Konrad-Zuse-Zentrum
für Informationstechnik Berlin

Takustraße 7
D-14195 Berlin-Dahlem
Germany

TOBIAS ACHTERBERG

SCIP - a framework to integrate Constraint and Mixed Integer Programming

URL: http://www.zib.de/projects/integer-optimization/MIP

ZIB-Report 04-19 (June 2004)

SCIP - a framework to integrate Constraint and Mixed Integer Programming

Tobias Achterberg*

January 2, 2005

Abstract

Constraint Programs and Mixed Integer Programs are closely related optimization problems originating from different scientific areas. Today's state-of-the-art algorithms of both fields have several strategies in common, in particular the branch-and-bound process to recursively divide the problem into smaller subproblems. On the other hand, the main techniques to process each subproblem are different, and it was observed that they have complementary strengths.

We present the programming framework SCIP that integrates techniques from both fields in order to exploit the strengths of both, Constraint Programming and Mixed Integer Programming. In contrast to other proposals of recent years to combine both fields, SCIP does not focus on easy implementation and rapid prototyping, but is tailored towards expert users in need of full, in-depth control and high performance.

Keywords: Mixed Integer Programming, MIP, Constraint Programming, CP, branch-and-bound

1 Introduction

In recent years, it was shown that combining techniques from Constraint Programming and Integer Programming can help to solve problems, that were intractable with either of the two methods. For example, Timpe applied a hybrid approach to solve chemistry industry planning problems that include lot-sizing, assignment, and sequencing as subproblems [49]. He used the CHIP C++ library [18] for the CP part and Dash's XPRESS-MP library [17] for the IP part. Other examples of successful integration include the assembly line balancing problem [12] and the parallel machine scheduling problem [29].

There are already different approaches to integrate Constraint and Integer Programming into a single framework. Bockmayr and Kasper developed

*Konrad-Zuse-Zentrum für Informationstechnik Berlin, achterberg@zib.de

the framework COUPE [11], that unifies CP and IP by observing that both techniques rely on branching and inference. In this setting, cutting planes and domain propagation are just specific types of inference. Althaus et al. propose the system SCIL, which introduces symbolic constraints on top of mixed integer programming solvers [4]. Aron et al. developed SIMPL [6], a system for integrated modelling and solution. They view both, CP and IP, as a special case of an infer-relax-restrict cycle in which CP and IP techniques closely interact at any stage.

This paper introduces the Constraint Integer Programming framework SCIP, which is is oriented towards the needs of Constraint and Mathematical Programming experts who want to have total control of the solution process and access to detailed information down to the guts of the solver. It includes the following features:

- It is a framework for branching, cutting, pricing and propagation.
- It is highly flexible through many possible user plugins:
  - constraint handlers to implement arbitrary constraints,
  - variable pricers to dynamically create problem variables,
  - domain propagators to apply constraint independent domain propagations,
  - cut separators to apply cutting planes on the LP relaxation,
  - relaxators to provide relaxations and dual bounds in addition to the LP relaxation,
  - primal heuristics to search for feasible solutions with specific support for probing and diving,
  - node selectors to guide the search,
  - branching rules to split the problem into subproblems,
  - presolvers to simplify the solved problem,
  - file readers to parse different input file formats,
  - event handlers to be informed on specific events, e. g., when a node was solved, a specific variable changed its bounds, or a new primal solution was found,
  - display handlers to create additional columns in the solver's output.
  - dialog handlers to extend the included command shell.
- Every existing unit is implemented as a plugin, leading to an interface flexible enough to meet the needs of most additional user extensions.
- A dynamic cut pool management is included.

- The user may mix preprocessed and active problem variables in expressions: they are automatically transformed onto corresponding active problem variables.
- Arbitrarily many children per node can be created, and the different children can be arbitrarily defined.
- It has open LP solver support (currently supporting CPlex [27], SoPlex [51], and CLP [21]),
- The LP relaxation need not to be solved at every single node (it can even be turned off completely, mimicing a pure Constraint Solver).
- Additional relaxations (e. g., semidefinite relaxations or Lagrangian relaxations) can be included, working in parallel or interleaved.
- Conflict analysis can be applied to learn from infeasible subproblems.
- Dynamic memory management reduces the number of operation system calls with automatic memory leakage detection in debug mode.

The remaining part of the paper is organized as follows. Section 2 compares the Constraint Programming and Mixed Integer Programming problems and the different techniques to solve them. An intermediate problem class called *Constraint Integer Program* is defined. Section 3 introduces the basic concepts of our CIP framework Scip and describes the different types of external plugins that can be included to extend Scip's functionality. Section 4 illustrates the algorithmic design of Scip and describes how the framework and the external plugins interact to solve a CIP instance. Section 5 gives computational results on some MIP instances and compares Scip with an academic and a state-of-the-art commercial MIP solver.

2 Comparison CP/MIP

In this section, we introduce the definitions of Constraint Programs and Mixed Integer Programs and derive a problem class which we call *Constraint Integer Program*. We briefly present and compare the basic solution strategies of both fields and highlight the key ideas that make the two approaches efficient in praxis.

2.1 Constraint Programming

The optimization version of a Constraint Program (CP) can be defined as follows:

Definition 2.1 (Constraint Program) *A* Constraint Program *is a triple $CP = (\mathcal{C}, \mathcal{D}, f)$ and consists of solving*

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

with $\mathcal{D} = D_1 \times \ldots \times D_n$ representing the domains of finitely many variables $x_j \in D_j$, $j = 1, \ldots, n$, $\mathcal{C} = \{C_1, \ldots, C_m\}$ being a finite set of constraints $C_i : \mathcal{D} \to \{0, 1\}$, $i = 1, \ldots, m$, and $f : \mathcal{D} \to \mathbb{R}$ being the objective function. For $x \in \mathcal{D}$ we define $\mathcal{C}(x) :\Leftrightarrow \forall i \in \{1, \ldots, m\} : C_i(x) = 1$.

Note that there are no further restrictions on the constraint predicates $C_i \in \mathcal{C}$ and the objective function f.

Existing Constraint (Logic) Programming solvers like Prolog III [15], Clp($\mathcal{R}$) [28], Cal [3], Chip [18], or ILOG Solver [42], are usually restricted to Finite Domain Constraint Programming (CP(FD)). In this setting, all domains $D_1, \ldots, D_m$ have to be finite.

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

The key element for solving Constraint Programs in praxis is the efficient implementation of domain propagation algorithms which exploit the structure of the involved constraints. A CP solver usually includes a library of constraint types with specifically tailored propagators.

Another important feature is the provided infrastructure for managing the local domains and representing the subproblems in the tree. Currently, two different techniques are used in existing software: trailing and copying (see [46] for a comparison). For each node in the tree, trailing stores only the differences of the node and its parent with respect to the variables' domains and the current set of active constraints. This reduces the memory consumption with the cost of additional effort for switching between subproblems. On the other hand, copying physically duplicates the data of the parent node to define the child nodes and applies the necessary modifications to the child nodes afterwards. This can result in a large memory overhead but allows for fast switching between subproblems.

2.2 Mixed Integer Programming

A Mixed Integer Program (MIP) is defined as:

Definition 2.2 (Mixed Integer Program) *Given a matrix $A \in \mathbb{R}^{m \times n}$, vectors $b \in \mathbb{R}^m$ and $c \in \mathbb{R}^n$, and a subset $I \subseteq N = \{1, \ldots, n\}$, the* Mixed Integer Program *$MIP = (A, b, c, I)$ is to solve*

$$(MIP) \quad c^\star = \min\{c^T x \mid Ax \leq b, x \in \mathbb{Z}^I \times \mathbb{R}^{N \setminus I}\}.$$

Common restrictions of MIP are Integer Programs (IPs) with $I = N$ and Binary Programs (BPs) with $I = N$ and $0 \leq x \leq 1$. Note that in contrast to CP, we are now restricted to

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

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

Just like CP solvers, modern MIP solvers recursively split the problem into smaller subproblems, thereby generating a branching tree. However, the processing of the nodes is different. For each node of the tree, the LP relaxation is solved, which can be constructed from the MIP by removing the integrality conditions. The relaxation can be strengthened by cutting planes which use the LP information and the integrality restrictions to derive valid inequalities cutting off the optimal LP solution without removing integral solutions. The LP provides a lower bound for the whole subtree, and if this bound exceeds the value of the currently best primal solution, the node and its subtree can be discarded. The LP relaxation usually gives a much stronger bound than the simple dual propagation of CP solvers can provide.

The most important ingredients of a MIP solver implementation are a fast and numerically stable LP solver, cutting plane separators, primal heuristics, and presolving algorithms (see [10]). Additionally, the applied branching rule is of major importance (see [2]). Necessary infrastructure includes the management of subproblem modifications, LP warmstart information, and a cut pool.

Modern MIP solvers like CBC [20], CPLEX [27], LINDO [34], MINTO [40, 39], SIP [36], SYMPHONY [43], or XPRESS [17] offer a variety of different general purpose separators, that can be activated for solving the problem instance at hand (see [7]). It is also possible to add problem specific cuts through callback mechanisms, thus providing some of the flexibility a full MIP framework offers. These mechanisms are in many cases sufficient to solve a given problem instance. With the help of modelling tools like Ampl [22], Gams [13], or Zimpl [30] it is sometimes even possible to formulate the model in a mathematical fashion, automatically transform the model and data into solver input and solve the instance within reasonable time. In this setting, the user does not need to know the internals of the MIP solver, which is used as black box tool.

Unfortunately, this rapid mathematical prototyping chain (see [31]) does not yield results in acceptable solving time for every problem class, sometimes not even for small instances. For these problem classes, the user has to develop special purpose code with problem specific algorithms. To provide him with the necessary infrastructure like the branching tree and LP management, or to support him with standard general purpose algorithms like LP based cutting plane separators or primal heuristics, he can use a MIP framework like Abacus [48] or the tools provided by the Coin project [14].

2.3 Constraint Integer Programming

As described in the previous sections, current solvers for Constraint Programming and Mixed Integer Programming share the idea of dividing the problem into smaller subproblems and implicitly enumerating all potential solutions. They differ in the way of processing the subproblems. Because MIP is a very specific restriction of CP, MIP solvers can apply sophisticated problem specific algorithms, that operate on the subproblem as a whole, in particular the simplex algorithm [16] to solve the LP relaxations, and cutting plane separators like the Gomory cut separator [23].

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

The hope of combining CP and MIP techniques is to take advantage of both strengths and to compensate for the different weaknesses. We propose the following slight restriction of a CP:

Definition 2.3 (Constraint Integer Prograim) *The* Constraint Integer Program *$CIP = (\mathcal{C}, I, c)$ consists of solving*

$$(CIP) \quad c^\star = \min\{c^T x \mid \mathcal{C}(x), x \in \mathbb{Z}^I \times \mathbb{R}^{N \setminus I}\}$$

with a finite set $\mathcal{C} = \{C_1, \ldots, C_m\}$ of constraints $C_i : \mathbb{Z}^I \times \mathbb{R}^{N \setminus I} \to \{0,1\}$, $i = 1, \ldots, m$, a subset $I \subseteq N = \{1, \ldots, n\}$ of the variable index set, and an objective function vector $c \in \mathbb{R}^n$, and has to fulfill the following restriction:

$$\forall x_I \in \mathbb{Z}^I \ \exists (A_{x_I}, b_{x_I}) : \{x_N \mid A_{x_I} x_N \leq b_{x_I}\} = \{x_N \mid \mathcal{C}(x_I, x_N)\}. \qquad (1)$$

Restriction (1) ensures, that the remaining subproblem after fixing the integral variables is always a Linear Program. This means, that the problem can be completely solved by enumerating all values of the integral variables and solving the corresponding LPs.[1]

The linearity restriction of the objective function can easily be compensated by introducing an auxiliary objective variable that is linked to the actual non-linear objective function with a non-linear constraint. We just demand a linear objective function in order to simplify the derivation of the LP relaxation. The same holds true for omitting the general variable domains $\mathcal{D}$, that exist in Definition 2.1 of the Constraint Program. They can also be represented as additional constraints. Therefore, every CP that meets condition (1) can be represented as Constraint Integer Program.

In the remaining part of the paper, we will describe SCIP, a framework to solve CIPs.

3 Basic Concepts of SCIP

SCIP is a framework for Constraint Integer Programming and provides the necessary infrastructure to implement algorithms for solving CIPs. It manages the branching tree along with all subproblem data, it automatically updates the LP relaxations, and handles all necesarry transformations due to the preprocessing problem modifications. Additionally, a cut pool and pricing store management and a SAT like conflict analysis mechanism (see [47]) is available. SCIP provides an efficient memory allocation shell, which also includes a simple leak detection if compiled in debug mode. Finally, a lot of statistical output can be generated to support the user's diagnosis of his algorithms, in particular the branching tree can be visualized with the help of VBC TOOL [32].

Despite the infrastructure mentioned above, all the main algorithms are part of external *plugins*. These are user defined callback objects that interact with the framework through a very detailed interface. The current distribution of SCIP already contains necessary plugins to solve MIPs (see Section 5 for computational results on some MIP instances). In the following, we will describe the different plugin types and their role in solving a CIP.

[1] Note that this does not forbid quadratic or even more involved expressions. Only the remaining part after fixing (and thus eliminating) the integral variables must be linear in the continuous variables.

3.1 Constraint Handlers

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

The primary task of a constraint handler is to check a given solution for feasibility with respect to all constraints of its type existing in the problem instance. This feasibility test already suffices to produce a correct algorithm for solving CIPs with constraints of the supported type. However, the algorithm would resemble a complete enumeration of all potential solutions, because no additional primal information would be available.

To help pruning the search tree, constraint handlers may provide additional information about its constraints to the framework, namely

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

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

$$a^T x \le b \tag{2}$$

with positive integral right hand side $b \in \mathbb{Z}_+$, positive integral coefficients $a_j \in \mathbb{Z}_+$ and binary variables $x_j \in \{0, 1\}$.

The feasibility test of the knapsack constraint handler is very simple: it only adds up the coefficients a_j of variables x_j set to 1 in the given solution and compares the result with the right hand side b. Presolving algorithms for knapsack constraints include modifying the coefficients and right hand side in order to tighten the LP relaxation, and fixing variables with $a_j > b$ to 0 (see [45]).

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

The linear relaxation of the knapsack constraint initially consists of the knapsack inequality (2) itself. Additional cutting planes like lifted cover cuts [8, 9, 37] or GUB cover cuts [50] are dynamically generated to enrich the knapsack's relaxation and cut off the current LP solution.

3.2 Presolvers

In addition to the constraint based (primal) presolving mechanisms provided by the individual constraint handlers, additional presolving algorithms can be applied with the help of *presolvers*. They usually perform dual presolving operations, taking the objective function into account.

For example, if the value of a variable x_j can always be decreased without rendering any constraint infeasible (an information, the constraint handlers have to provide), and if the objective value c_j of the variable is non-negative, the *dual fixing presolver* fixes the variable to its lower bound. In the setting of a MIP, this condition is satisfied, iff $A_{\cdot j} \geq 0$ and $c_j \geq 0$.[2]

3.3 Cut Separators

In SCIP, we distinct two different types of cutting planes. The first type are the constraint based cutting planes, that are valid inequalities or even facets of the polyhedron described by a single constraint or a subset of the constraints. The may also be strengthened by lifting procedures that take more information about the full problem into account. These cutting planes are generated by the constraint handlers of the corresponding constraint types. Prominent examples are the different types of knapsack cuts that are generated in the knapsack constraint handler, or the cuts for TSP tours like subtour elimination and comb inequalities [24, 25] that are separated by the tour constraint handler.

The second type of cutting planes are general purpose cuts, which are using the current LP relaxation and the integrality conditions to generate valid inequalities. Generating those cuts is the task of the *cut separators*. Examples are Gomory fractional cuts [23], complemented mixed integer rounding cuts [35], and strong Chvátal-Gomory cuts [33].

3.4 Domain Propagators

Constraint based (primal) domain propagation algorithms are part of the corresponding constraint handlers. For example, the *alldifferent*[3] constraint handler excludes certain values of the variables' domains with the help of a bipartite matching algorithm.

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

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

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

[2] with $A_{\cdot j}$ being the j'th column of the coefficient matrix A

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

3.5 Branching Rules

In SCIP, even the integrality conditions are enforced by an external plugin, the *integrality constraint handler*. However, they slightly differ from the other constraints in the way, their data is stored (i. e., which variables should take integral values): the integrality restriction is attached directly to the variables and is therefore globally available to all algorithms.

The integrality constraint handler has to make sure, that no solution is accepted which contains integral variables with fractional values. If the current LP solution is fractional, the integrality restriction is enforced by branching. This branching is performed by calling the *branching rules*.

A branching rule usually creates two subproblems by splitting a single variable's domain. If applied on a fractional LP solution, commonly an integral variable x_j with fractional value $\bar{x}_j$ is selected, and the two branches $x_j \leq \lfloor \bar{x}_j \rfloor$ and $x_j \geq \lceil \bar{x}_j \rceil$ are created. The well known *most infeasible*, *pseudocost*, *reliability* and *strong branching* rules are examples of this type (see [2]). However, it is also possible to implement much more general branching schemes, for example by creating more than two subproblems, or by adding additional constraints to the subproblems instead of tightening a variable's domain.

3.6 Variable Pricers

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

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

3.7 Primal Heuristics

Feasible solutions can be found in two different ways during the traversal of the branching tree. On the one hand, the solution of a node's relaxation may be feasible. On the other hand, feasible solutions can be discovered by *primal heuristics*. They are called periodically during the search.

SCIP provides specific infrastructure for diving and probing heuristics. *Diving heuristics* iteratively resolve the LP after making a few changes to the current subproblem, usually aiming at driving the fractional values of integral variables to integrality. *Probing heuristics* call the domain propagation algorithms of the constraint handlers after applying changes to the variables' domains. Other heuristics without special support in SCIP include local search heuristics like *tabu search*, and *rounding heuristics*, which try to round the current fractional LP solution to a feasible integral solution.

3.8 Node Selectors

Node selectors decide, which of the leaves in the branching tree is selected as next subproblem to be processed. This choice can have huge impact on the solver's performance, because it influences the finding of feasible solutions.

Constraint Programming was originally developed for *Constraint Satisfaction Problems* (CSPs). These are CPs without objective function. In this setting, the solver only has to find out whether there is a feasible solution or not. Therefore, many of the available CP solvers employ *depth first search*.

With the addition of an objective function, depth first search is usually an inferior strategy. It tends to evaluate many nodes in the tree, that could have been discarded if the optimal solution was known earlier. In Mixed Integer Programming, several node selection strategies are known, that try to discover good feasible solutions early during the search process. Examples of those strategies are *best first* and *best estimate* search.

3.9 Relaxators

SCIP provides specific support for LP relaxations: constraint handlers possess virtual methods for generating the LP, additional cut separators may be included to further tightening the LP relaxation, and there exist a lot of interface methods to access the LP information at the current subproblem.

In addition, it is also possible to include other relaxations, e. g., Lagrange relaxations or semidefinite relaxations. This is possible through *relaxator* objects. The relaxator manages the necessary data structures and calls the relaxation solver to generate dual bounds and primal solution candidates. However, the data to define a single relaxation must either be extracted by the relaxator itself (e. g., from the user defined problem data, the LP information, or the integrality conditions), or be provided by the constraint handlers. In the latter case, the constraint handlers have to be extended to support this specific relaxation.

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

3.10 Event Handlers

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

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

An *event handler* is a special plugin that is called to process events of selected types. This handler usually passes the information to other objects, e. g., to a constraint handler. It is very common in SCIP, that a constraint handler closely interacts with an event handler in order to improve its own run time performance.

3.11 Conflict Handlers

Current state-of-the-art SAT solvers employ analysis of infeasible subproblems to generate so-called *conflict clauses* (see [47]). These are induced constraints that may help to prune the branching tree at other nodes. In the CP community, a generalization of those clauses is known as *no-goods*.

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

3.12 File Readers

File readers are called to parse an input file and generate a CIP model. It creates constraints and variables, and activates variable pricers if necessary. Each file reader is hooked to a single file name extension. It is automatically called if the user wants to read in a problem file of corresponding name. Examples of file formats are the *MPS format* [26] and *LP format* [27] for Linear and Mixed Integer Programs, the *CNF format* for SAT instances in conjunctive normal form, and the *TSP format* [44] for traveling salesman tour instances.

3.13 Dialog Handlers

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

3.14 Display Columns

While solving a Constraint Integer Program, SCIP displays status information in a column-like fashion. For example, the current number of processed branching tree nodes, the solving time, and the relative gap between primal and dual bound are three of those *display columns*. There already exist a wide variety of display columns which can be activated or deactivated on demand. Additionally, the user can implement his own display columns in order to track problem or algorithm specific values.

4 Algorithmic Design

This section focuses on the algorithmic design of SCIP. During the execution of SCIP, different *operational stages* are distinguished (see Figure 1). These stages are described below, and it is specified which callback methods of the different plugins are executed and which operations the user may perform during the different stages. It is explained, how the problem is represented in SCIP's data structures, and which transformations are being applied during the course of the algorithm.

4.1 Init Stage

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

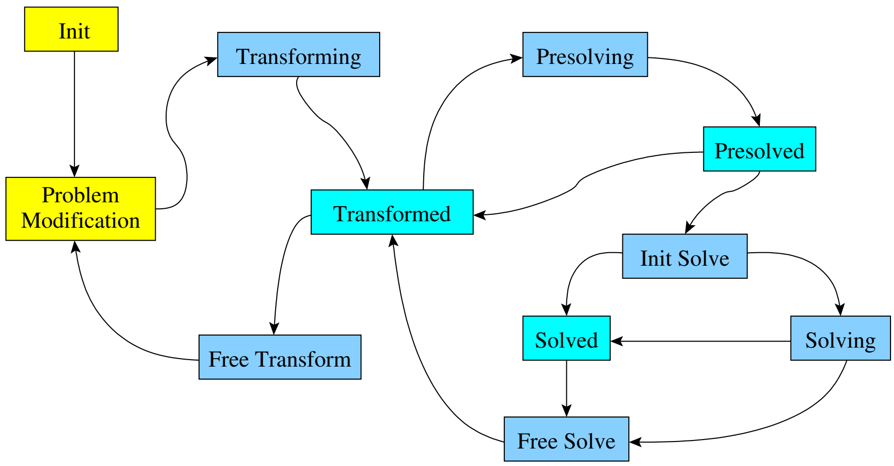

Figure 1. Operational stages of SCIP. The arrows represent possible transitions between stages.

4.2 Problem Modification Stage

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

4.3 Transforming Stage

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

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

4.4 Transformed Stage

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

4.5 Presolving Stage

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

One of the main tasks of presolving is to detect fixings and aggregations of variables. These variables can be deleted from the problem by replacing their occurences in the constraints with the corresponding counterpart. The fixings and aggregations are stored as variable aggregation graph, which is used by the framework to automatically convert any operations on those variables to equivalent ones on active problem variables.

Example 4.1 Consider the linear constraints

$$
\begin{aligned}
3x_1 \qquad\qquad\quad &= 9 \qquad (3)\\
2x_1 + 4x_2 - x_3 \quad &= 0 \qquad (4)\\
x_3 + x_4 &= 1 \qquad (5)
\end{aligned}
$$

on integer variables x_1, x_2, x_3, and x_4. The presolving of constraint (3) fixes $x_1 = 3$. The linear constraint handler then replaces the occurence of x_1 in (4) with its fixed value, resulting in $4x_2 - x_3 = -6$. Now, x_3 can be aggregated to $x_3 = 4x_2 + 6$. Constraint (5) inserts the aggregation $x_4 = 1 - x_3$ into the aggregation graph.

Figure 2 shows the complete aggregation graph of this example. On the left hand side, the original problem variables created in the *problem modification stage* are shown. They are linked to their transformed problem counterpart. Additional links between transformed variables are introduced by aggregations. Assume now, some constraint handler or cut separator adds the inequality

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

to the LP relaxation. This inequality is constructed out of a mixture of original, fixed, aggregated, and active problem variables, and is thereby automatically transformed onto active problem variables. This results in the inequality

$$8x_2 + 11 \leq 23,$$

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

Constraint handlers may also upgrade their constraints to a more specific constraint type. For example, the linear constraint handler provides an upgrading mechanism for its constraints

$$l \leq a^T x \leq r.$$

Other constraint handlers can be hooked into this mechanism to be called for converting linear constraints into constraints of their own type. For example,

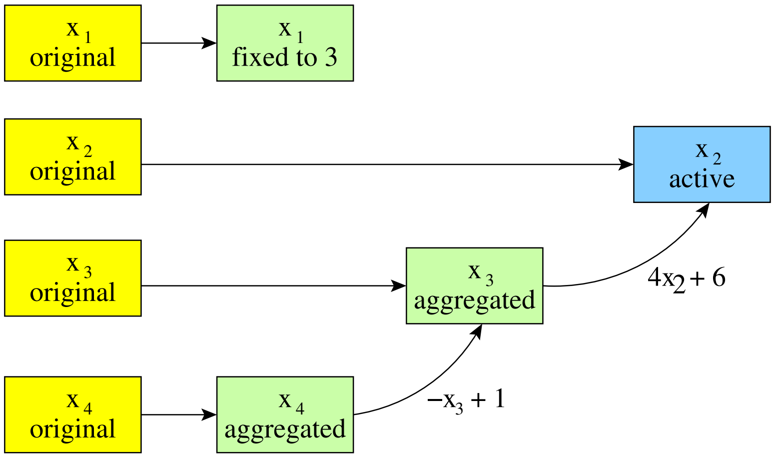

Figure 2. Variable aggregation graph of Example 4.1.

the knapsack constraint handler (see Example 3.1) checks whether the linear constraint consists of only binary variables, integral weights, and only one finite side l or r. If the check succeeds, the linear constraint is converted into a knapsack constraint, possibly by negating some of the binary variables or inverting the inequality.

4.6 Presolved Stage

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

4.7 Init Solve Stage

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

4.8 Solving Stage

If the problem was not already solved in the *presolving stage*, the branch and bound process to implicitly enumerate the potential solutions is performed in the *solving stage*. This stage contains the main solving loop of SCIP which

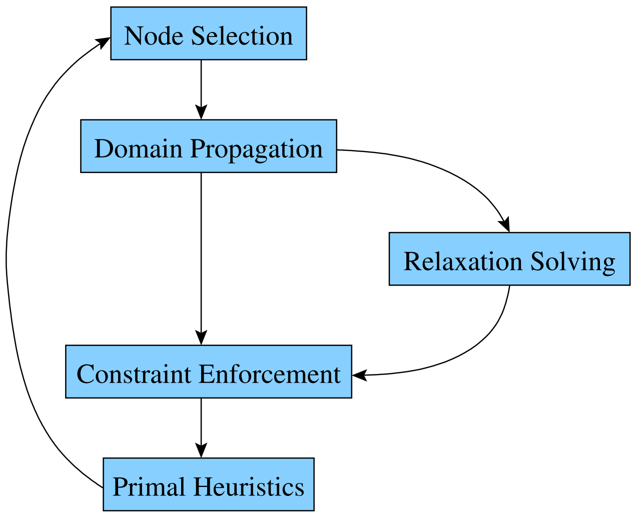

Figure 3. Main solving loop of the *solving stage*.

consists of five different steps that are called successively until the problem is solved or the solving process is interrupted (see Figure 3).

4.8.1 Node Selection

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

Successively choosing a child or sibling of the current node is called *plunging*. Selecting the best leave of the tree ends the current plunging sequence and starts the next one. During plunging, the set up of the subproblems to be processed is computationally less expensive, since the children and siblings are most likely to be closely related to the current node. Switching to the best leave of the tree is more expensive, but has the advantage that the search can be brought to regions in the search space that are more promising to contain a good feasible solution. Efficient node selectors for MIP employ a mixture of plunging and best first search.

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

4.8.2 Domain Propagation

After a node is selected to be processed and the corresponding subproblem is set up, the domain propagators and the domain propagation methods of the constraint handlers are called to tighten the variables' local domains. This propagation is applied iteratively until no more reductions are found or a propagation limit set by the user is reached.

Domain propagation need not to be applied at every node. Every constraint handler and domain propagator can decide itself, if it wants to spend the effort trying to tighten the variables' domains.

4.8.3 Relaxation Solving

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

In SCIP we make the following notational distinctions between the CIP subproblem and its LP relaxation. The CIP consists of *variables* and *constraints*. The variables are marked to be integer or continuous, and their domains may contain holes. The constraints are stored in constraint handler specific data structures. Their semantics is unknown to the framework and only implicitly given by the actions performed in the constraint handlers' callback methods.

The LP relaxation consists of *columns* and *rows*. For each column, the lower and upper bounds are known. Every column belongs to exactly one CIP variable, but not every CIP variable needs to be represented by a column in the LP. The rows are defined as linear combinations of columns and have left and right hand sides as additional data. A single constraint can give rise to multiple rows in the LP, but rows can also live on their own, e. g., if they were created by a general purpose separator.

The LP solving cosists of an inner loop as can be seen in Figure 4. It is executed as long as changes to the LP were applied in the separation or reduced cost strengthening steps. Note that resolving the LP after adding cuts or modifying the columns' bounds can be efficiently done with the dual simplex algorithm.

Calling LP Solver. The first step is to call the LP solver to solve the initial LP relaxation of the subproblem. In the root node, this is defined by

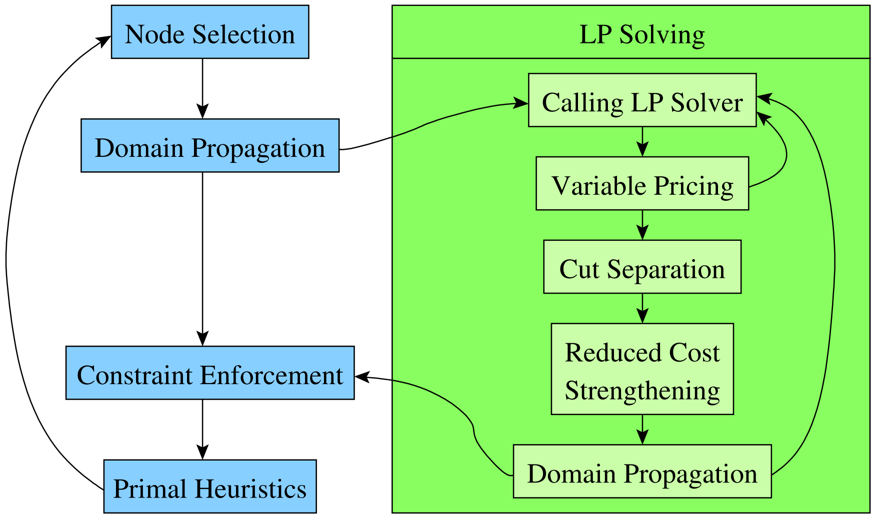

Figure 4. LP solving loop.

the relaxations of constraints that are marked to be *initial*: the constraint handlers are asked to enrich the LP with rows that correspond to their *initial* constraints before the first LP is solved. The initial LP relaxation of a subsequent node equals its parent's relaxation modified by the additional bound changes of the node. Note that branching on constraints does not change the LP relaxation of the child nodes directly. It only modifies the CIP subproblem, and the corresponding constraint handlers may then modify the LP in their cut separation or constraint enforcement methods (see Section 4.8.4 below).

The LP is solved with the primal or dual simplex algorithm, depending on the feasibility status of the current basis. It is also possible to use an interior point method like the barrier algorithm to solve the LP relaxations, if provided by the included LP solver. The resulting LP solution is checked for stability. In a numerically unstable situation, different LP solver parameter settings are tried in order to achieve a stable solution. If this fails, the LP relaxation of the current subproblem is discarded, and the solving process continues as if the LP was not solved at the current node. Note that this is a valueable feature for solving numerically difficult problems. Due to the fact, that SCIP does not need to solve the LP at every node, it can easily leap over numerical troubles in the LP solver without having to abandon the whole solving process.

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

The pricing is performed in rounds. In each round, several new variables are created, and after each round, the LP solver is called again to resolve the relaxation. Note that the primal simplex algorithm can be used to quickly resolve the LP after new columns have been added.

Cut Separation. After the pricing was performed and the LP was resolved, the cut separators and the separation methods of the constraint handlers are called to tighten the LP relaxation with additional cutting planes. In each iteration of the LP solving loop, cutting planes are collected in a separation store, and only some of them are added to the LP afterwards. The selection of the cuts to be added to the LP is a crucial decision which affects the performance and the stability of the LP solver in the subsequent calls. In SCIP, the cuts are selected with respect to two different criteria (see [5]):

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

It is tried to select a nearly orthogonal subset of cutting planes cutting as deep as possible into the current LP polyhedron. The user has the possibility to change the employed distance norm. In the default settings, the euclidean norm is used to measure the efficacy of the cuts.

Reduced Cost Strengthening. A simplex based LP solver provides reduced cost values for each column, denoting the change in the objective value for a change in the column's solution value. This information can be used for columns being currently at one of their bounds to tighten the opposite bound (see [41]). Reduced cost strengthening can be viewed as a special kind of general purpose cutting plane separator using dual problem information. In future versions of SCIP it will be moved out of the core framework and implemented as external cut separator plugin.

Domain Propagation. If a bound of the columns was changed in the cut separation or reduced cost strengthening steps, domain propagation is applied again to further tighten the variables' domains (see Section 4.8.2).

4.8.4 Constraint Enforcement

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

In contrast to the constraint handlers' *feasibility tests* which only check a given primal solution (generated by a primal heuristic) for feasibility, the *enforcement* methods should also try to resolve an infeasibility. The constraint handler has different options of dealing with an infeasibility (see Figure 5):

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

The remaining answer of a constraint enforcement method is that the current solution is feasible for all constraints of the constraint handler.

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

The constraint enforcement cycle can have three different outcomes:

1. A constraint handler resolved an infeasibility, after which the node processing is continued appropriately.
2. All constraint handlers declared the solution to be feasible, which means a new feasible solution has been found.

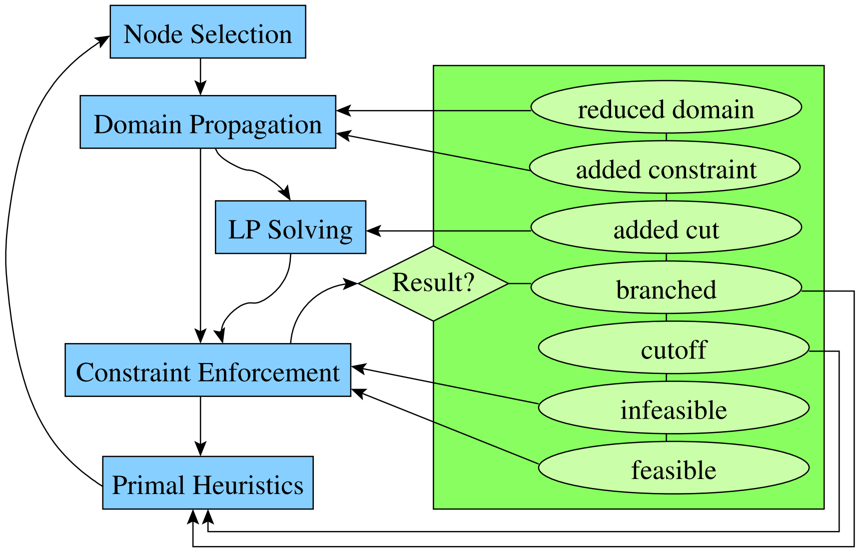

Figure 5. Constraint enforcement results.

3. At least one constraint handler detected an infeasibility, but none of them resolved it. In this case, the branching rules are called to create a branching.

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

4.8.5 Primal Heuristics

The processing of a subproblem concludes with calling the primal heuristics. Like every plugin in SCIP, the primal heuristics do not need to process every single node. They are usually called with a certain frequency, i. e., at specific depth levels in the branching tree. Primal heuristics can generate primal solutions, that are passed to the constraint handlers for checking their feasibility. If a feasible solution was found, the leaves of the branching tree exceeding the new primal bound are discarded.

5 Computational Results

SCIP version 0.78 already includes all necessary plugins to solve Mixed Integer Programs. In particular, it supports the following cutting planes:

- Gomory's fractional cuts [23],
- Marchand's version of complemented mixed integer rounding cuts [35],
- strong Chvátal-Gomory cuts [33],
- lifted knapsack cover cuts [8, 9, 37].

Cutting planes like clique cuts that depend on the set packing relaxation are still missing but will be added in one of the next versions.

SCIP includes various preprocessing algorithms, but does not yet provide probing techniques. There are 8 different variants of diving heuristics, including a modification of the *feasibility pump* [19]. Additionally, one quick and simple and one more involved rounding heuristic use the LP solution as starting point for generating primal solutions.

Various branching rules are available, including the very simple (and inefficient) *least infeasible* and *most infeasible* branching rules, as well as the very sophisticated *reliability branching* and variants of the costly *strong branching* rule (see [2] for a review). The node selection rule can be configured to resemble any mixture of *depth first* and *best first search*. LP solver interfaces exist for CPLEX [27], SOPLEX [51], and CLP [21].

In the following we present computational results on several MIP instances, comparing SCIP 0.78 with CPLEX 9.03 and SIP [36]. Note that CPLEX 9.03 was used as embedded LP solver in SCIP. All calculations were performed on a 3.2 GHz Pentium-4EE workstation with 2 GB RAM.

Our test set consists of instances from MIPLIB 2003 [1] and instances collected by Mittelmann [38]. We selected all instances where CPLEX 9.02 needed at least 1000 branching nodes and at most one hour CPU time for solving.[4] In all runs, we used a time limit of 3600 seconds and a memory limit of 1.5 GB.

Table 1 shows the results on our test set. Obviously, SCIP 0.78 is not strictly competitive to CPLEX 9.03 – there is a factor of 2 in both, the geometric mean and the total running time. However, we take these numbers

[4] CPLEX was run with default settings, except that "absolute mipgap" was set to 10^{-9} and "relative mipgap" to 0.0, which are the corresponding values in SCIP. The test set was assembled a few months ago when CPLEX 9.03 was not yet available.

	Scip 0.78		Sip		CPlex 9.03	
Name	Nodes	Time	Nodes	Time	Nodes	Time
aflow30a	41921	129.0	126209	127.7	37116	85.8
cap6000	5030	45.6	4320	15.6	4573	14.5
gesa2-o	21	7.4	42702	69.3	658	1.1
mas74	6809511	1639.4	4238858	934.0	5311878	1297.4
mas76	512719	109.0	430112	90.6	375729	56.1
misc07	42540	51.1	58155	67.4	72420	59.6
mzzv11	>3258	>3608.0	>150	>3642.2	3820	432.8
pk1	468423	133.0	344984	99.3	365710	86.2
pp08aCUTS	627	1.9	233	2.0	1518	1.8
qiu	12958	130.6	10099	108.3	11431	105.9
rout	32599	53.8	112780	149.3	843504	1832.6
vpm2	21467	12.2	14981	10.2	3240	1.1
ran8x32	23457	29.2	55375	46.2	7679	7.6
ran10x26	35591	69.0	47795	37.0	20176	41.4
ran12x21	91477	157.5	117050	85.6	81525	157.2
ran13x13	160566	139.4	95981	68.5	62936	62.6
binkar10_1	>801005	>1746.0	>1964203	>3600.0	5458	24.1
mas284	23554	12.3	20507	11.1	37405	12.2
prod1	82707	43.3	63157	42.7	97507	67.8
bc1	129174	2227.7	25050	430.1	9939	266.9
bienst1	11298	70.1	11492	54.6	12292	113.6
bienst2	105515	1126.9	102947	383.2	161843	2368.3
mkc1	>457536	>3600.0	>953912	>3600.0	14236	47.8
neos2	170245	664.0	23921	90.0	40193	81.2
neos3	>1033592	>3600.0	783556	1624.8	835129	2599.2
neos7	266071	720.0	203640	467.9	113861	415.6
seymour1	6918	890.2	5097	747.2	10459	1020.8
swath1	344	30.2	26105	131.2	15860	41.5
swath2	50467	341.9	24281	134.8	150740	372.5
Total (29)	11400591	21388.7	9907652	16870.8	8708835	11675.5
Geom. Mean	40564	162.5	49434.4	133.9	34668	81.4

Table 1. Computational results of Scip 0.78 and CPlex 9.03. Results marked with '>' were not solved to optimality within the time and memory limits.

as an indication that SCIP's performance is not *very far* away from a state-of-the-art MIP solver.

The comparison of SCIP with SIP shows that SIP is slightly superior. This is not surprising, since SCIP is the successor of SIP, and the algorithms reimplemented in SCIP are not yet tuned as exhaustive as the ones in SIP. The parameter settings are not yet carefully adjusted as well. Additionally, since SIP is exclusively implemented to solve completely specified MIPs, there is an overhead incorporated in SCIP for the support of general constraints, variable pricing, and other subproblem relaxations.

References

[1] T. Achterberg, T. Koch, and A. Martin. The mixed integer programming library: MIPLIB 2003, 2003. `http://miplib.zib.de`.

[2] T. Achterberg, T. Koch, and A. Martin. Branching rules revisited. *Operations Research Letters*, 33:42–54, 2005.

[3] A. Aiba, K. Sakai, Y. Sato, D.J. Hawley, and R. Hasegawa. Constraint logic programming language CAL. In *FGCS-88: Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 263–276, Tokyo, 1988.

[4] E. Althaus, A. Bockmayr, M. Elf, M. Jünger, T. Kasper, and K. Mehlhorn. SCIP – symbolic constraints in integer linear programming. Technical Report ALCOMFT-TR-02-133, MPI Saarbrücken, May 2002.

[5] G. Andreello, A. Caprara, and M. Fischetti. Embedding cuts in a branch&cut framework: a computational study with $\{0, \frac{1}{2}\}$-cuts. Preliminary Draft, October 2003.

[6] I. Aron, J.N. Hooker, and T.H. Yunes. SIMPL: A system for integrating optimization techniques. In J.-C. Regin and Michel Rueher, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems: First International Conference, CPAIOR 2004*, pages 21–36, Nice, France, April 2004.

[7] A. Atamtürk and M.W.P. Savelsbergh. Integer-programming software systems. *Annals of Operations Research*, 2004. forthcoming, `http://www.isye.gatech.edu/faculty/Martin_Savelsbergh/publications/ipsoftware-final.pdf`.

[8] E. Balas. Facets of the knapsack polytope. *Mathematical Programming*, 8:146–164, 1975.

[9] E. Balas and E. Zemel. Facets of the knapsack polytope from minimal covers. *SIAM Journal on Applied Mathematics*, 34:119–148, 1978.

[10] R. Bixby, M. Fenelon, Z. Gu, E. Rothberg, and R. Wunderling. MIP: Theory and practice – closing the gap. In M.J.D. Powell and S. Scholtes, editors, *Systems Modelling and Optimization: Methods, Theory, and Applications*, pages 19–49. Kluwer, 2000.

[11] 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.

[12] 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.

[13] A. Brooke, D. Kendrick, A. Meeraus, R. Raman, and R.E. Rosenthal. GAMS - a user's guide, December 1998. `http://www.gams.com`.

[14] COIN-OR. Computational infrastructure for operations research. `http://www.coin-or.org`.

[15] A. Colmerauer. An introduction to Prolog III. In *Communications of the ACM*, volume 33, pages 69–90, July 1990.

[16] G.B. Dantzig. Maximization of a linear function of variables subject to linear inequalities. In T.C. Koopmans, editor, *Activity Analysis of Production and Allocation*, pages 330–335. John Wiley & Sons, New York, 1951.

[17] Dash Optimization. XPress-MP. `http://www.dashoptimization.com`.

[18] M. Dincbas, P. van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The constraint logic programming language CHiP. In *FGCS-88: Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 693–702, Tokyo, 1988.

[19] M. Fischetti, F. Glover, and A. Lodi. The feasibility pump. Technical Report OR-04-4, Università di Bologna – D.E.I.S. – Operations Research, 2004.

[20] J. Forrest. COIN branch and cut, 2004. `http://www.coin-or.org`.

[21] J. Forrest, D. de la Nuez, and R. Lougee-Heimer. CLP user guide, August 2004. `http://www.coin-or.org/Clp/userguide`.

[22] R. Fourer, D.M. Gay, and B.W. Kernighan. *AMPL: A Modelling Language for Mathematical Programming*. Duxbury Press, Brooks/Cole Publishing Company, 2nd edition, November 2002.

[23] R.E. Gomory. An algorithm for integer solutions to linear programming. In R.L. Graves and P. Wolfe, editors, *Recent Advances in Mathematical Programming*, pages 269–302, New York, 1963. McGraw-Hill.

[24] M. Grötschel and M.W. Padberg. On the symmetric traveling salesman problem I: Inequalities. *Mathematical Programming*, 16:265–280, 1979.

[25] M. Grötschel and M.W. Padberg. On the symmetric traveling salesman problem II: Lifting theorems and facets. *Mathematical Programming*, 16:281–302, 1979.

[26] IBM. Mathematical Programming System Extended/370 (MPSX/370) program reference manual, 1979. SH19-1095-3, 4th Ed.

[27] ILOG CPLEX. Reference Manual, 2004. `http://www.ilog.com/products/cplex`.

[28] J. Jaffar and J.-L. Lassez. Constraint logic programming. In *Proceedings of the 14th ACM Symposium on Principles of Programming Languages*, pages 111–119, Munich, 1987.

[29] 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.

[30] T. Koch. Zimpl user guide. Technical Report 01-20, Konrad-Zuse-Zentrum für Informationstechnik Berlin, Takustr. 7, Berlin, 2001. http://www.zib.de/koch/zimpl.

[31] T. Koch. *Rapid Mathematical Prototyping.* PhD thesis, TU Berlin, 2004.

[32] S. Leipert. The tree interface – version 1.0 user manual. Technical Report 96.242, Institut für Informatik, Universität zu Köln, 1996. http://www.informatik.uni-koeln.de/ls_juenger/research/vbctool.

[33] A.N. Letchford and A. Lodi. Strengthening Chvátal-Gomory cuts and Gomory fractional cuts. *Operations Research Letters*, 30(2):74–82, 2002.

[34] LINDO. API Users Manual, 2003. http://www.lindo.com.

[35] H. Marchand and L.A. Wolsey. Aggregation and mixed integer rounding to solve MIPs. *Operations Research*, 49(3):363–371, 2001.

[36] A. Martin. Integer programs with block structure. Habilitations-Schrift, Technische Universität Berlin, 1998.

[37] A. Martin and R. Weismantel. The intersection of knapsack polyhedra and extensions. In R.E. Bixby, E.A. Boyd, and R.Z.Ríos-Mercado, editors, *Integer Programming and Combinatorial Optimization*, Proceedings of the 6th IPCO Conference, pages 243–256, 1998.

[38] H. Mittelmann. Decision tree for optimization software: Benchmarks for optimization software, 2003. http://plato.asu.edu/bench.html.

[39] G. Nemhauser and M.W.P. Savelsbergh. MINTO 3.1, 2004. http://coral.ie.lehigh.edu/~minto.

[40] G.L. Nemhauser, M.W.P. Savelsbergh, and G.C. Sigismondi. MINTO, a Mixed INTeger Optimizer. *Operations Research Letters*, 15:47–58, 1994.

[41] G.L. Nemhauser and L.A. Wolsey. *Integer and Combinatorial Optimization.* John Wiley & Sons, 1988.

[42] J.-F. Puget. A C++ implementation of CLP. Technical Report 94-01, ILOG S.A., Gentilly, France, 1994.

[43] T.K. Ralphs. SYMPHONY version 5.0 user's manual. Technical Report 04T-020, Lehigh University Industrial and Systems Engineering, 2004. http://branchandcut.org/SYMPHONY.

[44] G. Reinelt. TSPLIB 95, 1995. Institut für Angewandte Mathematik, Universität Heidelberg, http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95.

[45] M.W.P. Savelsbergh. Preprocessing and probing techniques for mixed integer programming problems. *ORSA Journal on Computing*, 6:445–454, 1994.

[46] C. Schulte. Comparing trailing and copying for constraint programming. In D. De Schreye, editor, *Proceedings of the 1999 International Conference on Logic Programming*, pages 275–289, Las Cruces, NM, USA, November 1999. The MIT Press.

[47] J.P. Marques Silva and K.A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions of Computers*, 48:506–521, 1999.

[48] S. Thienel. *ABACUS - A Branch-and-Cut System*. PhD thesis, Institut für Informatik, Universität zu Köln, 1995.

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

[50] L.A. Wolsey. Valid inequalities for 0-1 knapsacks and MIPs with generalized upper bound constraints. *Discrete Applied Mathematics*, 29:251–261, 1990.

[51] R. Wunderling. *Paralleler und objektorientierter Simplex-Algorithmus*. PhD thesis, Konrad-Zuse-Zentrum für Informationstechnik, Berlin, 1996.