0.3	0.2	0.3	0.2	1.1	0.1	2739	1465	2681	1482	12013	1255
neos-849702	0.3	0.1	0.3	0.2	1.1	0.1	3198	1021	2439	1459	11779	1275
neos-850681	0.1	0.1	0.1	0.2	0.6	0.1	1180	1108	1111	1113	8348	1175
neos-856059	0.2	0.1	0.1	0.1	0.1	0.1	1085	450	767	463	450	571
neos-859770	0.5	0.5	0.6	0.6	0.6	0.6	75	109	120	102	112	136
neos-860244	0.3	0.3	0.3	0.3	0.3	0.3	38	51	54	46	59	45
neos-860300	0.2	0.2	0.3	0.2	0.2	0.2	265	234	281	250	415	313
neos-862348	0.2	0.2	0.2	0.2	0.2	0.2	766	1233	587	744	610	1034
neos-863472	0.0	0.0	0.0	0.0	0.0	0.0	64	81	126	136	87	108
neos-872648	4.2	4.3	4.6	5.1	6.7	4.2	4269	3774	6252	3764	3908	3916
neos-873061	2.5	2.6	2.6	3.0	3.6	2.5	1880	1765	2672	1759	1800	1779
neos-876808	55.4	22.2	71.4	71.2	106.1	8.9	100931	58199	84105	69049	109207	80578
neos-880324	0.0	0.0	0.0	0.0	0.0	0.0	79	69	95	78	73	56
neos-881765	0.0	0.0	0.0	0.1	0.1	0.0	278	265	595	860	2734	486
neos-885086	0.6	1.4	0.5	2.4	0.7	0.4	3059	5328	3346	4038	4665	864
neos-885524	1.5	1.5	1.5	1.5	1.5	1.5	147	199	477	69	162	193
neos-886822	0.0	0.0	0.1	0.1	0.1	0.1	993	607	643	671	3028	1093
neos-892255	0.1	0.1	0.1	0.2	0.1	0.2	728	1200	704	1672	1434	2777
neos-905856	0.0	0.0	0.0	0.1	0.1	0.0	772	510	583	568	3001	859
neos-906865	0.1	0.1	0.1	0.1	0.1	0.0	410	736	558	564	801	661
neos-911880	0.0	0.0	0.0	0.0	0.0	0.0	189	148	190	174	737	380
neos-911970	0.0	0.0	0.0	0.0	0.0	0.0	192	146	251	148	616	310
neos-912015	0.0	0.0	0.1	0.1	0.2	0.0	791	317	485	448	4948	669
neos-912023	0.1	0.0	0.1	0.1	0.1	0.1	952	585	720	706	2758	995
neos-913984	1.1	7.7	1.2	1.3	10.9	1.1	1145	9959	2254	1387	11457	1355
neos-914441	0.8	0.5	0.5	1.1	0.8	0.6	6358	4115	4844	5393	9198	8672
neos-916173	0.2	0.2	0.2	0.2	0.2	0.2	197	250	362	271	430	318
neos-916792	0.3	0.4	0.4	0.4	0.4	0.4	291	311	397	455	531	476
neos-930752	0.6	7.9	0.3	2.1	3.6	2.5	9602	29891	9292	8612	15921	17066
neos-931517	0.5	0.7	0.4	0.3	1.9	0.5	7733	6268	6585	1896	13590	7716

Continued on next page

Table B.1.: LP solver comparison (`allLP` test set, SCIP 6.0.2)

	time						iters					
LP solver	CPLEX	Clp	Gurobi	MOSEK	SoPlex	Xpress	CPLEX	Clp	Gurobi	MOSEK	SoPlex	Xpress
instance	12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09	12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09
neos-931538	0.5	2.1	0.5	0.3	1.1	0.8	8989	11962	6911	1891	11739	9838
neos-932721	0.7	3.5	0.3	1.2	2.1	0.8	7734	15646	3864	6512	16079	8192
neos-932816	1.0	3.9	0.7	1.3	2.6	1.1	6561	14586	4733	4776	16476	7649
neos-933364	0.0	0.1	0.0	0.1	0.0	0.0	980	933	1069	1105	1597	1108
neos-933550	0.1	0.0	0.1	0.1	0.1	0.1	622	457	799	888	1368	759
neos-933562	0.1	0.1	0.2	0.2	0.9	0.2	1240	895	1805	1430	10796	1576
neos-933638	5.5	11.4	4.6	2.0	14.1	7.0	16010	21565	14386	4594	25071	19078
neos-933815	0.0	0.0	0.0	0.0	0.0	0.0	116	138	211	108	212	191
neos-933966	3.8	22.6	1.0	1.9	9.8	5.2	13841	41051	7080	5147	21418	17272
neos-934184	0.0	0.0	0.0	0.1	0.0	0.0	980	933	1069	1105	1597	1108
neos-934278	7.2	13.3	5.9	2.5	16.0	12.7	20461	25766	17318	6179	28571	32784
neos-934441	5.8	17.2	6.8	2.4	18.0	13.9	18183	31677	18851	5852	30779	35912
neos-934531	0.4	0.4	0.4	2.1	0.5	0.3	337	396	2859	1848	1254	209
neos-935234	6.0	17.4	1.4	3.3	18.3	13.9	19264	35775	10613	9086	30818	36378
neos-935348	6.1	13.4	1.1	2.6	16.1	10.9	18888	30266	11188	7867	29196	31677
neos-935496	0.1	0.1	0.2	0.2	0.8	0.1	1013	645	1352	903	10687	1171
neos-935627	6.4	17.7	1.6	2.8	17.2	13.3	20237	37467	12313	8853	30899	37057
neos-935674	0.1	0.1	0.2	0.2	0.9	0.1	936	627	1498	963	10370	1150
neos-935769	3.6	30.6	1.2	2.1	8.6	5.9	15081	65485	10472	7611	21054	22536
neos-936660	4.8	43.2	1.4	3.1	15.4	7.2	17714	85833	11877	9755	29285	25133
neos-937446	3.9	17.5	1.3	2.8	16.2	6.0	14883	36763	10901	8507	28452	21248
neos-937511	4.3	38.5	1.4	3.0	14.7	6.5	16473	71328	12030	9909	27382	22307
neos-937815	9.4	23.7	3.5	4.9	24.9	21.7	25260	44176	20938	12410	37328	51345
neos-941262	5.2	19.9	7.1	3.1	15.2	12.4	19176	45314	23334	10240	31534	38952
neos-941313	9.9	126.3	11.5	25.2	94.9	31.1	41718	79311	44583	51076	53635	89441
neos-941698	0.1	0.0	0.1	0.1	0.3	0.1	608	428	531	489	5308	737
neos-941717	0.1	0.1	0.1	0.2	0.7	0.1	1388	555	1338	980	10714	1562
neos-941782	0.1	0.1	0.1	0.1	0.6	0.1	1233	875	901	694	10288	1095
neos-942323	0.0	0.0	0.1	0.1	0.1	0.1	380	281	421	285	751	575
neos-942830	0.1	0.1	0.1	0.1	0.2	0.1	806	459	701	558	3889	967
neos-942886	0.0	0.0	0.0	0.0	0.1	0.0	467	175	312	245	1815	694
neos-948126	8.3	16.9	2.6	5.7	19.9	22.6	25704	35480	16879	15852	35898	55656
neos-948268	0.5	0.9	1.1	0.9	3.6	0.8	3129	3193	7074	3967	14227	3411
neos-948346	2.7	36.3	2.5	6.0	12.2	3.9	4229	52508	28367	14276	27784	9721

Continued on next page

Table B.1.: LP solver comparison (`allLP` test set, SCIP 6.0.2)

	time						iters					
LP solver	CPLEX	Clp	Gurobi	MOSEK	SoPlex	Xpress	CPLEX	Clp	Gurobi	MOSEK	SoPlex	Xpress
instance	12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09	12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09
neos-950242	0.3	0.3	0.3	0.5	0.3	0.3	797	897	279	853	524	677
neos-952987	0.3	0.3	0.3	0.4	0.3	0.3	451	595	523	608	542	500
neos-953928	6.5	46.5	0.5	2.7	19.9	2.4	30 338	91 093	10 499	11 942	28 703	15 516
neos-954925	7.5	135.6	4.6	48.9	30.4	12.0	11 679	118 266	34 595	28 315	23 448	18 662
neos-955215	0.0	0.0	0.0	0.0	0.0	0.0	80	88	81	96	122	87
neos-955800	0.1	0.1	0.1	0.1	0.0	0.1	249	341	1568	277	353	352
neos-956971	2.4	78.5	2.3	8.1	10.4	3.6	3571	110 755	23 954	12 703	23 015	8246
neos-957143	2.3	55.5	2.4	6.7	9.5	3.3	3117	75 270	30 311	13 029	15 332	10 220
neos-957270	1.1	1.1	1.2	1.2	1.0	1.1	185	484	614	485	794	280
neos-957323	1.8	18.0	1.6	5.4	7.0	2.3	6799	38 446	4538	12 330	15 081	8529
neos-957389	0.5	0.6	0.6	0.6	0.6	0.5	375	827	564	971	1173	480
neos-960392	1.7	5.3	1.0	3.7	11.9	1.2	7650	8573	31 298	14 090	20 638	5138
neos-983171	6.7	27.5	2.1	11.5	14.6	16.0	22 331	59 643	14 320	32 292	29 448	47 484
neos-984165	7.4	32.5	2.6	5.8	19.9	19.5	21 988	67 469	16 182	15 822	35 284	47 282
neos1	1.8	170.7	5.3	18.5	15.7	37.2	1945	46 393	44 935	4980	12 102	331 253
neos13	0.5	1.0	1.4	0.6	0.5	0.4	542	3372	2064	405	2007	391
neos15	0.0	0.0	0.0	0.0	0.0	0.0	216	209	192	215	198	197
neos16	0.0	0.0	0.0	0.0	0.0	0.0	250	262	355	249	249	248
neos17	0.0	0.0	0.0	0.0	0.0	0.0	483	464	607	511	809	487
neos18	0.1	0.1	0.1	0.1	0.1	0.1	1244	898	1099	860	958	1262
neos2	2.8	498.4	6.1	69.4	19.3	49.1	3197	108 939	38 903	10 934	13 923	402 948
neos3	3599.4*	1769.4	6.6	3600.0*	3600.0*	844.3	7 061 538*	66 433	14 033	68 000*	156 321*	750 135
neos5	0.0	0.0	0.0	0.0	0.0	0.0	132	79	71	108	113	101
neos6	0.5	0.4	0.4	0.9	1.2	0.5	1630	829	1026	2819	5869	1916
neos788725	0.0	0.0	0.0	0.1	0.1	0.1	680	606	868	710	1989	1281
neos8	1.4	1.2	1.2	1.5	1.3	1.1	588	471	644	583	1269	557
neos808444	0.6	1.9	0.5	0.6	4.4	0.5	2166	4602	2717	2263	10 867	5840
neos858960	0.0	0.0	0.0	0.0	0.0	0.0	3	5	3	6	6	3
neos859080	0.0	0.0	0.0	0.0	0.0	0.0	3	3	3	31	72	3
nesm	0.1	0.1	0.1	0.1	0.2	0.1	1152	1532	1239	1373	5027	1344
net12	1.3	0.5	0.4	1.1	1.0	0.4	7520	2038	2729	3224	7475	3359
netdiversion	9.0	7.9	7.1	95.7	15.6	7.2	19 536	26 333	24 076	51 983	21 402	33 243
netlarge2	33.9	1311.1	36.9	125.2	3600.0*	146.5	35 667	118 516	35 469	45 874	41 505*	46 609
netlarge3	95.8	3600.0*	101.9	148.2	3600.0*	162.6	33 980	91 713*	33 634	38 345	13 702*	40 947

Continued on next page

Table B.1.: LP solver comparison (`allLP` test set, SCIP 6.0.2)

	time						iters					
LP solver	CPLEX	Clp	Gurobi	MOSEK	SoPlex	Xpress	CPLEX	Clp	Gurobi	MOSEK	SoPlex	Xpress
instance	12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09	12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09
newdano	0.0	0.0	0.0	0.0	0.0	0.0	114	354	315	459	393	356
nexp-150-20-8-5	0.1	0.2	0.2	0.2	0.2	0.2	923	987	1297	1034	1582	1446
nl	0.6	0.8	0.6	1.3	2.9	1.1	8367	7454	7424	7809	13768	10493
nobel-eu-DBE	0.1	0.1	0.1	0.1	0.1	0.1	1464	1406	1311	877	3145	1879
noswot	0.0	0.0	0.0	0.0	0.0	0.0	133	120	251	86	142	106
npmv07	5.4*	5.8*	4.2*	7.2*	10.4*	4.1*	33795*	38588*	19343*	47410*	51061*	31745*
ns1111636	6.0	13.6	3.3	5.8	23.7	8.3	29279	20484	13562	12917	44412	46295
ns1116954	5.1	5.0	7.2	242.7	53.0	13.9	8243	3510	29365	38072	26273	18910
ns1208400	0.8	0.6	0.8	0.8	1.6	0.3	4486	2668	4236	2344	12456	2002
ns1456591	0.3	0.2	0.4	0.3	0.5	0.2	1048	556	1489	668	2430	650
ns1606230	3.1	9.2	2.2	3.1	5.7	2.2	12000	26648	7982	7079	19327	11457
ns1631475	23.0	7.3	16.9	22.9	39.2	26.1	33881	11447	28350	20288	36941	38626
ns1644855	183.7	555.7	149.3	433.4	205.6	91.2	95141	168056	83131	107792	76980	96547
ns1663818	51.0	47.5	53.8	58.1	50.3	50.2	496	1755	1207	7980	1594	15713
ns1685374	181.7	126.2	163.3	236.4	344.4	110.3	165932	57408	101371	63233	91471	76017
ns1686196	0.2	0.1	0.2	0.2	0.2	0.2	110	266	226	206	194	288
ns1687037	3577.8*	3600.0*	3600.0*	3600.0*	3600.0*	3589.2*	471089*	0*	551378*	457689*	127420*	680827*
ns1688347	0.2	0.2	0.3	0.3	0.2	0.2	375	423	1084	436	225	260
ns1688926	11.7	18.3	11.9	3599.1*	180.9	193.1	13784	13853	7711	951605*	28701	156671
ns1696083	0.9	0.8	0.9	1.0	0.8	0.9	448	403	501	433	411	564
ns1702808	0.0	0.0	0.0	0.0	0.0	0.0	53	92	252	109	64	59
ns1745726	0.2	0.2	0.2	0.2	0.2	0.2	124	311	243	224	218	267
ns1758913	7.1	41.6	9.2	49.0	4.1	4.5	5228	28732	22526	5598	776	4009
ns1760995	15.0	354.8	33.1	93.0	17.0	15.0	17474	112951	81154	10898	14847	22960
ns1766074	0.0	0.0	0.0	0.0	0.0	0.0	43	32	35	42	31	31
ns1769397	0.3	0.3	0.3	0.3	0.3	0.2	113	192	314	255	254	301
ns1778858	1.5	1.3	0.8	2.0	3.8	1.5	11710	7579	6820	8299	13448	10298
ns1830653	0.3	0.2	0.2	0.4	0.3	0.3	1971	1170	1071	1595	1793	2036
ns1853823	3495.5	3600.0*	2210.6	3600.0*	3600.0*	608.0	489600	606935*	775045	254000*	266145*	293320
ns1854840	3577.3*	3600.0*	139.0	3600.0*	3600.0*	3599.3*	698425*	687186*	146766	354000*	316411*	941278*
ns1856153	0.3	0.3	0.4	0.7	0.3	0.4	854	452	1388	1670	176	1124
ns1904248	26.5	7.0	64.8	154.8	16.1	4.8	19482	31285	53405	33436	18796	7392
ns1905797	0.8	1.4	0.8	1.8	4.8	1.0	1386	2699	962	2055	10612	3018
ns1905800	0.2	0.3	0.2	0.3	0.3	0.2	492	1125	427	677	1408	954

Continued on next page

Table B.1.: LP solver comparison (`allLP` test set, SCIP 6.0.2)

		time						iters					
instance	LP solver	CPLEX 12.8.0.0	CLP 1.16.11	GUROBI 8.1.0	MOSEK 8.1.0.21	SOPLEX 4.0.2	XPRESS 33.01.09	CPLEX 12.8.0.0	CLP 1.16.11	GUROBI 8.1.0	MOSEK 8.1.0.21	SOPLEX 4.0.2	XPRESS 33.01.09
ns1952667		0.8	0.8	0.8	0.9	0.9	0.9	136	64	68	131	143	310
ns2017839		9.5	20.1	5.5	97.8	125.1	9.0	57 597	48 848	33 112	107 217	77 096	49 136
ns2081729		0.0	0.0	0.0	0.0	0.0	0.0	99	31	237	261	101	237
ns2118727		95.9	95.1	223.1	56.7	70.2	51.3	47 949	37 460	168 752	19 822	22 139	48 075
ns2122603		361.0	3.3	6.0	13.4	5.4	3.8	511 778	13 908	39 759	27 278	12 707	20 381
ns2124243		5.9	89.0	2.8	19.3	98.0	18.0	39 891	105 542	21 702	30 017	74 517	52 265
ns2137859		4.5	4.9	4.7	4.5	6.8	4.4	1252	2294	5096	863	4926	1654
ns4-pr3		0.1	0.2	0.2	0.2	1.0	0.2	1903	1786	2847	2052	7999	4627
ns4-pr9		0.1	0.1	0.1	0.1	0.6	0.1	1633	1378	2248	1510	5492	3133
ns894236		1.8	2.2	1.6	3.0	4.9	2.6	10 001	9293	10 017	9966	19 087	17 171
ns894244		15.5	10.8	11.2	12.5	19.8	7.9	45 948	24 093	34 249	25 210	30 948	25 596
ns894786		9.6	3.1	12.3	28.9	15.1	6.7	33 546	7344	36 873	57 984	19 861	24 398
ns894788		0.1	0.2	0.2	0.2	0.8	0.2	1991	1797	1856	1725	10 998	3380
ns903616		20.9	18.7	6.0	20.2	23.4	5.6	69 905	39 098	22 393	34 677	33 727	19 683
ns930473		2.7	0.9	1.4	1.7	1.6	1.7	17 524	1082	3918	3931	2103	4209
nsa		0.0	0.0	0.0	0.0	0.0	0.0	139	88	113	128	116	115
nsct1		1.2	1.2	1.3	2.0	1.4	1.1	1036	1093	1073	5116	2053	1291
nsct2		1.9	2.0	1.9	2.1	2.3	1.8	0	0	0	0	0	0
nsic1		0.0	0.0	0.0	0.0	0.0	0.0	122	121	128	99	124	173
nsic2		0.0	0.0	0.0	0.0	0.0	0.0	211	246	227	234	328	256
nsir1		0.2	0.2	0.3	0.3	0.3	0.2	806	733	865	1582	782	634
nsir2		0.3	0.3	0.3	0.4	0.6	0.3	2542	2488	2578	2917	5089	4003
nsr8k		104.7	170.3	136.8	1962.1	241.1	188.4	84 391	110 400	110 623	693 887	102 901	148 760
nsrand-ipx		0.2	0.1	0.2	0.2	0.2	0.1	222	161	224	165	190	214
nu120-pr3		0.1	0.2	0.2	0.2	0.7	0.2	2013	1557	2582	1657	5192	2651
nu25-pr12		0.1	0.1	0.1	0.1	0.1	0.1	1404	1290	929	1212	1710	1326
nu60-pr9		0.1	0.1	0.1	0.2	0.4	0.1	1792	1303	2448	1465	3971	2290
nug05		0.0	0.0	0.0	0.0	0.0	0.0	118	150	178	196	241	179
nug06		0.0	0.0	0.0	0.0	0.1	0.0	595	511	567	632	1491	805
nug07		0.1	0.1	0.1	0.2	0.4	0.2	2041	1251	1579	2033	9095	2563
nug08		0.6	0.4	0.1	0.9	1.2	0.9	7738	4652	1147	5931	12 873	8116
nug08-3rd		293.3	191.2	22.0	586.0	1031.9	296.8	39 980	34 929	16 810	58 391	44 066	77 975
nug12		55.6	146.5	1.3	93.2	188.8	100.1	121 812	206 578	9717	105 882	164 885	178 046
nug15		884.4	3600.0*	16.2	2082.3	3600.0*	1837.6	689 651	1 882 531*	48 343	971 443	863 880*	1 176 743

Continued on next page

Table B.1.: LP solver comparison (`allLP` test set, SCIP 6.0.2)

	time						iters					
LP solver	CPLEX	Clp	Gurobi	MOSEK	SoPlex	Xpress	CPLEX	Clp	Gurobi	MOSEK	SoPlex	Xpress
instance	12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09	12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09
nug20	3600.0*	3600.0*	694.8	3598.8*	3600.0*	3599.9*	469145*	566094*	308925	555000*	375884*	1118795*
nug30	3586.5*	3600.0*	3596.3*	3600.0*	3600.0*	3590.3*	143049*	111803*	660981*	325000*	91660*	367011*
nursesched-medium-hint03	4.6	7.7	4.4	8.0	12.5	5.2	8891	8279	7370	7471	14947	8401
nursesched-sprint02	0.6	0.8	0.6	0.8	1.1	0.6	1690	2239	1779	1424	6511	2408
nw04	5.4	5.4	5.7	5.5	5.7	5.5	201	147	3319	270	487	351
nw14	11.1	11.0	11.2	11.2	12.6	11.2	315	311	1678	226	859	406
ofi	110.6	158.4	3600.0*	72.5	58.4	13.9	70765	107600	0*	63980	98210	78255
opm2-z10-s2	22.0	41.2	16.8	136.9	90.4	32.8	22808	19484	43578	20869	41816	23454
opm2-z10-s4	18.4	37.3	15.9	118.6	70.3	32.7	21721	18633	41233	19693	38452	25650
opm2-z11-s8	32.6	78.3	32.5	275.9	135.4	81.1	35402	23508	63359	29644	49885	32936
opm2-z12-s14	72.0	151.8	61.4	712.2	282.9	427.7	57631	32205	85004	45097	68591	116585
opm2-z12-s7	74.0	158.6	62.4	718.9	395.9	128.0	58121	32244	85856	45815	79850	44848
opm2-z7-s2	1.3	2.0	1.4	4.0	7.1	2.1	5978	5012	12141	4020	14414	5701
opt1217	0.0	0.0	0.0	0.0	0.0	0.0	190	618	145	231	578	160
orna1	0.0	0.0	3600.0*	3600.0*	0.1	0.0	938	922	0*	0*	1801	983
orna2	0.0	0.0	3600.0*	3600.0*	0.1	0.0	945	930	0*	0*	1624	1000
orna3	0.0	0.0	3600.0*	3600.0*	0.1	0.0	845	921	0*	0*	1548	966
orna4	0.0	0.1	3600.0*	3600.0*	0.1	0.1	602	947	0*	0*	2129	1032
orna7	0.0	0.0	3600.0*	3600.0*	0.1	0.0	546	950	0*	0*	1644	946
orswq2	0.0	0.0	0.0	0.0	0.0	0.0	77	89	59	86	117	97
osa-07	0.5	0.5	0.5	0.5	0.6	0.5	567	589	841	530	1223	626
osa-14	1.7	1.8	1.8	1.8	2.2	1.7	1109	1207	1512	1081	2736	969
osa-30	5.1	5.4	5.2	5.3	6.6	5.0	2270	2807	2783	2140	4182	2250
osa-60	26.9	28.8	27.1	27.7	34.9	26.8	4481	5400	5162	4235	8237	4373
p0033	0.0	0.0	0.0	0.0	0.0	0.0	10	17	13	15	14	14
p0040	0.0	0.0	0.0	0.0	0.0	0.0	16	16	4	17	16	17
p010	0.8	0.9	0.5	0.6	2.6	0.7	16185	14615	13082	12806	13842	14317
p0201	0.0	0.0	0.0	0.0	0.0	0.0	56	54	49	71	65	85
p0282	0.0	0.0	0.0	0.0	0.0	0.0	35	26	27	30	38	34
p0291	0.0	0.0	0.0	0.0	0.0	0.0	7	7	7	7	7	7
p05	0.3	0.3	0.2	0.2	0.6	0.3	7580	7462	7094	6413	7469	7271
p0548	0.0	0.0	0.0	0.0	0.0	0.0	15	22	14	16	12	12
p100x588b	0.0	0.0	0.0	0.0	0.0	0.0	104	104	99	95	103	99
p19	0.0	0.0	0.0	0.0	0.0	0.0	184	238	226	211	308	219

Continued on next page

Table B.1.: LP solver comparison (`allLP` test set, SCIP 6.0.2)

	time						iters					
LP solver	CPLEX	Clp	Gurobi	MOSEK	SoPlex	Xpress	CPLEX	Clp	Gurobi	MOSEK	SoPlex	Xpress
instance	12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09	12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09
p200x1188c	0.0	0.0	0.0	0.0	0.0	0.0	46	513	68	47	333	659
p2756	0.0	0.0	0.0	0.0	0.0	0.0	1	1	1	1	1	1
p2m2p1m1p0n100	0.0	0.0	0.0	0.0	0.0	0.0	0	0	0	1	0	0
p6000	0.1	0.1	0.2	0.1	0.1	0.1	129	145	1618	145	189	124
p6b	0.1	0.0	0.1	0.1	0.0	0.0	867	468	701	519	462	481
p80x400b	0.0	0.0	0.0	0.0	0.0	0.0	0	0	0	0	0	0
pcb1000	0.2	0.2	0.2	0.3	0.2	0.3	1596	1932	1845	1977	2422	2451
pcb3000	0.7	0.7	0.7	0.9	1.0	0.9	4421	5638	5232	5626	8098	7114
pds-02	0.1	0.1	0.1	0.1	0.1	0.1	370	889	619	436	1103	833
pds-06	0.4	0.6	0.5	0.5	0.8	0.5	2229	5553	2849	2213	9157	5406
pds-10	0.9	1.6	1.0	1.0	3.0	1.1	4465	11233	4992	3945	14650	11115
pds-100	35.9	1070.7	51.2	74.0	776.6	74.8	121261	372872	136093	170471	318167	224627
pds-20	2.4	17.8	2.6	2.6	16.1	4.5	13237	40883	14353	9369	35613	30670
pds-30	5.0	63.0	5.7	5.0	41.4	9.2	26491	82720	29664	16892	63079	49872
pds-40	8.4	149.1	9.3	7.6	136.6	17.1	43280	122906	42396	26156	119565	76802
pds-50	11.1	312.6	12.9	16.9	204.7	24.4	52552	192384	55195	56075	147516	98405
pds-60	14.1	493.2	25.3	16.9	365.3	33.8	60663	252632	73099	53356	209065	125244
pds-70	17.7	601.8	62.6	25.6	472.2	42.9	71248	294633	119432	73264	238870	154441
pds-80	24.3	869.1	46.2	30.4	440.9	48.9	86872	345671	111924	81393	257884	177540
pds-90	26.9	1496.5	55.8	41.4	769.1	63.2	100799	480472	137197	104361	330349	212807
peg-solitaire-a3	1.2	0.7	1.2	1.8	2.8	2.1	9152	3823	7076	4551	12400	12675
perold	0.1	0.2	0.1	0.1	0.3	0.1	1432	2685	1289	1348	5720	1478
pf2177	0.2	0.2	0.1	2.3	0.3	0.3	1319	801	1007	5986	1871	1385
pg	0.0	0.0	0.0	0.0	0.0	0.0	369	304	202	236	419	203
pg5_34	0.0	0.0	0.0	0.0	0.0	0.0	328	250	94	251	341	251
pgp2	0.1	0.1	0.1	0.1	0.2	0.1	3974	3917	4089	3897	3763	3903
physiciansched3-3	36.6	2242.3	75.4	247.4	178.7	2015.9	61241	911311	70913	94782	86262	932662
physiciansched6-2	2.8	3.9	2.8	3.6	2.9	2.7	2035	5974	2933	4804	4724	3042
pigeon-10	0.0	0.0	0.0	0.0	0.0	0.0	401	341	323	354	379	503
pigeon-11	0.0	0.0	0.0	0.0	0.0	0.0	379	453	379	446	459	531
pigeon-12	0.0	0.0	0.0	0.0	0.0	0.0	429	481	572	550	511	524
pigeon-13	0.0	0.0	0.0	0.0	0.0	0.0	490	686	687	608	550	562
pigeon-19	0.1	0.1	0.1	0.1	0.1	0.0	1016	1297	1396	1412	1174	1009
pilot	0.5	1.6	0.6	0.9	2.3	0.9	2507	6606	3097	3513	12039	4483

Continued on next page

Table B.1.: LP solver comparison (`allLP` test set, SCIP 6.0.2)

	time						iters					
LP solver	CPLEX	Clp	Gurobi	MOSEK	SoPlex	Xpress	CPLEX	Clp	Gurobi	MOSEK	SoPlex	Xpress
instance	12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09	12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09
pilot-ja	0.1	0.5	0.1	0.2	0.3	0.2	1633	6112	1565	1677	4213	1853
pilot-we	0.2	0.2	0.1	0.2	0.5	0.1	2441	3192	2001	2216	7501	2404
pilot4	0.0	0.0	0.1	0.1	0.1	0.1	862	745	619	545	1384	742
pilot87	2.6	5.7	4.9	3600.0*	6.0	3.6	8215	12108	39549	0*	14252	9584
pilotnov	0.1	0.1	0.1	0.1	0.1	0.1	1454	982	1056	897	1543	1267
piperout-08	0.6	0.5	0.6	0.9	0.4	0.4	0	0	0	0	0	0
piperout-27	0.9	0.7	0.4	1.3	0.5	0.3	10343	6233	6515	6479	9519	9151
pk1	0.0	0.0	0.0	0.0	0.0	0.0	52	47	45	55	98	26
pldd000b	0.1	0.1	0.1	0.1	0.1	0.1	558	1499	1297	1797	1914	1583
pldd001b	0.1	0.1	0.1	0.1	0.1	0.1	564	1619	1328	1961	2219	1625
pldd002b	0.1	0.1	0.1	0.1	0.1	0.1	566	1632	1270	1928	1887	1578
pldd003b	0.1	0.1	0.1	0.1	0.1	0.1	569	1584	1164	1887	1933	1576
pldd004b	0.1	0.1	0.1	0.1	0.1	0.1	560	1590	1270	1930	1957	1609
pldd005b	0.1	0.1	0.1	0.1	0.1	0.1	567	1581	1300	1875	1978	1566
pldd006b	0.1	0.1	0.1	0.1	0.1	0.1	567	1604	1267	1940	1922	1605
pldd007b	0.1	0.1	0.1	0.1	0.1	0.1	555	1586	1323	1964	2008	1568
pldd008b	0.1	0.1	0.1	0.2	0.1	0.1	606	1655	1314	2079	1817	1639
pldd009b	0.1	0.1	0.1	0.1	0.1	0.1	610	1617	1499	1994	1927	1650
pldd010b	0.1	0.1	0.1	0.1	0.1	0.1	623	1587	1474	1957	2109	1570
pldd011b	0.1	0.1	0.1	0.2	0.1	0.1	949	1602	1255	2036	1628	1546
pldd012b	0.1	0.1	0.1	0.2	0.1	0.1	665	1590	1353	2116	1816	1561
pltexpa2-16	0.0	0.0	0.1	0.0	0.0	0.1	1115	1649	701	947	1515	1761
pltexpa2-6	0.0	0.0	0.0	0.0	0.0	0.0	460	648	355	385	567	625
pltexpa3_16	0.8	1.5	1.0	1.9	1.6	0.9	20364	30376	19310	30430	26068	24117
pltexpa3_6	0.1	0.1	0.1	0.1	0.1	0.1	3096	5332	3247	4362	4794	4157
pltexpa4_6	0.9	1.6	1.0	3600.0*	1.9	0.9	20606	31814	17948	0*	26197	24908
ppo8a	0.0	0.0	0.0	0.0	0.0	0.0	81	79	80	79	80	76
ppo8aCUTS	0.0	0.0	0.0	0.0	0.0	0.0	169	203	153	147	210	237
primagaz	0.1	0.1	0.1	0.1	0.2	0.1	444	1646	68	1625	2188	1598
problem	0.0	0.0	0.0	0.0	0.0	0.0	1	9	2	4	7	9
probportfolio	0.0	0.0	0.0	0.0	0.0	0.0	125	175	125	127	126	173
prod1	0.0	0.0	0.0	0.0	0.0	0.0	0	16	0	0	7	7
prod2	0.0	0.0	0.0	0.0	0.0	0.0	0	36	0	0	10	10
progas	0.2*	0.1	0.1	0.1	0.3	0.2	1153*	1667	676	804	3738	1539

Continued on next page

Table B.1.: LP solver comparison (`allLP` test set, SCIP 6.0.2)

	time						iters					
LP solver	CPLEX	Clp	Gurobi	MOSEK	SoPlex	Xpress	CPLEX	Clp	Gurobi	MOSEK	SoPlex	Xpress
instance	12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09	12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09
proteindesign121hz512p9	3.0	3.0	3.1	3.0	3.8	2.9	1319	1747	998	293	2848	1241
proteindesign122trx11p8	2.5	2.4	2.6	2.5	2.8	2.2	1169	1328	609	212	2011	500
protfold	0.4	0.2	0.2	0.4	1.4	0.2	3419	1522	1303	2048	11388	1763
pw-myciel4	0.1	0.2	0.1	0.3	0.1	0.2	1722	1415	1597	1392	1386	2119
qap10	6.4	16.1	4.4	9.8	23.2	9.8	34104	51638	17550	26000	67470	40516
qiu	0.0	0.0	0.0	0.0	0.0	0.0	1146	821	970	559	1189	1245
qiulp	0.0	0.0	0.0	0.0	0.0	0.0	1146	821	970	559	1189	1245
qnet1	0.0	0.0	0.0	0.1	0.0	0.0	1007	449	750	974	675	1107
qnet1_o	0.0	0.0	0.0	0.0	0.0	0.0	558	183	463	350	1069	501
queens-30	2.4	2.7	1.5	6.5	7.6	6.2	2376	2656	1076	3712	12386	6303
r05	0.3	0.3	0.3	0.3	0.7	0.3	7677	7381	6930	6419	7420	7251
r80x800	0.0	0.0	0.0	0.0	0.0	0.0	89	176	90	98	190	203
radiationm18-12-05	0.4	0.4	0.5	0.4	0.4	0.4	3394	4163	4011	767	5329	3238
radiationm40-10-02	2.0	2.2	2.2	2.0	2.0	1.9	8793	14511	11746	2364	4747	7646
rail01	53.9	105.5	69.5	120.4	161.5	41.2	98398	125800	73144	102230	138797	136319
rail02	533.2	1941.7	647.6	1528.3	3600.0*	587.4	429163	837732	411643	636792	797615*	1101222
rail03	1062.7	3600.0*	987.0	2863.7	3600.0*	707.9	527934	0*	383128	571381	626754*	656403
rail2586	438.3	548.4	266.3	502.4	621.1	508.1	38028	55109	142362	36499	53945	70182
rail4284	874.4	1157.5	306.7	904.6	1380.3	833.4	73481	103108	276371	63279	96048	107690
rail507	2.4	3.0	2.6	2.8	5.3	3.0	2976	3734	3318	2975	11959	5189
rail516	1.9	3.1	1.8	1.9	8.4	2.3	1817	3343	4311	1420	12859	3506
rail582	2.5	2.9	2.1	2.7	6.2	3.1	4971	3678	7518	3509	13446	6746
ramos3	27.2	22.2	3.6	57.6	56.4	20.2	17208	14018	2860	14867	21079	18212
ran10x10a	0.0	0.0	0.0	0.0	0.0	0.0	24	24	14	24	23	26
ran10x10b	0.0	0.0	0.0	0.0	0.0	0.0	27	19	10	26	23	19
ran10x10c	0.0	0.0	0.0	0.0	0.0	0.0	32	32	15	27	26	28
ran10x12	0.0	0.0	0.0	0.0	0.0	0.0	28	26	10	29	26	26
ran10x26	0.0	0.0	0.0	0.0	0.0	0.0	34	54	19	59	51	46
ran12x12	0.0	0.0	0.0	0.0	0.0	0.0	34	29	20	31	23	28
ran12x21	0.0	0.0	0.0	0.0	0.0	0.0	39	44	17	48	52	41
ran13x13	0.0	0.0	0.0	0.0	0.0	0.0	24	37	19	35	33	35
ran14x18	0.0	0.0	0.0	0.0	0.0	0.0	36	50	23	54	54	54
ran14x18-disj-8	0.1	0.1	0.1	0.1	0.1	0.1	638	994	646	606	1136	893
ran14x18_1	0.0	0.0	0.0	0.0	0.0	0.0	34	41	23	53	58	50

Continued on next page

Table B.1.: LP solver comparison (`allLP` test set, SCIP 6.0.2)

	time						iters					
LP solver	CPLEX	Clp	Gurobi	MOSEK	SoPlex	Xpress	CPLEX	Clp	Gurobi	MOSEK	SoPlex	Xpress
instance	12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09	12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09
ran16x16	0.0	0.0	0.0	0.0	0.0	0.0	41	39	36	41	37	41
ran17x17	0.0	0.0	0.0	0.0	0.0	0.0	40	40	20	46	45	42
ran4x64	0.0	0.0	0.0	0.0	0.0	0.0	10	81	3	93	68	62
ran6x43	0.0	0.0	0.0	0.0	0.0	0.0	36	67	15	66	85	61
ran8x32	0.0	0.0	0.0	0.0	0.0	0.0	28	46	13	68	54	52
rat1	0.4	17.6	0.4	0.4	0.6	21.8	1872	10 021	1799	1612	2870	18 114
rat5	0.4	0.4	0.4	0.5	0.5	0.4	1998	1882	1917	1959	2115	2034
rat7a	4.5	40.0	4.2	3.6	11.6	54.2	3459	7556	2755	2545	11 129	16 481
rd-rplusc-21	2.9	2.9	3.0	3.1	2.8	2.9	174	233	498	160	219	168
reblock115	0.2	0.1	0.1	0.5	0.3	0.3	2391	1431	1595	1860	3531	2514
reblock166	0.9	0.5	0.4	1.1	0.8	0.9	5247	1931	4677	2131	4499	4564
reblock354	1.9	1.7	2.5	7.0	5.7	5.0	9254	5550	16 942	7470	14 823	17 735
reblock420	6.8	3.0	1.7	10.9	10.6	3.6	11 787	5047	9913	5790	12 950	16 127
reblock67	0.1	0.1	0.1	0.1	0.1	0.1	1362	754	832	930	1861	1257
recipe	0.0	0.0	0.0	0.0	0.0	0.0	0	0	0	0	0	0
refine	0.0	0.0	0.0	0.0	0.0	0.0	13	13	10	13	13	13
rentacar	0.2	0.2	0.2	0.2	0.3	0.3	507	1575	850	1321	3126	2105
rgn	0.0	0.0	0.0	0.0	0.0	0.0	65	97	64	55	75	73
rlfddd	0.6	0.7	0.7	0.7	0.6	0.6	1016	611	791	554	612	749
rlfdual	1.5	1.2	1.5	3.0	4.1	2.8	5392	5080	6350	4707	11 162	9102
rlfprim	0.7	0.6	0.8	1.3	1.1	0.6	2897	2568	5593	3290	6484	2591
rlp1	0.0	0.0	0.0	0.0	0.0	0.0	164	135	105	120	196	76
rmatr100-p10	0.4	0.5	0.4	1.0	1.7	0.5	1712	1389	1838	1950	8687	1848
rmatr100-p5	0.7	0.9	0.7	2.0	3.5	1.0	3941	2682	3893	3723	11 203	3322
rmatr200-p10	7.1	9.3	5.8	21.9	19.1	9.0	10 226	6780	10 247	10 894	14 799	9917
rmatr200-p20	4.0	4.8	3.6	8.3	10.9	4.6	5708	3276	5055	4363	11 557	4655
rmatr200-p5	12.2	17.1	9.5	63.9	26.1	14.3	19 914	13 674	17 424	28 469	20 002	17 209
rmine10	7.7	9.6	11.5	22.9	15.6	17.4	13 919	13 506	43 643	13 407	17 267	35 582
rmine14	358.2	466.5	481.8	1007.7	591.3	188.8	79 225	94 240	361 336	64 768	106 187	277 051
rmine21	3600.0*	3600.0*	3594.3*	3600.0*	3600.0*	3597.2*	279 318*	232 770*	478 445*	152 000*	310 795*	838 726*
rmine25	3600.0*	3600.0*	3589.1*	3600.0*	3600.0*	3577.4*	469 212*	189 507*	336 230*	235 000*	272 800*	587 125*
rmine6	0.1	0.1	0.1	0.3	0.1	0.1	1372	1143	2222	1335	1604	1713
rocI-4-11	0.2	0.1	0.1	0.2	0.2	0.1	1461	2151	1605	1469	2113	2432
rocII-4-11	0.9	0.9	1.0	0.9	0.9	0.9	560	578	520	88	496	709

Continued on next page

Table B.1.: LP solver comparison (`allLP` test set, SCIP 6.0.2)

	time						iters					
LP solver	CPLEX	Clp	Gurobi	MOSEK	SoPlex	Xpress	CPLEX	Clp	Gurobi	MOSEK	SoPlex	Xpress
instance	12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09	12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09
rocII-5-11	1.0	0.9	1.1	1.0	1.0	0.9	679	606	194	140	579	833
rocII-7-11	1.5	1.4	1.6	1.5	1.5	1.4	922	672	279	194	822	1066
rocII-9-11	2.0	1.9	2.1	2.1	2.0	1.9	1166	809	459	235	995	1480
rococoB10-011000	0.1	0.1	0.2	0.1	0.5	0.2	3798	1679	3808	1295	6661	4810
rococoC10-001000	0.0	0.0	0.1	0.1	0.1	0.1	1384	441	1346	1146	2582	1553
rococoC11-011100	0.2	0.1	0.2	0.2	0.9	0.2	2688	1301	2477	1337	8958	3085
rococoC12-111000	0.3	0.1	0.5	0.8	1.0	0.4	5462	962	7787	3407	5743	4498
roi2alpha3n4	1.2	1.4	1.1	1.1	1.3	0.8	796	3728	869	543	3525	605
roi5alpha10n8	3.6	7.5	4.3	4.2	7.8	3.7	2642	5645	2969	2308	10 400	4323
roll3000	0.1	0.1	0.1	0.1	0.1	0.1	1134	807	1002	918	1610	1092
rosen1	0.0	0.0	0.0	0.1	0.0	0.0	649	346	367	358	482	353
rosen10	0.1	0.1	0.1	0.2	0.2	0.1	2062	1488	1481	1386	2492	1386
rosen2	0.1	0.1	0.1	0.1	0.1	0.1	1123	721	768	695	933	764
rosen7	0.0	0.0	0.0	0.0	0.0	0.0	261	177	177	175	227	190
rosen8	0.0	0.0	0.0	0.0	0.0	0.0	560	397	352	384	863	382
rout	0.0	0.0	0.0	0.0	0.0	0.0	403	293	267	244	386	361
route	1.4	1.4	1.4	1.4	1.5	1.3	2377	917	1357	2111	1351	1037
roy	0.0	0.0	0.0	0.0	0.0	0.0	81	97	85	79	98	96
rvb-sub	3.2	3.9	4.5	4.0	12.0	3.9	1657	1823	22 383	1852	10 667	2737
s100	531.9	1446.8	1134.2	3600.0*	2040.4	205.2	218 806	319 872	531 286	0*	293 478	129 272
s250r10	29.8	453.8	146.9	3600.0*	326.9	51.8	69 023	181 977	150 369	0*	143 753	145 235
satellites1-25	0.3	0.3	0.4	0.7	5.0	0.7	1139	1856	2272	3562	16 265	4007
satellites2-40	8.7	12.3	1.6	20.0	166.2	11.4	15 856	22 084	5225	25 334	108 367	21 160
satellites2-60	10.6	11.7	1.7	20.4	157.3	12.7	16 523	20 544	5654	25 535	104 131	23 490
satellites2-60-fs	3.8	7.7	2.5	8.1	121.7	8.5	9286	18 826	13 516	17 359	97 178	16 828
satellites3-40	287.9	87.6	5.6	447.2	1600.2	40.8	193 498	56 275	12 065	100 548	352 689	38 457
satellites3-40-fs	214.0	129.8	8.7	165.0	1680.8	54.6	148 599	77 359	25 712	78 975	467 690	44 998
savsched1	420.0	3600.0*	722.5	1046.4	3600.0*	426.1	141 664	480 073*	184 271	286 675	233 768*	184 060
sc105	0.0	0.0	0.0	0.0	0.0	0.0	31	38	37	31	47	49
sc205	0.0	0.0	0.0	0.0	0.0	0.0	90	100	73	71	116	151
sc205-2r-100	0.0	0.0	0.0	0.0	0.0	0.0	58	35	34	47	28	45
sc205-2r-16	0.0	0.0	0.0	0.0	0.0	0.0	5	7	6	14	6	11
sc205-2r-1600	0.8	0.8	0.8	0.8	0.8	0.8	693	190	119	405	455	326
sc205-2r-200	0.0	0.0	0.0	0.0	0.0	0.1	89	40	39	42	73	32

Continued on next page

Table B.1.: LP solver comparison (`allLP` test set, SCIP 6.0.2)

	time						iters					
LP solver	CPLEX	Clp	Gurobi	MOSEK	SoPlex	Xpress	CPLEX	Clp	Gurobi	MOSEK	SoPlex	Xpress
instance	12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09	12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09
sc205-2r-27	0.0	0.0	0.0	0.0	0.0	0.0	25	10	19	18	29	23
sc205-2r-32	0.0	0.0	0.0	0.0	0.0	0.0	6	7	5	26	5	15
sc205-2r-4	0.0	0.0	0.0	0.0	0.0	0.0	9	8	9	10	6	8
sc205-2r-400	0.1	0.1	0.1	0.1	0.1	0.1	165	117	61	63	93	61
sc205-2r-50	0.0	0.0	0.0	0.0	0.0	0.0	62	61	60	62	63	60
sc205-2r-64	0.0	0.0	0.0	0.0	0.0	0.0	6	11	12	7	6	27
sc205-2r-8	0.0	0.0	0.0	0.0	0.0	0.0	13	12	13	6	6	12
sc205-2r-800	0.3	0.2	0.2	0.3	0.3	0.3	321	192	262	166	246	89
sc50a	0.0	0.0	0.0	0.0	0.0	0.0	13	22	12	8	19	18
sc50b	0.0	0.0	0.0	0.0	0.0	0.0	9	16	13	13	17	19
scagr25	0.0	0.0	0.0	0.0	0.0	0.0	211	213	187	246	339	259
scagr7	0.0	0.0	0.0	0.0	0.0	0.0	47	45	44	51	50	48
scagr7-2b-16	0.0	0.0	0.0	0.0	0.0	0.0	209	128	176	202	204	211
scagr7-2b-4	0.0	0.0	0.0	0.0	0.0	0.0	57	34	48	60	59	56
scagr7-2b-64	0.1	0.1	0.2	0.2	0.2	0.1	3298	1878	2628	2980	3190	2613
scagr7-2c-16	0.0	0.0	0.0	0.0	0.0	0.0	232	131	182	203	199	212
scagr7-2c-4	0.0	0.0	0.0	0.0	0.0	0.0	59	35	47	57	56	60
scagr7-2c-64	0.0	0.0	0.1	0.1	0.0	0.0	919	531	671	836	657	714
scagr7-2r-108	0.1	0.1	0.1	0.1	0.1	0.1	1127	915	975	1265	1322	1149
scagr7-2r-16	0.0	0.0	0.0	0.0	0.0	0.0	213	125	180	218	208	211
scagr7-2r-216	0.1	0.1	0.1	0.2	0.2	0.1	2404	1822	2458	2495	2607	2276
scagr7-2r-27	0.0	0.0	0.0	0.0	0.0	0.0	335	216	294	345	340	288
scagr7-2r-32	0.0	0.0	0.0	0.0	0.0	0.0	419	242	464	397	408	417
scagr7-2r-4	0.0	0.0	0.0	0.0	0.0	0.0	56	38	49	48	57	54
scagr7-2r-432	0.2	0.3	0.3	0.4	0.4	0.2	5212	3636	4910	5022	4813	4546
scagr7-2r-54	0.0	0.0	0.0	0.1	0.0	0.0	629	459	576	660	648	598
scagr7-2r-64	0.0	0.0	0.0	0.0	0.0	0.0	840	478	745	805	678	713
scagr7-2r-8	0.0	0.0	0.0	0.0	0.0	0.0	105	69	91	97	106	102
scagr7-2r-864	0.7	0.9	0.7	1.3	0.9	0.6	10043	7462	9867	9901	9880	8048
scfxm1	0.0	0.0	0.0	0.0	0.0	0.0	244	283	269	256	636	327
scfxm1-2b-16	0.1	0.1	0.1	0.1	0.1	0.1	1440	2016	1606	1243	2851	2030
scfxm1-2b-4	0.0	0.0	0.0	0.0	0.0	0.0	350	531	430	337	780	518
scfxm1-2b-64	1.2	1.8	1.4	1.9	2.8	1.4	11162	15691	12720	10252	17490	16551
scfxm1-2c-4	0.0	0.0	0.0	0.0	0.0	0.0	360	508	441	345	832	566

Continued on next page

Table B.1.: LP solver comparison (`allLP` test set, SCIP 6.0.2)

	time						iters					
LP solver	CPLEX	Clp	Gurobi	MOSEK	SoPlex	Xpress	CPLEX	Clp	Gurobi	MOSEK	SoPlex	Xpress
instance	12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09	12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09
scfxm1-2r-128	1.2	1.4	1.4	1.9	4.1	1.1	10939	14593	13240	10063	18689	15733
scfxm1-2r-16	0.1	0.1	0.1	0.1	0.1	0.1	1296	2034	1791	1251	2694	2020
scfxm1-2r-256	3.5	5.8	3.0	5.8	19.6	3.4	20618	34430	29738	19698	39555	34026
scfxm1-2r-27	0.1	0.2	0.2	0.2	0.2	0.1	2106	2937	2741	2131	5017	3352
scfxm1-2r-32	0.2	0.2	0.2	0.2	0.2	0.2	2551	3386	3193	2516	4688	3853
scfxm1-2r-4	0.0	0.0	0.0	0.0	0.0	0.0	349	512	457	335	741	577
scfxm1-2r-64	0.5	0.5	0.5	0.6	0.9	0.5	5377	6867	6551	5062	11243	8033
scfxm1-2r-8	0.0	0.1	0.1	0.1	0.1	0.1	673	1037	873	637	1555	1035
scfxm1-2r-96	0.9	1.0	0.9	1.1	1.8	1.0	8087	11115	9776	7454	13609	12828
scfxm2	0.0	0.0	0.0	0.0	0.0	0.0	608	616	514	563	1104	629
scfxm3	0.1	0.1	0.1	0.1	0.1	0.1	803	944	835	871	1660	1005
scorpion	0.0	0.0	0.0	0.0	0.0	0.0	53	55	43	46	67	56
scrs8	0.0	0.0	0.0	0.0	0.0	0.0	336	464	275	300	802	438
scrs8-2b-16	0.0	0.0	0.0	0.0	0.0	0.0	16	16	0	16	16	16
scrs8-2b-4	0.0	0.0	0.0	0.0	0.0	0.0	4	4	0	4	4	4
scrs8-2b-64	0.0	0.0	0.0	0.0	0.0	0.0	124	125	113	151	136	135
scrs8-2c-16	0.0	0.0	0.0	0.0	0.0	0.0	22	22	22	22	22	22
scrs8-2c-32	0.0	0.0	0.0	0.0	0.0	0.0	38	38	38	38	38	38
scrs8-2c-4	0.0	0.0	0.0	0.0	0.0	0.0	4	4	0	4	4	4
scrs8-2c-64	0.0	0.0	0.0	0.0	0.0	0.0	76	76	76	76	76	76
scrs8-2c-8	0.0	0.0	0.0	0.0	0.0	0.0	10	10	10	10	10	10
scrs8-2r-128	0.0	0.0	0.0	0.0	0.0	0.0	254	263	204	305	271	271
scrs8-2r-16	0.0	0.0	0.0	0.0	0.0	0.0	32	32	32	32	32	32
scrs8-2r-256	0.1	0.1	0.1	0.1	0.1	0.1	518	527	799	634	543	567
scrs8-2r-27	0.0	0.0	0.0	0.0	0.0	0.0	50	48	38	58	51	51
scrs8-2r-32	0.0	0.0	0.0	0.0	0.0	0.0	64	64	64	64	64	64
scrs8-2r-4	0.0	0.0	0.0	0.0	0.0	0.0	8	8	8	8	8	8
scrs8-2r-512	0.2	0.2	0.2	0.1	0.2	0.2	1148	1173	1689	1420	1188	1264
scrs8-2r-64	0.0	0.0	0.0	0.0	0.0	0.0	128	128	128	128	128	128
scrs8-2r-64b	0.0	0.0	0.0	0.0	0.0	0.0	127	132	104	153	135	136
scrs8-2r-8	0.0	0.0	0.0	0.0	0.0	0.0	19	18	14	21	21	20
scsd1	0.0	0.0	0.0	0.0	0.0	0.0	93	101	89	127	92	116
scsd6	0.0	0.0	0.0	0.0	0.0	0.0	254	247	276	326	532	375
scsd8	0.0	0.1	0.1	0.1	0.1	0.1	906	835	997	1247	2069	1341

Continued on next page

Table B.1.: LP solver comparison (`allLP` test set, SCIP 6.0.2)

	time						iters					
LP solver	CPLEX	Clp	Gurobi	MOSEK	SoPlex	Xpress	CPLEX	Clp	Gurobi	MOSEK	SoPlex	Xpress
instance	12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09	12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09
scsd8-2b-16	0.0	0.0	0.0	0.0	0.0	0.0	191	170	231	205	248	144
scsd8-2b-4	0.0	0.0	0.0	0.0	0.0	0.0	35	42	35	56	46	41
scsd8-2b-64	0.5	0.5	0.5	0.6	0.5	0.5	1996	2393	2744	2901	2136	1816
scsd8-2c-16	0.0	0.0	0.0	0.0	0.0	0.0	125	150	160	216	191	147
scsd8-2c-4	0.0	0.0	0.0	0.0	0.0	0.0	35	42	35	46	46	41
scsd8-2c-64	0.5	0.5	0.5	0.6	0.5	0.5	1993	2309	2168	2748	2056	1974
scsd8-2r-108	0.1	0.2	0.2	0.2	0.2	0.2	980	1180	986	985	1114	990
scsd8-2r-16	0.0	0.0	0.0	0.0	0.0	0.0	119	150	111	232	181	142
scsd8-2r-216	0.4	0.4	0.4	0.5	0.4	0.3	2016	2412	2064	2015	2273	2229
scsd8-2r-27	0.0	0.0	0.1	0.0	0.1	0.0	227	255	229	224	258	268
scsd8-2r-32	0.1	0.1	0.0	0.1	0.1	0.1	221	295	208	354	488	265
scsd8-2r-4	0.0	0.0	0.0	0.0	0.0	0.0	35	42	35	44	46	43
scsd8-2r-432	0.9	1.0	0.9	1.1	0.9	0.9	4035	4699	4052	3991	4589	4236
scsd8-2r-54	0.1	0.1	0.1	0.1	0.1	0.1	478	572	484	481	549	481
scsd8-2r-64	0.1	0.1	0.1	0.1	0.1	0.1	435	594	405	726	1037	396
scsd8-2r-8	0.0	0.0	0.0	0.0	0.0	0.0	56	78	57	97	96	77
scsd8-2r-8b	0.0	0.0	0.0	0.0	0.0	0.0	56	78	57	97	96	77
sct1	1.7	1.8	2.5	3600.0*	5.6	1.7	10358	6561	16540	0*	17976	7507
sct2	0.1	0.1	0.1	0.1	0.1	0.1	793	1539	1642	1176	1718	1118
sct32	1.4	3.2	1.1	1.9	2.3	1.1	12260	15645	11973	7285	13774	14667
sct5	0.6	3.6	0.7	3600.0*	2.1	0.7	3303	10617	7788	0*	10308	4887
sctap1	0.0	0.0	0.0	0.0	0.0	0.0	244	197	262	160	252	177
sctap1-2b-16	0.0	0.0	0.0	0.0	0.0	0.0	165	230	174	173	218	169
sctap1-2b-4	0.0	0.0	0.0	0.0	0.0	0.0	45	59	42	47	53	46
sctap1-2b-64	0.4	0.5	0.5	0.5	0.6	0.5	2632	4086	2852	3028	3826	3255
sctap1-2c-16	0.0	0.0	0.0	0.0	0.0	0.0	174	250	182	186	270	179
sctap1-2c-4	0.0	0.0	0.0	0.0	0.0	0.0	48	64	49	49	52	47
sctap1-2c-64	0.1	0.1	0.1	0.1	0.1	0.1	650	862	1625	640	853	619
sctap1-2r-108	0.1	0.2	0.1	0.2	0.2	0.2	1249	1431	929	1018	1284	1073
sctap1-2r-16	0.0	0.0	0.0	0.0	0.0	0.0	170	198	460	159	171	207
sctap1-2r-216	0.3	0.4	0.3	0.4	0.4	0.4	2410	2880	1874	2051	2570	2328
sctap1-2r-27	0.0	0.0	0.1	0.1	0.0	0.1	302	351	288	268	354	285
sctap1-2r-32	0.1	0.1	0.1	0.1	0.1	0.0	338	390	855	308	342	372
sctap1-2r-4	0.0	0.0	0.0	0.0	0.0	0.0	44	54	40	40	42	45

Continued on next page

Table B.1.: LP solver comparison (`allLP` test set, SCIP 6.0.2)

	time						iters					
LP solver	CPLEX	Clp	Gurobi	MOSEK	SoPlex	Xpress	CPLEX	Clp	Gurobi	MOSEK	SoPlex	Xpress
instance	12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09	12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09
sctap1-2r-480	0.9	1.0	1.1	1.0	1.2	0.9	4960	6303	11471	4475	5569	4993
sctap1-2r-54	0.1	0.1	0.1	0.1	0.1	0.1	588	683	583	501	668	554
sctap1-2r-64	0.1	0.1	0.1	0.1	0.1	0.1	675	770	1731	625	700	681
sctap1-2r-8	0.0	0.0	0.0	0.0	0.0	0.0	79	100	85	81	84	103
sctap1-2r-8b	0.0	0.0	0.0	0.0	0.0	0.0	88	111	90	83	112	87
sctap2	0.0	0.0	0.0	0.1	0.1	0.0	443	327	368	260	580	356
sctap3	0.1	0.1	0.1	0.1	0.1	0.1	523	401	639	342	699	532
seba	0.0	0.0	0.0	0.0	0.0	0.0	1	1	1	1	1	1
self	25.4	21.4	22.4	109.9	85.7	46.7	11979	5328	6796	39795	26005	20208
set1ch	0.0	0.0	0.0	0.0	0.0	0.0	290	282	276	264	346	290
set3-10	0.1	0.1	0.1	0.1	0.0	0.1	712	1531	703	466	1645	1506
set3-15	0.1	0.1	0.1	0.1	0.1	0.1	780	1576	807	504	1706	1581
set3-20	0.1	0.1	0.1	0.1	0.1	0.1	719	1548	779	458	1660	1555
seymour	0.4	0.7	0.7	1.7	0.7	0.9	3811	2850	3481	4815	6535	6607
seymour-disj-10	1.3	4.6	1.3	2.9	1.9	2.1	8246	9692	4057	5172	9311	7615
seymour1	0.4	0.6	0.7	1.7	0.7	0.9	3811	2850	3481	4815	6535	6607
seymourl	0.4	0.7	0.7	1.7	0.7	0.9	3811	2850	3481	4815	6535	6607
sgpf5y6	8.6	8.7	8.6	3600.0*	8.9	8.4	5470	9523	30064	0*	6322	5718
share1b	0.0	0.0	0.0	0.0	0.0	0.0	206	150	97	122	226	118
share2b	0.0	0.0	0.0	0.0	0.0	0.0	104	109	84	83	133	98
shell	0.0	0.0	0.0	0.0	0.0	0.0	232	308	236	343	406	300
ship04l	0.0	0.0	0.0	0.0	0.0	0.0	194	358	235	400	504	460
ship04s	0.0	0.0	0.0	0.0	0.0	0.0	136	280	184	315	363	322
ship08l	0.0	0.0	0.1	0.1	0.1	0.0	395	638	372	642	1037	701
ship08s	0.0	0.0	0.0	0.0	0.0	0.0	287	376	266	368	518	418
ship12l	0.1	0.1	0.1	0.1	0.1	0.1	638	868	655	845	1135	990
ship12s	0.0	0.0	0.0	0.0	0.0	0.0	337	517	370	513	658	545
shipsched	0.4	0.5	0.5	0.4	0.4	0.5	0	0	0	0	0	0
shs1023	94.2	181.6	65.5	573.8	806.8	55.1	211562	176172	144691	328558	275819	165779
siena1	8.0	11.8	7.9	16.6	13.6	13.4	17225	21068	20247	24176	24330	28539
sierra	0.0	0.0	0.1	0.1	0.0	0.0	318	411	316	325	436	414
sing161	227.9	1941.0	248.4	1053.8	3600.0*	862.8	171610	359561	178170	162248	322505*	344706
sing2	2.9	7.2	1.3	4.5	23.7	4.5	26643	27753	14218	11554	30678	28944
sing245	29.3	387.0	67.5	117.4	891.9	110.5	64107	215856	86842	80522	194375	156228

Continued on next page

Table B.1.: LP solver comparison (`allLP` test set, SCIP 6.0.2)

	time						iters					
LP solver	CPLEX	Clp	Gurobi	MOSEK	SoPlex	Xpress	CPLEX	Clp	Gurobi	MOSEK	SoPlex	Xpress
instance	12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09	12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09
sing326	3.8	8.7	2.9	12.1	33.2	5.6	26442	31435	18175	21066	35275	31898
sing359	403.1	1484.7	231.6	716.8	3600.0*	762.6	151480	291915	147380	127809	296976*	294162
sing44	4.2	10.9	3.8	15.3	45.6	6.8	29596	35556	21495	24190	41796	35404
slptsk	0.5	0.6	0.4	0.7	0.7	0.7	4535	3281	2289	2210	4731	4974
small000	0.0	0.0	0.0	0.0	0.0	0.0	182	673	236	369	453	479
small001	0.0	0.0	0.0	0.0	0.0	0.0	231	679	291	381	598	524
small002	0.0	0.0	0.0	0.0	0.0	0.0	272	956	257	403	794	527
small003	0.0	0.0	0.0	0.0	0.0	0.0	266	762	275	385	609	595
small004	0.0	0.0	0.0	0.0	0.0	0.0	210	1254	273	381	521	476
small005	0.0	0.0	0.0	0.0	0.0	0.0	223	988	219	368	615	514
small006	0.0	0.0	0.0	0.0	0.0	0.0	241	767	268	318	799	481
small007	0.0	0.0	0.0	0.0	0.0	0.0	242	771	252	306	914	451
small008	0.0	0.0	0.0	0.0	0.0	0.0	222	674	234	281	884	408
small009	0.0	0.0	0.0	0.0	0.0	0.0	192	685	205	295	720	407
small010	0.0	0.0	0.0	0.0	0.0	0.0	206	549	224	280	818	394
small011	0.0	0.0	0.0	0.0	0.0	0.0	198	700	201	296	616	371
small012	0.0	0.0	0.0	0.0	0.0	0.0	181	569	176	256	444	356
small013	0.0	0.0	0.0	0.0	0.0	0.0	189	412	177	286	419	356
small014	0.0	0.0	0.0	0.0	0.0	0.0	157	457	170	292	444	387
small015	0.0	0.0	0.0	0.0	0.0	0.0	154	479	168	321	414	384
small016	0.0	0.0	0.0	0.0	0.0	0.0	145	432	185	320	506	383
snp-02-004-104	8.0	16.6	6.4	10.9	35.2	8.5	83639	78291	67547	77951	97810	75194
sorrell3	1.5	1.9	1.3	2.9	1.7	1.0	2217	1035	2211	1283	1030	1280
south31	7.6	8.6	7.5	6.5	15.5	6.6	26909	17859	28503	21279	22812	21298
sp150x300d	0.0	0.0	0.0	0.0	0.0	0.0	24	255	44	44	273	250
sp97ar	0.7	0.7	0.7	0.8	2.7	0.8	1395	1212	1327	1252	8487	1881
sp97ic	0.1	0.1	0.1	0.2	0.2	0.1	785	857	735	903	2554	1435
sp98ar	0.6	0.8	0.7	0.8	2.5	0.8	2465	2599	2827	2058	10623	4054
sp98ic	0.2	0.2	0.2	0.3	0.6	0.2	1146	1029	1117	1108	4177	1591
sp98ir	0.1	0.1	0.1	0.2	0.2	0.1	1171	668	789	583	1708	1200
spal_004	3600.0*	3600.0*	3560.3*	3600.0*	3600.0*	3600.0*	15408*	20540*	16183*	5000*	15173*	203591*
splan1	3592.3*	3600.0*	3584.1*	3599.4*	3600.0*	3594.3*	853040*	1048502*	617295*	413000*	522513*	786036*
splice1k1	13.1	20.5	14.1	38.1	24.8	103.9	2429	2562	2421	8727	3420	11873
square15	22.5	60.4	23.5	57.8	154.6	33.4	44172	46193	44369	55427	66121	52197

Continued on next page

Table B.1.: LP solver comparison (`allLP` test set, SCIP 6.0.2)

	time						iters					
LP solver	CPLEX	Clp	Gurobi	MOSEK	SoPlex	Xpress	CPLEX	Clp	Gurobi	MOSEK	SoPlex	Xpress
instance	12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09	12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09
square41	55.0	62.6	17.1	3598.7*	81.4	85.9	8911	8598	3091	650273*	16209	21451
square47	109.1	124.5	39.5	3600.0*	246.9	310.2	12886	9130	3791	140012*	21709	41488
stair	0.0	0.0	0.0	0.0	0.0	0.0	245	359	291	207	459	428
standata	0.0	0.0	0.0	0.0	0.0	0.0	84	99	77	106	130	88
standmps	0.0	0.0	0.0	0.0	0.0	0.0	103	152	63	144	229	152
stat96v1	27.8	3600.0*	30.5	94.2	88.5	113.1	14248	139848*	14893	29742	22934	53900
stat96v2	3596.0*	3600.0*	1609.0	3600.0*	3600.0*	3586.8*	244356*	0*	134098	169000*	104452*	312737*
stat96v3	3587.6*	3600.0*	2473.2	3600.0*	3600.0*	3595.1*	195296*	58190*	173853	135000*	51872*	270075*
stat96v4	38.9	60.1	26.0	117.5	62.6	33.5	41294	84196	40016	45596	45137	40426
stat96v5	4.2	22.6	3.6	8.8	15.1	7.7	3350	12520	3933	5346	11179	6830
stein27	0.0	0.0	0.0	0.0	0.0	0.0	57	47	91	55	42	43
stein45	0.0	0.0	0.0	0.0	0.0	0.0	126	144	275	187	121	135
stocfor1	0.0	0.0	0.0	0.0	0.0	0.0	27	63	24	55	66	55
stocfor2	0.1	0.1	0.1	0.1	0.1	0.1	654	1587	674	1254	1876	1488
stocfor3	1.0	1.5	1.1	1.6	2.3	1.1	6614	12683	4732	11002	14616	12159
stockholm	2.6	422.6*	1.6	1.9	11.5	5.5	0	55316*	0	0	0	0
stormG2_1000	799.5	325.8	49.7	359.5	3600.0*	97.8	558711	562679	478832	453917	630952*	870894
stormg2-125	10.4	8.0	4.1	7.0	28.1	5.3	68974	71055	56431	55534	78478	101659
stormg2-27	0.8	0.8	0.7	0.8	2.2	0.7	15074	15793	12230	12101	17117	20302
stormg2-8	0.2	0.2	0.2	0.2	0.3	0.2	3990	4492	3690	3589	5919	5741
stormg2_1000	810.0	325.9	49.5	354.6	3600.0*	98.8	558711	562679	478832	453917	630952*	870894
stp3d	115.7	154.8	196.6	430.2	463.2	305.4	0	0	0	0	0	0
sts405	0.2	0.5	0.6	0.7	0.3	0.2	679	634	27271	621	504	1081
sts729	5.5	2.5	2.7	4.8	1.0	0.6	10370	1146	88453	1316	901	2388
supportcase10	35.1	47.7	15.0	160.8	232.7	38.6	59694	38194	87126	44929	101278	205378
supportcase12	9.4	13.7	9.0	8.7	23.5	10.1	37927	32851	16536	7394	59833	43255
supportcase18	0.1	0.2	0.2	0.2	0.8	0.2	474	874	745	494	5199	1074
supportcase19	3582.1*	3600.0*	3593.8*	3600.0*	3600.0*	3595.6*	1006609*	2289748*	1995118*	529000*	127730*	1350862*
supportcase22	8.5	8.0	3.8	3.9	7.1	9.0	5078	2471	3003	804	4065	4505
supportcase26	0.0	0.0	0.0	0.0	0.0	0.0	13	220	4	195	16	220
supportcase33	0.6	1.2	0.7	1.1	4.5	0.6	1321	2703	1387	1637	10256	1162
supportcase40	0.3	0.4	0.4	1.1	0.9	0.5	4652	3410	4102	3934	6045	4378
supportcase42	1.1	23.0	1.2	4.6	1.1	1.7	1794	15279	2943	3088	1861	3741
supportcase6	4.3	6.3	3.0	5.7	16.8	5.4	4429	4633	28331	4738	9655	4601

Continued on next page

Table B.1.: LP solver comparison (`allLP` test set, SCIP 6.0.2)

	time						iters					
LP solver	CPLEX	Clp	Gurobi	MOSEK	SoPlex	Xpress	CPLEX	Clp	Gurobi	MOSEK	SoPlex	Xpress
instance	12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09	12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09
supportcase7	4.6	6.3	4.7	5.2	7.7	4.5	7092	10224	8299	6595	15993	10294
swath	0.1	0.1	0.1	0.1	0.1	0.1	109	108	206	120	128	141
swath1	0.1	0.1	0.1	0.1	0.1	0.1	109	108	206	120	128	141
swath3	0.1	0.1	0.1	0.1	0.1	0.1	109	108	206	120	128	141
sws	0.2	0.2	0.2	0.2	0.2	0.2	0	0	0	0	0	0
t0331-4l	3.8	5.2	4.6	7.1	9.1	6.8	6133	6473	6931	8240	15572	9868
t1717	2.9	3.8	3.1	4.0	7.2	5.7	6779	7409	7622	7023	16046	11849
t1722	0.7	0.9	0.8	0.8	2.0	1.2	2672	2705	2892	2462	11967	4470
tanglegram1	0.8	0.7	0.7	1.0	0.7	0.7	311	316	616	358	292	319
tanglegram2	0.1	0.1	0.1	0.1	0.1	0.1	163	162	310	155	149	165
tbfp-network	11.9	7.8	7.6	6.8	18.4	6.2	22349	8279	11917	10178	17929	9925
testbig	0.5	0.5	0.5	0.5	0.5	0.5	4	1644	206	94	2708	1554
thor50dday	1.0	1.0	1.0	1.1	1.1	1.0	14	15	14	37	15	14
timtab1	0.0	0.0	0.0	0.0	0.0	0.0	13	16	11	17	15	18
timtab2	0.0	0.0	0.0	0.0	0.0	0.0	24	27	20	64	28	28
toll-like	0.1	0.0	0.1	0.1	0.1	0.0	695	705	724	696	672	733
tp-6	3600.0*	3600.0*	3600.0*	3600.0*	3600.0*	3600.0*	0*	0*	0*	0*	0*	0*
tr12-30	0.0	0.0	0.0	0.0	0.0	0.0	328	680	328	328	680	680
traininstance2	0.1	0.1	0.2	0.1	0.2	0.1	106	2705	1526	350	2709	158
traininstance6	0.1	0.1	0.1	0.1	0.1	0.1	29	2100	1365	89	2068	80
transportmoment	1.0	1.2	1.0	1.2	1.1	1.1	3128	5529	2800	4199	6276	6737
trento1	1.8	2.7	2.3	4.8	4.5	3.9	8859	10423	11939	13564	17927	18043
triptim1	22.6	48.5	17.4	45.4	53.3	14.4	34820	47896	25111	41112	47751	38329
triptim2	227.1	625.8	217.1	380.6	706.3	214.4	190978	335812	186998	164075	302280	246998
triptim3	148.2	238.3	123.8	165.2	135.5	70.7	145402	155163	116668	102792	87970	108504
truss	1.8	3.0	1.8	4.0	1.8	1.0	17391	19518	17399	21071	13562	8979
ts-palko	3600.0*	3600.0*	3583.4*	3600.0*	3600.0*	3596.0*	147398*	58247*	80610*	138000*	58004*	169859*
tuff	0.0	0.0	0.0	0.0	0.0	0.0	64	132	110	126	173	174
tw-myciel4	0.1	0.3	0.2	3.2	0.3	1.3	1866	1955	1046	8753	3559	7797
uc-case11	10.3	13.2	4.3	28.6	15.7	7.6	15778	19092	16518	19935	25326	22269
uc-case3	2.9	12.2	1.6	5.2	21.8	4.4	15943	19980	12940	8831	25908	19010
uccase12	3.8	26.1	3.7	127.5	22.8	3.0	43619	43932	40594	83957	59771	40701
uccase9	1.9	9.1	1.4	5.2	25.5	3.5	16607	17740	14126	8822	31403	17621
uct-subprob	0.1	0.1	0.1	0.1	0.4	0.1	879	1596	774	792	5444	2127

Continued on next page

Table B.1.: LP solver comparison (`allLP` test set, SCIP 6.0.2)

	time						iters					
LP solver	CPLEX	Clp	Gurobi	MOSEK	SoPlex	Xpress	CPLEX	Clp	Gurobi	MOSEK	SoPlex	Xpress
instance	12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09	12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09
ulevimin	4.9	12.6	26.0	3600.0*	22.0*	5.1	40 619	33 333	349 886	0*	36 698*	22 862
umts	0.1	0.2	0.1	0.2	0.4	0.1	470	1442	704	1350	5525	1441
unitcal_7	1.1	1.0	0.8	1.7	5.2	0.8	24 597	14 959	12 160	12 088	17 879	16 927
us04	0.6	0.6	0.6	0.6	0.6	0.6	239	284	201	352	427	370
usAbbrv-8-25_70	0.2	0.2	0.2	0.2	0.2	0.2	1413	1435	1573	1272	1441	1400
van	2.3	12.3	2.5	5.0	5.4	3.8	8930	8518	8953	8747	13 864	15 100
var-smallemery-m6j6	1.6	1.8	1.6	1.9	4.6	1.9	2189	2387	2035	2083	10 513	2546
vpm1	0.0	0.0	0.0	0.0	0.0	0.0	30	35	28	19	39	28
vpm2	0.0	0.0	0.0	0.0	0.0	0.0	40	49	54	43	58	40
vpphard	5.8	2.4	1.8	5.0	9.8	2.9	5640	1959	3174	2812	11 673	2763
vpphard2	6.2	6.2	6.2	6.6	6.8	5.9	0	0	0	0	0	0
vtp-base	0.0	0.0	0.0	0.0	0.0	0.0	20	25	15	17	24	25
wachplan	0.1	0.1	0.1	0.2	0.7	0.1	1118	874	797	1291	10 450	1970
watson_1	25.7	56.6	23.7	28.7	210.1	13.0	83 546	291 328	429 833	50 550	234 474	176 098
watson_2	38.3	164.7	25.4	30.7	1257.0	23.1	516 322	607 493	181 780	66 966	314 683	270 321
wide15	23.7	47.6	24.3	55.5	127.5	32.2	44 375	45 741	44 644	55 761	64 118	52 140
wnq-n100-mw99-14	6.0	6.1	7.4	14.1	6.0	7.2	706	487	2860	602	775	1269
wood1p	0.2	0.2	0.2	0.2	0.2	0.2	114	143	101	198	121	387
woodw	0.1	0.1	0.1	0.1	0.1	0.1	464	727	705	712	1144	753
world	9.2	18.6	7.2	21.4	66.2	37.0	32 287	32 558	31 079	31 836	50 315	64 969
zed	0.0	0.0	0.0	0.0	0.0	0.0	12	20	18	17	22	17
zib54-UUE	0.1	0.1	0.1	0.2	0.1	0.2	1709	1538	2217	2018	2035	5158

Table B.2.: LP solution polishing on the first LP relaxation (MIPLIB 2017 benchmark)

	fractionality		iterations		time		condition	
settings	polish	w/o	polish	w/o	polish	w/o	polish	w/o
instance								
30n20b8	122.3	123.0	7231.7	7227.7	0.64	0.64	4.89	4.89
50v-10	29.0	29.0	220.0	220.0	0.0	0.0	2.0	2.0
CMS750_4	879.0	1835.0	7747.0	5344.0	0.22	0.18	5.15	5.15
academictimetablesmall	901.0	1015.3	10334.7	10017.0	2.97	2.87	4.39	3.82
air05	222.0	222.0	6362.0	6362.0	0.75	0.75	4.04	4.04
app1-1	24.7	24.7	1334.7	1314.3	0.03	0.02	5.34	5.36
app1-2	245.0	245.0	14573.7	14573.3	9.95	9.95	4.71	5.36
assign1-5-8	114.0	114.0	1146.0	1146.0	0.02	0.02	2.0	2.0
atlanta-ip	1671.7	1676.0	19484.3	19474.0	12.46	12.39	6.29	6.29
b1c1s1	246.0	246.0	1584.0	1584.0	0.03	0.02	3.82	3.82
bab2	643.7	643.7	135639.0	135632.3	117.75	117.12	4.26	4.26
bab6	997.3	997.3	103940.7	103934.3	58.49	57.81	4.58	4.58
beasleyC3	153.0	153.0	1114.0	1106.0	0.02	0.02	2.51	2.51
binkar10_1	38.0	38.0	693.0	693.0	0.01	0.01	2.51	2.51
blp-ar98	129.0	130.0	895.0	894.0	0.16	0.15	2.94	2.94
blp-ic98	53.0	53.0	445.0	445.0	0.09	0.09	2.69	2.69
bnatt400	592.3	708.7	865.7	753.0	0.12	0.02	3.65	2.39
bnatt500	713.3	902.0	1153.0	980.3	0.19	0.03	4.1	2.53
bppc4-08	48.7	48.7	2534.0	2534.0	0.11	0.1	2.92	2.92
brazil3	915.3	921.7	16276.3	16269.3	3.81	3.82	3.78	3.62
buildingenergy	8043.3	8043.3	143851.7	143851.7	521.6	441.79	6.33	6.33
cbs-cta	143.7	174.7	11723.3	10164.0	1.78	1.47	7.93	4.23
chromaticindex1024-7	47.7k	47.7k	98520.0	94537.3	258.51	244.48	5.12	5.41
chromaticindex512-7	23.3k	23.3k	56480.7	54530.3	67.11	64.06	4.87	5.0
cmflsp50-24-8-8	491.0	491.0	10336.0	10336.0	2.05	2.04	3.34	3.34
co-100	203.0	212.0	3285.0	3237.0	0.76	0.76	4.86	4.87
cod105	694.0	694.0	12305.3	12305.3	6.56	6.55	4.66	4.66
comp07-2idx	1088.3	1147.7	9689.7	9393.7	5.96	5.13	4.15	3.69
comp21-2idx	734.3	793.7	4497.0	4220.7	1.4	1.15	3.65	3.39
cost266-UUE	56.0	56.0	2038.0	2038.0	0.07	0.07	5.84	5.84
cryptanalysiskb128n5obj14	19.7k	21.7k	39752.7	36445.7	36.19	20.08	4.03	4.16
cryptanalysiskb128n5obj16	19.7k	21.7k	41488.3	38295.3	38.38	22.97	4.15	4.21
csched007	85.0	85.0	1371.0	1371.0	0.05	0.05	3.96	3.96
csched008	86.7	92.7	5837.0	5825.0	0.25	0.26	4.79	4.32

Continued on next page

Table B.2.: LP solution polishing on the first LP relaxation (MIPLIB 2017 benchmark)

	fractionality		iterations		time		condition	
settings	polish	w/o	polish	w/o	polish	w/o	polish	w/o
instance								
cvs16r128-89	3210.0	3210.0	11398.7	11397.7	2.58	2.56	4.1	4.1
dano3_3	12.0	12.0	86605.0	86605.0	36.21	36.19	6.45	6.45
dano3_5	25.0	25.0	84882.3	84882.3	36.23	36.17	6.43	6.43
decomp2	236.7	236.7	3497.0	2421.0	0.06	0.07	2.44	3.43
drayage-100-23	205.3	215.3	6032.7	6019.0	0.34	0.34	3.79	3.78
drayage-25-23	183.7	194.7	4966.3	4952.3	0.26	0.26	3.53	3.52
dws008-01	11.0	29.0	348.0	234.0	0.01	0.01	2.91	3.04
eil33-2	30.0	30.0	1114.0	1114.0	0.11	0.11	2.51	2.51
eilA101-2	71.0	71.0	10702.0	10702.0	22.51	22.21	2.61	2.61
enlight_hard	-	-	0.0	0.0	0.0	0.0	-	-
ex10	-	-	0.0	0.0	0.0	0.0	-	-
ex9	-	-	0.0	0.0	0.0	0.0	-	-
exp-1-500-5-5	134.0	134.0	617.0	617.0	0.01	0.0	3.91	3.91
fast0507	255.3	260.3	12421.0	12410.0	3.48	3.47	2.92	2.93
fastxgemm-n2r6s0t2	24.7	24.7	1307.0	1306.7	0.1	0.08	2.65	2.65
fhnw-binpack4-4	26.7	192.3	583.3	383.3	0.0	0.0	1.85	1.99
fhnw-binpack4-48	809.0	3156.0	5554.0	3207.0	0.06	0.05	2.93	2.11
fiball	258.7	266.7	3014.0	2955.0	0.3	0.26	2.94	2.09
gen-ip002	18.0	18.0	44.0	44.0	0.0	0.0	2.43	2.43
gen-ip054	15.0	15.0	29.0	29.0	0.0	0.0	2.11	2.11
germanrr	210.7	210.7	8031.3	8031.3	1.97	1.97	3.83	3.83
gfd-schedulen180f7d50m30k18	37.3k	56.7k	79862.3	69882.0	590.95	142.51	4.74	4.48
glass-sc	101.0	101.0	864.0	864.0	0.13	0.13	2.49	2.49
glass4	72.0	72.0	74.0	74.0	0.0	0.0	1.18	1.18
gmu-35-40	11.0	11.0	745.0	745.0	0.01	0.02	4.08	4.08
gmu-35-50	16.7	16.7	1424.7	1423.3	0.03	0.03	4.19	4.19
graph20-20-1rand	88.0	91.7	131.0	124.7	0.0	0.0	2.02	2.1
graphdraw-domain	88.0	100.0	218.0	208.0	0.01	0.0	3.04	3.04
h80x6320d	108.0	116.0	212.0	208.0	0.01	0.02	3.81	3.81
highschool1-aigio	-	-	0.0	0.0	3552.76	3553.17	6.35	6.3
hypothyroid-k1	161.0	161.0	182.0	182.0	0.02	0.02	4.69	4.69
ic97_potential	131.3	206.3	808.3	732.3	0.01	0.01	1.87	1.87
icir97_tension	511.7	586.0	891.3	807.3	0.0	0.0	2.96	2.96
irish-electricity	4429.7	4491.0	69702.3	69623.7	155.3	152.42	9.03	9.03

Continued on next page

Table B.2.: LP solution polishing on the first LP relaxation (MIPLIB 2017 benchmark)

	fractionality		iterations		time		condition	
settings	polish	w/o	polish	w/o	polish	w/o	polish	w/o
instance								
irp	17.0	17.0	604.0	604.0	0.17	0.17	2.59	2.59
istanbul-no-cutoff	13.0	13.0	3150.0	3150.0	0.22	0.12	3.6	3.6
k1mushroom	416.7	416.7	5767.3	5767.3	1.85	1.8	6.22	6.22
lectsched-5-obj	572.3	2770.0	5511.3	3272.7	0.21	0.12	4.17	3.97
leo1	65.0	65.0	855.0	855.0	0.09	0.09	2.67	2.67
leo2	70.0	70.0	1424.0	1424.0	0.28	0.28	3.08	3.08
lotsize	470.0	525.0	1553.0	1498.0	0.02	0.02	6.08	6.11
mad	16.0	16.0	67.0	67.0	0.0	0.0	1.72	1.72
map10	65.0	65.0	19061.0	19061.0	7.6	7.16	4.21	4.21
map16715-04	69.0	69.0	19828.0	19828.0	8.56	8.11	4.07	4.07
markshare2	7.0	7.0	18.0	18.0	0.0	0.0	0.92	0.92
markshare_4_0	4.0	4.0	5.0	5.0	0.0	0.0	0.94	0.94
mas74	12.0	12.0	45.0	45.0	0.0	0.0	1.96	1.96
mas76	11.0	11.0	33.0	33.0	0.0	0.0	1.99	1.99
mc11	363.0	363.0	1755.0	1755.0	0.05	0.05	2.59	2.59
mcsched	1259.0	1259.0	6040.0	6040.0	0.39	0.39	3.11	3.11
mik-250-20-75-4	75.0	75.0	81.0	81.0	0.0	0.0	0.26	0.26
milo-v12-6-r2-40-1	185.7	353.7	3687.0	3573.7	0.19	0.19	3.88	3.87
momentum1	231.0	231.0	1933.0	1933.0	0.12	0.12	5.43	5.43
mushroom-best	21.0	21.0	694.0	694.0	0.48	0.48	5.21	5.21
mzzv11	880.0	984.3	43724.0	43649.7	24.2	24.24	5.81	5.83
mzzv42z	746.3	851.0	15180.3	15105.7	5.07	5.04	6.14	6.16
n2seq36q	177.0	181.3	6492.0	5617.0	0.58	0.58	3.89	3.83
n3div36	24.0	24.0	523.0	523.0	0.24	0.24	2.08	2.08
n5-3	31.0	32.0	938.0	937.0	0.02	0.02	2.89	2.89
neos-1122047	-	-	0.0	0.0	0.0	0.0	-	-
neos-1171448	66.3	130.3	8715.7	8177.0	1.55	1.05	3.55	3.56
neos-1171737	59.0	118.7	2979.0	2828.0	0.2	0.17	3.29	3.26
neos-1354092	897.3	897.3	144560.7	144560.7	105.38	105.07	4.43	4.43
neos-1445765	145.0	145.0	2521.0	2521.0	0.14	0.14	3.56	3.56
neos-1456979	105.0	125.7	922.0	896.0	0.11	0.11	4.6	4.25
neos-1582420	290.7	292.3	2713.0	2697.0	0.18	0.19	3.93	3.93
neos-2075418-temuka	-	-	0.0	0.0	3488.12	3488.14	6.16	5.8
neos-2657525-crna	41.0	58.0	138.0	129.0	0.0	0.0	1.95	1.76

Continued on next page

Table B.2.: LP solution polishing on the first LP relaxation (MIPLIB 2017 benchmark)

	fractionality		iterations		time		condition	
settings	polish	w/o	polish	w/o	polish	w/o	polish	w/o
instance								
neos-2746589-doon	447.0	469.0	13347.3	13286.7	5.13	5.12	4.94	4.66
neos-2978193-inde	24.3	26.3	1838.7	1836.0	0.09	0.09	2.52	2.52
neos-2987310-joes	-	-	12217.7	10206.3	0.64	0.63	2.88	3.07
neos-3004026-krka	1696.0	1722.0	3120.0	3097.0	0.09	0.08	2.3	3.11
neos-3024952-loue	350.7	395.0	11108.0	10826.7	0.9	0.89	3.18	3.17
neos-3046615-murg	41.3	72.0	148.0	116.3	0.0	0.0	1.57	1.72
neos-3083819-nubu	33.0	33.0	1027.0	1027.0	0.03	0.03	2.76	2.76
neos-3216931-puriri	682.0	693.3	8087.3	8065.3	1.69	1.69	4.95	4.28
neos-3381206-awhea	437.3	437.3	1733.0	1733.0	0.02	0.03	3.13	3.13
neos-3402294-bobin	168.7	180.7	5810.3	5796.7	3.56	3.46	3.1	3.04
neos-3402454-bohle	-	-	0.0	0.0	3384.02	3383.97	6.42	6.25
neos-3555904-turama	4128.3	4327.7	17729.3	17420.3	9.64	8.14	5.54	5.19
neos-3627168-kasai	130.0	135.0	2410.0	2396.0	0.07	0.07	2.7	2.7
neos-3656078-kumeu	1570.7	1802.3	48358.0	47896.0	19.69	19.59	10.52	10.06
neos-3754480-nidda	41.0	41.0	596.0	596.0	0.0	0.01	5.99	5.99
neos-3988577-wolgan	2352.3	2535.3	84648.3	81330.0	134.79	133.25	4.61	4.44
neos-4300652-rahue	92.7	114.3	4518.0	4448.0	1.42	0.57	2.73	2.86
neos-4338804-snowy	153.0	166.7	473.7	335.3	0.0	0.0	2.17	2.19
neos-4387871-tavua	34.0	34.0	2514.3	2514.3	0.12	0.12	3.41	3.41
neos-4413714-turia	1019.3	2089.3	9298.7	8573.7	10.42	10.38	5.66	6.17
neos-4532248-waihi	12.0k	27.3k	31140.7	87896.3	1380.04	637.83	6.07	5.26
neos-4647030-tutaki	778.7	782.3	5989.7	5986.0	4.19	4.18	6.48	6.41
neos-4722843-widden	2779.7	4087.7	5675.3	4345.3	0.64	0.51	6.48	6.43
neos-4738912-atrato	232.3	232.3	12477.7	12477.0	0.49	0.49	4.8	4.8
neos-4763324-toguru	1563.0	1563.0	12496.0	12496.0	18.37	18.2	3.87	3.87
neos-4954672-berkel	50.0	51.0	476.0	469.0	0.01	0.0	2.94	2.94
neos-5049753-cuanza	172.3	232.0	11849.0	11434.7	6.66	6.48	3.25	3.25
neos-5052403-cygnet	652.0	697.3	205452.3	204758.3	613.05	610.43	5.3	5.37
neos-5093327-huahum	62.7	64.0	4190.0	4188.7	0.31	0.3	3.46	3.46
neos-5104907-jarama	661.7	835.0	29168.7	28832.7	66.69	66.59	4.53	4.47
neos-5107597-kakapo	61.0	68.0	176.0	169.0	0.01	0.01	3.04	3.08
neos-5114902-kasavu	221.7	259.0	21454.0	20901.7	94.39	94.78	3.65	3.7
neos-5188808-nattai	22.0	24.0	3009.0	3008.0	0.78	0.23	3.04	3.04
neos-5195221-niemur	1068.3	1239.7	2759.3	2500.7	0.95	0.18	4.05	4.6

Continued on next page

Table B.2.: LP solution polishing on the first LP relaxation (MIPLIB 2017 benchmark)

	fractionality		iterations		time		condition	
settings	polish	w/o	polish	w/o	polish	w/o	polish	w/o
instance								
neos-631710	1438.3	1456.3	46654.0	46579.7	686.2	686.26	4.68	4.6
neos-662469	362.0	363.0	11943.0	11936.0	3.19	3.21	3.46	3.34
neos-787933	-	-	165.0	151.0	0.0	0.0	2.36	3.15
neos-827175	5.7	59.3	11779.0	11641.3	2.36	2.33	3.76	4.09
neos-848589	616.0	616.0	1401.0	1401.0	0.66	0.61	3.04	3.04
neos-860300	105.0	105.0	396.0	396.0	0.02	0.03	4.46	4.46
neos-873061	91.7	91.7	37238.7	37233.3	139.41	137.12	4.09	4.09
neos-911970	46.7	51.0	545.0	540.7	0.0	0.01	1.85	2.01
neos-933966	1208.7	1220.7	21569.3	21482.7	10.32	10.3	4.25	4.07
neos-950242	265.3	305.7	5051.7	4905.7	1.76	1.75	2.87	3.06
neos-957323	139.7	153.3	13040.0	13007.0	4.1	4.09	3.09	3.25
neos-960392	365.7	384.7	16066.0	15895.3	7.4	7.36	4.67	4.34
neos17	171.0	171.0	1202.0	1202.0	0.03	0.03	4.11	4.11
neos5	35.0	35.0	140.0	139.0	0.0	0.0	2.72	2.61
neos8	29.7	30.3	126.3	125.7	0.0	0.0	2.62	2.62
neos859080	3.0	3.0	48.0	37.0	0.0	0.0	2.18	2.2
net12	377.0	405.0	5307.3	5256.7	0.68	0.61	4.35	4.32
netdiversion	4146.7	4176.3	27131.7	25187.7	19.47	19.13	4.0	4.07
nexp-150-20-8-5	79.7	79.7	1744.0	1744.0	0.07	0.07	4.82	4.82
ns1116954	463.3	480.0	27953.7	27557.7	59.09	57.67	3.76	3.74
ns1208400	383.7	402.0	12557.7	12527.0	1.74	1.72	3.94	3.78
ns1644855	326.3	335.7	76300.0	76284.7	179.61	178.71	7.78	7.83
ns1760995	683.3	946.3	7029.7	6365.0	6.47	5.78	7.95	6.45
ns1830653	190.7	195.7	2046.0	2037.0	0.17	0.16	3.48	3.39
ns1952667	39.0	39.0	134.0	134.0	0.12	0.12	2.43	2.43
nu25-pr12	33.0	36.0	1675.0	1670.0	0.04	0.04	2.28	2.3
nursesched-medium-hint03	1162.3	1246.0	16145.0	15835.3	12.6	12.34	4.09	4.19
nursesched-sprint02	358.3	399.3	9608.7	9511.0	1.41	1.4	3.22	3.25
nw04	6.0	6.0	475.0	475.0	0.56	0.55	2.23	2.23
opm2-z10-s4	5584.0	5584.0	35887.0	35887.0	68.97	68.53	4.28	4.28
p200x1188c	5.0	5.0	553.0	553.0	0.01	0.01	2.32	2.32
peg-solitaire-a3	982.7	997.3	11842.3	11816.7	5.05	5.05	4.14	3.91
pg	93.0	93.0	247.0	247.0	0.0	0.0	0.86	0.86
pg5_34	88.0	88.0	415.0	415.0	0.01	0.01	1.23	1.23

Continued on next page

Table B.2.: LP solution polishing on the first LP relaxation (MIPLIB 2017 benchmark)

	fractionality		iterations		time		condition	
settings	polish	w/o	polish	w/o	polish	w/o	polish	w/o
instance								
physiciansched3-3	1586.7	1661.0	95443.7	95243.7	127.3	126.78	6.44	6.44
physiciansched6-2	999.7	1087.7	6594.3	6246.0	1.06	0.8	4.47	3.96
piperout-08	135.7	1554.7	3705.7	2187.3	0.04	0.02	3.09	3.36
piperout-27	242.0	2388.3	5535.0	3305.7	0.06	0.05	3.96	4.16
pk1	15.0	15.0	58.0	58.0	0.0	0.0	3.51	3.51
proteindesign121hz512p9	202.0	212.0	1486.0	1476.0	0.56	0.55	7.0	7.0
proteindesign122trx11p8	150.0	162.0	1119.0	1107.0	0.35	0.34	7.15	7.18
qap10	1226.0	1226.0	43730.0	43730.0	15.33	15.32	5.5	5.5
radiationm18-12-05	564.7	685.0	4434.7	4009.3	0.09	0.08	3.35	3.36
radiationm40-10-02	2273.3	2878.3	17462.7	15611.7	2.44	2.36	3.99	3.96
rail01	7581.7	7596.3	142061.7	141057.3	170.41	167.92	5.93	5.93
rail02	11.0k	11.0k	837453.3	836192.7	3077.34	3052.87	5.83	5.84
rail507	249.7	253.7	12139.7	12125.7	3.21	3.2	3.33	3.34
ran14x18-disj-8	86.0	86.0	1474.0	1474.0	0.05	0.05	5.04	5.04
rd-rplusc-21	55.0	102.0	538.3	504.0	0.05	0.04	3.21	2.89
reblock115	878.0	878.0	3640.7	3640.7	0.3	0.29	2.74	2.74
rmatr100-p10	51.0	51.0	8160.0	8160.0	1.65	1.66	3.45	3.45
rmatr200-p5	66.0	66.0	19840.7	19840.7	22.23	22.19	4.61	4.61
rocI-4-11	413.0	461.3	1201.7	1143.3	0.04	0.02	3.46	3.16
rocII-5-11	150.3	168.7	455.0	429.7	0.02	0.01	3.35	3.41
rococoB10-011000	261.0	261.0	6180.7	6180.7	0.4	0.41	2.46	2.46
rococoC10-001000	136.3	149.7	2544.7	2529.0	0.08	0.09	2.7	2.7
roi2alpha3n4	36.3	36.3	3058.3	3058.0	0.35	0.35	3.28	3.28
roi5alpha10n8	89.0	89.0	10405.7	10405.7	6.02	6.0	3.92	3.92
roll3000	179.0	192.0	1717.0	1702.0	0.07	0.07	3.63	3.65
s100	206.7	208.0	291486.7	291486.3	2020.78	2007.26	4.66	4.66
s250r10	216.3	231.7	125716.3	125711.7	267.98	267.6	4.03	4.06
satellites2-40	2829.3	2849.3	78026.0	78011.3	114.52	114.72	8.34	5.33
satellites2-60-fs	3087.3	3100.7	66928.7	66918.0	79.3	79.35	6.3	4.75
savsched1	-	-	0.0	0.0	3359.27	3373.22	5.17	5.21
sct2	57.0	58.0	1734.3	1717.3	0.05	0.05	3.25	3.12
seymour	509.0	546.7	5317.0	5283.3	0.47	0.47	3.01	3.08
seymour1	92.0	120.0	5929.0	5916.0	0.6	0.6	3.08	3.08
sing326	944.0	946.0	42930.0	41114.0	43.83	43.64	4.67	4.67

Continued on next page

Table B.2.: LP solution polishing on the first LP relaxation (MIPLIB 2017 benchmark)

		fractionality		iterations		time		condition	
	settings	polish	w/o	polish	w/o	polish	w/o	polish	w/o
instance									
sing44		760.0	760.0	41242.0	40281.0	47.88	48.02	4.63	4.63
snp-02-004-104		41.0	68.0	97487.0	97452.0	40.41	40.6	5.29	5.29
sorrell3		1024.0	1024.0	1031.0	1031.0	0.84	0.74	2.08	2.08
sp150x300d		41.0	41.0	276.0	261.0	0.0	0.0	2.2	2.2
sp97ar		192.0	192.0	5991.0	5991.0	1.57	1.57	3.65	3.65
sp98ar		156.0	159.0	3068.0	3064.0	0.85	0.84	3.79	3.81
splice1k1		316.0	316.0	2893.3	2893.3	3.52	3.53	6.38	6.38
square41		357.0	357.0	16316.7	16316.7	69.15	67.16	6.04	6.04
square47		347.0	347.0	19508.0	19508.0	175.64	174.46	6.05	6.05
supportcase10		7247.7	7550.3	381688.3	381251.0	880.14	852.82	5.47	5.38
supportcase12		189.7	189.7	70242.7	70242.7	18.43	18.16	4.62	4.62
supportcase18		92.7	92.7	5493.3	5493.3	0.52	0.51	1.53	1.53
supportcase19		-	-	0.0	0.0	3406.22	3423.08	7.68	7.6
supportcase22		57.0	62.0	3892.0	3633.3	5.04	2.79	3.82	4.1
supportcase26		47.0	207.0	380.0	220.0	0.0	0.0	1.93	2.08
supportcase33		236.7	270.0	5171.7	5134.3	0.62	0.6	3.92	3.47
supportcase40		38.0	38.0	6021.0	6021.0	0.64	0.64	3.92	3.92
supportcase42		142.0	142.0	1805.0	1805.0	0.74	0.66	5.0	5.0
supportcase6		140.0	140.0	11702.0	11702.0	16.86	16.73	3.71	3.71
supportcase7		249.3	250.7	10368.3	10368.0	1.73	1.73	7.08	7.08
swath1		13.0	13.0	128.0	128.0	0.0	0.01	2.04	2.04
swath3		16.0	16.0	125.0	125.0	0.0	0.01	2.18	2.18
tbfp-network		216.7	216.7	16403.7	16403.7	16.67	16.69	5.0	5.0
thor50dday		49.0	49.0	46804.0	46801.3	125.9	125.63	3.48	3.48
timtab1		96.0	101.0	218.0	213.0	0.0	0.0	1.99	1.99
tr12-30		326.0	326.0	687.0	687.0	0.0	0.0	1.93	1.93
traininstance2		71.7	110.7	1921.3	1891.7	0.06	0.06	6.98	5.44
traininstance6		17.7	27.7	680.0	669.7	0.01	0.02	4.17	3.9
trento1		483.0	483.0	17727.0	17727.0	4.33	4.36	4.69	4.69
triptim1		3354.7	3365.3	67198.3	66766.3	98.1	97.87	5.32	5.2
uccase12		98.0	98.0	70892.7	70892.7	30.03	28.98	6.66	6.66
uccase9		422.0	422.0	31909.3	31886.3	25.77	25.82	4.78	4.78
uct-subprob		208.3	217.0	2154.3	2130.7	0.11	0.1	4.0	4.03
unitcal_7		683.0	704.3	23270.7	23138.3	5.7	3.18	5.4	5.4

Continued on next page

Table B.2.: LP solution polishing on the first LP relaxation (MIPLIB 2017 benchmark)

	fractionality		iterations		time		condition	
settings	polish	w/o	polish	w/o	polish	w/o	polish	w/o
instance								
var-smallemery-m6j6	395.0	396.0	10472.0	10471.0	3.68	3.65	6.9	6.9
wachplan	294.3	304.3	9518.0	9492.0	0.61	0.6	3.45	3.35

Table B.3.: Root gap comparison (MIPLIB 2017 benchmark)

	gap						time					
LP solver	Clp	CPLEX	Gurobi	MOSEK	SoPlex	Xpress	Clp	CPLEX	Gurobi	MOSEK	SoPlex	Xpress
instance	12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09	12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09
30n20b8	84.17	269.56	-	272.41	-	404.47	108.6+	64.2+	28.2+	85.8+	105.6+	72.1+
50v-10	12.58	16.79	12.40	11.95	15.68	14.31	35.7+	16.1+	16.2+	25.0+	16.1+	15.4+
CMS750_4	198.40	198.40	198.40	198.40	11.60	22.40	116.7+	7.5+	5.8+	26.8+	13.9+	8.9+
academictimetablesmall	-	-	-	-	-	-	3612.7*	72.4+	63.3+	323.2+	160.6+	74.4+
air05	-	15.38	-	17.44	17.44	17.44	5.2+	2.7+	3.6+	7.3+	6.0+	5.7+
app1-1	1.54k	98.42	235.27	135.29	445.11	143.67	248.1+	5.5+	5.0+	6.8+	5.5+	1.0+
app1-2	1.05k	-	702.28	501.84	134.52	131.04	3600.2*	72.1+	269.9+	444.7+	716.6+	153.6+
assign1-5-8	30.93	30.32	30.68	30.84	31.04	30.84	0.9+	1.2+	0.9+	1.3+	0.8+	1.1+
atlanta-ip	-	-	-	-	19.12	21.28	176.2+	93.9+	102.6+	219.2+	110.1+	103.0+
b1c1s1	236.56	68.78	71.60	62.87	56.54	62.12	3600.0*	32.0+	36.0+	50.4+	62.7+	35.7+
bab2	-	-	-	-	-	-	1425.6+	551.6+	653.5+	1922.0+	2177.4+	708.6+
bab6	-	-	-	-	-	-	536.0+	376.6+	405.7+	984.4+	1361.6+	577.8+
beasleyC3	1.80	5.11	1.85	21.68	1.06	2.65	1039.8+	25.8+	33.3+	44.1+	14.0+	19.8+
binkar10_1	0.75	0.55	3.05	1.20	0.75	1.15	5.6+	2.6+	1.1+	3.3+	1.6+	2.3+
blp-ar98	9.51	10.25	9.60	6.60	12.46	12.33	164.0+	136.3+	145.0+	168.1+	147.4+	148.9+
blp-ic98	15.71	14.62	20.73	13.71	18.52	10.62	39.9+	36.4+	39.3+	47.0+	32.5+	33.0+
bnatt400	-	-	-	-	-	-	12.8+	10.7+	9.3+	46.1+	10.2+	7.1+
bnatt500	-	-	-	-	-	-	21.5+	21.7+	15.6+	32.9+	14.7+	9.7+
bppc4-08	13.46	19.23	19.23	19.23	19.23	17.31	3.2+	1.5+	1.1+	5.4+	1.6+	1.7+
brazil3	-	-	-	-	-	-	211.7+	87.4+	56.4+	180.4+	140.5+	77.4+
buildingenergy	0.26	0.15	0.27	3.03	3.00	0.24	1328.1+	1682.3+	739.3+	3606.7*	3604.5*	1433.4+
cbs-cta	>999k	0.00	0.00	0.00	0.00	0.00	3600.0*	3.2	17.6	60.0	8.7	2.7
chromaticindex1024-7	-	33.33	33.33	33.33	33.33	33.33	1162.7+	1129.6+	1262.6+	2104.3+	1420.6+	1085.0+
chromaticindex512-7	-	33.33	33.33	33.33	33.33	33.33	322.0+	291.1+	295.3+	474.6+	463.0+	304.0+
cmflsp50-24-8-8	-	-	-	-	-	-	3601.4*	27.0+	43.8+	54.2+	53.8+	37.3+
co-100	38.67	32.70	1.10k	35.41	36.26	35.22	1643.8+	1022.8+	54.8+	1364.8+	1336.0+	848.0+
cod105	103.17	103.17	103.17	103.17	103.17	103.17	83.8+	61.2+	81.6+	325.4+	219.2+	65.9+
comp07-2idx	13.62k	13.62k	13.62k	13.62k	13.62k	13.62k	129.8+	270.8+	129.4+	574.4+	798.6+	103.6+
comp21-2idx	1.64k	1.66k	1.57k	1.55k	1.60k	1.62k	414.2+	121.1+	65.9+	284.8+	274.6+	88.0+
cost266-UUE	-	25.35	25.37	26.47	25.83	26.31	1337.9+	6.2+	7.0+	8.3+	5.8+	4.3+
cryptanalysiskb128n50obj14	-	-	-	-	-	-	2044.1+	1273.2+	1268.8+	1911.4+	1152.6+	1178.4+
cryptanalysiskb128n50obj16	-	-	-	-	-	-	1426.9+	1023.4+	1044.4+	2172.2+	1445.1+	1184.4+
csched007	-	-	-	-	-	-	106.5+	5.1+	5.0+	8.4+	9.4+	4.2+
csched008	-	-	-	-	-	-	124.4+	2.2+	1.2+	10.9+	3.0+	1.7+

Continued on next page

Table B.3.: Root gap comparison (MIPLIB 2017 benchmark)

	gap						time					
LP solver	Clp	CPLEX	Gurobi	MOSEK	SoPlex	Xpress	Clp	CPLEX	Gurobi	MOSEK	SoPlex	Xpress
instance	12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09	12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09
cvs16r128-89	90.88	90.85	82.76	82.13	90.38	90.29	24.4+	28.0+	50.5+	71.7+	30.2+	33.1+
dano3_3	0.21	0.20	0.20	0.20	0.20	0.20	486.0+	32.3+	87.3+	140.5+	112.9+	41.0+
dano3_5	0.68	0.51	0.35	0.32	0.10	0.44	364.7+	54.1+	82.1+	170.6+	144.9+	56.3+
decomp2	0.00	0.00	0.00	0.00	0.00	0.00	3.0	2.9	2.9	2.8	2.9	2.8
drayage-100-23	722.24	16.43	712.54	110.43	112.03	712.54	50.1+	6.7+	7.8+	45.3+	9.3+	8.7+
drayage-25-23	806.98	229.70	797.88	28.38	21.35	263.24	73.2+	6.8+	6.6+	60.8+	9.0+	7.6+
dws008-01	-	-	-	-	-	-	75.9+	32.6+	27.5+	41.1+	36.2+	37.0+
eil33-2	25.09	25.09	25.09	25.09	25.09	25.09	3.8+	3.1+	3.5+	4.4+	4.9+	4.0+
eilA101-2	63.05	67.41	73.00	66.61	73.00	71.61	120.3+	117.9+	126.4+	123.6+	193.8+	141.4+
enlight_hard	0.00	0.00	0.00	0.00	0.00	0.00	0.0	0.0	0.0	0.0	0.0	0.0
ex10	0.00	0.00	0.00	0.00	0.00	0.00	571.0	575.0	574.9	572.4	571.8	575.0
ex9	0.00	0.00	0.00	0.00	0.00	0.00	38.1	37.9	37.9	38.1	37.9	37.9
exp-1-500-5-5	0.00	0.00	0.00	0.00	0.00	0.00	3.9	4.4	3.7	4.8	3.9	3.8
fast0507	3.34	3.35	2.19	2.19	3.35	2.77	19.1+	10.9+	12.2+	20.4+	18.2+	17.0+
fastxgemm-n2r6s0t2	3.13k	3.13k	3.13k	3.13k	3.13k	3.13k	7.2+	3.3+	1.8+	16.6+	1.6+	1.7+
fhnw-binpack4-4	-	-	-	-	-	-	0.8+	0.4+	0.4+	1.9+	0.6+	0.5+
fhnw-binpack4-48	-	-	-	-	-	-	9.5+	3.4+	4.1+	27.5+	7.1+	4.5+
fiball	4.35	8.70	7.97	7.25	5.07	6.52	46.8+	12.3+	17.5+	42.9+	26.6+	14.0+
gen-ip002	3.48	3.24	3.24	3.38	3.48	3.24	0.1+	0.1+	0.1+	0.2+	0.1+	0.1+
gen-ip054	4.59	4.59	4.59	4.59	4.59	4.59	0.0+	0.0+	0.0+	0.0+	0.0+	0.0+
germanrr	-	-	-	-	-	-	68.5+	73.3+	60.4+	97.5+	98.9+	74.5+
gfd-schedulen180f7d50m30k18	-	-	-	-	-	-	3634.9*	3600.1*	2289.4+	3607.9*	3600.1*	2813.7+
glass-sc	65.54	65.16	72.04	68.99	65.16	59.89	90.7+	42.0+	42.9+	142.4+	34.6+	75.9+
glass4	450.00	590.63	215.63	241.67	431.25	187.50	0.7+	0.4+	1.7+	2.1+	0.7+	0.5+
gmu-35-40	0.48	0.70	0.28	0.47	0.60	0.11	1.1+	0.7+	0.6+	1.3+	0.9+	1.2+
gmu-35-50	1.06	0.36	0.64	0.78	0.84	0.36	2.9+	1.1+	1.0+	1.6+	1.1+	1.0+
graph20-20-1rand	1.37k	1.33k	2.80k	2.66k	2.78k	2.61k	36.1+	25.4+	38.3+	73.4+	29.0+	32.1+
graphdraw-domain	186.02	192.27	189.36	186.02	191.55	193.79	0.8+	0.5+	0.8+	1.8+	0.5+	0.4+
h80x6320d	0.68	0.58	0.72	0.27	0.18	0.27	1653.8+	65.3+	71.9+	113.6+	139.7+	134.6+
highschool1-aigio	-	-	-	-	-	-	3600.0*	3590.7+	3593.3+	3600.5*	3601.2*	3596.5+
hypothyroid-k1	0.00	0.00	0.00	0.00	0.00	0.00	23.9	23.9	24.1	25.0	23.9	23.9
ic97_potential	6.20	5.79	6.20	6.20	6.20	6.20	2.6+	0.6+	0.6+	2.2+	0.9+	0.7+
icir97_tension	-	-	1.84	-	-	-	9.2+	4.9+	8.0+	12.3+	8.3+	7.1+
irish-electricity	-	-	-	-	-	-	1249.4+	1142.1+	3023.3+	2700.8+	3057.1+	1057.2+

Continued on next page

Table B.3.: Root gap comparison (MIPLIB 2017 benchmark)

	gap						time					
LP solver	Clp	CPLEX	Gurobi	MOSEK	SoPlex	Xpress	Clp	CPLEX	Gurobi	MOSEK	SoPlex	Xpress
instance	12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09	12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09
irp	0.11	0.10	0.06	0.16	0.39	0.08	8.8+	8.9+	9.7+	10.9+	10.3+	10.5+
istanbul-no-cutoff	89.87	90.65	87.80	104.25	91.84	96.38	48.8+	51.8+	35.1+	76.3+	33.6+	31.6+
k1mushroom	1.28k	1.20k	1.28k	1.29k	1.29k	1.26k	2171.1+	1745.2+	1773.2+	2426.4+	1898.6+	1708.8+
lectsched-5-obj	-	-	-	-	-	-	33.0+	18.6+	15.0+	102.4+	21.9+	19.4+
leo1	12.93	9.56	60.89	12.16	59.26	12.66	37.5+	27.4+	6.8+	35.7+	21.5+	29.8+
leo2	-	62.41	-	-	62.67	-	4.3+	8.5+	5.2+	4.4+	6.1+	4.3+
lotsize	4.10	-	-	286.03	242.91	-	171.4+	98.9+	148.8+	163.1+	230.8+	102.1+
mad	-	-	-	-	-	-	0.8+	0.4+	0.3+	0.4+	0.2+	0.3+
map10	68.30	69.73	186.91	187.73	184.86	185.30	46.0+	43.6+	30.3+	81.8+	54.7+	28.2+
map16715-04	496.49	471.61	485.44	501.68	493.64	484.16	68.9+	56.5+	39.1+	84.6+	42.9+	46.1+
markshare2	-	-	-	-	-	-	0.2+	0.1+	0.1+	0.1+	0.1+	0.1+
markshare_4_0	-	-	-	-	-	-	0.1+	0.0+	0.0+	0.0+	0.0+	0.1+
mas74	21.74	21.74	17.51	21.74	21.74	21.44	1.0+	0.9+	0.9+	1.0+	0.9+	0.9+
mas76	2.57	2.58	2.57	2.57	2.57	2.58	0.9+	0.8+	0.9+	1.2+	0.6+	0.8+
mc11	3.91	2.39	2.00	0.97	2.02	2.38	1734.4+	52.4+	48.7+	40.6+	29.9+	26.2+
mcsched	20.17	20.17	20.17	20.17	20.17	20.15	9.7+	7.0+	9.5+	12.1+	8.6+	8.2+
mik-250-20-75-4	3.14	3.17	3.25	3.15	3.11	3.11	4.6+	5.8+	7.1+	5.2+	4.5+	6.8+
milo-v12-6-r2-40-1	301.68	295.70	72.48	39.80	290.71	69.94	834.2+	43.6+	81.0+	14.1+	45.0+	38.0+
momentum1	253.39	259.82	259.95	26.73	253.37	253.42	1078.2+	61.0+	116.3+	102.1+	44.1+	24.0+
mushroom-best	>999k	362.40k	2.82k	>999k	>999k	872.42k	15.8+	8.2+	8.4+	32.4+	37.2+	6.5+
mzzv11	-	21.46	22.97	14.99	33.95	18.18	648.6+	74.5+	74.2+	142.6+	96.2+	78.7+
mzzv42z	2.49k	3.15	5.27	9.78	2.49k	3.89	3601.3*	79.7+	76.8+	143.6+	79.1+	73.6+
n2seq36q	0.38	4.23	30.38	73.85	66.54	73.85	11.2+	23.1+	28.7+	76.1+	54.2+	22.6+
n3div36	17.09	12.11	17.57	16.11	22.43	19.53	95.3+	64.6+	82.8+	103.9+	78.5+	87.1+
n5-3	45.91	48.21	46.57	55.96	43.58	52.91	25.2+	8.3+	6.2+	10.5+	8.6+	9.6+
neos-1122047	0.00	0.00	0.00	0.00	0.00	0.00	12.0	11.8	12.4	12.2	12.1	12.3
neos-1171448	0.16	2.32	0.32	1.46	0.32	1.98	67.7+	5.2+	7.5+	29.2+	16.5+	6.3+
neos-1171737	2.63	2.90	4.28	3.72	2.63	4.28	31.0+	5.8+	18.0+	30.6+	23.1+	11.3+
neos-1354092	-	-	-	-	-	-	744.8+	31.6+	34.8+	108.1+	552.5+	35.2+
neos-1445765	3.51	3.51	3.42	1.70	3.52	3.51	66.8+	44.1+	47.7+	53.2+	45.6+	45.4+
neos-1456979	-	-	-	-	-	-	82.6+	4.5+	5.7+	28.8+	5.8+	6.2+
neos-1582420	-	4.62	-	2.48	-	4.90	195.6+	7.3+	6.5+	13.0+	7.5+	6.8+
neos-2075418-temuka	-	0.00	0.00	0.00	-	0.00	3600.5*	1287.1!	935.3!	1381.4!	3600.0*	298.8!
neos-2657525-crna	-	-	-	-	-	-	0.3+	0.2+	0.3+	0.4+	0.2+	0.4+

Continued on next page

Table B.3.: Root gap comparison (MIPLIB 2017 benchmark)

	gap						time					
LP solver	Clp	CPLEX	Gurobi	MOSEK	SoPlex	Xpress	Clp	CPLEX	Gurobi	MOSEK	SoPlex	Xpress
instance	12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09	12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09
neos-2746589-doon	-	-	-	-	-	-	88.6+	264.6+	168.8+	419.0+	221.2+	253.2+
neos-2978193-inde	90.45	1.28	1.28	3.29	1.28	1.28	105.7+	3.4+	2.6+	3.6+	4.9+	7.2+
neos-2987310-joes	0.00	0.00	0.00	0.00	0.00	0.00	33.8	19.1	19.2	19.4	19.6	19.3
neos-3004026-krka	-	-	-	-	-	-	11.7+	4.9+	5.5+	22.9+	9.8+	9.0+
neos-3024952-loue	-	-	-	-	-	-	413.0+	9.7+	8.1+	28.3+	12.8+	13.2+
neos-3046615-murg	496.68	618.23	621.34	490.36	601.22	493.73	0.6+	0.6+	0.3+	0.4+	0.4+	0.4+
neos-3083819-nubu	0.72	0.73	0.72	0.72	0.72	0.72	0.8+	0.6+	0.8+	1.3+	0.9+	1.1+
neos-3216931-puriri	-	-	-	-	-	-	908.6+	64.6+	55.2+	111.0+	75.7+	45.7+
neos-3381206-awhea	0.00	0.00	0.00	0.00	0.00	0.00	1.4	1.4	1.4	3.9	1.5	1.4
neos-3402294-bobin	-	-	-	-	-	-	217.2+	141.9+	65.6+	621.3+	96.2+	173.1+
neos-3402454-bohle	-	-	-	-	-	-	3600.1*	3600.0*	3600.1*	3777.9*	3600.1*	3600.2*
neos-3555904-turama	63.80	-	23.34	63.80	23.34	-	1110.0+	341.3+	336.8+	1750.8+	795.8+	246.7+
neos-3627168-kasai	8.14	4.12	4.43	4.63	5.39	4.45	11.3+	7.1+	9.8+	13.1+	9.3+	6.7+
neos-3656078-kumeu	-	-	-	-	-	-	3600.0*	484.3+	255.9+	791.2+	2524.4+	171.2+
neos-3754480-nidda	-	-	-	-	-	-	0.2+	0.1+	0.2+	0.2+	0.2+	0.1+
neos-3988577-wolgan	-	-	-	-	-	-	367.7+	75.3+	94.0+	863.3+	521.3+	131.8+
neos-4300652-rahue	-	-	-	-	-	-	522.8+	438.8+	339.7+	1314.1+	606.3+	559.1+
neos-4338804-snowy	182.79	182.79	182.79	182.79	182.79	182.79	1.3+	1.0+	1.0+	6.4+	1.2+	0.9+
neos-4387871-tavua	48.20	49.92	93.57	30.78	51.59	-	3600.0*	228.6+	36.0+	129.6+	107.7+	4.7+
neos-4413714-turia	6.84	6.97	12.32	937.70	16.49	18.92	1224.2+	1853.6+	631.5+	3600.0*	421.1+	1243.2+
neos-4532248-waihi	-	-	-	-	-	-	3600.9*	3680.6*	2012.6+	3603.1*	3600.0*	3606.5*
neos-4647030-tutaki	0.01	0.01	0.03	0.02	0.02	0.01	1367.9+	88.7+	218.4+	3600.2*	138.8+	86.0+
neos-4722843-widden	106.68	244.24	111.03	156.42	157.50	126.47	784.5+	2140.2+	758.6+	3603.4*	559.8+	1078.9+
neos-4738912-atrato	4.86	7.87	5.96	7.90	9.98	5.19	10.4+	7.2+	17.3+	14.1+	16.4+	11.1+
neos-4763324-toguru	106.84	82.94	81.80	105.72	96.26	78.51	364.8+	490.4+	461.4+	1318.5+	727.4+	550.4+
neos-4954672-berkel	88.52	116.81	57.03	83.35	73.25	90.44	12.7+	11.1+	13.9+	16.0+	13.4+	12.0+
neos-5049753-cuanza	-	11.44	8.27	-	11.75	6.70	3600.3*	135.2+	134.9+	1843.8+	2045.9+	247.4+
neos-5052403-cygnet	1.39	2.48	2.51	61.56	1.95	2.49	409.0+	153.0+	153.9+	3600.4*	1435.6+	215.8+
neos-5093327-huahum	-	75.37	-	-	-	-	121.5+	22.4+	30.2+	68.7+	35.6+	27.3+
neos-5104907-jarama	-	-	-	-	-	-	3600.0*	3600.1*	3599.9+	3608.0*	3600.0*	3599.3+
neos-5107597-kakapo	-	-	-	13.96k	-	-	479.3+	13.0+	12.7+	23.9+	3.4+	7.0+
neos-5114902-kasavu	-	-	-	-	-	-	3600.2*	527.4+	351.6+	3603.1*	3600.0*	885.7+
neos-5188808-nattai	-	-	-	-	-	-	74.3+	13.3+	11.5+	27.2+	17.0+	10.1+
neos-5195221-niemur	-	-	-	-	-	-	103.0+	43.0+	43.6+	164.2+	57.3+	63.4+

Continued on next page

Table B.3.: Root gap comparison (MIPLIB 2017 benchmark)

	gap						time					
LP solver	Clp	CPLEX	Gurobi	MOSEK	SoPlex	Xpress	Clp	CPLEX	Gurobi	MOSEK	SoPlex	Xpress
instance	12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09	12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09
neos-631710	14.21	14.21	14.21	14.21	14.21	14.21	3600.0*	3600.0*	2013.6+	3600.1*	3600.0*	3148.2+
neos-662469	575.59	559.29	570.15	591.86	564.72	581.03	120.4+	20.2+	22.2+	62.6+	38.3+	26.4+
neos-787933	0.00	0.00	0.00	0.00	0.00	0.00	1.6	1.8	1.8	1.6	2.0	2.0
neos-827175	0.00	0.00	0.00	0.00	0.00	0.00	31.3	1.9	2.6	5.2	28.5	1.6
neos-848589	2.09	2.11	2.09	2.09	2.73	2.12	407.1+	882.1+	392.9+	565.2+	956.6+	413.1+
neos-860300	2.18	2.55	12.26	12.04	2.31	11.92	60.5+	16.2+	21.0+	27.8+	19.3+	22.8+
neos-873061	-	4.07	3.63	4.73	7.57	3.34	-	362.5+	345.6+	2396.9+	3600.0*	1053.6+
neos-911970	15.73	10.13	11.48	11.19	9.61	12.10	4.7+	2.5+	2.5+	5.8+	2.5+	2.9+
neos-933966	1.28k	2.20	15.72	961.01	1.28k	13.52	186.4+	42.9+	63.3+	255.6+	207.4+	34.9+
neos-950242	-	-	-	-	-	-	84.3+	86.0+	84.1+	202.2+	59.4+	61.5+
neos-957323	0.00	0.00	0.00	0.00	0.00	0.00	62.8+	29.6	34.8+	86.1+	78.5+	31.2+
neos-960392	-	2.15	1.71	0.85	-	0.00	944.6+	40.0+	19.6+	99.4+	452.9+	14.2
neos17	11.03k	415.01	416.30	421.88	415.01	415.01	0.3+	0.4+	0.4+	0.6+	0.5+	0.4+
neos5	15.28	14.91	26.55	15.41	19.07	18.88	1.3+	2.2+	1.6+	1.3+	2.1+	1.8+
neos8	0.00	0.00	0.00	0.00	0.00	0.00	2.3	2.4	2.4	2.4	2.5	2.5
neos859080	-	-	-	-	-	-	0.1+	0.0+	0.1+	0.1+	0.1+	0.1+
net12	317.96	266.19	274.12	298.73	292.77	296.07	90.0+	32.3+	18.7+	84.1+	17.8+	15.2+
netdiversion	>999k	8.28	>999k	>999k	>999k	>999k	486.3+	320.0+	191.3+	1901.3+	298.7+	398.5+
nexp-150-20-8-5	6.64	9.75	9.56	9.29	3.51	8.28	481.5+	275.5+	240.9+	297.3+	408.1+	265.9+
ns1116954	-	-	-	-	-	-	3600.1*	636.2+	416.9+	1236.3+	2936.8+	226.0+
ns1208400	-	-	-	-	-	-	6.8+	24.6+	4.8+	41.7+	17.0+	3.8+
ns1644855	0.00	0.00	0.00	4.69	7.37	7.37	267.0	528.6	572.7	3603.5*	2211.5+	362.6+
ns1760995	0.00	0.00	0.00	4.69	7.37	7.37	3610.6*	3611.6*	3611.7*	3600.9*	3600.8*	3603.2*
ns1830653	-	-	-	-	-	-	73.4+	9.0+	6.7+	25.2+	9.4+	7.2+
ns1952667	-	-	-	-	-	-	4.4+	3.8+	5.3+	5.3+	5.1+	4.6+
nu25-pr12	0.27	0.28	0.17	0.17	0.36	0.10	2.6+	2.3+	2.4+	4.0+	1.8+	2.9+
nursesched-medium-hint03	10.90k	10.46k	10.69k	10.65k	10.55k	10.54k	506.4+	286.6+	360.1+	1700.1+	896.4+	393.7+
nursesched-sprint02	303.31	3.80	7.51	0.00	12.07	303.43	20.3+	10.9+	12.6+	53.0	26.1+	16.5+
nw04	3.34	11.66	4.11	4.11	3.35	4.11	22.6+	19.2+	22.2+	15.5+	21.3+	25.0+
opm2-z10-s4	59.04	57.59	59.25	60.14	59.62	59.03	751.0+	675.2+	536.7+	1807.7+	579.0+	740.1+
p200x1188c	10.72	0.00	0.00	6.00	12.67	7.55	4.6+	3.8	4.6	4.3+	3.6+	3.7+
peg-solitaire-a3	-	-	-	-	-	-	88.1+	74.2+	40.5+	137.2+	113.9+	30.1+
pg	-	0.30	0.37	0.38	0.30	0.31	-	3.6+	3.4+	5.7+	6.1+	3.7+
pg5_34	-	0.55	0.55	0.55	0.55	0.55	-	2.6+	2.0+	3.5+	2.9+	2.0+

Continued on next page

Table B.3.: Root gap comparison (MIPLIB 2017 benchmark)

	gap						time					
LP solver	Clp	CPLEX	Gurobi	MOSEK	SoPlex	Xpress	Clp	CPLEX	Gurobi	MOSEK	SoPlex	Xpress
instance	12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09	12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09
physiciansched3-3	-	-	-	-	-	-	3600.2*	329.7+	220.8+	978.5+	711.2+	365.7+
physiciansched6-2	-	-	-	-	-	-	943.5+	44.2+	33.3+	119.2+	71.6+	44.8+
piperout-08	-	285.14	-	-	337.46	-	60.1+	70.6+	66.1+	77.2+	69.0+	58.0+
piperout-27	35.97	35.97	35.97	35.97	34.93	35.97	122.8+	126.0+	109.9+	142.6+	142.0+	120.4+
pk1	-	-	-	-	-	-	0.3+	0.3+	0.2+	0.4+	0.4+	0.4+
proteindesign121hz512p9	-	-	-	-	-	-	3229.0+	2328.3+	2239.0+	2383.0+	2066.2+	2662.7+
proteindesign122trx11p8	2.17	2.15	2.08	2.11	2.08	1.95	1913.6+	1524.6+	1646.3+	2110.8+	1897.0+	2658.9+
qap10	30.89	0.00	31.10	31.10	21.13	0.00	90.5+	56.6	68.1+	103.7+	107.7+	66.0
radiationm18-12-05	318.42	344.35	320.29	353.60	107.42	192.58	15.6+	9.1+	8.5+	33.6+	14.4+	9.7+
radiationm40-10-02	227.83	239.17	230.92	234.02	216.49	228.86	102.0+	56.0+	68.6+	215.8+	39.6+	98.1+
rail01	-	27.73	-	-	-	28.49	-	1853.5+	1556.1+	3287.6+	3600.0*	1138.8+
rail02	-	44.36	-	-	-	-	3600.0*	3252.4+	3595.0+	3615.2*	3600.0*	3594.9+
rail507	3.35	2.19	2.19	2.19	2.77	3.33	23.3+	12.0+	15.3+	31.1+	21.9+	26.0+
ran14x18-disj-8	12.22	18.62	12.40	16.73	10.18	14.52	13.5+	8.3+	8.9+	12.0+	9.0+	9.8+
rd-rplusc-21	-	-	-	-	-	-	123.8+	56.3+	63.6+	75.3+	47.6+	44.3+
reblock115	21.55	21.02	21.30	21.57	21.40	21.30	39.5+	43.2+	37.1+	76.8+	39.2+	37.8+
rmatr100-p10	76.85	16.48	15.17	37.17	38.43	26.77	9.3+	5.0+	19.0+	11.2+	9.3+	5.9+
rmatr200-p5	37.75	38.69	38.73	38.90	38.44	40.16	1051.6+	303.4+	418.6+	1103.8+	1620.7+	449.4+
rocI-4-11	-	-	-	-	-	-	6.9+	3.3+	3.2+	8.6+	5.0+	4.8+
rocII-5-11	-	-	-	-	-	-	47.0+	26.7+	24.8+	49.6+	32.1+	32.2+
rococoB10-011000	115.08	135.89	132.70	122.90	117.63	109.58	51.0+	11.6+	12.0+	31.3+	20.6+	14.8+
rococoC10-001000	24.58	15.07	19.85	21.09	18.20	39.52	20.5+	7.6+	6.4+	14.0+	8.1+	7.9+
roi2alpha3n4	47.97	48.10	48.05	43.00	45.38	47.27	100.7+	74.2+	79.4+	82.2+	81.3+	70.6+
roi5alpha10n8	142.78	127.63	135.43	110.85	128.00	146.46	296.4+	262.6+	288.5+	555.6+	303.0+	318.9+
roll3000	-	-	5.99	8.77	-	10.51	32.6+	3.8+	5.4+	10.1+	5.6+	5.7+
s100	-	24.12	-	14.55	-	-	3641.4*	1367.2+	3610.1*	2211.5+	3600.1*	1373.5+
s250r10	-	1.14	2.26	-	0.52	1.68	4764.4*	245.3+	737.6+	285.0+	747.7+	426.2+
satellites2-40	-	-	-	-	-	-	3600.0*	979.1+	401.3+	1012.1+	3600.0*	397.6+
satellites2-60-fs	-	314.29	-	-	-	-	3600.0*	580.3+	276.2+	2515.5+	3600.0*	446.7+
savsched1	-	13.46k	3.60k	12.89k	-	306.74	3600.0*	3599.9+	3599.9+	3602.6*	3600.0*	3095.0+
sct2	3.83	6.19	0.13	1.74	0.80	2.14	1336.9+	2.2+	2.6+	10.6+	2.5+	2.5+
seymour	9.19	9.33	9.40	9.46	9.18	9.23	22.3+	14.2+	17.0+	34.7+	13.7+	19.2+
seymour1	7.55	7.69	7.71	6.75	6.65	7.20	10.3+	6.6+	6.4+	18.8+	8.8+	7.8+
sing326	7.94	1.10	2.76	0.73	0.97	2.85	282.2+	110.8+	104.0+	317.5+	371.0+	125.5+

Continued on next page

Table B.3.: Root gap comparison (MIPLIB 2017 benchmark)

	gap						time					
LP solver	Clp	CPLEX	Gurobi	MOSEK	SoPlex	Xpress	Clp	CPLEX	Gurobi	MOSEK	SoPlex	Xpress
instance	12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09	12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09
sing44	0.39	1.60	0.55	1.89	2.82	0.48	190.7+	112.0+	110.1+	269.2+	270.8+	146.0+
snp-02-004-104	27.11	21.37	25.69	21.81	31.80	25.60	3705.0*	643.3+	1189.1+	129.4+	523.7+	237.1+
sorrell3	31.29	30.87	31.33	85.62	22.74	31.32	1884.8+	3059.3+	1013.7+	3600.0*	1133.6+	2331.6+
sp150x300d	9.52	10.12	9.52	9.52	9.52	9.54	0.7+	1.3+	0.7+	0.4+	0.2+	1.5+
sp97ar	8.22	7.74	12.13	6.52	7.59	7.07	64.0+	53.7+	64.1+	86.0+	60.6+	65.6+
sp98ar	5.93	7.06	4.91	8.01	7.63	7.50	63.1+	61.8+	74.7+	100.0+	62.6+	58.9+
splice1k1	1.26k	1.26k	1.26k	1.26k	1.26k	1.26k	2336.5+	1943.7+	1781.7+	2048.0+	1782.3+	1781.6+
square41	193.23	192.46	193.95	194.13	193.71	192.63	447.5+	763.2+	821.1+	1708.4+	848.5+	700.7+
square47	230.52	229.38	231.11	231.21	230.70	230.00	839.6+	970.0+	1111.9+	3630.6*	1350.3+	1133.0+
supportcase10	448.30	440.18	424.83	434.27	1.24k	428.48	3600.0*	3599.8+	3599.8+	3606.6*	3600.0*	3598.9+
supportcase12	2.18	2.03	1.83	2.86	1.64	2.62	155.5+	47.8+	31.2+	50.5+	216.8+	26.2+
supportcase18	16.56	14.44	12.32	35.63	31.39	31.39	18.6+	5.5+	7.0+	10.8+	26.6+	5.1+
supportcase19	-	-	-	-	-	-	3600.0*	3593.2+	3590.5+	3601.6*	3600.1*	3591.8+
supportcase22	-	-	-	-	-	-	1898.4+	3600.0*	3600.0*	3614.1*	80.6+	3256.1+
supportcase26	40.48	36.19	40.93	46.71	41.45	43.54	9.2+	8.0+	4.9+	7.1+	4.8+	5.5+
supportcase33	7.07k	2.77k	2.75k	3.00k	2.76k	2.84k	3600.1*	126.9+	139.2+	172.1+	138.6+	137.2+
supportcase40	21.54	23.46	21.34	21.48	20.69	21.47	195.6+	7.9+	6.4+	19.4+	13.6+	10.8+
supportcase42	8.65	7.70	10.58	8.23	7.70	8.23	41.8+	20.5+	4.4+	37.9+	23.1+	8.9+
supportcase6	300.80	14.22	302.41	15.07	14.75	300.41	363.4+	333.8+	367.0+	418.0+	399.3+	396.7+
supportcase7	-	5.50	3.96	-	6.43	6.11	346.6+	149.5+	135.6+	198.7+	118.4+	146.7+
swath1	-	24.39	31.10	21.63	-	-	-	3.5+	4.1+	6.4+	4.5+	4.7+
swath3	-	43.21	64.00	53.80	68.97	82.84	-	4.1+	4.7+	5.6+	4.7+	5.5+
tbfp-network	462.93	170.04	464.27	465.16	465.16	463.86	154.1+	170.1+	134.7+	157.2+	254.4+	144.5+
thor50dday	36.99	37.12	28.07	-	29.98	39.56	3600.1*	1405.6+	1516.1+	3600.6*	3600.0*	2743.8+
timtab1	-	-	-	-	-	-	1.8+	1.6+	1.2+	2.5+	1.4+	2.2+
tr12-30	1.93	0.56	0.84	1.40	1.06	0.73	6.4+	2.5+	2.6+	3.6+	2.9+	3.4+
traininstance2	-	-	-	-	-	-	11.1+	10.1+	12.0+	21.8+	12.2+	17.6+
traininstance6	-	-	-	-	-	-	11.4+	5.1+	9.9+	10.1+	6.6+	4.9+
trento1	401.03	387.13	580.68	582.65	394.85	387.10	52.1+	16.5+	17.2+	32.6+	36.8+	23.4+
triptim1	0.00	0.00	0.00	0.00	0.00	0.00	192.0+	157.5	203.0+	614.8	645.6+	124.5
uccase12	0.00	0.00	0.00	0.00	0.00	0.00	76.8+	29.4+	38.5+	212.2+	61.0+	63.4+
uccase9	217.28	7.67	7.02	4.37	16.41	12.92	376.0+	235.8+	499.9+	518.8+	304.2+	213.9+
uct-subprob	40.04	36.16	40.79	39.93	39.70	39.05	11.1+	6.7+	12.6+	21.7+	10.2+	9.2+
unitcal_7	0.06	0.10	0.07	0.08	0.10	0.07	3600.1*	84.4+	68.2+	145.0+	157.1+	52.4+

Continued on next page

Table B.3.: Root gap comparison (MIPLIB 2017 benchmark)

	gap						time					
LP solver	Clp	CPLEX	Gurobi	MOSEK	SoPlex	Xpress	Clp	CPLEX	Gurobi	MOSEK	SoPlex	Xpress
instance	12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09	12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09
var-smallemery-m6j6	6.58	6.83	6.83	8.44	6.83	6.83	49.2+	26.6+	24.2+	44.5+	35.7+	34.3+
wachplan	-	12.50	-	-	12.50	-	10.0+	4.0+	4.6+	12.6+	6.2+	4.5+

Table B.4.: κ_{LP} and κ statistics for various MILPs

		attention level $\sqrt{\alpha}$		max log κ (tree condition number)	
instance	log κ_{LP}	CPLEX 12.10.0.0	FICO Xpress v8.8.3	CPLEX 12.10.0.0	FICO Xpress v8.8.3
10teams	∞	0.0	0.0	5.2	4.98
22433	9.42	0.0	0.0	5.4	4.31
23588	8.68	0.0	0.0	4.46	4.58
30n20b8	9.04	0.0	0.0	6.11	6.78
50v-10	∞	0.0196	0.0054	9.87	8.15
Test3	∞	0.0	0.0	6.26	5.54
a1c1s1	7.03	0.0088	0.0153	9.88	9.83
acc-tight4	∞	0.0145	0.0	8.95	5.29
acc-tight5	∞	0.0093	0.0	7.08	6.08
acc-tight6	∞	0.0096	0.0	7.57	5.78
aflow30a	5.97	0.0024	0.0	8.71	6.34
aflow40b	6.27	0.002	0.0	8.89	7.12
air03	7.74	0.0	0.0	2.93	2.85
air04	∞	0.0	0.0	6.11	5.44
air05	∞	0.0	0.0	5.15	4.87
aligninq	8.73	0.0026	0.0	8.1	4.82
ash608gpia-3col	4.74	0.0936	0.0	12.3	6.16
b2c1s1	7.06	0.0314	0.0306	11.9	8.91
bab1	∞	0.0	0.0	6.97	6.41
bab5	7.73	0.0	0.0022	7.4	7.45
bc	5.04	0.0236	0.0	11.87	6.98
bc1	5.04	0.0058	0.0	8.48	6.79
beasleyC3	∞	0.0063	0.0447	7.81	7.31
bell3a	8.86	0.0	0.0	5.28	3.91
bell5	10.12	0.0	0.0	4.39	4.38
berlin_5_8_0	∞	0.0	0.0017	10.59	9.25
bg512142	∞	0.0172	0.0399	10.07	9.7
bienst1	∞	0.001	0.008	7.15	7.56
bienst2	∞	0.0	0.001	7.53	7.31
binkar10_1	8.26	0.0	0.0	4.66	6.66
blend2	∞	0.002	0.0	7.2	4.48
blp-ar98	∞	0.0	0.0	6.84	6.69
blp-ic97	∞	0.0	0.0	6.53	7.09
bnatt350	∞	0.0	0.002	6.71	7.74
bnatt400	∞	0.0	0.0022	7.24	7.24
co-100	∞	0.0091	0.0022	8.57	7.28
cov1075	3.5	0.0	0.0	5.82	6.89
csched007	∞	0.0022	0.0	8.48	6.5
csched008	∞	0.012	0.001	12.33	7.87
csched010	∞	0.0017	0.0	10.21	6.07
d10200	∞	0.0	0.0	7.08	7.3
d20200	6.39	0.0	0.0022	6.58	7.57
dano3_3	∞	0.0237	0.1	7.81	7.57
dano3_4	∞	0.056	0.1	8.61	8.0
dano3_5	∞	0.0536	0.1	8.24	8.67
dano3mip	∞	0.0603	0.0987	8.8	8.92
danoint	∞	0.0054	0.0014	10.08	7.73
dcmulti	6.88	0.0	0.0	5.9	4.14
dfn-gwin-UUM	∞	0.0042	0.0014	7.59	7.54
disctom	∞	0.0	0.0	3.65	3.63
egout	∞	0.0	-	2.87	-
eil33-2	6.09	0.0	0.0	4.69	4.77
eilA101-2	7.04	0.0	0.0	6.25	5.12
eilB101	5.58	0.0	0.0	5.76	5.48

Continued on next page

Table B.4.: κ_{LP} and MILP κ statistics for various MILPs

		attention level $\sqrt{\alpha}$		max log κ (tree condition number)	
instance	log κ_{LP}	CPLEX 12.10.0.0	FICO Xpress v8.8.3	CPLEX 12.10.0.0	FICO Xpress v8.8.3
enigma	∞	0.003	0.0	7.34	5.1
enlight13	2.49	0.0	-	0.3	-
enlight14	2.55	0.0	-	0.3	-
enlight15	2.62	0.0	-	0.3	-
enlight16	2.67	0.0	-	0.3	-
enlight9	2.15	0.0	-	0.3	-
ex9	∞	0.0	0.0	5.84	0.0
f2000	4.65	0.0	0.0	5.09	6.48
fast0507	∞	0.0	0.0	4.52	5.18
fiball	∞	0.0	0.0	6.47	4.65
fiber	∞	0.0	-	3.8	-
fixnet6	∞	0.0	0.0	6.58	6.21
flugpl	6.89	0.0	0.0	2.2	0.0
g200x740i	∞	0.0064	0.0089	10.14	7.96
gen	8.77	0.0	0.0	3.18	3.38
germany50-DBM	∞	0.0	0.0	6.91	7.01
gesa2	10.11	0.0	-	4.2	-
gesa2-o	10.11	0.0	-	4.44	-
gesa3	10.03	0.0	0.0	5.27	4.98
gesa3_o	10.03	0.0	0.0	4.52	5.2
gmu-35-40	7.88	0.001	0.002	11.33	8.73
gmu-35-50	8.17	0.0125	0.0	12.2	7.9
go19	∞	0.0	0.0	6.82	6.28
gt2	5.95	0.0	0.0	3.6	3.1
hanoi5	∞	0.001	0.0118	7.33	9.99
haprp	∞	0.0	-	2.8	-
ic97_potential	∞	0.001	0.0	10.34	7.13
iis-100-0-cov	3.96	0.0	0.0	5.86	6.27
iis-bupa-cov	4.11	0.0	0.0	6.87	6.33
iis-pima-cov	4.31	0.0	0.0	6.84	6.35
janos-us-DDM	∞	0.0	0.0	7.27	7.0
k16x240	∞	0.0049	0.001	9.18	8.75
khb05250	∞	0.0266	0.0	7.24	6.72
l152lav	5.88	0.0	0.0	5.64	4.27
lectsched-1	∞	0.0	0.0	4.32	6.7
lectsched-1-obj	∞	0.0	0.0085	6.52	8.53
lectsched-2	∞	0.0	0.0	3.71	5.21
lectsched-3	∞	0.0	0.0	3.72	5.84
lectsched-4-obj	∞	0.0	0.0	3.46	5.8
liu	7.27	0.0	0.0077	7.58	9.81
lotsize	9.7	0.1385	0.103	14.97	11.99
lrsa120	∞	0.0216	0.0017	12.19	7.62
lseu	5.01	0.0	0.0	3.22	4.88
m100n500k4r1	3.3	0.0014	0.0	8.68	5.26
macrophage	3.64	0.0	0.0	4.5	5.13
manna81	4.5	0.0	0.0	1.21	4.19
map06	∞	0.0039	0.0048	7.42	7.08
map10	∞	0.0	0.004	6.7	7.42
map14	∞	0.0	0.0045	6.56	7.3
map18	∞	0.0	0.0	6.23	6.49
map20	∞	0.0	0.0	6.13	6.04
markshare1	4.16	0.0	0.0	8.08	6.45
markshare2	4.32	0.0	0.0	8.32	6.15
markshare_5_0	4.03	0.0	0.0	5.12	6.42

Continued on next page

Table B.4.: κ_{LP} and MILP κ statistics for various MILPs

		attention level $\sqrt{\alpha}$		max log κ (tree condition number)	
instance	log κ_{LP}	CPLEX 12.10.0.0	FICO Xpress v8.8.3	CPLEX 12.10.0.0	FICO Xpress v8.8.3
mas74	∞	0.0	0.0	5.41	4.54
mas76	∞	0.0	0.0	4.06	4.35
maxgasflow	8.98	0.1608	0.4991	13.23	12.96
mc11	∞	0.0112	0.0269	9.84	8.66
mcsched	∞	0.0	0.0	5.75	5.17
methanosarcina	4.4	0.0	0.0	5.64	5.85
mik-250-1-100-1	4.44	0.0	0.0	4.99	5.08
misc03	∞	0.0	0.0	3.12	4.26
misc06	∞	0.0	0.0	3.61	4.27
misc07	∞	0.0	0.0	3.76	5.14
mitre	8.76	0.0	0.0	4.16	3.8
mkc	5.13	0.0026	0.001	8.85	7.7
mkc1	5.13	0.0	0.0	5.78	6.83
mod008	2.5	0.0	0.0	3.77	3.77
mod010	∞	0.0	0.0	3.83	3.38
modglob	∞	0.0	0.0	6.82	5.65
momentum1	∞	0.1184	0.1181	14.47	12.77
mspp16	∞	0.0	0.0	4.36	3.46
mzzv11	∞	0.0107	-	7.19	-
mzzv42z	∞	0.0	0.0	6.05	6.27
n15-3	∞	0.0803	0.0817	9.18	7.48
n3700	∞	0.0901	0.0826	10.42	9.33
n3705	∞	0.0877	0.0844	10.06	9.81
n370a	∞	0.0876	0.0888	10.32	9.26
n3div36	8.81	0.0	0.0	5.28	6.23
n3seq24	∞	0.0	0.0	5.51	6.0
n4-3	∞	0.0041	0.0055	7.66	8.0
n9-3	∞	0.0036	0.0098	7.6	7.77
neos-1053234	∞	0.0	0.0995	6.7	9.22
neos-1056905	5.23	0.0	0.0022	5.8	9.31
neos-1061020	∞	0.0323	0.0175	9.44	8.03
neos-1067731	∞	0.0	0.0	7.56	6.91
neos-1109824	4.98	0.0	0.0	3.63	5.34
neos-1120495	4.86	0.0	0.0	3.57	3.66
neos-1121679	4.16	0.0	0.0	7.47	6.45
neos-1151496	4.57	0.0	0.0	3.65	4.71
neos-1171448	6.49	0.0	0.0	4.36	5.11
neos-1171692	6.42	0.0	0.0	4.41	5.04
neos-1171737	6.28	0.0039	0.0	7.39	7.16
neos-1173026	∞	0.0014	0.0	7.15	5.16
neos-1200887	5.87	0.0	0.0	5.15	5.67
neos-1208069	4.62	0.0	0.0	5.72	6.01
neos-1208135	4.56	0.0	0.0	4.95	5.27
neos-1211578	5.79	0.0	0.0	4.6	4.35
neos-1215259	4.86	0.0095	0.0	13.08	5.59
neos-1215891	∞	0.0	0.0	4.25	4.98
neos-1228986	5.96	0.0	0.0	4.32	4.19
neos-1281048	5.86	0.0	0.0	6.53	5.63
neos-1311124	6.33	0.0	0.0	4.62	4.97
neos-1324574	∞	0.0	0.0	5.4	5.44
neos-1330346	∞	0.0	0.0	5.96	5.74
neos-1330635	∞	0.0	-	1.62	-
neos-1337307	7.73	0.0	0.001	6.5	7.44
neos-1346382	6.31	0.0	0.0	7.25	7.17

Continued on next page

Table B.4.: κ_{LP} and MILP κ statistics for various MILPs

		attention level $\sqrt{\alpha}$		max log κ (tree condition number)	
instance	log κ_{LP}	CPLEX 12.10.0.0	FICO Xpress v8.8.3	CPLEX 12.10.0.0	FICO Xpress v8.8.3
neos-1396125	∞	0.002	0.0014	7.67	7.38
neos-1407044	∞	0.0263	0.0	11.95	6.51
neos-1413153	7.51	0.0	0.0	4.24	5.06
neos-1415183	7.64	0.0	0.0	4.16	4.47
neos-1417043	∞	0.0	0.0	4.16	4.32
neos-1420790	4.37	0.0	0.0	8.87	7.27
neos-1423785	7.65	0.0	0.001	5.92	7.17
neos-1425699	12.35	0.0	0.0	1.86	1.77
neos-1426662	5.62	0.0	0.0	5.26	7.85
neos-1427181	5.92	0.0	0.0	4.5	5.94
neos-1427261	6.01	0.0	0.0	5.76	6.64
neos-1429185	5.79	0.0	0.0	5.11	6.55
neos-1429212	∞	0.0065	0.011	9.46	7.44
neos-1429461	5.89	0.0	0.0	5.21	5.74
neos-1430701	5.79	0.0	0.0	4.48	5.82
neos-1436709	6.0	0.0	0.0	5.58	6.7
neos-1436713	6.09	0.0	0.0	5.62	5.79
neos-1437164	6.87	0.0	-	3.44	-
neos-1439395	6.16	0.0	0.0	6.91	6.19
neos-1440225	∞	0.0151	0.0	9.49	6.1
neos-1440447	5.89	0.0	0.0	4.4	4.48
neos-1440457	6.14	0.0	0.0	5.6	6.68
neos-1440460	6.14	0.0	0.001	4.54	7.65
neos-1441553	6.63	0.0	-	4.03	-
neos-1442119	6.16	0.0	0.0	5.21	7.31
neos-1442657	6.09	0.0	0.0	4.98	7.56
neos-1445532	4.79	0.0	0.0	3.67	4.09
neos-1445738	4.78	0.0	0.0	5.88	5.34
neos-1445743	4.79	0.0	0.0	6.17	5.64
neos-1445755	4.78	0.0	0.0	5.79	5.44
neos-1445765	4.79	0.0	0.0	5.79	5.57
neos-1451294	∞	0.0014	0.0014	7.7	7.59
neos-1456979	∞	0.0	0.0	6.94	6.75
neos-1460246	5.33	0.0	0.0	6.63	6.17
neos-1460265	5.89	0.0	0.0	4.15	3.77
neos-1460543	5.9	0.001	0.0	7.54	6.79
neos-1460641	6.5	0.001	0.0	7.54	6.72
neos-1461051	∞	0.0	0.0	3.6	4.0
neos-1464762	6.48	0.0	0.0	3.6	6.72
neos-1467067	3.67	0.0	0.0	3.96	4.78
neos-1467371	6.47	0.0	0.0	3.96	6.68
neos-1467467	6.47	0.0	0.0	6.17	6.72
neos-1480121	5.38	0.0035	0.0037	7.79	7.47
neos-1489999	3.78	0.0	0.0	2.61	4.18
neos-1516309	7.23	0.0	0.0	2.96	3.16
neos-1582420	5.03	0.0	0.0	6.26	6.31
neos-1593097	8.46	0.0	0.0	4.42	4.95
neos-1595230	3.77	0.0	0.0	5.35	5.5
neos-1599274	7.22	0.0	0.0	2.56	3.39
neos-1601936	∞	0.0	0.0	5.53	6.68
neos-1603512	∞	0.0	0.0	3.7	5.34
neos-1603518	∞	0.0	0.0	5.59	5.21
neos-1605061	∞	0.0	0.0056	6.18	8.17
neos-1605075	5.3	0.0	0.0	6.24	5.47

Continued on next page

Table B.4.: κ_{LP} and MILP κ statistics for various MILPs

		attention level $\sqrt{\alpha}$		max log κ (tree condition number)	
instance	log κ_{LP}	CPLEX 12.10.0.0	FICO Xpress v8.8.3	CPLEX 12.10.0.0	FICO Xpress v8.8.3
neos-1616732	3.9	0.0	0.0	5.38	5.76
neos-1620770	4.56	0.0	0.0	4.85	6.29
neos-1620807	3.68	0.0	0.0	3.73	4.17
neos-1622252	4.57	0.0	0.001	6.14	7.04
neos-430149	∞	0.0033	0.0085	7.25	8.77
neos-480878	7.65	0.0054	0.0182	7.84	8.02
neos-494568	5.63	0.0	0.0	2.6	3.17
neos-495307	1.81	0.0	0.0	4.1	4.24
neos-498623	∞	0.0	0.0	2.84	4.56
neos-501453	6.93	0.0	0.0	3.92	1.94
neos-501474	6.93	0.0	0.0	4.29	4.21
neos-503737	∞	0.0	0.0	4.4	4.61
neos-504674	5.18	0.0	0.0	6.89	6.89
neos-504815	4.94	0.001	0.0	7.25	5.78
neos-506422	5.47	0.0	0.0	4.19	5.78
neos-512201	5.18	0.0	0.0	5.97	5.88
neos-522351	10.86	0.0902	0.0	8.11	6.23
neos-525149	∞	0.0	0.0	2.8	3.75
neos-538867	∞	0.0	0.0	4.92	5.56
neos-538916	4.27	0.0	0.0	4.18	5.08
neos-544324	6.25	0.0	0.0	4.53	5.15
neos-547911	5.79	0.0	0.0	4.84	5.24
neos-548047	4.73	0.0	0.0	7.38	5.93
neos-548251	4.79	0.0099	0.0028	9.63	9.27
neos-551991	5.26	0.001	0.0	7.52	5.68
neos-555001	∞	0.0	0.0	3.42	4.06
neos-555298	6.46	0.0	0.0	4.54	4.67
neos-555343	∞	0.0	0.0	4.55	5.42
neos-555424	∞	0.0	0.0	4.12	6.32
neos-555694	∞	0.0	0.0	3.2	3.71
neos-555771	∞	0.0	0.0	2.94	3.36
neos-555884	∞	0.0	0.0	6.49	7.35
neos-555927	∞	0.0	0.0	4.07	5.31
neos-565815	∞	0.0	0.0	5.24	5.76
neos-570431	4.79	0.0	0.0	6.38	5.11
neos-582605	∞	0.0	0.0	6.73	6.69
neos-583731	∞	0.0	0.0	1.32	4.28
neos-584146	∞	0.0	0.0	6.13	6.57
neos-584851	2.96	0.0	0.0	4.71	4.56
neos-584866	3.84	0.001	0.0	8.43	6.27
neos-585192	∞	0.0156	0.0391	9.33	9.71
neos-585467	∞	0.0216	0.016	9.62	7.97
neos-595904	7.18	0.004	0.0	8.66	2.26
neos-595905	6.61	0.0	-	6.26	-
neos-595925	6.93	0.0024	0.0	7.91	6.65
neos-598183	6.39	0.0026	0.0	7.39	5.24
neos-603073	6.36	0.0014	0.003	7.66	7.42
neos-611135	7.7	0.001	0.0014	9.23	7.62
neos-611838	6.55	0.0	0.0	6.1	6.71
neos-612125	6.53	0.0	0.0	5.57	6.61
neos-612143	6.54	0.0	0.0	5.79	6.47
neos-612162	6.55	0.0	0.0	6.36	6.76
neos-619167	∞	0.1314	1.0	15.68	19.54
neos-631164	11.96	0.0	0.0	11.28	5.37

Continued on next page

Table B.4.: κ_{LP} and MILP κ statistics for various MILPs

		attention level $\sqrt{\alpha}$		max log κ (tree condition number)	
instance	log κ_{LP}	CPLEX 12.10.0.0	FICO Xpress v8.8.3	CPLEX 12.10.0.0	FICO Xpress v8.8.3
neos-631517	11.92	0.0095	0.001	10.91	7.23
neos-631694	5.63	0.0	0.0	5.2	3.97
neos-632335	∞	0.0	0.0	2.0	4.4
neos-633273	∞	0.0	0.0	1.76	4.27
neos-655508	10.16	0.0	-	0.3	-
neos-662469	7.11	0.0	0.0	5.57	5.73
neos-686190	∞	0.0039	0.0037	8.18	7.3
neos-691058	∞	0.0	0.0	3.44	4.46
neos-691073	∞	0.0	0.0	3.52	4.7
neos-693347	∞	0.0	0.0045	6.3	8.16
neos-709469	∞	0.0045	0.0	7.1	4.25
neos-717614	∞	0.0032	0.0	7.25	6.43
neos-738098	∞	0.0	0.0	5.76	5.39
neos-775946	∞	0.0	0.0	3.48	4.37
neos-777800	∞	0.0	0.0	6.26	3.98
neos-780889	∞	0.0	0.0	5.62	5.22
neos-785899	∞	0.0	0.0	4.36	4.25
neos-785912	∞	0.0017	0.0	8.02	5.0
neos-785914	∞	0.0	-	3.88	-
neos-787933	4.62	0.0	0.0	1.93	3.14
neos-791021	∞	0.0	0.0	4.28	4.67
neos-796608	6.91	0.0	0.0	3.01	2.84
neos-799838	∞	0.0	0.0	5.94	6.18
neos-801834	7.04	0.0	0.0	5.36	5.87
neos-803219	5.72	0.0	0.0	6.52	5.97
neos-803220	5.74	0.001	0.0	7.0	6.16
neos-806323	7.19	0.0072	0.0068	7.42	7.64
neos-807454	∞	0.0	0.0	3.84	4.62
neos-807456	∞	0.0832	0.0	13.64	6.2
neos-807639	8.57	0.0037	0.0	7.95	5.99
neos-807705	8.23	0.0024	0.0048	7.07	7.59
neos-808072	∞	0.0	0.0	5.33	6.21
neos-808214	∞	0.0	0.0	4.31	5.81
neos-810286	∞	0.0	0.0	4.7	5.19
neos-810326	∞	0.0	0.0	6.25	5.12
neos-820146	3.42	0.0	0.0	5.11	5.12
neos-820157	∞	0.0	0.0	5.98	5.15
neos-820879	∞	0.0	0.0	6.0	5.35
neos-825075	∞	0.0	0.0	3.56	4.31
neos-826250	6.03	0.0	0.0	5.08	4.8
neos-826650	∞	0.0	0.0	6.92	6.89
neos-826694	∞	0.0	0.0	5.12	4.88
neos-826841	6.56	0.0	0.0	5.3	5.49
neos-830439	5.46	0.0	0.0	3.31	3.3
neos-831188	4.15	0.005	0.0	7.54	5.37
neos-841664	7.14	0.0751	0.1	8.5	8.68
neos-847302	∞	0.001	0.0	7.36	6.46
neos-848150	∞	0.0	0.0	4.34	5.09
neos-848198	5.96	0.0461	0.0848	9.12	9.07
neos-848845	∞	0.0017	0.0	7.37	6.83
neos-849702	∞	0.0075	0.0	10.13	5.9
neos-850681	7.18	0.0	0.0	3.69	4.08
neos-856059	4.85	0.0	0.0	4.98	6.74
neos-859770	∞	0.0071	0.0	8.58	6.51

Continued on next page

Table B.4.: κ_{LP} and MILP κ statistics for various MILPs

		attention level $\sqrt{\alpha}$		max log κ (tree condition number)	
instance	log κ_{LP}	CPLEX 12.10.0.0	FICO Xpress v8.8.3	CPLEX 12.10.0.0	FICO Xpress v8.8.3
neos-860244	∞	0.0	0.0	5.15	4.57
neos-860300	∞	0.0	0.0	6.53	5.82
neos-862348	∞	0.0	0.0	3.88	4.62
neos-863472	∞	0.0	0.0	5.62	5.22
neos-880324	∞	0.0	0.0	5.88	5.06
neos-881765	∞	0.0	0.0	3.24	3.71
neos-886822	10.34	0.0	0.0	7.81	4.87
neos-892255	∞	0.0	0.0	5.16	5.77
neos-905856	∞	0.0	0.0	6.85	7.15
neos-906865	5.43	0.0	0.0	6.21	5.68
neos-911880	4.87	0.001	0.0	8.77	7.66
neos-911970	5.1	0.0	0.0	6.64	7.28
neos-912015	∞	0.0	0.0	4.33	5.02
neos-912023	∞	0.0	0.0	4.36	6.19
neos-913984	5.59	0.0	-	4.09	-
neos-916173	7.3	0.029	0.0026	10.43	7.43
neos-916792	∞	0.0037	0.0	10.59	6.71
neos-930752	5.86	0.0	0.0	6.47	5.21
neos-931517	4.77	0.0	0.0	6.44	6.05
neos-931538	4.77	0.0	0.0	5.84	5.2
neos-933364	4.8	0.0	0.0	6.92	7.45
neos-933550	∞	0.0	-	0.0	-
neos-933562	∞	0.0	0.0	6.03	6.85
neos-933815	4.63	0.0	0.0	5.68	6.59
neos-934531	∞	0.0	-	3.12	-
neos-935496	∞	0.0	0.0	6.77	6.9
neos-935674	∞	0.001	0.0	7.41	6.62
neos-935769	5.2	0.0	0.0	6.02	5.43
neos-941698	∞	0.0	0.0	4.36	4.33
neos-941717	∞	0.0	0.0	6.41	6.86
neos-941782	∞	0.0	0.0	6.23	6.55
neos-942323	∞	0.0	0.0	6.52	5.05
neos-942830	∞	0.0	0.0	5.69	6.53
neos-942886	∞	0.0	0.0	5.69	3.58
neos-948268	∞	0.0	-	0.0	-
neos-948346	7.04	0.0	0.0	4.38	4.96
neos-952987	5.97	0.0102	0.0	10.17	6.9
neos-953928	6.33	0.0	0.0	4.87	4.19
neos-954925	6.99	0.0	0.0	4.98	5.34
neos-955215	4.58	0.0	0.0	5.05	5.84
neos-955800	∞	0.0	0.0024	4.45	7.42
neos-956971	6.51	0.0	0.0	5.25	4.67
neos-957143	6.65	0.0	0.0	6.23	4.79
neos-957270	7.13	0.0	0.0	2.02	3.87
neos-957323	6.65	0.0	0.0	4.0	4.38
neos-957389	7.2	0.0	-	2.62	-
neos-960392	7.03	0.0	0.0	5.6	5.82
neos-983171	5.31	0.0	0.002	6.64	7.57
neos13	6.36	0.0017	0.0	7.48	6.0
neos15	6.38	0.0099	0.012	10.9	9.3
neos16	6.49	0.0	0.0	5.57	5.67
neos18	∞	0.0	0.0	4.44	6.99
neos6	6.14	0.0	0.0	4.42	6.24
neos788725	∞	0.0	0.0	4.96	5.84

Continued on next page

Table B.4.: κ_{LP} and MILP κ statistics for various MILPs

		attention level $\sqrt{\alpha}$		max log κ (tree condition number)	
instance	log κ_{LP}	CPLEX 12.10.0.0	FICO Xpress v8.8.3	CPLEX 12.10.0.0	FICO Xpress v8.8.3
neos858960	∞	0.0082	0.0	10.41	6.71
net12	∞	0.0	0.0	5.37	6.67
newdano	∞	0.0	0.0014	7.25	7.78
nobel-eu-DBE	∞	0.0033	0.0261	9.42	8.87
noswot	3.8	0.0	0.0	5.27	6.34
ns1158817	∞	0.1	-	9.68	-
ns1606230	∞	0.0	0.001	5.36	7.07
ns1631475	∞	0.0076	0.0521	9.89	7.56
ns1663818	∞	0.1	0.0745	8.94	8.41
ns1685374	∞	0.0119	0.0	9.01	6.91
ns1686196	∞	0.0	0.0	6.7	6.3
ns1688347	∞	0.0	0.0	5.89	6.77
ns1696083	∞	0.0315	0.1011	10.44	11.31
ns1702808	∞	0.0035	0.5478	9.47	13.61
ns1745726	∞	0.0	0.0	6.09	3.89
ns1766074	∞	0.0	0.0	4.65	4.47
ns1769397	∞	0.011	0.0	8.01	6.82
ns1830653	∞	0.0	0.0	5.13	6.79
ns1854840	∞	0.0	-	5.21	-
ns1905797	∞	0.0058	0.0068	11.15	8.36
ns1905800	∞	0.0159	0.001	12.71	7.38
ns1952667	4.92	0.014	0.0	8.52	6.11
ns2081729	6.35	0.0	0.0089	5.96	10.17
ns2137859	∞	0.0047	0.0032	8.27	7.73
ns4-pr9	∞	0.0	0.0	5.69	6.3
ns894236	∞	0.0057	0.0	9.97	6.94
ns894244	∞	0.0199	0.0039	12.36	7.52
ns894786	∞	0.0099	0.0139	9.83	8.19
ns894788	∞	0.014	0.0	10.15	6.49
ns903616	∞	0.0093	0.0128	11.91	8.15
ns930473	∞	0.0372	0.0812	10.43	9.75
nsa	5.33	0.0	0.0	4.58	4.4
nsrand-ipx	12.1	0.0	0.0	5.94	5.55
nu120-pr3	∞	0.0	0.0	6.05	6.86
nu60-pr9	∞	0.0	0.0	7.56	6.54
nug08	∞	0.0	0.0	5.8	6.23
nw04	8.68	0.0	0.0	3.91	3.04
opm2-z7-s2	7.82	0.0022	0.0	7.42	6.17
opt1217	3.66	0.0	0.0	3.91	3.11
p0033	∞	0.0	0.0	2.17	3.4
p0201	∞	0.0	-	3.48	-
p0282	6.42	0.0	0.0	3.57	1.23
p0548	∞	0.0	-	3.64	-
p100x588b	∞	0.0074	0.0098	12.15	8.99
p2756	6.56	0.0	0.0	3.87	3.65
p6b	4.07	0.0	0.0	6.61	5.57
p80x400b	∞	0.0115	0.0078	10.14	9.44
pb-simp-nonunif	∞	0.0094	0.0	8.19	6.49
pg	2.92	0.0	0.0	6.21	5.81
pg5_34	3.0	0.0024	0.0	7.07	6.13
pigeon-10	∞	0.0	0.0	4.64	5.33
pigeon-11	∞	0.0	0.0	5.31	5.87
pigeon-12	∞	0.0	0.0	4.42	2.92
pigeon-13	∞	0.0	0.0	5.24	2.51

Continued on next page

Table B.4.: κ_{LP} and MILP κ statistics for various MILPs

		attention level $\sqrt{\alpha}$		max log κ (tree condition number)	
instance	log κ_{LP}	CPLEX 12.10.0.0	FICO Xpress v8.8.3	CPLEX 12.10.0.0	FICO Xpress v8.8.3
pigeon-19	∞	0.0	0.0	5.37	3.61
pk1	4.57	0.0	0.0	6.5	5.91
pp08a	4.27	0.0	0.0	6.38	6.39
pp08aCUTS	4.39	0.0	0.0	5.58	6.2
probportfolio	4.79	0.0	0.0	7.16	6.83
prod1	3.8	0.001	0.0	7.91	4.9
prod2	4.02	0.0022	0.0	9.57	6.34
protfold	∞	0.0073	0.0217	10.59	11.91
pw-myciel4	4.53	0.0	0.0	5.06	5.33
qap10	∞	0.0174	0.1	7.0	7.55
qiu	5.4	0.002	0.0	7.41	6.33
qnet1	∞	0.0	0.0	4.73	3.43
qnet1_o	∞	0.0	0.0	4.73	2.98
queens-30	4.09	0.0	0.0	7.06	6.17
r80x800	∞	0.0024	0.0017	8.4	8.57
rail507	∞	0.0	0.0	6.08	4.97
ramos3	3.67	0.0	0.0	6.6	7.0
ran14x18	∞	0.001	0.0	7.63	6.96
ran14x18-disj-8	∞	0.001	0.0	9.11	7.81
ran14x18_1	∞	0.0	0.0	7.73	7.38
ran16x16	∞	0.0	0.0	6.69	6.38
rd-rplusc-21	∞	0.0039	0.0288	11.35	10.72
rgn	3.72	0.0	0.0	3.49	3.03
rlp1	∞	0.0	0.0	4.05	4.04
rmatr100-p10	4.81	0.0	0.0	4.81	5.23
rmatr100-p5	5.09	0.0	0.0	5.87	5.84
rmine6	5.32	0.0028	0.0	8.11	6.62
rocII-4-11	∞	0.002	0.001	8.2	7.25
rocII-7-11	∞	0.001	0.0036	9.67	7.67
rocII-9-11	∞	0.0024	0.0014	10.67	7.76
rococoB10-011000	∞	0.0	0.0	7.46	6.7
rococoC10-001000	∞	0.0017	0.0052	8.68	8.14
rococoC11-011100	∞	0.001	0.001	8.02	7.44
rococoC12-111000	∞	0.0014	0.0075	9.13	7.72
rout	7.42	0.0	0.0	6.83	5.36
roy	4.7	0.0	0.0	3.81	4.82
rvb-sub	∞	0.0079	0.0	13.29	5.08
satellites1-25	∞	0.0987	0.0973	11.02	11.34
satellites2-60-fs	∞	0.3341	0.1099	11.02	10.47
set1ch	5.98	0.0	0.0	5.99	6.35
set3-10	9.94	0.0141	0.0315	10.06	8.7
seymour	∞	0.0	0.0	7.43	6.62
seymour-disj-10	4.94	0.003	0.0014	9.92	7.31
shipsched	∞	0.0108	0.1054	10.11	11.11
shs1023	∞	0.0532	0.1	7.75	8.24
sp97ar	∞	0.0	0.0	7.72	6.6
sp97ic	∞	0.0	0.0	7.75	6.76
sp98ar	∞	0.0	0.0	7.44	6.61
sp98ic	11.69	0.0	0.0	6.07	6.35
sp98ir	∞	0.0	0.0	6.17	6.0
stein27	2.58	0.0	0.0	2.46	3.05
stein45	3.32	0.0	0.0	4.06	3.92
stockholm	∞	0.0208	0.1005	11.05	10.91
sts729	5.42	0.0	0.0277	6.91	7.14

Continued on next page

Table B.4.: κ_{LP} and MILP κ statistics for various MILPs

		attention level $\sqrt{\alpha}$		max log κ (tree condition number)	
instance	log κ_{LP}	CPLEX 12.10.0.0	FICO Xpress v8.8.3	CPLEX 12.10.0.0	FICO Xpress v8.8.3
swath	∞	0.0	0.0	7.06	5.92
tanglegram2	4.13	0.0	0.0	4.39	3.65
timtab1	5.22	0.0014	0.0	8.23	6.35
timtab2	5.38	0.002	0.0	10.94	7.16
toll-like	3.83	0.0	0.0	6.91	6.36
tr12-30	5.8	0.0058	0.0236	8.51	9.33
tw-myciel4	∞	0.0	0.0	6.66	6.73
uc-case11	∞	0.0356	0.0359	12.1	9.71
uct-subprob	∞	0.0	0.0	6.43	6.2
umts	∞	0.0037	0.0032	8.91	8.58
usAbbrv-8-25_70	∞	0.0	0.0049	7.01	9.96
vpm1	∞	0.0	0.0	2.52	3.07
vpm2	∞	0.0	0.0	5.49	4.21
vpphard	∞	0.0	0.0	7.91	6.18
vpphard2	∞	0.0	0.0	4.93	5.8
wachplan	∞	0.001	0.0	7.57	7.04
zib54-UUE	∞	0.0286	0.0101	9.99	8.24

Table B.5.: LP method comparison (MIPLIB 2017 benchmark, SCIP 6.0.2/MOSEK 8.1.0.21)

	first LP iterations		first LP time			first LP fractionality		
settings	barrier	simplex	barrier	crossover	simplex	barrier	crossover	simplex
instance								
30n20b8	25	1199	0.73	0.79	0.8	193	131	130
50v-10	29	285	0.01	0.1	0.01	31	29	29
CMS750_4	23	5650	0.35	-	0.24	6418	-	1697
academictimetablesmall	38	3530	10.19	10.58	2.51	22k	1263	1276
air05	16	1402	0.24	0.21	0.4	227	222	222
app1-1	16	106	0.34	0.28	0.1	1152	24	25
app1-2	32	26k	4.89	5.36	11.55	12k	244	244
assign1-5-8	9	221	0.01	-	0.01	114	-	114
atlanta-ip	142	15k	19.7	21.68	10.29	0	0	1676
b1c1s1	38	783	0.3	0.39	0.11	248	248	248
bab2	36	64k	27.77	22.37	63.17	3240	1138	672
bab6	20	45k	8.79	8.74	32.23	3679	1595	1082
beasleyC3	18	817	0.08	0.08	0.1	162	145	148
binkar10_1	28	568	0.06	0.07	0.06	38	38	38
blp-ar98	38	638	93.08	-	92.85	189	-	129
blp-ic98	18	221	1.88	2.47	2.27	112	53	54
bnatt400	8	809	0.1	0.09	0.18	1940	923	639
bnatt500	-	-	-	-	-	-	-	-
bppc4-08	13	488	0.04	0.06	0.15	1454	68	44
brazil3	11	6245	0.32	0.43	2.53	4570	1076	851
buildingenergy	34	164k	12.08	13.76	1126.25	8647	8438	8107
cbs-cta	124	4073	0.52	0.41	0.6	2467	190	238
chromaticindex1024-7	8	69k	11.01	23.19	374.13	73k	52k	45k
chromaticindex512-7	8	34k	3.57	9.86	72.06	36k	0	18k
cmflsp50-24-8-8	44	7551	1.03	0.6	1.57	0	491	491
co-100	67	1466	11.01	7.43	4.36	0	221	213
cod105	10	4368	0.23	1.73	9.73	1024	694	694
comp07-2idx	34	4061	10.89	17.65	5.86	3134	1374	1035
comp21-2idx	22	2262	2.97	5.78	2.19	2194	914	725
cost266-UUE	53	1429	0.26	0.17	0.05	0	56	56
cryptanalysiskb128n50obj14	-	60k	-	3.07	398.1	-	14k	23k
cryptanalysiskb128n50obj16	10	63k	2.56	2.75	439.48	26k	12k	23k
csched007	18	585	0.15	0.16	0.08	90	85	84
csched008	15	1346	0.15	0.13	0.13	1267	114	83

Continued on next page

Table B.5.: LP method comparison (MIPLIB 2017 benchmark, SCIP 6.0.2/MOSEK 8.1.0.21)

	first LP iterations		first LP time			first LP fractionality		
settings	barrier	simplex	barrier	crossover	simplex	barrier	crossover	simplex
instance								
cvs16r128-89	13	14k	2.32	1.51	3.63	3472	3210	3210
dano3_3	-	116k	-	10.85	59.3	-	12	12
dano3_5	202	125k	15.58	13.47	66.18	0	0	25
decomp2	1	1	0.01	0.01	0.01	0	0	0
drayage-100-23	121	8480	0.73	8.91	2.63	4795	278	244
drayage-25-23	193	9824	0.99	7.81	2.88	5109	272	207
dws008-01	36	298	0.67	0.44	0.31	56	31	28
eil33-2	20	218	0.25	0.35	0.25	30	30	30
eilA101-2	33	1502	41.67	42.09	28.09	71	71	71
enlight_hard	1	1	0.01	0.01	0.01	0	0	0
ex10	1	1	0.01	0.01	0.01	0	0	0
ex9	1	1	0.01	0.01	0.01	0	0	0
exp-1-500-5-5	41	925	0.08	0.08	0.08	135	135	135
fast0507	17	4622	0.5	0.51	2.29	456	266	254
fastxgemm-n2r6s0t2	14	1000	0.06	-	0.36	48	-	24
fhnw-binpack4-4	7	380	0.08	-	0.07	381	-	378
fhnw-binpack4-48	11	3227	0.27	0.28	0.18	3225	2994	3190
fiball	44	2441	9.87	30.37	6.76	32k	285	274
gen-ip002	10	69	0.01	0.01	0.01	18	18	18
gen-ip054	18	42	0.01	0.01	0.01	15	15	15
germanrr	32	1799	1.21	0.99	1.01	0	213	214
gfd-schedulen180f7d50m30k18	18	214k	8.23	9.99	3598.94	120k	57k	0
glass-sc	16	490	0.12	0.11	0.31	101	101	101
glass4	-	88	-	-	0.09	-	-	72
gmu-35-40	66	645	0.11	0.28	0.17	0	11	11
gmu-35-50	70	766	0.09	0.28	0.17	0	16	16
graph20-20-1rand	11	2237	0.23	-	0.42	1924	-	424
graphdraw-domain	23	236	0.1	0.09	0.08	158	100	90
h80x6320d	15	213	0.59	0.45	0.48	118	116	122
highschool1-aigio	13	210k	90.43	1276.37	3614.37	204k	35k	0
hypothyroid-k1	17	213	0.73	0.68	0.63	161	161	161
ic97_potential	14	541	0.06	-	0.05	522	-	358
icir97_tension	10	36	0.08	-	0.08	788	-	730
irish-electricity	520	90k	523.79	540.81	331.78	0	0	4537

Continued on next page

Table B.5.: LP method comparison (MIPLIB 2017 benchmark, SCIP 6.0.2/MOSEK 8.1.0.21)

		first LP iterations		first LP time			first LP fractionality		
instance	settings	barrier	simplex	barrier	crossover	simplex	barrier	crossover	simplex
irp		37	397	0.78	1.21	0.01	23	17	17
istanbul-no-cutoff		18	2717	0.39	0.48	0.95	13	13	13
k1mushroom		32	999	4.47	6.31	4.35	359	353	353
lectsched-5-obj		11	2517	0.45	-	0.73	6318	-	2153
leo1		33	342	2.39	1.29	1.29	0	64	65
leo2		31	416	3.9	4.02	3.79	0	0	0
lotsize		52	1590	0.1	0.1	0.08	0	524	526
mad		22	196	0.08	0.08	0.08	200	19	19
map10		44	18k	2.35	2.36	11.87	65	65	65
map16715-04		47	17k	2.55	2.58	11.7	69	69	69
markshare2		5	21	0.01	0.01	0.01	60	7	7
markshare_4_0		5	15	0.01	0.01	0.01	30	4	4
mas74		34	91	0.01	0.01	0.01	12	12	12
mas76		16	74	0.01	0.01	0.01	11	11	11
mc11		20	2301	0.08	0.08	0.01	365	363	363
mcsched		56	4002	0.23	0.14	0.54	0	1259	1259
mik-250-20-75-4		15	80	0.01	0.01	0.01	75	75	75
milo-v12-6-r2-40-1		50	6938	0.19	0.3	1.19	369	350	369
momentum1		123	2132	2.31	1.27	0.59	0	213	231
mushroom-best		19	392	0.52	0.31	0.41	29	21	27
mzzv11		257	6806	28.75	14.74	3.18	0	1013	857
mzzv42z		163	3310	13.27	3.86	1.59	4182	901	746
n2seq36q		12	5099	1.86	1.97	1.75	9445	389	222
n3div36		20	133	20.6	19.67	19.96	180	24	23
n5-3		12	576	0.06	0.05	0.04	38	31	36
neos-1122047		1	1	0.01	0.01	0.01	0	0	0
neos-1171448		56	10k	1.14	1.15	3.66	2457	105	76
neos-1171737		37	4951	0.42	0.4	1.41	1170	89	77
neos-1354092		7	13k	0.71	2.02	5.72	13k	891	926
neos-1445765		28	608	0.98	0.01	0.01	145	145	145
neos-1456979		26	688	0.66	-	0.24	390	-	113
neos-1582420		17	1162	0.25	0.17	0.2	299	294	291
neos-2657525-crna		14	249	0.01	0.01	0.01	405	113	85
neos-2746589-doon		12	12k	3.77	3.82	8.21	10k	480	638

Continued on next page

Table B.5.: LP method comparison (MIPLIB 2017 benchmark, SCIP 6.0.2/MOSEK 8.1.0.21)

		first LP iterations		first LP time			first LP fractionality		
	settings	barrier	simplex	barrier	crossover	simplex	barrier	crossover	simplex
instance									
neos-2978193-inde		20	1668	0.18	0.21	0.14	56	45	27
neos-2987310-joes		74	3479	4.53	3.21	1.64	0	0	0
neos-3004026-krka		7	1934	0.62	0.64	0.34	8320	3396	2778
neos-3024952-loue		18	5751	0.26	0.27	0.75	782	439	378
neos-3046615-murg		24	63	0.01	0.01	0.01	136	75	75
neos-3083819-nubu		28	515	0.11	0.09	0.09	33	33	33
neos-3216931-puriri		21	9115	1.6	1.73	4.0	1606	709	687
neos-3381206-awhea		7	589	0.06	0.06	0.06	2375	414	355
neos-3402294-bobin		14	3685	1.36	1.42	5.81	768	168	212
neos-3402454-bohle		12	19k	27.59	30.44	3586.17	2484	195	0
neos-3555904-turama		15	14k	5.17	9.11	24.64	17k	7621	729
neos-3627168-kasai		31	960	0.05	0.15	0.06	174	0	136
neos-3656078-kumeu		15	31k	1.06	1.65	17.39	12k	3688	3161
neos-3754480-nidda		59	238	0.1	0.01	0.01	0	41	41
neos-3988577-wolgan		11	37k	2.27	2.98	39.93	24k	4266	3214
neos-4300652-rahue		15	1869	11.35	12.95	13.4	10k	2138	206
neos-4338804-snowy		31	436	0.1	-	0.09	1281	-	420
neos-4387871-tavua		12	1484	0.29	0.34	0.3	107	34	34
neos-4413714-turia		90	130k	319.1	437.01	1872.15	190k	2606	2973
neos-4532248-waihi		48	7207	665.95	657.29	594.68	86k	28k	2667
neos-4647030-tutaki		49	4938	30.57	28.6	8.19	1093	721	680
neos-4722843-widden		44	1820	9.3	9.47	7.14	54k	3077	1589
neos-4738912-atrato		26	1270	0.16	0.14	0.15	0	230	229
neos-4763324-toguru		156	7864	209.01	210.0	20.37	1563	1563	1563
neos-4954672-berkel		51	446	0.08	0.01	0.01	76	51	50
neos-5049753-cuanza		20	8192	203.72	236.68	8.12	602	233	231
neos-5052403-cygnet		16	154k	19.52	19.77	293.61	5486	1126	715
neos-5093327-huahum		27	6147	6.78	6.82	3.32	0	64	64
neos-5104907-jarama		31	46k	467.87	482.88	248.94	2566	949	943
neos-5107597-kakapo		19	1579	0.26	0.24	0.23	2908	1536	1555
neos-5114902-kasavu		31	26k	1766.81	2431.42	140.33	679	262	258
neos-5188808-nattai		21	5666	1.41	-	2.61	162	-	38
neos-5195221-niemur		10	7413	1.22	1.29	9.92	4792	3584	2371
neos-631710		-	84k	-	348.75	711.42	-	1847	1500

Continued on next page

Table B.5.: LP method comparison (MIPLIB 2017 benchmark, SCIP 6.0.2/MOSEK 8.1.0.21)

		first LP iterations		first LP time			first LP fractionality		
	settings	barrier	simplex	barrier	crossover	simplex	barrier	crossover	simplex
instance									
neos-662469		73	4816	40.39	176.48	11.09	447	347	344
neos-787933		12	113	0.4	0.44	0.42	310	18	0
neos-827175		14	7209	1.68	1.87	2.69	14k	137	148
neos-848589		17	647	8.85	9.18	4.71	747	622	637
neos-860300		20	335	0.01	0.01	0.01	107	106	105
neos-873061		37	72k	6.12	6.41	205.19	103	91	92
neos-911970		12	160	0.03	0.02	0.05	840	62	45
neos-933966		28	18k	3.27	3.97	12.48	7124	1614	1009
neos-950242		28	461	2.02	2.15	0.41	4272	966	308
neos-957323		29	10k	10.08	15.18	6.02	31k	444	114
neos-960392		9	10k	1.89	2.11	2.91	59k	1556	265
neos17		33	618	0.01	0.1	0.1	171	0	171
neos5		8	140	0.01	-	0.01	35	-	35
neos8		37	85	0.24	0.24	0.2	0	29	30
net12		84	1268	8.08	4.13	0.53	728	531	345
netdiversion		33	39k	64.43	110.11	93.14	56k	0	3881
nexp-150-20-8-5		49	1201	40.63	54.04	4.33	307	80	82
ns1116954		10	16k	86.98	88.33	79.32	7482	868	570
ns1208400		8	1706	0.87	0.97	0.58	2524	417	340
ns1644855		76	83k	1423.37	1428.38	411.29	1124	541	488
ns1760995		1	1	0.01	0.01	0.01	0	0	0
ns1830653		12	1035	0.12	-	0.24	338	-	162
ns1952667		12	188	0.74	0.76	0.65	13k	40	40
nu25-pr12		11	1106	0.05	0.01	0.01	262	36	36
nursesched-medium-hint03		28	16k	40.85	180.98	35.64	4983	1723	1214
nursesched-sprint02		24	6450	1.53	1.63	2.2	2039	687	474
nw04		40	405	5.34	4.6	4.01	8	6	6
opm2-z10-s4		51	20k	49.59	27.94	123.44	5584	5584	5584
p200x1188c		19	635	0.09	0.09	0.06	5	5	5
peg-solitaire-a3		12	9904	0.63	0.91	8.71	4169	1292	1230
pg		32	272	0.09	0.07	0.08	93	93	93
pg5_34		19	355	0.09	0.08	0.03	88	88	88
physicianshed3-3		176	37k	156.24	118.71	38.46	0	1961	1637
physicianshed6-2		31	6869	0.94	1.25	2.79	4198	1555	1282

Continued on next page

Table B.5.: LP method comparison (MIPLIB 2017 benchmark, SCIP 6.0.2/MOSEK 8.1.0.21)

	first LP iterations		first LP time			first LP fractionality		
settings	barrier	simplex	barrier	crossover	simplex	barrier	crossover	simplex
instance								
piperout-08	40	1881	2.28	2.59	1.81	4568	1492	1591
piperout-27	31	3110	5.5	4.53	3.12	5218	2249	2400
pk1	12	61	0.01	0.01	0.01	55	15	15
proteindesign121hz512p9	79	1175	346.66	293.17	310.13	21k	216	218
proteindesign122trx11p8	53	599	55.86	10.28	31.71	37k	160	176
qap10	18	43k	0.01	0.86	22.57	1501	1231	1225
radiationm18-12-05	73	2586	0.59	0.58	0.58	3475	837	850
radiationm40-10-02	81	8300	2.77	2.47	2.82	14k	3617	2989
rail01	115	69k	84.46	86.52	105.53	11k	8114	7785
rail02	277	378k	1333.6	666.69	1322.83	0	12k	12k
rail507	23	4603	0.75	0.66	2.46	456	266	249
ran14x18-disj-8	29	711	0.06	0.05	0.05	86	86	86
rd-rplusc-21	38	538	2.36	2.19	2.09	429	246	64
reblock115	46	1525	0.86	0.55	0.39	0	878	878
rmatr100-p10	26	1773	0.31	0.36	0.72	51	51	51
rmatr200-p5	34	26k	3.0	3.15	61.32	66	66	66
rocI-4-11	30	694	0.19	0.12	0.1	808	525	472
rocII-5-11	12	129	0.9	1.0	0.47	2035	320	273
rococoB10-011000	22	3421	0.18	0.19	0.48	261	261	261
rococoC10-001000	19	1588	0.52	0.24	0.22	192	185	134
roi2alpha3n4	16	1030	5.43	5.72	4.08	46	37	36
roi5alpha10n8	18	3893	21.56	22.91	18.56	100	89	89
roll3000	18	807	0.13	-	0.13	251	-	190
s100	40	102k	73.71	94.93	401.61	1091	0	182
s250r10	37	135k	41.29	49.13	205.25	3782	0	0
satellites2-40	32	15k	57.46	29.91	24.42	13k	4282	1382
satellites2-60-fs	23	18k	12.91	0.01	23.45	16k	4529	1673
savsched1	28	152k	115.89	216.97	1138.96	234k	41k	17k
sct2	76	2784	0.21	0.23	0.5	994	80	50
seymour	14	4702	0.17	0.18	1.65	655	561	562
seymour1	42	7793	0.42	0.49	2.58	169	130	116
sing326	46	18k	7.54	4.22	12.04	0	973	950
sing44	41	37k	8.34	4.86	30.36	0	760	760
snp-02-004-104	220	157k	45.48	26.23	22.76	0	68	68

Continued on next page

Table B.5.: LP method comparison (MIPLIB 2017 benchmark, SCIP 6.0.2/MOSEK 8.1.0.21)

		first LP iterations		first LP time			first LP fractionality		
	settings	barrier	simplex	barrier	crossover	simplex	barrier	crossover	simplex
instance									
sorrell3		13	1288	2.92	3.44	3.79	1024	1024	1024
sp150x300d		23	176	0.01	0.01	0.01	45	41	41
sp97ar		51	1203	7.45	3.62	3.62	0	194	200
sp98ar		41	1008	9.1	7.32	7.01	0	158	156
splice1k1		27	1188	14.76	17.09	6.74	320	316	316
square41		38	89k	128.96	133.16	944.81	414	372	370
square47		36	123k	367.37	389.05	2584.14	367	347	347
supportcase10		15	31k	23.64	23.54	84.3	8654	5444	7218
supportcase12		17	7105	3.17	4.45	2.69	196	196	162
supportcase18		6	981	0.57	0.47	0.55	13k	90	85
supportcase19		19	420k	43.76	47.24	3425.4	8393	2421	0
supportcase22		11	1826	4.46	4.67	7.93	62	62	62
supportcase26		28	228	0.05	0.01	0.02	396	207	207
supportcase33		27	7772	20.76	-	21.01	3168	-	275
supportcase40		45	6661	0.68	0.57	0.77	38	38	38
supportcase42		48	3710	3.59	2.15	6.58	0	139	445
supportcase6		52	5731	22.13	18.64	23.72	166	140	140
supportcase7		33	5538	2.11	1.56	2.09	253	253	249
swath1		13	147	0.74	0.7	0.6	17	13	13
swath3		13	129	1.03	-	0.93	22	-	16
tbfp-network		26	11k	16.11	16.74	24.54	1952	229	220
thor50dday		24	1	1.33	1.44	0.01	104	49	0
timtab1		56	230	0.1	-	0.01	0	-	110
tr12-30		27	364	0.05	0.06	0.06	326	326	326
traininstance2		20	185	0.44	0.54	0.48	2598	138	91
traininstance6		23	74	0.26	0.32	0.28	1442	49	39
trento1		73	13k	3.79	2.57	6.78	697	475	478
triptim1		22	44k	16.64	17.78	44.53	14k	7841	6057
uccase12		41	88k	40.71	20.28	133.37	854	168	211
uccase9		129	22k	20.6	11.26	31.52	0	422	422
uct-subprob		12	745	0.06	0.05	0.16	323	233	203
unitcal_7		130	16k	16.09	8.39	2.8	0	712	654
var-smallemery-m6j6		150	1957	8.92	4.33	1.71	0	397	396
wachplan		17	1392	0.24	-	0.33	1806	-	288

Table B.6.: Node throughput comparison (MIPLIB 2017 benchmark, heuristics and separation deactivated)

	nodes						tree time					
LP solver	Clp	CPLEX	Gurobi	MOSEK	SoPlex	Xpress	Clp	CPLEX	Gurobi	MOSEK	SoPlex	Xpress
instance	12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09	12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09
30n20b8	240	1k	10k	12k	6k	8k	3600.0*	210.4	889.7	1077.0	1139.8	520.8
50v-10	2212k	2998k	2907k	1192k	2697k	1512k	3600.0*	3600.0*	3600.0*	3600.0*	3600.0*	3600.0*
CMS750_4	891	459k	379k	185k	402k	188k	3599.9*	3599.9*	3599.9*	3599.8*	3599.7*	3599.9*
academictimetablesmall	3	5k	14k	761	1k	7k	3787.7*	3599.2*	3599.3*	3598.1*	3596.2*	3598.0*
air05	689	331	353	366	380	455	3599.9*	16.5	19.2	30.3	32.7	30.3
app1-1	1k	13	28	12	5	24	3600.0*	5.6	6.4	10.1	0.6	2.8
app1-2	14	30k	21k	7k	14	143	3594.0*	3599.4*	3598.6*	3599.1*	410.3	162.7
assign1-5-8	126k	5490k	6430k	3585k	6298k	4150k	3600.0*	3600.0*	3599.9*	3599.9*	3599.9*	3600.0*
atlanta-ip	3	11k	7k	1k	9k	16k	3595.0*	3591.0*	3591.6*	3590.0*	3580.3*	3439.1
b1c1s1	728	1143k	1474k	536k	803k	897k	3600.0*	3600.0*	3600.0*	3600.0*	3600.0*	3600.0*
bab2	20	6k	6k	1k	957	850	5812.0*	3560.7*	3584.2*	3546.8*	3390.1*	3566.8*
bab6	15	8k	8k	1k	5k	2k	3730.3*	3584.5*	3591.0*	3570.2*	3481.4*	3585.1*
beasleyC3	76k	4504k	3086k	2184k	2346k	2079k	3599.9*	3600.0*	3600.0*	3600.1*	3600.0*	3600.0*
binkar10_1	2195k	3349k	3781k	1475k	3324k	2079k	3600.0*	3599.9*	3600.0*	3600.0*	3600.0*	3599.9*
blp-ar98	718	236k	198k	98k	159k	157k	3600.6*	3599.8*	3599.8*	3599.7*	3599.7*	3599.8*
blp-ic98	243k	398k	308k	97k	194k	137k	3599.8*	3599.8*	3599.8*	3599.8*	3599.8*	3599.8*
bnatt400	8k	12k	7k	9k	6k	7k	226.0	301.0	200.1	448.9	127.9	107.5
bnatt500	25k	26k	26k	24k	29k	31k	573.2!	733.7!	713.9!	1270.6!	532.3!	434.6!
bppc4-08	8k	1251k	1129k	224k	855k	988k	3600.0*	3600.0*	3600.0*	3600.0*	3599.8*	3600.0*
brazil3	550	4k	18k	1k	1k	5k	3683.5*	3598.4*	3598.7*	3597.7*	3595.5*	3598.2*
buildingenergy	308	414	390	117	42	233	3552.3*	3578.4*	3592.8*	2640.5*	3123.5*	3536.0*
cbs-cta	395k	278	1k	76k	2k	421	3599.9*	16.7	23.4	1686.1	73.7	52.9
chromaticindex1024-7	1k	2k	3k	1k	1k	6k	3510.7*	3495.9*	3306.8*	3232.9*	3311.9*	3170.8
chromaticindex512-7	834	4k	4k	4k	17k	3k	1341.9	3575.7*	3557.8*	3530.0*	3460.8*	1735.8
cmflsp50-24-8-8	745	125k	44k	32k	31k	38k	3606.7*	3599.4*	3599.4*	3598.6*	3598.1*	3599.5*
co-100	828	49k	36k	7k	22k	38k	3598.5*	3598.6*	3598.3*	3598.4*	3596.4*	3597.8*
cod105	2k	644	156	155	99	87	775.0	130.9	149.1	322.2	356.9	80.8
comp07-2idx	11k	14k	41k	2k	3k	27k	3598.1*	3595.7*	3598.3*	3598.5*	3593.9*	3599.1*
comp21-2idx	29k	132k	139k	6k	17k	68k	3599.2*	3599.5*	3599.6*	3599.7*	3598.8*	3599.8*
cost266-UUE	253	1297k	1163k	562k	1026k	745k	3600.0*	3600.0*	3600.0*	3600.0*	3599.8*	3600.0*
cryptanalysiskb128n50bj14	31	133	272	84	862	2k	3569.0*	3570.7*	3451.7*	3196.0*	3575.8*	3595.6*
cryptanalysiskb128n50bj16	35	105	120	61	344	2k	3567.2*	3574.1*	3439.4*	3173.0*	3592.9*	3594.3*
csched007	1k	890k	907k	677k	937k	733k	3600.0*	3600.0*	3600.0*	3600.0*	3599.9*	3599.9*
csched008	2k	57k	46k	60k	398k	73k	3599.9*	169.5	126.0	972.6	1715.3	226.5

Continued on next page

Table B.6.: Node throughput comparison (MIPLIB 2017 benchmark, heuristics and separation deactivated)

	nodes						tree time					
LP solver	Clp	CPLEX	Gurobi	MOSEK	SoPlex	Xpress	Clp	CPLEX	Gurobi	MOSEK	SoPlex	Xpress
instance	12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09	12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09
cvs16r128-89	44k	129k	89k	29k	35k	89k	3599.0*	3599.2*	3599.8*	3598.3*	3596.9*	3598.8*
dano3_3	30	52	35	62	45	29	915.3	43.1	56.1	92.9	76.7	73.5
dano3_5	337	624	450	1k	311	205	3541.5*	173.3	251.6	756.2	391.9	154.1
decomp2	1k	1k	7k	18k	1k	2k	245.8	414.4	1403.2	3599.8*	309.5	656.5
drayage-100-23	321	412	505	5k	502	584	53.7	15.2	33.4	1362.6	40.4	55.8
drayage-25-23	4k	830k	504k	12k	250k	482k	3601.2*	3600.0*	3599.9*	3599.4*	1301.6	3521.8
dws008-01	32	53k	68k	106k	59k	76k	3600.0*	3599.9*	3599.8*	3599.7*	3599.8*	3599.8*
eil33-2	9k	615	641	675	717	631	65.2	116.1	43.5	98.2	95.0	110.2
eilA101-2	9k	3k	16k	10k	3k	7k	3596.0*	3597.0*	3598.7*	3595.1*	3565.6*	3598.9*
enlight_hard	1	1	1	1	1	1	0.0	0.0	0.0	0.0	0.0	0.0
ex10	1	1	1	1	1	1	568.5	576.0	571.3	574.5	573.3	574.6
ex9	1	1	1	1	1	1	37.7	37.8	37.8	37.6	37.8	37.8
exp-1-500-5-5	5241k	6722k	6762k	3657k	5319k	4069k	3600.1*	3600.1*	3600.1*	3600.1*	3600.1*	3600.0*
fast0507	23k	917	1k	2k	810	1k	1940.8	70.1	104.7	223.4	179.9	391.1
fastxgemm-n2r6s0t2	56k	52k	59k	73k	281k	61k	3570.8	754.4	489.8	3599.9*	1421.9	428.5
fhnw-binpack4-4	9494k	10 102k	10 105k	5663k	8618k	7998k	3600.0*	3600.0*	3600.0*	3599.9*	3600.0*	3599.9*
fhnw-binpack4-48	1402k	2081k	1889k	414k	1556k	738k	3599.9*	3599.9*	3599.8*	3600.0*	3599.9*	3599.9*
fiball	4k	4k	4k	4k	4k	4k	3600.2*	3599.9*	3600.0*	3600.1*	3599.5*	3599.9*
gen-ip002	4379k	4427k	4945k	4653k	4139k	4128k	1728.3	2139.9	1540.9	2617.6	1530.8	2137.9
gen-ip054	3539k	3994k	7695k	5199k	4161k	5194k	1423.9	1846.6	2116.6	2356.0	1590.8	2543.1
germanrr	3	275k	176k	70k	101k	115k	3668.4*	3599.7*	3599.6*	3599.5*	3596.6*	3599.5*
gfd-schedulen180f7d50m30k18	469	5k	2k	1	5	6k	3620.1*	3538.8*	3573.9*	6.7*	3452.2*	3596.3*
glass-sc	47k	140k	171k	81k	264k	107k	3599.7*	3599.9*	3599.9*	3599.7*	3599.8*	3599.6*
glass4	-	4442k	4511k	2025k	4180k	4763k	-	3600.0*	3600.0*	3600.0*	3600.0*	3600.0*
gmu-35-40	3729k	4948k	5341k	2466k	4889k	3372k	3600.0*	3600.0*	3600.0*	3599.9*	3600.0*	3600.0*
gmu-35-50	380k	2906k	2445k	1563k	2416k	2112k	3600.0*	3600.0*	3600.0*	3600.0*	3599.9*	3600.0*
graph20-20-1rand	8k	545k	577k	56k	425k	744k	3599.9*	3599.7*	3600.0*	3599.6*	3599.7*	3599.7*
graphdraw-domain	4099k	4740k	6394k	2997k	5491k	3844k	3600.0*	3595.4	3599.9*	3600.0*	2915.8	3600.0*
h80x6320d	6k	876k	857k	363k	713k	405k	3601.4*	3599.9*	3599.9*	3599.9*	3599.9*	3599.9*
highschool1-aigio	1	1	1	1	1	1	7.2*	7.7*	7.2*	7.0*	7.2*	7.4*
hypothyroid-k1	39	37	28	38	2	45	48.0	35.1	36.1	45.9	31.8	35.0
ic97_potential	4064k	5178k	5209k	3155k	5280k	4075k	3600.0*	3600.0*	3600.0*	3600.0*	3600.0*	3600.0*
icir97_tension	15k	2868k	2322k	1081k	2282k	1719k	3600.0*	3600.0*	3600.0*	3600.0*	3600.0*	3600.0*
irish-electricity	84	145	164	79	42	1k	3420.0*	3535.5*	3475.2*	3264.3*	3424.4*	3566.8*

Continued on next page

Table B.6.: Node throughput comparison (MIPLIB 2017 benchmark, heuristics and separation deactivated)

	nodes						tree time					
LP solver	Clp	CPLEX	Gurobi	MOSEK	SoPlex	Xpress	Clp	CPLEX	Gurobi	MOSEK	SoPlex	Xpress
instance	12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09	12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09
irp	50	31	37	39	33	25	65.0	28.0	20.2	41.9	28.8	33.6
istanbul-no-cutoff	2k	1k	1k	1k	1k	1k	859.8	258.5	121.7	540.1	124.6	89.7
k1mushroom	144	40	49	71	6	92	1600.8	1388.1	1383.0	1521.6	1362.3	1374.9
lectsched-5-obj	130k	218k	230k	36k	177k	163k	3599.8*	3599.8*	3599.7*	3599.5*	3599.7*	3599.8*
leo1	414k	852k	138k	184k	310k	137	3599.9*	3599.9*	3599.9*	3599.9*	3599.8*	18.7*
leo2	58k	205k	13k	49k	112k	10	3599.8*	3599.5*	3599.0*	3599.8*	3599.5*	3.8*
lotsize	1910k	3124k	2897k	1323k	1980k	1497k	3600.0*	3600.0*	3600.1*	3600.0*	3599.9*	3599.9*
mad	1297k	11294k	11477k	6436k	12629k	8947k	3600.0*	3600.0*	3600.0*	3600.0*	3600.0*	3600.0*
map10	3k	1k	1k	1k	1k	3k	3596.6*	727.3	313.2	3593.3*	840.1	716.1
map16715-04	1k	2k	2k	1k	2k	2k	3595.9*	1751.9	922.5	3592.4*	2191.1	1477.8
markshare2	18407k	14978k	18937k	13605k	19826k	15456k	3600.0*	3600.0*	3600.1*	3600.1*	3600.0*	3600.0*
markshare_4_0	1984k	1786k	1960k	2373k	2074k	1961k	299.3	348.3	292.4	455.8	284.7	357.5
mas74	4762k	3049k	3201k	3137k	3063k	2952k	3600.0*	1066.0	928.3	1563.6	859.0	1257.7
mas76	574k	274k	325k	392k	355k	332k	424.1	92.3	81.8	187.5	100.9	134.2
mc11	1k	3303k	3027k	1423k	1731k	1538k	3599.9*	3600.0*	3600.1*	3600.0*	3599.9*	3599.9*
mcsched	273k	18k	18k	39k	21k	19k	3599.7*	132.8	128.2	446.9	180.5	159.9
mik-250-20-75-4	7470k	7562k	7045k	3074k	7374k	3954k	3600.1*	3600.1*	3600.1*	3600.0*	3600.1*	3600.0*
milo-v12-6-r2-40-1	3	462k	735k	457k	810k	657k	3599.6*	3599.9*	3600.0*	3599.8*	3599.8*	3599.9*
momentum1	83	4k	-	1k	23k	47k	3604.6*	3599.4*	-	3599.9*	3599.8*	3600.0*
mushroom-best	1k	145k	44k	44k	11k	35k	3599.5*	3599.7*	3599.7*	3599.6*	3599.1*	3599.7*
mzzv11	268	14k	13k	8k	29k	19k	3569.5*	3594.1*	3598.9*	3597.1*	3589.9*	3598.9*
mzzv42z	1k	20k	8k	5k	19k	17k	3599.4*	3594.5*	1662.7	3598.9*	3431.4	3599.2*
n2seq36q	7k	340k	420k	35k	62k	198k	3600.7*	3599.6*	3599.5*	3599.3*	3599.1*	3599.4*
n3div36	47k	71k	46k	41k	45k	38k	3600.1*	3599.9*	3599.8*	3599.7*	3599.5*	3599.7*
n5-3	786k	817k	866k	532k	790k	571k	3600.0*	2935.9*	3302.3*	3600.0*	3600.0*	3599.9*
neos-1122047	22	19	24	37	8	34	20.8	12.2	35.8	1193.9	34.4	13.5
neos-1171448	506	754	296	9k	1k	104k	117.9	57.0	43.9	3596.3*	3598.3*	3599.7*
neos-1171737	51k	76k	105k	53k	24k	262k	3599.5*	1159.2	2641.8	3598.4*	3599.8*	3600.0*
neos-1354092	2	16k	17k	475	5k	17k	3737.4*	3598.2*	3598.2*	3594.7*	3385.5*	3596.0*
neos-1445765	1k	156	302	233	358	225	93.6	33.0	39.6	45.0	43.0	36.0
neos-1456979	180	435k	381k	52k	49k	236k	3607.4*	3600.0*	3600.0*	3600.0*	3599.9*	3600.0*
neos-1582420	3k	12k	37k	5k	26k	5k	3599.8*	107.3	210.4	113.1	234.4	66.7
neos-2075418-temuka	1	1	1	1	1	1	100.9*	103.6!	100.0!	101.0!	111.4*	100.3!
neos-2657525-crna	3229k	6070k	5034k	4420k	5378k	4644k	3600.0*	3600.0*	3600.0*	3600.0*	3600.0*	3600.0*

Continued on next page

Table B.6.: Node throughput comparison (MIPLIB 2017 benchmark, heuristics and separation deactivated)

	nodes						tree time					
LP solver	Clp	CPLEX	Gurobi	MOSEK	SoPlex	Xpress	Clp	CPLEX	Gurobi	MOSEK	SoPlex	Xpress
instance	12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09	12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09
neos-2746589-doon	1	24k	20k	6k	12k	13k	3640.2*	3596.1*	3598.9*	3593.4*	3592.4*	3598.1*
neos-2978193-inde	332k	347k	509k	233k	312k	279k	3599.8*	1507.1	3599.9*	3599.9*	3599.9*	3599.9*
neos-2987310-joes	1	1	1	1	1	1	17.6	17.5	17.4	17.6	17.6	17.4
neos-3004026-krka	511	2k	793k	57k	2k	4k	104.3	42.3	3599.9*	3599.9*	40.6	220.9
neos-3024952-loue	813k	1243k	1008k	492k	755k	573k	3599.5*	3599.6*	3599.9*	3599.9*	3598.8*	3599.8*
neos-3046615-murg	6489k	5748k	6091k	4378k	5801k	5372k	3600.0*	3600.0*	3600.1*	3600.1*	3600.1*	3600.0*
neos-3083819-nubu	49	4k	8k	5k	11k	7k	3600.0*	26.4	38.8	61.0	74.8	51.8
neos-3216931-puriri	101	8k	6k	2k	5k	13k	3596.6*	3597.2*	3598.3*	3596.0*	3598.2*	3598.4*
neos-3381206-awhea	3740k	4480k	3582k	2094k	3166k	2397k	3600.0*	3600.0*	3600.0*	3600.0*	3600.0*	3600.0*
neos-3402294-bobin	13k	24k	208k	346	69k	88k	3595.4*	3599.1*	3598.8*	3600.6*	3595.7*	3585.2*
neos-3402454-bohle	1	2	3	1	1	3	3292.2*	3584.7*	3087.7*	189.7*	190.6*	3196.0*
neos-3555904-turama	17	86	-	136	1k	520	3996.3*	3548.0*	-	3578.5*	3601.0*	3590.1*
neos-3627168-kasai	17k	3506k	3759k	1617k	2495k	1823k	3600.0*	3600.0*	3600.0*	3600.0*	3600.0*	3600.0*
neos-3656078-kumeu	125	3k	1k	2k	334	6k	3638.8*	3597.2*	3598.4*	3584.7*	3527.8*	3598.3*
neos-3754480-nidda	2225k	4957k	5678k	3745k	7331k	5272k	3600.0*	3600.0*	3600.0*	3600.0*	3600.0*	3600.0*
neos-3988577-wolgan	2k	25k	16k	3k	1k	9k	3369.7*	3593.7*	3598.8*	3560.6*	3378.1*	3596.6*
neos-4300652-rahue	4k	9k	5k	530	8k	4k	3595.3*	3594.4*	3593.8*	3579.2*	3595.9*	3595.2*
neos-4338804-snowy	1868k	4377k	3635k	1283k	3025k	2567k	3600.0*	3600.0*	3600.1*	3600.0*	3600.0*	3600.0*
neos-4387871-tavua	80k	976k	645k	436k	421k	502k	3600.0*	3600.0*	3600.0*	3599.9*	3599.9*	3599.9*
neos-4413714-turia	3k	9	166	1	2k	9k	3580.5*	2949.8*	3580.7*	68.2*	2168.5	3590.4*
neos-4532248-waihi	-	5	1k	31	3	3k	-	1788.6*	3578.8*	3569.8*	3340.4*	3579.0*
neos-4647030-tutaki	24k	29k	36k	6k	13k	22k	3596.3*	3597.1*	3597.0*	3597.1*	3592.7*	3598.8*
neos-4722843-widden	628	291	343	1k	3k	1k	3616.8*	3598.5*	3598.5*	3597.1*	3598.2*	3598.6*
neos-4738912-atrato	11k	17k	23k	156k	26k	38k	3599.7*	69.0	93.7	1798.2	216.0	346.7
neos-4763324-toguru	7k	15k	20k	8k	5k	16k	3602.5*	3593.1*	3592.7*	3581.8*	3578.4*	3594.2*
neos-4954672-berkel	6190k	7565k	6346k	3919k	6088k	4356k	3600.1*	3600.1*	3600.1*	3600.1*	3600.1*	3600.0*
neos-5049753-cuanza	157	8k	5k	1k	155	4k	3596.4*	3597.1*	3596.5*	3594.3*	3587.1*	3597.9*
neos-5052403-cygnet	168	519	527	5	74	1k	3351.3*	3506.5*	3529.8*	2280.0*	3056.4*	3532.0*
neos-5093327-huahum	2k	107k	126k	41k	116k	110k	3599.6*	3598.4*	3599.0*	3597.4*	3599.2*	3598.9*
neos-5104907-jarama	1	204	147	30	19	569	3582.9*	3576.9*	3588.6*	3361.7*	3534.2*	3582.0*
neos-5107597-kakapo	1k	795k	455k	165k	862k	412k	3600.0*	3600.0*	2726.0	3599.9*	3600.0*	2764.8
neos-5114902-kasavu	1k	5k	3k	26	60	1k	3585.5*	3589.8*	3588.1*	3493.3*	3506.8*	3593.3*
neos-5188808-nattai	46	18k	9k	1k	22k	12k	3600.0*	963.9	358.3	3598.6*	2078.2	391.1
neos-5195221-niemur	75	32k	29k	13k	40k	26k	3600.7*	2455.4	1603.7	3590.6*	2562.2	1487.3

Continued on next page

Table B.6.: Node throughput comparison (MIPLIB 2017 benchmark, heuristics and separation deactivated)

	nodes						tree time					
LP solver	Clp	CPLEX	Gurobi	MOSEK	SoPlex	Xpress	Clp	CPLEX	Gurobi	MOSEK	SoPlex	Xpress
instance	12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09	12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09
neos-631710	2	5	5	3	5	147	4114.5*	3125.0*	1094.8*	2792.4*	2599.2*	2658.1*
neos-662469	22k	134k	70k	37k	35k	47k	3673.0*	3599.3*	3599.3*	3598.5*	3595.6*	3599.3*
neos-787933	1	1	1	1	1	1	1.0	1.0	0.9	0.9	0.9	1.0
neos-827175	42	1	1	3	4	1	2878.8	1.1	1.5	34.8	38.8	0.5
neos-848589	2k	4k	10k	1k	8k	2k	3598.5*	3597.4*	3596.5*	3596.7*	3596.6*	3597.9*
neos-860300	1k	37	9	9	9	7	326.0	25.0	17.5	33.3	20.1	20.5
neos-873061	272	14k	12k	8k	59	762	3530.8*	3596.0*	3595.2*	3594.0*	3427.6*	3512.7*
neos-911970	415k	6293k	6076k	3924k	5839k	4756k	3600.0*	3600.0*	3600.0*	3600.0*	3600.0*	3600.0*
neos-933966	4k	82	129	432	1k	89	3561.6*	106.8	116.5	1252.5	3587.0*	67.3
neos-950242	88	167	782	166	400	311	68.4	200.5	680.3	1225.5	366.5	56.0
neos-957323	27	1	3	11	7	19	114.4	13.3	19.6	177.1	195.3	57.5
neos-960392	42	754	14k	1k	490	128	4476.4*	343.4	3598.8*	3596.9*	3586.4*	243.1
neos17	-	13k	12k	11k	7k	15k	-	13.3	11.1	19.4	10.2	20.4
neos5	1292k	1835k	1473k	2423k	689k	1431k	488.5	682.0	500.7	1330.2	229.5	695.0
neos8	1	46	64	10	1	1	1.9	2.3	2.9	2.0	1.8	1.8
neos859080	374	312	1k	115	2k	96	1.2!	0.2!	0.6!	0.1!	0.6!	0.1!
net12	855	2k	3k	1k	5k	4k	3600.0*	615.1	283.6	3600.2*	1164.4	734.2
netdiversion	892	40	27	982	383	147	3361.5	302.3	170.6	3488.4*	2245.8	248.8
nexp-150-20-8-5	250k	890k	462k	258k	293k	315k	3599.8*	3599.8*	3599.8*	3599.9*	3599.8*	3599.9*
ns1116954	15	779	38k	85	45	37k	7293.1*	3593.4*	3112.8	3351.2*	3525.9*	3572.5*
ns1208400	2	687	1k	1k	784	1k	3599.8*	66.8	62.3	649.6	167.8	73.4
ns1644855	11	109	201	1	231	17	3336.6*	3226.0	3351.8*	3248.1*	3271.7*	379.9
ns1760995	0	0	0	0	0	0	3612.5*	3610.7*	3612.8*	3606.3*	3610.5*	3611.7*
ns1830653	2k	21k	45k	26k	18k	28k	3599.8*	176.8	373.3	1158.6	123.0	208.9
ns1952667	131	228	14k	20k	4k	7k	3615.9*	80.7	3601.0*	3600.8*	367.4	2677.0
nu25-pr12	729	23k	9k	10k	13k	8k	3600.0*	64.8	39.2	79.8	65.1	59.9
nursesched-medium-hint03	1k	3k	5k	1k	568	3k	3593.8*	3596.1*	3596.4*	3592.6*	3585.1*	3597.0*
nursesched-sprint02	526	47k	35k	41k	30k	33k	3599.6*	607.2	1127.2	3599.3*	1697.8	1039.0
nw04	1	1	1	1	1	1	8.7	8.3	8.6	8.8	9.0	9.4
opm2-z10-s4	792	1k	3k	280	6k	2k	3557.9*	3575.4*	3577.8*	3468.2*	3515.9*	3554.2*
p200x1188c	17k	1828k	1765k	1351k	966k	932k	3600.0*	3600.0*	3600.0*	3600.0*	2942.3	3072.2
peg-solitaire-a3	140	4k	12k	793	353	4k	3600.0*	3597.5*	3595.6*	3591.7*	790.8	3599.6*
pg	3463k	5069k	4474k	2152k	4008k	2665k	3600.0*	3600.1*	3600.1*	3600.0*	3600.0*	3600.0*
pg5_34	3377k	4745k	4123k	1967k	3840k	2523k	3600.0*	3600.1*	3600.1*	3600.0*	3600.0*	3600.0*

Continued on next page

Table B.6.: Node throughput comparison (MIPLIB 2017 benchmark, heuristics and separation deactivated)

	nodes						tree time					
LP solver	Clp	CPLEX	Gurobi	MOSEK	SoPlex	Xpress	Clp	CPLEX	Gurobi	MOSEK	SoPlex	Xpress
instance	12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09	12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09
physiciansched3-3	8	495	693	370	294	650	4354.2*	3577.7*	3587.7*	3562.4*	3454.5*	3587.9*
physiciansched6-2	31	52k	35k	9k	4k	44k	3685.1*	3598.9*	3599.1*	3597.5*	3598.8*	3599.3*
piperout-08	989	3k	1k	3k	118	4k	532.3	284.2	229.3	308.6	413.4	237.0
piperout-27	2k	4k	1k	1k	84	2k	537.4	544.3	314.4	341.8	319.8	286.3
pk1	95k	334k	292k	316k	335k	280k	55.0	114.1	82.7	142.4	100.4	108.9
proteindesign121hz512p9	24	37	25	340	29	22	3601.2*	3600.8*	3601.6*	3602.0*	3600.0*	3603.7*
proteindesign122trx11p8	22	23	58	295	65	33	3604.8*	3600.4*	3604.1*	3600.7*	3601.0*	3600.7*
qap10	3	1	3	1	3	1	51.4	19.1	22.0	9.1	38.7	19.4
radiationm18-12-05	919k	1625k	1190k	389k	859k	692k	3599.8*	3599.9*	3599.9*	3599.8*	3599.8*	3599.8*
radiationm40-10-02	130k	206k	184k	41k	96k	108k	3599.0*	3599.1*	3599.0*	3598.3*	3596.4*	3598.8*
rail01	21	221	530	99	196	1k	3466.4*	3528.2*	3549.0*	3497.0*	3395.4*	3568.7*
rail02	8	28	51	10	1	23	1412.5*	3013.0*	2909.8*	2281.2*	47.5*	3198.2*
rail507	21k	945	1k	10k	1k	896	1962.1	66.6	95.7	1001.6	219.2	161.0
ran14x18-disj-8	639k	2690k	3546k	978k	2916k	1452k	3600.0*	3600.0*	3600.0*	3600.0*	3600.0*	3600.0*
rd-rplusc-21	47	124k	181k	154k	224k	236k	3599.8*	3599.5*	3599.6*	3599.6*	3599.7*	3599.8*
reblock115	1k	981k	960k	415k	1206k	722k	3600.0*	3599.8*	3599.8*	3599.4*	3599.6*	3599.8*
rmatr100-p10	1k	975	873	1k	822	754	162.6	50.8	49.4	295.1	105.5	53.2
rmatr200-p5	660	8k	4k	533	478	3k	3583.3*	3590.5*	3592.5*	3542.0*	3576.6*	3584.2*
rocI-4-11	16k	15k	13k	18k	29k	26k	183.4	61.6	67.0	284.2	117.4	104.8
rocII-5-11	2k	104k	351k	38k	107k	94k	3600.2*	3599.7*	3599.7*	3599.9*	3599.7*	3599.7*
rococoB10-011000	127k	673k	477k	106k	64k	128k	3599.9*	3599.9*	3599.8*	3599.9*	3599.4*	3599.8*
rococoC10-001000	91k	819k	1292k	1273k	867k	1308k	3600.0*	2323.0	3319.1	3599.9*	3204.1	3600.0*
roi2alpha3n4	27k	32k	42k	10k	42k	26k	3598.7*	3599.0*	3599.0*	3599.0*	3599.1*	3599.3*
roi5alpha10n8	5k	6k	5k	3k	6k	5k	3594.1*	3597.7*	3596.9*	3596.8*	3591.1*	3597.7*
roll3000	368	1258k	1447k	514k	1451k	879k	3600.2*	3600.0*	3599.9*	3600.0*	3599.9*	3600.0*
s100	35	896	3	2	1	428	2186.2*	3241.2*	2725.8*	3353.4*	791.3*	3212.8*
s250r10	251	5k	2k	32	4k	7k	3012.6*	1765.3	3549.6*	3517.4*	3261.8*	3482.8
satellites2-40	12	344	3k	259	9	5k	4579.3*	3564.9*	3589.0*	3589.0*	3443.7*	3583.8*
satellites2-60-fs	3	436	2k	189	91	2k	3719.8*	3579.3*	3588.8*	3588.8*	3489.9*	3584.3*
savsched1	1	2	81	3	1	34	12.9*	3099.2*	3139.1*	2945.7*	12.7*	3328.3*
sct2	132k	1066k	334k	278k	544k	738k	3637.5*	3600.0*	3599.9*	3599.8*	3599.9*	3599.9*
seymour	81k	148k	161k	72k	73k	59k	3599.4*	3599.5*	3599.4*	3598.4*	3599.4*	3599.4*
seymour1	4k	7k	6k	6k	7k	5k	513.5	377.1	170.8	418.4	155.4	151.4
sing326	448	15k	17k	7k	5k	12k	3599.1*	3595.7*	3596.7*	3584.9*	3551.0*	3591.0*

Continued on next page

Table B.6.: Node throughput comparison (MIPLIB 2017 benchmark, heuristics and separation deactivated)

	nodes						tree time					
LP solver	Clp	CPLEX	Gurobi	MOSEK	SoPlex	Xpress	Clp	CPLEX	Gurobi	MOSEK	SoPlex	Xpress
instance	12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09	12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09
sing44	10k	22k	9k	6k	3k	13k	3589.1*	3595.2*	3596.6*	3582.3*	3527.6*	3590.1*
snp-02-004-104	25k	47k	28k	8k	19k	13k	3581.7*	3595.1*	3595.5*	3588.9*	3548.4*	3587.7*
sorrell3	14k	85k	91k	1k	146k	70k	3598.6*	3598.9*	3599.2*	3597.8*	3598.8*	3599.5*
sp150x300d	5862k	6892k	7631k	4822k	6185k	4991k	3600.0*	3600.0*	3600.1*	3600.1*	3600.0*	3600.0*
sp97ar	56k	139k	46k	34k	47k	57k	3599.6*	3599.5*	3599.5*	3599.5*	3597.5*	3599.6*
sp98ar	72k	118k	35k	33k	53k	1k	3599.4*	3599.5*	3599.6*	3599.4*	3598.2*	179.0*
splice1k1	1k	390	510	670	486	438	2071.6	1752.7	1875.1	2046.5	1557.3	1722.7
square41	20	701	330	1	13	73	3555.9*	3538.8*	3591.0*	43.2*	3506.2*	3556.0*
square47	6	85	46	1	5	15	3490.0*	3473.8*	3573.7*	88.3*	3363.2*	3444.2*
supportcase10	11	162	592	46	27	10k	3369.7*	3489.7*	3581.4*	3353.1*	2340.9*	3465.8*
supportcase12	60k	117k	91k	19k	5k	56k	3595.4*	3596.5*	3597.4*	3597.7*	3579.7*	3596.1*
supportcase18	630k	1664k	975k	286k	808k	635k	3599.8*	3600.0*	3599.8*	3599.8*	3599.4*	3599.8*
supportcase19	1	1	1	1	1	1	176.0*	176.3*	176.0*	175.4*	175.9*	175.8*
supportcase22	16	14	25	9	78	19	3589.8*	3592.7*	3597.4*	3592.7*	3595.9*	3587.9*
supportcase26	4893k	3953k	4133k	4030k	5223k	3038k	3600.0*	3600.0*	3600.0*	3600.0*	3600.0*	3600.0*
supportcase33	2	60k	31k	29k	64k	36k	3602.3*	3598.4*	3598.4*	3597.3*	3598.0*	3598.5*
supportcase40	16k	132k	133k	53k	122k	118k	3599.7*	1774.6	1231.7	3599.0*	2158.0	2256.2
supportcase42	1k	129k	107k	14k	51k	97k	1415.4*	3589.9*	3599.1*	3598.3*	3598.4*	3599.4*
supportcase6	312	14k	13k	10k	5k	8k	3808.2*	3593.3*	3591.1*	3591.1*	3577.6*	3595.2*
supportcase7	20	3k	3k	5k	3k	2k	3643.7*	71.0	67.4	291.1	88.2	77.7
swath1	537	1k	1k	377	2k	1k	3601.4*	66.3	28.7	21.3	52.2	54.9
swath3	158	51k	49k	79k	46k	38k	3602.2*	647.8	329.6	1116.0	327.5	467.2
tbfp-network	75	219	112	2k	359	162	635.1	1145.3	720.5	2517.0	2508.8	570.7
thor50dday	43k	82k	83k	31k	26k	34k	3557.2*	3599.4*	3599.3*	3599.1*	3441.0*	3490.0*
timtab1	-	5447k	5703k	4034k	6152k	4212k	-	3600.0*	3600.0*	3600.0*	3600.0*	3600.0*
tr12-30	48k	4419k	4840k	2633k	3960k	2964k	3600.0*	3600.0*	3600.1*	3600.0*	3600.0*	3600.0*
traininstance2	6k	7k	9k	12k	3k	17k	3602.7*	3599.7*	3599.8*	3599.8*	3599.7*	3599.8*
traininstance6	34k	33k	44k	92k	20k	231k	3599.9*	3599.9*	3599.9*	3599.8*	3599.9*	3599.8*
trento1	5k	32k	7k	5k	23k	10k	3597.0*	2214.6	3597.4*	3594.6*	3594.8*	1571.1
triptim1	24	6	14k	750	117	19	3671.0*	60.3	1718.2	1585.8	520.8	118.9
uccase12	35k	42k	35k	19k	51k	18k	3566.1*	3594.8*	3596.9*	3465.8*	3570.4*	3597.5*
uccase9	11k	39k	21k	28k	3k	33k	3591.1*	3591.9*	3598.8*	3592.4*	3572.1*	3596.9*
uct-subprob	296k	918k	628k	203k	493k	627k	3599.8*	3599.9*	3599.9*	3599.8*	3599.8*	3599.9*
unitcal_7	77	89k	135k	47k	98k	65k	3599.1*	3599.0*	3599.4*	3596.8*	3595.8*	3599.2*

Continued on next page

Table B.6.: Node throughput comparison (MIPLIB 2017 benchmark, heuristics and separation deactivated)

	nodes						tree time					
LP solver	Clp	CPLEX	Gurobi	MOSEK	SoPlex	Xpress	Clp	CPLEX	Gurobi	MOSEK	SoPlex	Xpress
instance	12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09	12.8.0.0	1.16.11	8.1.0	8.1.0.21	4.0.2	33.01.09
var-smallemery-m6j6	56k	129k	166k	32k	79k	63k	3598.9*	3599.3*	3599.2*	3599.1*	3595.3*	3599.2*
wachplan	4k	35k	23k	24k	68k	58k	3599.8*	698.1	783.8	3599.8*	1436.1	1192.9