Konrad-Zuse-Zentrum
für Informationstechnik Berlin

Takustraße 7
D-14195 Berlin-Dahlem
Germany

TIMO BERTHOLD

GERALD GAMRATH

AMBROS M. GLEIXNER

STEFAN HEINZ

THORSTEN KOCH

YUJI SHINANO

Solving mixed integer linear and nonlinear problems using the SCIP Optimization Suite

Supported by the DFG Research Center MATHEON *Mathematics for key technologies* in Berlin.

ZIB-Report 12-27 (July 2012)

Herausgegeben vom
Konrad-Zuse-Zentrum für Informationstechnik Berlin
Takustraße 7
D-14195 Berlin-Dahlem

Telefon: 030-84185-0
Telefax: 030-84185-125

e-mail: bibliothek@zib.de
URL: http://www.zib.de

ZIB-Report (Print) ISSN 1438-0064
ZIB-Report (Internet) ISSN 2192-7782

Solving mixed integer linear and nonlinear problems using the SCIP Optimization Suite*

Timo Berthold
Gerald Gamrath
Ambros M. Gleixner
Stefan Heinz
Thorsten Koch
Yuji Shinano

Zuse Institute Berlin, Takustr. 7, 14195 Berlin, Germany,
{berthold,gamrath,gleixner,heinz,koch,shinano}@zib.de

July 31, 2012

Abstract

This paper introduces the SCIP Optimization Suite and discusses the capabilities of its three components: the modeling language Zimpl, the linear programming solver SoPlex, and the constraint integer programming framework SCIP. We explain how these can be used in concert to model and solve challenging mixed integer linear and nonlinear optimization problems. SCIP is currently one of the fastest non-commercial MIP and MINLP solvers. We demonstrate the usage of Zimpl, SCIP, and SoPlex by selected examples, give an overview of available interfaces, and outline plans for future development.

*A Japanese translation of this paper will be published in the Proceedings of the 24th RAMP Symposium held at Tohoku University, Miyagi, Japan, 27–28 September 2012, see http://orsj.or.jp/ramp/2012. Available as ZIB-Report 24, http://opus4.kobv.de/opus4-zib/frontdoor/index/index/docId/1559.

Contents

1 Introduction

Linear programming (LP) and *mixed integer linear programming* (MIP) are among the most essential techniques in operations research to model and solve optimization problems in practice. Since Dantzig's initial formulation of the simplex method for linear programs [12], Gomory's first complete cutting plane algorithm for general integer programs [17], and its combination with the branch-and-bound paradigm [24], continuing theoretical and computational advances have resulted in powerful solution algorithms for this class of optimization problems [21, 25].

In addition to the theoretical interest in integer programming, formulating a problem as a mixed integer linear program has many advantages for practitioners:

- Many real-world restrictions and optimization goals can be expressed, or at least approximated, using linear functions and integer variables. These are easily comprehensible without special experience in formulating MIPs.
- There exist a number of modern solver packages that can handle MIPs of surprisingly large size out-of-the-box [35, 39, 40, 41, 44]. Even if the problem at hand cannot be solved to proven optimality, often a near-optimal solution is produced.
- During the solution process, the dual bound given by the LP relaxation bounds the gap between the best found primal solution and the best possible solution. Thus, even if a problem cannot be solved to optimality within a specified time limit, the user has a proven guarantee about the quality of the best found solution, which often suffices in practice.

The progress in solving real-world MIP instances has been exceptional over the last decades. The pure algorithmic speedup recorded by commercial solver vendors such as CPLEX or XPRESS on their test sets collected from customers is as much as a factor of 55,000 over the last 20 years, see [22].

Among the most recent advances in state-of-the-art MIP solvers is the integration of solution techniques from *constraint programming* (CP) and *satisfiability solving* (SAT). It has been shown that several optimization problems that have been intractable by either of these methods alone could be solved by combining their ideas, see, e.g., [8, 20, 32]. As a consequence, various concepts for a general integration of MIP and CP have been investigated [3, 4, 7, 19, 27, 28] and MIP solvers have started to adapt ideas from CP and SAT for their algorithms.

To support this integration not only in algorithmic details, but on a conceptual level, the paradigm of *constraint integer programming* (CIP) has been developed [1]. It aims at restricting the modeling capabilities of CP as little as necessary to still gain the full power of all primal and dual MIP solving techniques. One showcase for the flexibility and richness of CIP is its recent extension to the general class of *nonconvex mixed integer nonlinear programming* (MINLP) [5, 33].

The goal of this article is to introduce the SCIP Optimization Suite, a software package that facilitates the modeling and solving of general CIPs with a special focus on mixed integer linear programs. It consists of three tools:

- SCIP, a highly customizable solver framework for constraint integer programming and branch-cut-and-price. Used standalone, it is currently one of the fastest non-commercial MIP and MINLP solvers.
- ZIMPL, an easy-to-learn modeling language whose syntax closely resembles standard mathematical notation and which is well-integrated into SCIP.

- SoPlex, an advanced implementation of the simplex method used to solve LPs standalone or within SCIP.

The paper is organized as follows. Section 2 describes the specific features and capabilities of SCIP, Zimpl, and SoPlex. Section 3 gives an overview of the various interfaces to other software products and file formats. A tutorial with simple examples for the different possibilities of how to use the SCIP Optimization Suite in practice is provided in Section 4. Section 5 explains how the SCIP Optimization Suite can be obtained and Section 6 presents a selection of projects in industry and academia where these tools are employed. We conclude by outlining future directions for the development of the SCIP Optimization Suite in Section 7.

2 Features

2.1 SCIP

SCIP – *Solving Constraint Integer Programs* – is a branch-cut-and-price framework for MIP, MINLP, and CIP. In particular, SCIP can be used as a callable library to embed its solving capabilities into third-party software or as a standalone solver with an interactive shell to solve optimization problems given in a standard file format. The goal of SCIP is to combine the advantages and compensate for the weaknesses of MIP, MINLP, and CP.

Plugin-based architecture. SCIP distinguishes itself through the modular design of its solution algorithm, which allows the combination of general-purpose and specialized solving techniques in a flexible fashion.

The central objects of SCIP are *constraint handlers*. Together with the domains of the variables they define the space of feasible solutions. There are handlers for integrality, linear, nonlinear, logical, combinatorial, and many other constraints. A constraint handler must be able to decide whether a given solution is feasible for all constraints of its type. To speed up the solution process, it may provide supplementary algorithms like constraint-specific presolving, domain propagation, separation methods, or a linear representation of its constraints.

Additional plugins such as branching rules, propagators, and primal heuristics allow for an efficient solution process. Altogether, SCIP 3.0 contains 126 plugins. The interaction among them is organized by the core of SCIP. In addition, the user has the possibility to influence the solution process by a variety of parameters.

The standard distribution of SCIP contains a collection of 26 constraint handlers. Typically, a subset of them suffice to represent and solve models of a particular class as explained in the following paragraphs. The relation between different problem classes is depicted in Figure 1.

LP, MIP, and MINLP. In addition to a constraint handler for general linear constraints, SCIP provides handlers for specific subclasses of linear constraints, e.g., knapsack, variable upper and lower bounds, set packing, covering, and set partitioning constraints. Those allow for even more efficient solution techniques. LPs are expressed by a selection of these linear constraints, MIPs by adding integrality constraints on some of the variables. Further, SCIP supports special mathematical programming constraints, in particular indicator constraints and special ordered sets (SOS) of type one and two.

The solution process is enhanced by many other MIP-specific plugins, most prominently by primal heuristics such as rounding heuristics, RINS, or the feasibility pump,

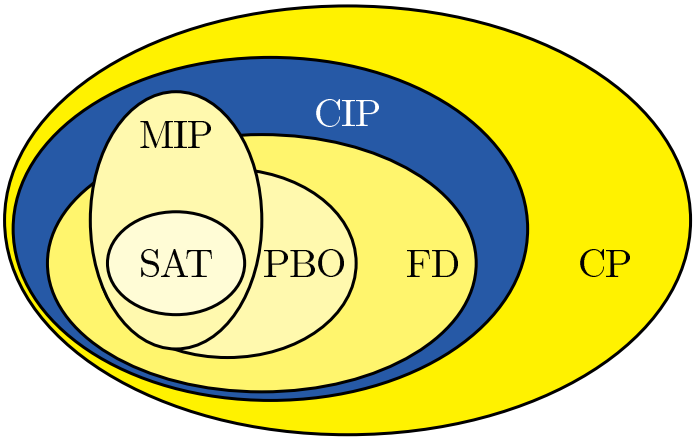

Figure 1: Inclusion diagram for different classes of mathematical optimization problems.

and separators for general cutting planes such as Gomory, c-MIR, strong Chvátal-Gomory, flowcover, MCF, clique cuts and others.

MINLPs can be modeled by linear, nonlinear, and integrality constraints. Besides supporting general nonlinear functions that are given via expression trees, SCIP has special support for quadratic, second-order cone, bivariate, and absolute power constraints. Note that the variety of MIP-specific plugins also support the MINLP solution process.

PBO, SAT, and scheduling. Pseudo-boolean optimization problems [9], can be modeled and solved by `linear`, `and`, and `integral` constraints. SAT instances by `logicor` and `integral` constraints. Scheduling problems can be modeled in a CP fashion using `linking` and `cumulative` constraints.

Branch-and-price. A widely used feature of SCIP is its branch-and-price capability. A user can customize SCIP as a complete branch-and-price solver for a specific class of problems by simply implementing a *variable pricer* plugin. In particular, the whole management of the search tree is carried out by SCIP. This distinguishes SCIP from most commercial MIP solvers, for which it is not easily possible to locally add variables via callbacks.

Performance. Although SCIP supports the much more general concept of constraint integer programming, it is competitive with state-of-the-art commercial and noncommercial MIP solvers [22]. In independent comparisons, SCIP consistently scores as one of the fastest non-commercial solvers for MIP [38] and PBO [46]. Further, it is currently one of the fastest solvers for nonconvex MINLP [33].

2.2 Zimpl

When developing an optimization model, the choice of the problem class is essential. It will determine the level of detail that can be incorporated in the model, the solution algorithms available, and consequently the size of the instances that one can expect to solve under given time restrictions. However, it is important to note that the solvability of a MIP, for example, is hard to predict and may depend very much on

the details of the modeling, see, e.g. [2]. Using an algebraic modeling language can support the modeling process tremendously, since it allows one to easily modify the model until a final and satisfactory formulation has been found.

In the following, we will give a short overview of ZIMPL, its goals, and its features. ZIMPL (Zuse Institute Mathematical Programming Language) is a powerful language to describe mathematical programs and a tool to translate these programs into `lp` and `mps` files, the standard description formats for linear and mixed integer linear programs. ZIMPL is tightly integrated with SCIP, and thus provides a simple way to model LP, MIP, or MINLP instances to be solved with SCIP. In particular, ZIMPL

- has a clear syntax close to the mathematical notation,
- is quick and easy to learn,
- allows for a clear separation between model and data,
- is stable,
- is freely available in C source code under the LGPL,
- is highly portable (Linux, Windows, Mac, Solaris, ...),
- is solver-independent,
- is available as a callable library,
- can be used standalone or linked to a solver,
- avoids rounding errors by using rational arithmetic when building the model.

History and trends. The development of algebraic modeling languages started in the 1980's with systems such as GAMS [6, 10] and AMPL [14, 15]. With these tools it became possible to state an LP in near-mathematical notation and have it automatically translated into a file that a solver could process or even have it directly read into the appropriate solver.

Unfortunately, many modeling languages available today are commercial products. None of these languages is available as source code for further development. None can be given to colleagues or used in classes, apart from very limited "student editions". Usually only a limited number of operating systems and architectures is supported, sometimes only Microsoft Windows.

In contrast, ZIMPL is freely available in source code under LGPL. This is very convenient both for teaching, as all students can use ZIMPL on their laptops and at home, and for use in industry projects because of the liberal licensing scheme. It should be noted that the general situation has improved since 1999 when the development of ZIMPL started. There are now at least some other open source languages, like, e.g., the GNU MATHPROG language, which implements a subset of the AMPL language and is part of the GNU Linear Programming Kit GLPK [36].

The current trend in commercial modeling languages is to further integrate features like database query tools, solvers, report generators, and graphical user interfaces into one modeling system. This sometimes enables the construction of complete graphical (business) applications around the mathematical model. Today the freely available modeling languages are no match in this regard. ZIMPL implements maybe 20% of the functionality of AMPL. Nevertheless, this subset proved to be sufficient for many real-world projects. Furthermore, the user manual for ZIMPL consists of about 25 pages. As we will see in the following, you can learn ZIMPL within a few hours and in contrast to graphical modelers, ZIMPL models are still very close to the mathematical notation.

Syntax. In general, each ZIMPL model consists of six types of statements:

- sets,
- parameters,
- variables,
- objective,
- constraints, and
- function definitions.

ZIMPL statements never change the already existing part of the model, but only add to it. This makes it much easier to understand ZIMPL models.

In addition to LP and MIP, ZIMPL also supports polynomial terms, hence in connection with SCIP it is now possible to model and solve non-convex quadratically and polynomially constrained integer programs [5, 33].

Rational arithmetic. A special feature of ZIMPL is the use of rational arithmetic. With a few noted exceptions, all computations in ZIMPL are done with infinite-precision rational arithmetic, ensuring that no rounding errors can occur during modeling. This is achieved by employing the GNU Multiple Precision Library[1].

Automatic reformulation. Constraints may contain the absolute value of a variable or be activated only if some boolean expression formed by bounded integer or binary variables evaluates to true. For example, a constraint may be activated only if an integer variable takes a specified value. ZIMPL converts these automatically into a system of linear inequalities.

Zimpl in practice. ZIMPL has been (and is continuously) used to model many real-world and educational problems, as diverse as location planning in telecommunications, 3D-Steiner tree packing for chip design, track auctioning, protein folding, etc. It is used in lectures at many universities and courses like Combinatorial Optimization at Work, see `http://co-at-work.zib.de`.

2.3 SoPlex

SOPLEX is an advanced implementation of the revised simplex algorithm for solving linear programs. It features preprocessing, exploits sparsity, and provides primal and dual solving routines. It can be used as a standalone solver reading `mps` or `lp` files or embedded into third-party software via a C++ class library. It is the default LP solver in SCIP. In the following, we outline the main features of SOPLEX. More details are described in [34].

Preprocessing. SOPLEX performs various presolving steps to reduce the size of the problem before applying the simplex algorithm. These include trivial reductions like handling of singleton rows and more elaborate steps such as the detection of linear dependencies; preprocessing helps to speed up the solution process. Matrix scaling is applied to improve numerics.

[1] `http://www.gmplib.org`

Primal and dual algorithm. SoPlex implements a composite simplex procedure automatically switching between primal and dual solving steps. The user can specify the initial algorithm type. Temporary shifting of the variable and constraint bounds is employed to guarantee numerical stability and a feasible starting basis. This renders an explicit "phase one" unnecessary.

Linear algebra operations. The linear algebra operations heavily exploit the sparsity of the constraint matrix, right-hand side, and objective function, which are present in the majority of LP instances. During pivoting, the sparse LU factorization is updated using Forrest-Tomlin or eta update steps [13, 31].

As a distinguishing feature, SoPlex can switch from the standard column-based computational form, which is used by default, to the so-called row representation of the basis. For LP instances with more constraints than variables, this results in a smaller dimension of the basis matrix and faster linear system solves.

High-precision solutions. SoPlex is a floating point LP solver and uses double-precision arithmetic by default. The precision can be increased to `long double` in order to address instances with particularly poor numerical properties.

A feature that distinguishes SoPlex from other LP solvers is the novel technique of iterative refinement [16] available since version 1.7. It uses the GNU Multiple Precision Library and iterated floating point solves to compute arbitrarily precise solutions.

SoPlex for LP-based branch-and-cut. As the default LP solver used in SCIP, SoPlex is continuously tested and improved. Consequently, it has become one of the most robust non-commercial LP solvers available and is particularly suited for repeatedly solving closely related LP instances.

3 Interfaces

There are several ways of accessing the SCIP Optimization Suite from other software packages or programming platforms.

3.1 File formats

The easiest way to load a problem into SCIP is via an input file, given in a format that SCIP can parse directly, see Section 4.2.1. SCIP 3.0 is capable of reading more than ten different file formats, including formats for nonlinear problems and constraint programs. This gives researchers from different communities an easy first access to the SCIP Optimization Suite. Table 1 lists the most used file formats in SCIP.

3.2 Modeling languages and Matlab interface

A natural way of formulating an optimization problem is to use a modeling language. Besides Zimpl there are several other modeling tools with a direct interface to SCIP. These include Comet [26], a modeling language for constraint programming, and GAMS [10], which is well-suited for modeling nonlinear optimization problems.

With SCIP 3.0, a first beta version of a functional Matlab interface has been released. It supports solving MIPs and LPs defined by Matlab's matrix and vector types.

Table 1: A selection of file formats that can be parsed by SCIP directly.

extension	short description
lp	file format for LPs and MIPs
mps	file format for LPs and MIPs
cnf	file format for satisfiability problems (SAT)
opb	file format for pseudo-boolean optimization problems
wbo	file format for weighted pseudo-boolean optimization problems
zpl	file extension for ZIMPL models
fzn	FlatZinc format, an input language for constraint programs
pip	file format for polynomially constrained mixed integer programs
cip	the standard SCIP format defined by the constraint handlers
osil	an XML-based file format that supports linear and nonlinear optimization problems

3.3 Python interfaces

Several Python-based software packages provide an interface to the SCIP Optimization Suite:

- NUMBERJACK [18], a constraint programming platform supporting a variety of different solvers.
- PICOS [37], a Python interface for conic optimization solvers developed by Guillaume Sagnol.
- PYTHON-ZIBOPT [42], a Python extension of the SCIP Optimization Suite developed and maintained by Ryan J. O'Neil.
- SAGE [43], a free open-source mathematics software system.

3.4 Java and C++

Since SCIP is written in C, its callable library can be directly accessed from C++. If a user wants to program his own plugins in C++, there are wrapper classes for all different types of plugins available in the `src/objscip` directory of the SCIP standard distribution. Also, SCIP 3.0 comes with a first version of a Java interface based on JNI providing the essential functions of the SCIP callable library.

4 Examples for usage

4.1 Zimpl

In this section, we demonstrate how to build a ZIMPL model, which can then be fed to SCIP either directly or via generating `lp` or `mps` files. As an example, we use an exponential description of the *symmetric traveling salesman problem* (TSP) as given, e.g., in [29, Section 58.5].

Let $G = (V, E)$ be a complete graph, with V being the set of cities, E being the set of links between the cities, and d_{ij} being the (symmetric) distance between cities i and j. Introducing binary variables x_{ij} for each $(i, j) \in E$ indicating if edge (i, j) is

part of the tour, the TSP can be written as:

$$\begin{aligned} \min \sum_{(i,j)\in E} d_{ij}x_{ij} & \qquad \text{subject to} \\ \sum_{(i,j)\in E, v\in\{i,j\}} x_{ij} = 2 & \qquad \text{for all } v \in V, \\ \sum_{(i,j)\in E, i,j\in U} x_{ij} \leq |U| - 1 & \qquad \text{for all } U \subseteq V, \emptyset \neq U \neq V, \\ x_{ij} \in \{0,1\} & \qquad \text{for all } (i,j) \in E. \end{aligned}$$

The data is read in from a file that gives the number of the city and the x and y coordinates. Distances between cities are assumed Euclidean. For example:

```
# City       X    Y
Berlin     5251 1340
Frankfurt  5011  864
Leipzig    5133 1237
Heidelberg 4941  867
```

```
Karlsruhe  4901  840
Hamburg    5356  998
Bayreuth   4993 1159
Trier      4974  668
Hannover   5237  972
```

```
Stuttgart  4874  909
Passau     4856 1344
Augsburg   4833 1089
Koblenz    5033  759
```

The formulation in ZIMPL follows below and demonstrates the style of the syntax. The input file `tsp.dat` containing the table above is used to initialize the set of nodes `V` and coordinate parameters `px` and `py`. Note that it is sufficient to include the no subtour constraints only for sets containing and excluding at least three nodes.

```
set V               := { read "tsp.dat" as "<1s>" comment "#" };
set E               := { <i,j> in V * V with i < j };
set P[]             := powerset(V);
set K               := indexset(P);

param px[V]         := read "tsp.dat" as "<1s> 2n" comment "#";
param py[V]         := read "tsp.dat" as "<1s> 3n" comment "#";

defnumb dist(a,b) := sqrt((px[a]-px[b])^2 + (py[a]-py[b])^2);

var x[E] binary;

minimize cost: sum <i,j> in E : dist(i,j) * x[i, j];

subto two_connected: forall <v> in V do
   (sum <v,j> in E : x[v,j]) + (sum <i,v> in E : x[i,v]) == 2;

subto no_subtour:
   forall <k> in K with
      card(P[k]) > 2 and card(P[k]) < card(V) - 2 do
      sum <i,j> in E with <i> in P[k] and <j> in P[k] : x[i,j]
     <= card(P[k]) - 1;
```

The user can pass this `zpl` file directly to SCIP or generate an `lp` file using the command `zimpl tsp.zpl`. Alternatively, ZIMPL can produce an `mps` file with `zimpl -t mps tsp.zpl`. A list of command line options is displayed with `zimpl -help`.

Note that `P[]` holds all subsets of the cities and hence the number of cities for which this model can be solved is very limited. For the 13 cities above, the resulting problem has 78 variables, 8 021 constraints, and 154 596 non-zero entries in the constraint matrix. Information on how to solve much larger instances can be found on the CONCORDE website.[2] An optimal tour for the data above is Berlin, Leipzig, Passau,

[2] http://www.tsp.gatech.edu

Augsburg, Stuttgart, Heidelberg, Karlsruhe, Trier, Koblenz, Frankfurt, Hannover, Hamburg, Berlin and can be computed by SCIP within less than one second.

For a precise definition of the ZIMPL syntax and more details and examples on modeling with ZIMPL, we refer to the manual on http://zimpl.zib.de.

4.2 SCIP

In the following section we briefly introducethe usage of SCIP. We describe first steps for using the interactive shell and the callable library of SCIP.

4.2.1 How to use the interactive shell

A first step towards working with SCIP is to use it via the interactive shell as a black box solver for MIPs, MINLPs and other problems. Therefore, a SCIP binary[3] and an example problem file are required. SCIP can read files in LP, MPS, ZPL, WBO, FZN, PIP, and other formats (see Table 1). MIP test instances can, e.g., be found at the MIPLIB homepage.[4] For the remainder of this section, the instance stein27 from MIPLIB 3.0 will serve as an example.

Reading and solving. The SCIP binary can be started without any arguments. This opens the interactive shell. The `help` command shows you a list of all available shell commands. Brackets indicate a submenu with further options.

```
SCIP> help

 <display>            display information
 <set>                load/save/change parameters
 [...]
 read                 read a problem
```

The most important commands to start with are `read <path/to/file>` to parse a problem file, `optimize` to solve it and `display solution` to show the nonzero variables of the best found solution.

```
SCIP> read check/instances/MIP/stein27.mps.gz
original problem has 27 variables (27 bin, 0 int, 0 impl, 0 cont) and 118 constraints
SCIP> optimize

presolving:
(round 1) 0 del vars, 0 del conss, 0 chg bounds, 0 chg sides, 0 chg coeffs, 118 upgd conss, 0 impls, 0 clqs

 time | node  | left  |LP iter|LP it/n|mdpt |frac |cols |rows |cuts |confs|strbr|  dualbound   | primalbound  |  gap
t 0.0s|     1 |     0 |    34 |     - |   0 |  21 |  27 | 118 |   0 |   0 |   0 | 1.300000e+01 | 2.700000e+01 | 107.69%
R 0.0s|     1 |     0 |    34 |     - |   0 |  21 |  27 | 118 |   0 |   0 |   0 | 1.300000e+01 | 2.600000e+01 | 100.00%
s 0.0s|     1 |     0 |    34 |     - |   0 |  21 |  27 | 118 |   0 |   0 |   0 | 1.300000e+01 | 2.500000e+01 |  92.31%
  0.0s|     1 |     0 |    44 |     - |   0 |  21 |  27 | 120 |   2 |   0 |   0 | 1.300000e+01 | 2.500000e+01 |  92.31%
[...]
  0.1s|     1 |     2 |   107 |     - |   0 |  24 |  27 | 131 |  13 |   0 |  24 | 1.300000e+01 | 1.900000e+01 |  46.15%
R 0.1s|    14 |    10 |   203 |   7.4 |  13 |   - |  27 | 124 |  13 |   0 | 164 | 1.300000e+01 | 1.800000e+01 |  38.46%
  0.1s|   100 |    54 |   688 |   5.9 |  13 |  20 |  27 | 124 |  13 |   0 | 206 | 1.300000e+01 | 1.800000e+01 |  38.46%
[...]
SCIP Status        : problem is solved [optimal solution found]
Solving Time (sec) : 0.73
Solving Nodes      : 4192
Primal Bound       : +1.80000000000000e+01 (283 solutions)
Dual Bound         : +1.80000000000000e+01
Gap                : 0.00 %

SCIP> display solution

objective value:                                 18
x0001                                             1   (obj:1)
x0003                                             1   (obj:1)
[...]
```

[3] Download at http://scip.zib.de/download.shtml, available for Linux, Mac, and Windows.

[4] http://miplib.zib.de, the stein27 example file is available at http://miplib.zib.de/miplib3/miplib3/stein27.mps.gz.

Analyzing the ouput. After starting the optimization procedure, SCIP will first do a few rounds of presolving and then start the branch-and-bound search. When the solving process is interrupted by the user or hits a predefined limit, e.g., on the number of branch-and-bound nodes or the time, a short report on the current solution status is printed; equally when the problem is solved completely.

For this particular instance, presolving is very short. The linear constraints merely get upgraded to more specific types. For each round of presolving, a line will report the reductions performed, with a short summary after the last round. Next, we see output for the actual solving process. The first three output lines in the above example indicate that new incumbent solutions were found by the primal heuristics with display characters 't', 'R', and 's'. Due to this, the "primalbound" column shows a reduction of the incumbent solution value from 27 to 25.

In the fourth line, two "cuts" are added. Up to here, SCIP (or rather SoPlex) needed 44 "LP iter"ations: 34 for the first LP and 10 more to resolve after adding cuts. Little later, the root node processing is finished. Note that there are now two open nodes in the "left" column; SCIP has performed a branching on one of the 24 "frac"tional variables. From now on, there will be an output line every hundredth node or whenever a new incumbent is found (e.g. at node 14 in the above example).

After SCIP finishes the optimization process or has been stopped otherwise, `display solution` will print the nonzero values of the incumbent solution.

The exact performance varies amongst different architectures, operating systems, and so on. Thus, different installations of SCIP might need more or less time or nodes to solve. Also, this instance has more than 2000 different optimal solutions. The optimal objective value always has to be 18, but the solution vector may differ. If you are interested in this behavior, which is called performance variability, please see [22].

Accessing detailed statistics. Of course, it is also possible to access more detailed information on the solution process, the problem instance, and different components of the solver. Information on certain plugin types (e.g., heuristics, branching rules, separators) are retrieved by `display <plugin-type>`, information on the solution process is accessed by `display statistics`, and `display problem` shows the current problem instance.

```
SCIP> display heuristics
 primal heuristic     c priority freq ofs  description
 ----------------     - -------- ---- ---  -----------
 [...]
 rounding             R    -1000    1   0  LP rounding heuristic with infeasibility recovering
 shifting             s    -5000   10   0  LP rounding heuristic with infeasibility recovering also using continuous variables
 [...]

SCIP> display statistics
 [...]
 oneopt          :       0.01          4          1
 coefdiving      :       0.02         57          0
 [...]
 primal LP       :       0.00          0          0       0.00          -
 dual LP         :       0.20       4187      14351       3.43   71755.00
 [...]
```

In the above example, rounding and shifting were the heuristics producing the solutions in the beginning, which we can identify by their display characters 'R' and 's'. Rounding is called at every node, shifting only at every tenth level of the tree, see the "freq" column of the above output.

The statistics are quite comprehensive and so in the following we only explain a few lines. There is information for all types of plugins and for the overall solving process. Besides others, we see that the oneopt heuristic found one solution in 4 calls, whereas coefdiving failed all 57 times it was called. All the LPs have been solved with the dual simplex algorithm, which took about 0.2 seconds of the 0.7 seconds overall solving time.

Changing parameters. Having information on the performance of single components, a user might want to change their behavior by altering some parameters. As an example, we will disable the rounding and the shifting heuristic.

```
SCIP> set
  [...]
  <heuristics>          change parameters for primal heuristics
  [...]

SCIP/set> heuristics
  [...]
  <shifting>            LP rounding heuristic with infeasibility recovering also using continuous variables
  [...]

SCIP/set/heuristics> shifting

  <advanced>            advanced parameters
  freq                  frequency for calling primal heuristic <shifting> (-1: never, 0: only at depth freqofs) [10]
  freqofs               frequency offset for calling primal heuristic <shifting> [0]

SCIP/set/heuristics/shifting> freq
current value: 10, new value [-1,2147483647]: -1
heuristics/shifting/freq = -1

SCIP> se he rou freq -1
heuristics/rounding/freq = -1

SCIP> re check/instances/MIP/stein27.mps
original problem has 27 variables (27 bin, 0 int, 0 impl, 0 cont) and 118 constraints
SCIP> o

feasible solution found by trivial heuristic, objective value  2.700000e+01
 [...]
z 0.1s|     3 |     4 |   140 |  10.5 |1060k|   2 |  22 |  27 | 118 |  27 | 123 |  14 |   0 |  66 | 1.300000e+01 | 1.900000e+01 |  46.15%
 [...]
SCIP Status        : problem is solved [optimal solution found]
Solving Time (sec) : 0.75
Solving Nodes      : 4253
Primal Bound       : +1.80000000000000e+01 (287 solutions)
Dual Bound         : +1.80000000000000e+01
Gap                : 0.00 %

SCIP>
```

We can navigate through the menus step-by-step and get a list of available options and submenus. Thus, we select `set` to change settings, `heuristics` to change settings of primal heuristics, `shifting` for that particular heuristic. Then we see a list of parameters (and yet another submenu for advanced parameters), and disable this heuristic by setting its calling frequency to -1. If we already know the path to a certain setting, we can directly type it (as for the rounding heuristic in the above example). Note that we do not have to use the full names, but we may use short versions, as long as they are unique. To solve the problem a second time, we have to read it and start the optimization process again.

To support parameter tuning, SCIP also provides several meta-settings for presolving, heuristics, separation, and the search process, accessible via the `emphasis` menus, see also the FAQ page. Let's consider a second example:

```
SCIP> set default
reset parameters to their default values
SCIP> set heuristics emphasis

  aggressive            sets heuristics <aggressive>
  fast                  sets heuristics <fast>
  off                   turns <off> all heuristics

SCIP/set/heuristics/emphasis> aggr
heuristics/veclendiving/freq = 5
[...]
heuristics/crossover/minfixingrate = 0.5
SCIP> read check/instances/MIP/stein27.mps
original problem has 27 variables (27 bin, 0 int, 0 impl, 0 cont) and 118 constraints

SCIP> opt
[...]
D 0.1s|     1 |     0 |   107 |     - | 971k|   0 |  24 |  27 | 122 |  27 | 131 |  13 |   4 |   0 | 1.300000e+01 | 1.800000e+01 |  38.46%
  0.1s|     1 |     0 |   107 |     - | 971k|   0 |  24 |  27 | 122 |  27 | 131 |  13 |   4 |   0 | 1.300000e+01 | 1.800000e+01 |  38.46%
  0.1s|     1 |     0 |   119 |     - |1111k|   0 |  24 |  27 | 122 |  27 | 132 |  14 |   4 |   0 | 1.300000e+01 | 1.800000e+01 |  38.46%
  0.1s|     1 |     2 |   119 |     - |1112k|   0 |  24 |  27 | 122 |  27 | 132 |  14 |   4 |  24 | 1.300000e+01 | 1.800000e+01 |  38.46%
 time | node  | left  |LP iter|LP it/n| mem |mdpt |frac |vars |cons |cols |rows |cuts |confs|strbr|  dualbound   | primalbound  |  gap
  0.2s|   100 |    59 |   698 |   5.8 |1138k|  14 |  11 |  27 | 122 |  27 | 123 |  14 |   4 | 204 | 1.300000e+01 | 1.800000e+01 |  38.46%
  0.2s|   200 |    91 |  1226 |   5.6 |1155k|  14 |   - |  27 | 122 |  27 | 123 |  14 |   4 | 207 | 1.300000e+01 | 1.800000e+01 |  38.46%
^Cpressed CTRL-C 1 times (5 times for forcing termination)

SCIP Status        : solving was interrupted [user interrupt]
Solving Time (sec) : 0.32
```

```
Solving Nodes      : 216
Primal Bound       : +1.80000000000000e+01 (283 solutions)
Dual Bound         : +1.30000000000000e+01
Gap                : 38.46 %

SCIP>
```

Here, we first reset all parameters to their default values using `set default`. Next, we load the meta settings to apply primal heuristics more aggressively. SCIP shows us which single parameters this changed. Now, the optimal solution is already found at the root node, by a heuristic which is deactivated by default. Then, after node 200, the user pressed `CTRL-C` which interrupts the solving process, We see that now in the short status report, primal and dual bound are different, thus, the problem is not solved yet. Nevertheless, we could access statistics, see the current incumbent solution, change parameters and so on. Entering `optimize`, we continue the solving process from the point at which it had been interrupted.

Writing. SCIP can also write information to files. E.g., we could store the incumbent solution to a file, or output the problem instance in another file format (the LP format is much more human readable than the MPS format, for example).

```
SCIP> write solution stein27.sol

written solution information to file <stein27.sol>

SCIP> write problem stein27.lp
written original problem to file <stein27.lp>

SCIP> q
[...]
```

4.2.2 How to use the callable library

The next step is to use SCIP via the callable library. This means that within your own program code, preferably written in C or C++, you create a SCIP data object, which you can access and modify via calling public functions of the SCIP program library which is written in C. An external user is allowed to call all functions that are declared in `scip.h`, `pub_<...>.h` and all header files of particular plugins, e.g., for the linear constraint handler `cons_linear.h`.

If you are looking for information about a particular object of SCIP, such as a variable or a constraint, you should first search the corresponding `pub_<...>.h` header. E.g., for constraints, look in `pub_cons.h`. If you need some information about the overall problem, you should start searching in scip.h. It contains all "involved" operations that affect or need data from more than one component of SCIP. For these methods, you always have to provide a SCIP pointer, see below.

If you want to use SCIP as a callable library, the first step will always be that you create a SCIP object via `SCIPcreate()`. This returns a SCIP pointer that you need for most other function calls. Then you typically include the default plugins using `SCIPincludeDefaultPlugins()`. Afterwards, you can start to build the problem that you want to solve via `SCIPcreateProb()`.

Next, you might want to create variables via `SCIPcreateVar()` and add them to the problem via `SCIPaddVar()`. The same has to be done for the constraints. For example, if you want to fill in the rows of a general MIP, you have to call `SCIPcreateConsBasicLinear()` (or the more advenced `SCIPcreateConsLinear()`), `SCIPaddCons()` and additionally `SCIPreleaseCons()` after finishing. If all variables and constraints are present, you can initiate the solution process via `SCIPsolve()`. Finally, with `SCIPgetBestSol()` you get the optimal (or incumbent, if aborted prematurely) solution. With the methods `SCIPgetSolVal()`, `SCIPgetSolVals()`, and `SCIPgetSolOrigObj()` you can access the value of a single variable in a solution, the

values of all variables and the objective function value, respectively. Make sure to also call `SCIPreleaseVar()` if you do not need the variable pointer anymore.

Note that most of these methods return a `SCIP_RETCODE`. If everything worked during a function call a `SCIP_OKAY` is returned. In any other case one of the more than 15 error codes is passed to the user and needs to be considered before continuing the program. See the file `type_retcode.h` for a complete list of all error codes used within SCIP.

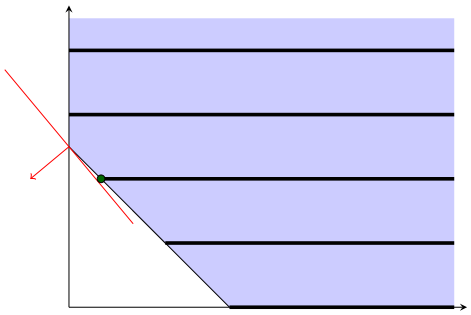

Figure 2: MIP with two constraints and two variables; one continuous and one integer.

Consider the toy MIP example $\min\{1.2x + y \mid x + y \geq 2.5, x \geq 0, y \in \mathbb{Z}_{\geq 0}\}$. This example is illustrated in Figure 2. The light shaded region shows the area corresponding to the LP relaxation; the parallel lines show the mixed integer set of feasible solutions. The line with its normal depicts the objective and the dot shows the optimal solution. Below we show how this problem could be modeled and solved using the SCIP callable library in a C programming environment.

```
#include "scip/scip.h"
#include "scip/scipdefplugins.h"
...
   SCIP* scip = NULL;
   SCIP_VAR* x = NULL;
   SCIP_VAR* y = NULL;
   SCIP_CONS* cons = NULL;
   SCIP_SOL* sol = NULL;

   /* initialize SCIP */
   SCIP_CALL( SCIPcreate(&scip) );
   SCIP_CALL( SCIPincludeDefaultPlugins(scip) );
   SCIP_CALL( SCIPcreateProbBasic(scip, "example") );

   /* create and add variables */
   SCIP_CALL( SCIPcreateVarBasic(scip, &x, "x", 0.0, SCIPinfinity(scip), 1.2, SCIP_VARTYPE_CONTINUOUS) );
   SCIP_CALL( SCIPcreateVarBasic(scip, &y, "y", 0.0, SCIPinfinity(scip), 1.0, SCIP_VARTYPE_INTEGER) );
   SCIP_CALL( SCIPaddVar(scip, x) );
   SCIP_CALL( SCIPaddVar(scip, y) );

   /* create empty constraint, add variables to it and release when done with modification */
   SCIP_CALL( SCIPcreateConsBasicLinear(scip, &cons, "constraint", 0, NULL, NULL, 2.5, SCIPinfinity(scip)) );
   SCIP_CALL( SCIPaddCoefLinear(scip, cons, x, 1.0) );
   SCIP_CALL( SCIPaddCoefLinear(scip, cons, y, 1.0) );
   SCIP_CALL( SCIPaddCons(scip, cons) );
   SCIP_CALL( SCIPreleaseCons(scip, &cons) );

   /* solve the problem and output solution values */
   SCIP_CALL( SCIPsolve(scip) );
   sol = SCIPgetBestSol(scip);
   printf("x: %f y: %f\n", SCIPgetSolVal(scip, sol, x), SCIPgetSolVal(scip, sol, y));

   /* release varibles and free SCIP */
   SCIP_CALL( SCIPreleaseVar(scip, &x) );
   SCIP_CALL( SCIPreleaseVar(scip, &y) );
   SCIP_CALL( SCIPfree(&scip) );
...
```

4.2.3 Implementing plugins

The most involved way to use SCIP is by implementing plugins. These are user defined callback objects that interact with the framework through a very detailed

interface. The plugin concept of SCIP facilitates the implementation of self-contained solver components that can be employed by a user to solve his particular constraint integer programming model.

For each type of plugin, there are template files `<plugintype>_xyz.{c,h}` available in the `src/scip` directory. The steps to implement a user plugin include adjusting the properties, such as the priorities and frequencies with which the plugin will be called by SCIP; providing a public interface method to include this plugin into SCIP; and implementing some of the callback methods. For example, to implement a variable pricer, a user would typically implement the `PRICERREDCOST` and the `PRICERFARKAS` callbacks, which are called during the price-and-cut loop for feasible and infeasible LPs, respectively. Further, there are callbacks to initialize and clean up data structures, as well as a local data structure object for each plugin.

4.2.4 How to debug and run tests

SCIP comes with support for debugging the code and running and evaluating tests automatically.

If you want to debug your own code that uses SCIP– or SCIP itself – the first thing a user should do is to compile it in debug mode, using `make OPT=dbg`. This activates asserts everywhere in the code as well as some additional consistency checking methods. To support debugging, users are strongly recommended to use asserts in their own code as well. If more information on the processing of a certain SCIP component is required, e.g., because it might be related to a bug, placing a `#define SCIP_DEBUG` at the top of the file activates a series of debug output. Again, it is recommended to use `SCIPdebugMessage()` in user's code. It further is possible to add a debug solution at the top of the file `src/scip/debug.h` to ensure that the code does not cut off an optimal solution.

Calling `make TEST=mytestset SETTINGS=mysettings test` starts an automatic test run for all instance files that are stored in `check/testset/mytestset.test`, using the settings stored in `settings/mysettings.set`. Other options such as `TIME`, `NODES`, `MEM` are available. During computation, SCIP automatically stores an output (`*.out`) and a result file (`*.res`) in `check/results`. The latter one reports on the time, branch-and-bound nodes, simplex iterations needed per instance and gives a short summary with total numbers and geometric means taken over all instances in the given testset. Further, the script `check/allcmpres.sh` can be used to compare two or more `*.res` files with respect to many different criteria.

4.2.5 How to find help

Online documentation. SCIP comes with a detailed online documentation accessible under http://scip.zib.de/doc/html/index.html. It features frequently asked questions, instructions for how to use the interactive shell, how to implement a specific type of plugin, or how to find public interface methods and parameters. For advanced users, the source code is linked directly from the online documentation and is very well-commented.

Parameters. SCIP 3.0 and its default plugins provide more than 1400 parameters to influence the solution process. These can be browsed in the `set` menu of the interactive shell and are organised according to functionality:

```
SCIP> set

  <branching>           change parameters for branching rules
  <conflict>            change parameters for conflict handlers
  <constraints>         change parameters for constraint handlers
  <display>             change parameters for display columns
```

```
<emphasis>          predefined parameter settings
<heuristics>        change parameters for primal heuristics
<limits>            change parameters for time, memory, objective value, and other limits
<lp>                change parameters for linear programming relaxations
<memory>            change parameters for memory management
<misc>              change parameters for miscellaneous stuff
<nlp>               change parameters for nonlinear programming relaxations
<nlpi>              change parameters for NLP solver interfaces
<nodeselection>     change parameters for node selectors
<numerics>          change parameters for numerical values
<presolving>        change parameters for presolving
<pricing>           change parameters for pricing variables
<propagating>       change parameters for constraint propagation
<reading>           change parameters for problem file readers
<separating>        change parameters for cut separators
<timing>            change parameters for timing issues
<vbc>               change parameters for VBC tool output
default             reset parameter settings to their default values
diffsave            save non-default parameter settings to a file
load                load parameter settings from a file
save                save parameter settings to a file
```

Many of the parameters are marked as advanced and displayed to the user only when explicitly entering an `<advanced>` menu. A list of all available parameters is available online and can be printed to a file using the shell command `set save`.

Public interface methods. The primary source for information on SCIP's methods and plugins is the online documentation of the corresponding header files, which can be found under the tab *Files*. The public interface methods of the SCIP core are located in `scip.h` and `pub_<...>.h`. The first file contains methods that perform complex operations that affect or need data from different components of SCIP; the other header files contain methods that only affect single objects, e.g., `pub_var.h` contains the method `SCIPvarGetType()` returning the type of a variable.

Additionally, SCIP methods follow a clear naming scheme. Often the beginning of a method's name can be guessed easily and searched for directly in the search box provided on the online documentation's web page.

The file `pub_misc.h` contains methods for data structures like priority queues, hash tables, and hash maps, as well as methods for sorting, numerics, random numbers, string operations, and file operations.

Implementing a plugin. The *How to add ...* section of the online documentation provides step-by-step instructions for how to implement your own SCIP plugin. If you are looking for a description of a callback method of a plugin that you want to implement, you have to look at the corresponding header file `type_<...>.h`, e.g., in `type_cons.h` for a description of the callbacks of a constraint handler.

Example projects. In addition to the general documentation, SCIP 3.0 comes with several examples, which show how to implement one or more plugins for specific problems.[5]

For the implementation of a constraint handler, see the linear ordering example `LOP` (in C) or the `TSP` example, which uses the C++ interface and also implements a primal heuristics. For a branch-and-price code, look at the `Binpacking` or the `Coloring` example, which contain a variable pricer, a branching rule, and show how to store additional data of the problem or of single variables.

Mailing list. For further help, we encourage users to subscribe to the SCIP mailing list `scip@zib.de`. Currently over 200 developers and users are active and try to give instant advice on more specific questions.[6]

[5] http://scip.zib.de/examples.shtml
[6] http://listserv.zib.de/mailman/listinfo/scip

Presentations, papers, etc. Under http://scip.zib.de/further.shtml, the SCIP web page collects introductions, presentations, exercises, and scientific publications about SCIP and CIP.

4.3 SoPlex

SoPlex provides a straightforward command line interface to solve linear programs as `lp` or `mps` files. Calling SoPlex on the small capri instance from the Netlib test set[7] by typing `soplex capri.mps` gives

```
Loading LP file capri.mps
LP has 271 rows 353 columns 1767 nonzeros
LP reading time: 0.00

Solving LP ...
IEQUSC01 Equilibrium scaling LP
IMAISM69 Main simplifier removed 26 rows, 43 columns, 246 nonzeros, 82 col bounds, 0 row bounds
IMAISM74 Reduced LP has 245 rows 310 columns 1521 nonzeros
ISOLVE01 iteration = 0          lastUpdate = 0          value = 0.000000e+00
ISTEEP01 initializing steepest edge multipliers
ISOLVE01 iteration = 180        lastUpdate = 180        value = 3.861101e+03
ISOLVE01 iteration = 311        lastUpdate = 131        value = 4.637225e+03
ISTEEP01 initializing steepest edge multipliers
ISOLVE01 iteration = 313        lastUpdate = 2          value = 4.860032e+03
ISOLVE02 Finished solving (status=OPTIMAL, iters=313, leave=311, enter=2, flips=0, objValue=4.860032e+03)

SoPlex statistics:
Factorizations     :          4
  Time spent       :       0.00
Solves             :        881
  Time spent       :       0.00
Solution time      :       0.02
Iterations         :        313

Solution value is: 2.6900129e+03
```

SoPlex displays the size of the original LP, applies equilibrium scaling to the constraint matrix, removes 43 variables and 26 constraints in presolving, and solves the instance after 313 simplex iterations: 311 dual iterations (`leave`) and two primal iterations (`enter`). The final statistics show that four clean factorizations were performed and 881 linear systems solved. The optimal objective value for the original problem is 2 690, not to be confused with the optimal value for the presolved problem, 4 860, which is displayed during the simplex phase.

Command line options. Calling `soplex` without input displays a list of options to control the solution algorithm. Options `-f` and `-o` set the primal and dual feasibility tolerance, respectively, which are `1e-6` by default. The number of iterations can be limited with `-L`. A time limit in seconds can be specified using `-l`. As an example, to solve to a higher primal feasibility tolerance of 10^{-8}, but use at most 1000 iterations, call `soplex -f1e-8 -L1000 capri.mps`.

The verbosity level can be increased by `-v`. To display not only the optimal objective value, but also the primal and dual solution, the user has to add `-x` and `-y`, respectively. An a posteriori check of the solution quality can be performed with option `-q`.

Writing and reading LP basis files. To obtain basis information, a basis file can be printed adding the option `-bw` and the name of the basis file after the LP file, e.g., `soplex -bw capri.mps capri.bas`. To warmstart from a basis file, use the option `-br`. Note that this automatically deactivates the simplifier of SoPlex.

Column versus row representation. By default, SoPlex uses the standard column-based computational form. For LP instances with more constraints than variables, it may be beneficial to switch to row representation. This often speeds up the linear algebra routines significantly and can be forced by option `-r`.

[7] http://www.netlib.org/netlib/lp

Primal and dual simplex. As explained in Section 2.3, SoPlex implements a composite simplex algorithm, automatically switching between primal and dual iterations. However, the user can specify with which type to use first.

The default starting algorithm depends on whether column or row representation is used. In column representation, SoPlex starts with the dual simplex; option `-e` changes this to primal. In row representation, SoPlex starts with the primal simplex; option `-e` changes this to dual. As an example, to use row representation, but still start with the dual simplex, call `soplex -r -e capri.mps`.

Pricing strategy, scaling, etc. The performance of the simplex algorithm may heavily depend on the pricing algorithm used. With option `-p[0-6]`, different strategies can be tried. The default is `-p4`, steepest edge pricing with unit initial norms.

Other options that may influence the performance or numerical stability are the scaling strategy (`-g`), the starting basis (`-c`), or the ratio test (`-t`).

Class interface. Besides the command line usage, SoPlex can be embedded into third-party software via its C++ class interface. Its object-oriented software design allows to reimplement single algorithmic components such as the pricing strategy or the scaling. For details, see the online documentation available under `http://soplex.zib.de`.

5 Licenses and availability

The SCIP Optimization Suite is available both under the *ZIB Academic License* and customized commercial licenses. The terms of the ZIB Academic License allow researchers at academic institutions to use the software freely for their research activities. Zimpl is additionally available under the LGPL3. For commercial use, specially tailored licenses are available upon request. For more details we refer to the web page of SCIP [44].

All licenses include full access to the source code of SCIP, SoPlex, and Zimpl. This allows researchers complete control over the solution process. The precise knowledge of what is happening in the algorithm and the ability to modify any part of the program is in our view a necessary precondition for successful research. In commercial applications, the availability of the source code grants the user a greater independence compared to closed-source commercial alternatives.

For the three major platforms Linux, Mac, and Windows, pre-compiled binaries can be downloaded from the web pages. If possible, these binaries are statically linked for maximal portability. Further, libraries, in particular DLLs, for Windows are provided for download. For Linux and Mac we provide build systems that allow for easy compilation of the source code.

6 SCIP Optimization Suite in practice

The SCIP Optimization Suite is currently used at more than one hundred universities and academic institutes worldwide both for pure research and industrial cooperations, as well as in many companies, from start-ups to blue chips. Global players like ABB, Google, SAP, and Siemens have licensed SCIP and employ it regularly in their projects.

It is the basis for many projects at the forefront of MIP and MINLP research such as exact integer programming, generic column-generation, and exploiting symmetries in integer programming. The practical applications for which SCIP has been used

successfully are as diverse as multi-layer telecommunication networks, chip design verification, service design in public transport, optimization of gas transport networks, et cetera.

This is only possible because the underlying mathematical algorithms have been implemented with a special focus on software engineering. SCIP, ZIMPL, and SOPLEX come with test suites that cover more than 75% of the program code. Assert statements are used extensively in the code to test preconditions and invariants. Special targets in the Makefile are available for performing valgrind[8] checks and the code is regularly run through FlexeLint,[9] a static program checker. All in all, this results in the high code quality and robustness that is crucial for application within real-world industry projects.

7 Future development

The history of what today constitutes the SCIP Optimization Suite started at the Zuse Institute about 20 years ago. Since then there has been a continuous development of tools and codes for linear and integer programming. Given the huge number of researchers and companies worldwide relying on the SCIP Optimiziation Suite we are confident that this progress will continue. The following is a selection of topics on which the future development of the SCIP Optimization Suite will focus.

Parallelization. With the chip industry switching from increasing clock speed to computing cores, the implementation of algorithms that run efficiently in parallel is one of the practically most important topics. The possibilities of using future Exascale systems to solve a single integer program are investigated in [23]. SCIP is already the basis for one of the most scalable frameworks for mixed integer programming [30].

Exact Integer Programming. For most commercial applications the solutions computed by today's floating-point MIP solvers within small numerical tolerances are perfectly sufficient. The question of exact feasibility and proven optimality[10] is less important than the task of finding good primal solutions as fast as possible. However, for particular classes of problems, especially feasibility problems, safeguarding against numerical inaccuracies is crucial. Further, for research purposes it is desirable that a MIP solution can be proven to be exactly optimal. Implementing a MIP solver that is theoretically exact and shows practically acceptable solution times remains an important task. As described in [11], significant progress has been made in this direction, with many open questions to solve.

MINLP and CP. The progress achieved in general-purpose MIP solving has been extraordinary over the last decades. It is our goal to combine these advances with techniques from the nonlinear optimization and constraint programming communities and hence make them available for further challenging problem classes, in particular MINLP. SCIP will continue to be one of the pioneers in this direction.

Citius, altius, fortius. Solving more and ever bigger instances faster is the all-time objective in mathematical programming and the continued goal of the developers of the SCIP Optimization Suite.

[8] http://www.valgrind.org

[9] http://www.gimpel.com

[10] By "proven" we mean that the problem is solved exactly, i.e., over the rational numbers, as opposed to numerically using floating-point arithmetic and that the optimality gap is reduced to zero, not only some small epsilon.

Acknowledgments

Many thanks to Erika Kovács, Domenico Salvagnin, and Daniel Steffy for their valuable comments. This research has been supported by the DFG Research Center MATHEON *Mathematics for key technologies* in Berlin, http://www.matheon.de.

References

[1] T. Achterberg. *Constraint Integer Programming*. PhD thesis, Technische Universität Berlin, 2007.

[2] T. Achterberg, T. Koch, and A. Tuchscherer. On the effects of minor changes in model formulations. Technical Report 08-29, Zuse Institute Berlin, Takustr. 7, Berlin, 2009. http://opus.kobv.de/zib/volltexte/2009/1115.

[3] E. Althaus, A. Bockmayr, M. Elf, M. Jünger, T. Kasper, and K. Mehlhorn. SCIL – symbolic constraints in integer linear programming. In *Algorithms – ESA 2002*, pages 75–87, 2002.

[4] I. D. Aron, J. N. Hooker, and T. H. Yunes. SIMPL: A system for integrating optimization techniques. In *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, CPAIOR 2004*, volume 3011 of *Lecture Notes in Computer Science*, pages 21–36, 2004.

[5] T. Berthold, S. Heinz, and S. Vigerske. Extending a CIP framework to solve MIQCPs. In J. Lee and S. Leyffer, editors, *Mixed Integer Nonlinear Programming*, volume 154 of *The IMA Volumes in Mathematics and its Applications*, pages 427–444. Springer, 2011.

[6] J. Bisschop and A. Meeraus. On the development of a general algebraic modeling system in a strategic planning environment. *Mathematical Programming Study*, 20:1–29, 1982.

[7] A. Bockmayr and T. Kasper. Branch-and-infer: A unifying framework for integer and finite domain constraint programming. *INFORMS Journal on Computing*, 10(3):287–300, 1998.

[8] A. Bockmayr and N. Pisaruk. Solving assembly line balancing problems by combining IP and CP. Sixth Annual Workshop of the ERCIM Working Group on Constraints, June 2001.

[9] E. Boros and P. L. Hammer. Pseudo-boolean optimization. *Discrete Appl. Math.*, 123(1-3):155–225, Nov. 2002.

[10] M. R. Bussieck and A. Meeraus. General algebraic modeling system (GAMS). In J. Kallrath, editor, *Modeling Languages in Mathematical Optimization*, pages 137–158. Kluwer, 2004.

[11] W. Cook, T. Koch, D. E. Steffy, and K. Wolter. An exact rational mixed-integer programming solver. In O. Günlük and G. J. Woeginger, editors, *IPCO 2011: Proceedings of the 15th International Conference on Integer Programming and Combinatoral Optimization*, volume 6655 of *Lecture Notes in Computer Science*, pages 104–116, 2011.

[12] G. Dantzig. *Linear Programming and Extensions*. Princeton University Press, 1963.

[13] J. J. Forrest and J. A. Tomlin. Updated triangular factors of the basis to maintain sparsity in the product form simplex method. *Mathematical Programming*, 2:263–278, 1972.

[14] R. Fourer, D. M. Gay, and B. W. Kernighan. A modelling language for mathematical programming. *Management Science*, 36(5):519–554, 1990.

[15] R. Fourer, D. M. Gay, and B. W. Kernighan. *AMPL: A Modelling Language for Mathematical Programming*. Brooks/Cole—Thomson Learning, 2nd edition, 2003.

[16] A. Gleixner, D. Steffy, and K. Wolter. Improving the accuracy of linear programming solvers with iterative refinement. Technical report, Zuse Institute Berlin, 2012.

[17] R. E. Gomory. Outline of an algorithm for integer solutions to linear programs. *Bulletin of the American Society*, 64:275–278, 1958.

[18] E. Hebrard, E. O'Mahony, and B. O'Sullivan. Constraint programming and combinatorial optimisation in numberjack. In A. Lodi, M. Milano, and P. Toth, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, volume 6140 of *LNCS*, pages 181–185. Springer, 2010.

[19] J. N. Hooker and M. A. Osorio. Mixed logical/linear programming. *Discrete Applied Mathematics*, 96-97(1):395–442, 1999.

[20] V. Jain and I. E. Grossmann. Algorithms for hybrid MILP/CP models for a class of optimization problems. *INFORMS Journal on Computing*, 13(4):258–276, 2001.

[21] M. Jünger, T. Liebling, D. Naddef, G. L. Nemhauser, W. R. Pulleyblank, G. Reinelt, G. Rinaldi, and L. A. Wolsey, editors. *50 Years of Integer Programming 1958-2008*. Springer, 2009.

[22] T. Koch, T. Achterberg, E. Andersen, O. Bastert, T. Berthold, R. E. Bixby, E. Danna, G. Gamrath, A. M. Gleixner, S. Heinz, A. Lodi, H. Mittelmann, T. Ralphs, D. Salvagnin, D. E. Steffy, and K. Wolter. MIPLIB 2010. *Mathematical Programming Computation*, 3(2):103–163, 2011.

[23] T. Koch, T. Ralphs, and Y. Shinano. Could we use a million cores to solve an integer program? *Mathematical Methods of Operations Research*, 76:67–93, 2012. http://dx.doi.org/10.1007/s00186-012-0390-9.

[24] A. H. Land and A. G. Doig. An automatic method of solving discrete programming problems. *Econometrica*, 28(3):497–520, 1960.

[25] A. Lodi. MIP computation. In M. Jünger, T. Liebling, D. Naddef, G. Nemhauser, W. Pulleyblank, G. Reinelt, G. Rinaldi, and L. Wolsey, editors, *50 Years of Integer Programming 1958-2008*, pages 619–645. Springer, 2009.

[26] L. Michel and P. V. Hentenryck. The comet programming language and system. In P. van Beek, editor, *Principles and Practice of Constraint Programming - CP 2005, 11th International Conference*, volume 3709 of *LNCS*, pages 881–881, 2005.

[27] P. Refalo. Tight cooperation and its application in piecewise linear optimization. In *Principles and Practice of Constraint Programming, CP 1999*, volume 1713 of *Lecture Notes in Computer Science*, pages 375–389, 1999.

[28] R. Rodosek, M. G. Wallace, and M. T. Hajian. A new approach to integrating mixed integer programming and constraint logic programming. *Annals of Operations Research*, 86(1):63–87, 1999.

[29] A. Schrijver. *Combinatorial Optimization*. Springer, 2003.

[30] Y. Shinano, T. Achterberg, T. Berthold, S. Heinz, and T. Koch. ParaSCIP – a parallel extension of SCIP. In C. Bischof, H.-G. Hegering, W. E. Nagel, and G. Wittum, editors, *Competence in High Performance Computing 2010*, pages 135–148. Springer, February 2012.

[31] L. M. Suhl and U. H. Suhl. A fast LU update for linear programming. *Annals of Operations Research*, 43:33–47, 1993.

[32] C. Timpe. Solving planning and scheduling problems with combined integer and constraint programming. *OR Spectrum*, 24(4):431–448, November 2002.

[33] S. Vigerske. *Decomposition in Multistage Stochastic Programming and a Constraint Integer Programming Approach to Mixed-Integer Nonlinear Programming*. PhD thesis, Humboldt-Universität zu Berlin, 2012. Submitted.

[34] R. Wunderling. *Paralleler und objektorientierter Simplex-Algorithmus*. PhD thesis, Technische Universität Berlin, 1996.

[35] FICO. Xpress Optimization Suite. http://www.fico.com/en/Products/DMTools/Pages/FICO-Xpress-Optimization-Suite.aspx.

[36] GNU. The GNU Linear Programming Kit. http://www.gnu.org/software/glpk.

[37] Guillaume Sagnol. PICOS: A python interface for conic optimization solvers. http://picos.zib.de.

[38] H. Mittelmann. Decision tree for optimization software: Benchmarks for optimization software. http://plato.asu.edu/bench.html.

[39] IBM ILOG CPLEX Optimizer. http://www.ilog.com/products/cplex.

[40] J. J. Forrest and R. Lougee-Heimer. COIN branch and cut, user guide. http://www.coin-or.org/Cbc.

[41] R. Bixby and Z. Gu and E. Rothberg. Gurobi optimization. http://www.gurobi.com.

[42] R. J O'Neil. PYTHON-ZIBOPT. http://code.google.com/p/python-zibopt.

[43] SAGE. A free open-source mathematics software system. http://www.sagemath.org.

[44] SCIP. Solving Constraint Integer Programs. http://scip.zib.de.

[45] SoPlex. The sequential object-oriented simplex. http://soplex.zib.de.

[46] V. Manquinho and O. Roussel. Pseudo-boolean competition 2012. http://www.cril.univ-artois.fr/PB12.

[47] ZIMPL. Zuse Institute Mathematical Programming Language. http://zimpl.zib.de.