Alma Mater Studiorum
Università di Bologna

SCUOLA DI INGEGNERIA E ARCHITETTURA
Corso di Laurea Magistrale in Ingegneria dell'Automazione

TESI DI LAUREA
in
Optimization Models and Algorithms

Computing Primal Solutions with exact arithmetics in SCIP

CANDIDATO:
Luca Fabbri

RELATORE:
Prof. Andrea Lodi

CORRELATORI:
Prof. Enrico Malaguti
Dr. Ambros Gleixner

III Sessione
Anno Accademico 2013-2014

Sommario

La ricerca di soluzioni esatte per problemi misti interi è un tema molto attuale all'interno della comunità scientifica. I risolutori MIP allo stato dell'arte utilizzano una rappresentazione numerica in formato floating-point, introducendo quindi approssimazioni. Sebbene tali risolutori di problemi misti interi forniscano risultati affidabili per la maggior parte dei problemi, ci sono casi in cui è necessaria una maggiore precisione. È noto, infatti, che per alcune applicazioni i risolutori floating-point restituiscano soluzioni errate, vale a dire soluzioni indicate come accettabili a causa delle approssimazioni, le quali non supererebbero un controllo con aritmetica esatta e non possono essere implementate nella pratica.

L'ambito in cui questa tesi è stata sviluppata è SCIP, un risolutore di programmi interi misti, sviluppato presso lo Zuse Institute di Berlino. In tale sede abbiamo considerato un nuovo approccio per risolvere problemi misti interi in modo esatto. In particolare abbiamo sviluppato un plug-in - un gestore di vincoli da inserire in SCIP - al fine di analizzare la precisione delle soluzioni floating-point ottenute e calcolare soluzioni primali esatte a partire da tali soluzioni floating-point.

Abbiamo condotto alcuni esperimenti computazionali per testare il gestore di vincoli per soluzioni primali esatte, attraverso l'utilizzo di due principali configurazioni: la *modalità di analisi* e la *modalità di applicazione*. La *modalità di analisi* ha permesso di raccogliere statistiche riguardanti l'attuale affidabilità di SCIP. I risultati hanno confermato che le soluzioni così ottenute sono sufficientemente accurate per un larga parte delle istanze. Tuttavia, la nostra analisi evidenzia anche la presenza di errori numerici aventi entità variabile. Utilizzando la *modalità di applicazione*, il gestore di vincoli suggerisce soluzioni esatte a partire dalla parte intera delle soluzioni floating-point. In tale configurazione abbiamo rilevato un generale miglioramento della qualità delle soluzioni finali trovate, senza tuttavia sopperire ad un significativo calo nelle prestazioni.

Abstract

The research for exact solutions of mixed integer problems is an active topic in the scientific community. State-of-the-art MIP solvers exploit a floating-point numerical representation, therefore introducing small approximations. Although such MIP solvers yield reliable results for the majority of problems, there are cases in which a higher accuracy is required. Indeed, it is known that for some applications floating-point solvers provide falsely feasible solutions, i.e. solutions marked as feasible because of approximations that would not pass a check with exact arithmetic and cannot be practically implemented.

The framework of the current dissertation is SCIP, a mixed integer programs solver mainly developed at Zuse Institute Berlin. In the same site we considered a new approach for exactly solving MIPs. Specifically, we developed a constraint handler to plug into SCIP, with the aim to analyze the accuracy of provided floating-point solutions and compute exact primal solutions starting from floating-point ones.

We conducted a few computational experiments to test the exact primal constraint handler through the adoption of two main settings. Analysis mode allowed to collect statistics about current SCIP solutions' reliability. Our results confirm that floating-point solutions are accurate enough with respect to many instances. However, our analysis highlighted the presence of numerical errors of variable entity. By using the enforce mode, our constraint handler is able to suggest exact solutions starting from the integer part of a floating-point solution. With the latter setting, results show a general improvement of the quality of provided final solutions, without a significant loss of performances.

Acknowledgements

This text is the last stage of a long journey. I did not walk alone, that's why I need to thank some people.

My experience in Berlin has been intense and gave me a personal and professional growth. I would like to express my sincere gratitude to Ambros Gleixner for teaching me a lot with his knowledges and his example, and for his support. Thank you Anna for the good time spent at ZIB and for your friendship. I would like to thank all the other ZIB guys that helped me, as well as Berlin itself to be so inspiring.

A special thank also to my supervisor, Professor Andrea Lodi, who gave me this opportunity and always let me feel his support and faith.

Colgo questa occasione per ringraziare chi mi ha sempre sostenuto in questi anni di studio e di vita. Non puó mancare un grazie ai miei amici Guido, Silvia e Marche, che hanno condiviso con me i loro spazi e il loro tempo. Le nostre esperienze e l'affetto che mi avete dato saranno sempre con me. Grazie anche a Gardo e Sonia per la loro grande e preziosa amicizia.

Un grazie immenso a Simona, per aver condiviso quotidianamente tutti i miei sforzi, riuscendo sempre a spronarmi con amore. Grazie per il tuo sostegno insostituibile e spontaneo, e anche per aver contribuito a questa tesi con la tua penna rossa sul mio inglese.

Il ringraziamento finale va alla mia famiglia. A partire dai miei nonni, nei quali vedo riflesso il mio passato, e dai quali ricevo incondizionato affetto. Grazie ad Annalisa, una sorella incredibile, che fra le tante cose mi ha accompagnato nei primi difficili passi a Berlino. Sei un grande riferimento per me. Grazie a Patrizia e Achille, i miei genitori. Custodisco gelosamente tutto quello che mi avete dato e la vostra fiducia totale, che mi ha sempre sorretto. Questo risultato lo dedico a voi.

Contents

Chapter 1

Floating-point Constraint Integer Programming

In this chapter an overview of the topic we are going to explore is given.

In section 1.1 we start from the definition of Mathematical Program, going through many different optimization models under different restrictions. The final model we want to obtain is Constraint Integer Programming, a quite general concept SCIP is based on.

In section 1.2 the main algorithms used in solving optimization problems are presented. In particular, original branch-and-bound and cutting planes are described. Nevertheless, it is worth to consider that state-of-the-art solvers use more sophisticated algorithms derived from these techniques.

In section 1.3 we discuss floating-point arithmetic, by considering how numbers are represented in a calculator, as well as advantages and disadvantages of such representations and how to deal with issues that may arise in solving optimization problems with floating-point data.

1.1 From Mathematical Programming to CIP

1.1.1 Mathematical Programming

Mathematical programming is the use of mathematical models to assist in taking decisions. In particular it makes use of optimization models with the aim to obtain the best solution of the problem associated with the mathematical model. Mathematical programming is more restrictive compared to other techniques (e.g., statistics, simulation, forecasting) because of its ambition to find out the optimal solution of a problem.

For a given problem it is possible to build an associated mathematical model. Since the model is an abstraction of the problem, it has some degree of approximation compared to the real-world problem. A few optimization algorithms is then applied to the model in order to find out the best solution depending on certain criteria. It is therefore possible to provide a definition of mathematical programming as follows.

Definition 1.1 (Mathematical Program)**.** Given n decision variables $x_1, x_2, \ldots, x_n$, an objective function $f(x_1, x_2, \ldots, x_n)$, and m constraints $g_i(x_1, x_2, \ldots, x_n) \leq b_i$, $i = 1, 2, \ldots, m$, a mathematical program is to solve

$$\begin{aligned} z^* = {} & \min f(x_1, x_2, \ldots, x_n) & & \\ & g_i(x_1, x_2, \ldots, x_n) \leq b_i & & i = 1, 2, \ldots, m \\ & x_j \geq 0 & & j = 1, 2, \ldots, n \end{aligned} \tag{1.1}$$

The optimal solution is given by the set of values $x_1^*, x_2^*, \ldots, x_n^*$ for decision variables such that $z^* = f(x_1^*, x_2^*, \ldots, x_n^*)$.

The mathematical model has to be built considering two opposite requirements. It is important to have a model as faithful as possible to the real-world problem. In fact the solution found is optimal for the built model, but it is not necessarily optimal for the real-world problem. The quality of the optimal solution is then strictly correlated to the quality of the model. On the other side, a model very close to reality is likely to be complex, and consequently harder to be solved.

In practice, mathematical models in the general form (1.1) are usually not solvable by state of the art algorithms within acceptable time. It is therefore necessary to introduce some limitations to the model in order to obtain models easier to solve, by avoiding to lose sufficient closeness to real-world problems.

It is possible to introduce a classification of the models. First, we can distinguish between linear programming and nonlinear programming. A linear program is a mathematical program where both objective function $f(x)$ and constraints $g_i(x)$, $(i = 1, 2, \ldots, m)$ are all linear functions. A Nonlinear Program is a mathematical program where at least one of such functions is nonlinear. Linear programs are clearly simpler and easier to solve.

Another classification can be done depending on the decision variables domain. If all the variables are continuous, i.e. they can have any value within a given interval, we have a continuous problem. If variables are discrete they can only have a finite number of possible values and they define a combinatorial problem. Mixed problems contains both continuous and discrete variables.

In the following paragraphs we will go deeper into the definition of many different models.

1.1.2 Linear Program

A Linear Program has linear objective function and constraints, and continuous variables.

Definition 1.2 (Linear Program)**.** Given a matrix $A \in \mathbb{R}^{m \times n}$, vectors $b \in \mathbb{R}^m$, a vector $c \in \mathbb{R}^n$, and a vector $x \in \mathbb{R}^n$ of variables or unknowns, the linear program (LP) is to solve

$$\begin{aligned} z^* = \min c^T x \\ Ax \leq b \\ x \geq 0 \end{aligned} \tag{1.2}$$

where m represents the number of constraints and n represents the number of variables.

The inequalities $Ax \leq b$ and $x \geq 0$ are the constraints which specify a convex polytope over which the objective function is to be optimized.

Linear Programs are solvable in polynomial time, which was first shown by Khatchiyan [22], by using the so-called ellipsoid method. However, the mainly used method to solve linear programs is simplex algorithm, invented by Dantzig [11]. Simplex algorithm is computationally exponential, but has good performances in the mean case.

1.1.3 Integer (Linear) Program

When all the decision variables x are restricted to be integer values, we have an integer linear program.

Definition 1.3 (Integer (Linear) Program)**.** Given a matrix $A \in \mathbb{R}^{m \times n}$, vectors $b \in \mathbb{R}^m$, a vector $c \in \mathbb{R}^n$, and a vector $x \in \mathbb{R}^n$ of variables or unknowns, the integer (linear) program (IP) is to solve

$$\begin{aligned} z^* = &\min c^T x \\ &Ax \le b \\ &x \ge 0, \qquad x \in \mathbb{Z} \end{aligned} \tag{1.3}$$

Since it is a combinatorial problem, it can, in principle, be solved by complete enumeration, which consists in considering all the possible combination of the n decisional variables. Then all the constraints are checked for each combination and the objective function value is computed. The combination providing the best objective value is the optimal solution.

In practice, complete enumeration is not used because the cost would increase exponentially with the number of variables and, consequently, it would be impossible to obtain a solution within acceptable time.

1.1.4 Mixed Integer (Linear) Program

If only a subset of the variables are integer, we have a Mixed Integer (Linear) Program.

Definition 1.4 (Mixed Integer (Linear) Program)**.** Given a matrix $A \in \mathbb{R}^{m \times n}$, vectors $b \in \mathbb{R}^m$, a vector $c \in \mathbb{R}^n$, a vector $x \in \mathbb{R}^n$ of variables or unknowns, and a subset $I \subset N = \{1, \dots, n\}$, the mixed integer (linear) program (MIP) is to solve

$$\begin{aligned} z^* = &\min c^T x \\ &Ax \le b \\ &x \ge 0, \qquad x_j \in \mathbb{Z} \quad \forall j \in I \end{aligned} \tag{1.4}$$

The vectors in the set $X_{MIP} = \{x \in \mathbb{R}^n \,|\, Ax \le b, x_j \in \mathbb{Z} \forall j \in I\}$ are called feasible solutions of MIP. A feasible solution $x^* \in X_{MIP}$ is called optimal if its objective value satisfies $c^T x^* = z^*$. MIP solvers usually treat simple bound constraints $l_j \le x_j \le u_j$, with $l_j, u_j \in \mathbb{R} \cup \{\pm\infty\}$ separately from the remaining constraints. In particular, integer variables with bounds $0 \le x_j \le 1$ play a special role in the solving algorithms and are a very important tool to model yes/no decisions.

Definition 1.5 (LP relaxation of a Mixed Integer Program)**.** Given a mixed integer program its LP relaxation is defined as

$$\begin{aligned} \check{z} = \min c^T x & \\ Ax \leq b & \\ x \geq 0 & \end{aligned} \tag{1.5}$$

$X_{LP} = \{x \in \mathbb{R}^n \,|\, Ax \leq b\}$ is the set of feasible solutions of the LP relaxation. An LP-feasible solution $\check{x} \in X_{LP}$ is called LP-optimal if $c^T\check{x} = \check{z}$. The LP relaxation can be strengthened by cutting planes which use the LP information and the integrality restrictions to derive valid inequalities that cut off the solution of the current LP relaxation without removing integral solutions. The objective value $\check{z}$ of the LP relaxation provides a lower bound for the whole sub-tree, and if this bound is not smaller than the value $\check{z} = c^T\check{x}$ of the current best primal solution $\check{x}$, the node and its sub-tree can be discarded. The LP relaxation usually gives a much stronger bound than the one that is provided by simple dual propagation of CP solvers. The solution of the LP relaxation usually requires much more time, however.

The most important ingredients of an MIP solver implementation are a fast and numerically stable LP solver, cutting plane separators, primal heuristics and presolving algorithms. Additionally, the applied branching rule is of major importance [1].

1.1.5 Constraint Program

A more general approach consists in constraint programs.

Definition 1.6 (Constraint Program)**.** A constraint program consists of solving

$$f^* = \min\{f(x) \,|\, x \in D, C(x)\} \tag{1.6}$$

with the set of domains $D = D_1 \times \ldots \times D_n$, the constraint set $C = \{C_1, \ldots, C_m\}$, and an objective function $f\colon D \to \mathbb{R}$.

We denote the set of feasible solutions by $X_{CP} = \{x \,|\, x \in D, C(x)\}$. A CP where all domains are finite is called a finite domain constraint program (CP(FD)). The key element for solving constraint programs in practice is the efficient implementation of domain propagation algorithms, which exploit the structure of the involved constraints. 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. If every variable's domain is reduced to a single value, a new primal solution is found. If any of the variables' domains becomes empty, the subproblem is discarded and a different leaf of the current branching tree is selected to continue the search [1].

1.1.6 Satisfiability Problems

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

Definition 1.7 (Satisfiability Problem)**.** Let $C = C_1 \wedge \cdots \wedge C_m$ be a logic formula in conjunctive normal form (CNF) on boolean variables $x_1, \ldots, x_n$. Each clause $C_i = l_1^i \vee \cdots \vee l_{k_1}^i$ is a disjunction of literals. A literal $l \in L = \{x_1, \ldots, x_n, \overline{x_1}, \ldots, \overline{x_n}\}$ is either a variable x_j or the negation of a variable $\overline{x_j}$. The task of the satisfiability problem is to either find an assignment $x^* \in \{0,1\}^n$, such that the formula C is satisfied, i.e., each clause C_i evaluates to 1, or to conclude that C is unsatisfiable, i.e., for all $x \in \{0,1\}^n$ at least one C_i evaluates to 0.

1.1.7 Comparing MIPs, CPs and SAT

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

Because MIP is a very specific case of CP, MIP solvers can apply advanced problem specific algorithms that operate on the subproblem as a whole. In particular, they use the simplex algorithm to solve the LP relaxations, and cutting plane separators. In contrast, due to the unrestricted definition of CPs, CP solvers cannot take such a global perspective, but they have to rely on the constraint propagators, each of them exploiting the structure of a single constraint class. An advantage of CP is, however, the possibility to model the problem more directly, using very expressive constraints which contain a lot of structure. Transforming those constraints into linear inequalities can conceal their structure from a MIP solver, and therefore lessen the solver's

MIP	linear objective function linear constraints real and integer variables
CP	arbitrary objective function arbitrary constraints arbitrary (discrete) variables
CIP	linear objective function arbitrary constraints real and integer variables after fixing all integer variables CIP becomes an LP

Table 1.1: Comparison between MIP, CP and CIP

ability to draw valuable conclusions about the instance or to make the right decisions during the search.

SAT is also a very specific case of CP with only one type of constraints. SAT solvers mainly exploit the special problem structure to speed up the domain propagation algorithm and to improve the underlying data structures.

1.1.8 Constraint Integer Program

Constraint integer programs allow to merge CP, SAT and MIP techniques, by combining their advantages and compensating for their individual weaknesses.

Definition 1.8 (Constraint Integer Program)**.** A constraint integer program consists of solving

$$c^* = \{\min c^T x \mid C(x) \in \mathbb{R}^n, x_j \in \mathbb{Z}\, \forall j \in I\} \tag{1.7}$$

with a finite set $C = \{C_1, \ldots, C_m\}$ of constraints $C_i \colon \mathbb{R}^n \to \{0,1\}, i = 1, \ldots, m$, a subset $I \subset N = \{1, \ldots, n\}$ of the variable index set, and an objective function vector $c \in \mathbb{R}^n$.

A CIP has to fulfill the following condition:

$$\forall\, \hat{x_I} \in \mathbb{Z}^I\, \exists\, (A', b') \colon \{x_C \in \mathbb{R}^C \mid C(\hat{x_I}, x_c)\} = \{x_C \in \mathbb{R}^C \mid A' x_C \leq b'\} \tag{1.8}$$

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

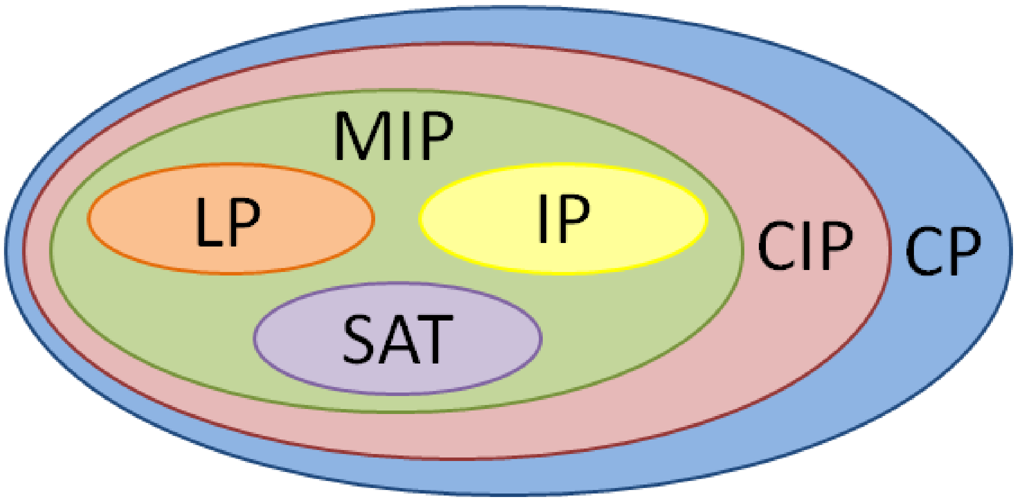

Figure 1.1: Outline of mathematical programs under different restrictions.

Restriction 1.8 ensures that the remaining subproblem, after integer variables have been fixed, is always a linear program. This means that in the case of finite domain integer variables, the problem can be, in principle, completely solved by enumerating all values of the integer variables and solving the corresponding LPs. Note that this does not forbid quadratic or even more involved expressions. Only the remaining part after fixing (and thus eliminating) the integer variables must be linear in the continuous variables.

1.2 Standard solution algorithms

1.2.1 Branch and Bound

The branch-and-bound procedure is a very general and widely used method to solve optimization problems. The key idea is to successively divide the given problem instance into smaller subproblems until the individual subproblems are easy to solve. The best among the subproblems' solutions is the global optimum.

Figure 1.2 show the fundamental steps of branch-and-bound algorithm. The splitting of a subproblem into two or more smaller subproblems in step 7 is called branching. During the course of the algorithm, a branching tree is created with each node representing one of the subproblems. The root of the tree corresponds to the initial problem R, while the leaves are either "easy" subproblems which have already been solved or subproblems in L which still

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

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

Figure 1.2: Branch-and-bound algorithm [1].

have to be processed. The intention of the bounding in step 5 is to avoid a complete enumeration of all potential solutions of R, which are usually exponentially many. In order for bounding to be effective, good lower (dual) bounds $\check{x}$ and upper (primal) bounds $\check{c}$ must be available. Lower bounds are calculated with the help of a relaxation Q_{relax} which should be easy to solve. Upper bounds can be found during the branch-and-bound algorithm in step 6, but they can also be generated by primal heuristics. The node selection in step 3 and the branching scheme in step 7 determine important decisions of a branch-and-bound algorithm that should be tailored to the given problem class. Both of them have a major impact on how early good primal solutions can be found in step 6 and how fast the lower bounds of open subproblems in L increase. They influence the bounding in step 5, which should cut off subproblems as early as possible and thereby prune large parts of the search tree. Even more important for a branch-and-bound algorithm to be effective is the type of relaxation that is solved in step 4. A reasonable relaxation must fulfill two usually opposite requirements: it should be easy to solve and it should yield strong dual bounds. In Mixed Integer Programming the most widely used relaxation is the LP relaxation, which proved to be very successful in practice.

1.2.2 Cutting planes

Besides splitting the current subproblem Q into two or more easier subproblems by branching, while solving MIPs one can also try to tighten the

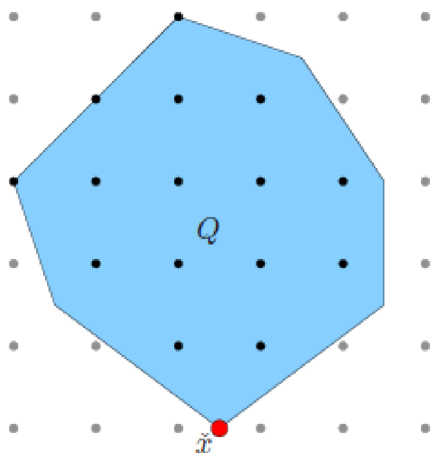
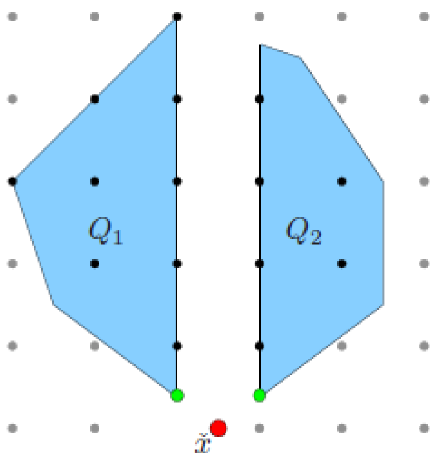

Figure 1.3: Branching on a single fractional variable [1].

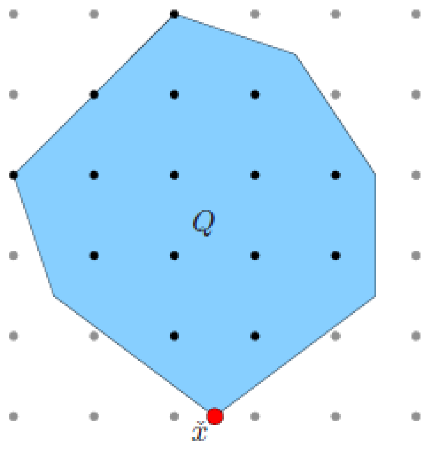
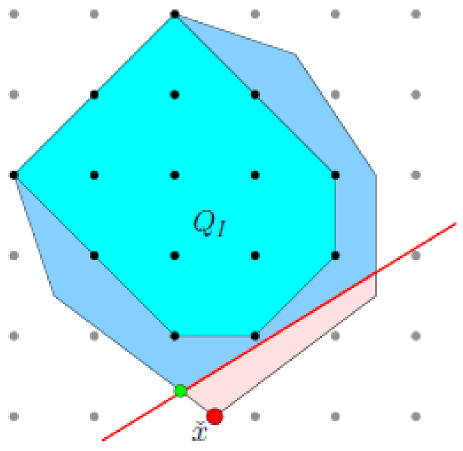

Figure 1.4: A cutting plane that separates the fractional LP solution $\check{x}$ from the convex hull Q_I of integer points of Q [1].

subproblem's relaxation in order to rule out the current solution $\check{x}$ and to obtain a different one.

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

1.2.3 Branch and cut

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

1.3 Floating point arithmetic

1.3.1 Floating point representation

In this section we will focus on some issues related to the data implementation of problems described in section 1.1 and the algorithms described in section 1.2. We will consider how data is represented in a computer and what descends from this representation.

The main limitation is the necessity of a finite representation. This is due to both the limited capacity of the memory and to the indefinitely large runtime that would be necessary to perform computations with indefinitely large numbers. Several different representations of real numbers have been proposed, but the most widely used is the floating-point representation. This representation is composed by three different parts: sign, significand and exponent. A generic number can therefore be written as $\pm\, significand \times base^{exponent}$. Since numbers implementation is binary, usually a base equals to 2 is used.

In the application we are going to discuss in this thesis the double-precision floating-point numeric format is the standard way to represent numbers. This format exploits 64 bits to represent each number. In particular, one bit is used for the sign, 11 bits for the exponent and 52 bits for the significand (see Figure 1.5).

The significand has an implicit integer bit of value 1. With the 52 bits of the fraction significand appearing in the memory format, the total precision is therefore 53 bits (approximately 16 decimal digits, since $53 \log_{10} 2 \approx 15.955$).

The most common reason why a real number might not be exactly representable as a floating-point number is when a rational number, i.e. a number having a finite representation with base 10, has an infinite binary representation. It is the case, for instance, of the decimal number 0.6. Its

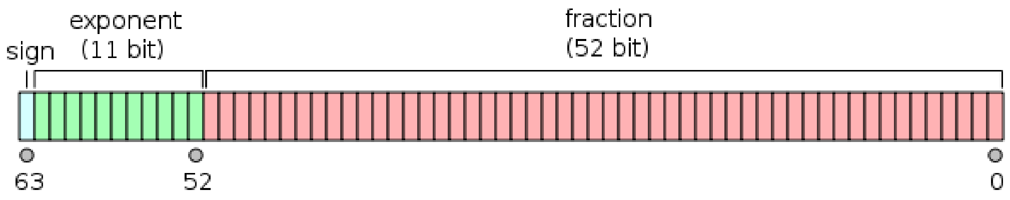

Figure 1.5: Double-precision floating-point format [33].

binary representation is

$$0.6_{(10)} = 0.\overline{1001}_{(2)}. \tag{1.9}$$

Thus, when *base* = 2, the number 0.6 lies strictly between two floating-point numbers and is exactly representable by neither of them. [19]

Double-precision floating-point format is widespread because of its computational efficiency and results obtained by floating-point computation are accurate enough for most scientific applications. Furthermore, many high precision floating-point algorithm has been developed in order to obtain an arbitrary precise computation.

Nevertheless, for some applications this is not sufficient to get reliable results. Before going deeper into this topic it is necessary to provide some other definitions.

1.3.2 Definition of feasibility and optimality

An optimization problem can be solved either for feasibility or for optimality.

When feasibility only is required, the solver is asked to return an answer to the question: does it exist a feasible solution for the problem? The answer is "yes" if at least one feasible solution is found, "no" if infeasibility is proved. The goal of the solver is to find out feasible solutions and, at the end, the optimal solution.

Optimality is more challenging to achieve. In this case the objective function is considered and the goal of the solver is to return the optimal solution. Obviously, the optimal solution exists only if the problem is feasible. The optimal solution will be chosen from the set of all the feasible solutions of the problem.

Let's further explore feasibility and optimality for MIPs.

Feasibility

The constraints of a MIP define a feasibility region. A solution is said to be feasible if it belongs to the feasibility region. Graphically, the feasibility region can be represented by a set of points in $\mathbb{R}^n$, where every point represents a feasible solution.

For LPs the feasibility region can always be represented by a convex polytope. In this case we have a continuous region and, therefore, infinitely many solutions, except for special cases where no solutions or a single solution occur.

In case we have an IP, a convex polytope is still defined by problem constraints, but also integrality has to be taken into account. The feasible region is represented by all integral points included in the polytope. If the problem is bounded, the number of feasible solutions is finite.

In practice, feasibility is proven by checking whether a given solution satisfies all the problem constraints.

Optimality

In order to declare that a certain solution is the optimal solution, it is necessary to find some optimality conditions that will provide stopping criteria in an algorithm for MIP. The ”naive” but nonetheless important reply is that we need to find a lower bound $\underline{z} \leq z$ and an upper bound $\overline{z} \geq z$ such that $\underline{z} = \overline{z} = z$. Practically, this means that any algorithm will find a decreasing sequence

$$\overline{z_1} > \overline{z_2} > \cdots > \overline{z_s} \geq z \tag{1.10}$$

of upper bounds, and an increasing sequence

$$\underline{z_1} < \underline{z_2} < \cdots < \underline{z_t} \leq z \tag{1.11}$$

of lower bounds, and stop when

$$\overline{z_s} - \underline{z_t} \leq \varepsilon \tag{1.12}$$

where ε is some suitably chosen small non-negative value. Thus, it is necessary to find ways of deriving such upper and lower bounds.

Primal bounds Every feasible solution $\hat{x} \in X$ provides a lower bound $\underline{z} = c(\hat{x}) \leq z$. This is essentially the only way we know to obtain lower bounds. For some IP problems, finding feasible solutions is easy, and the real question is how to find good solutions. For other IPs, finding feasible solutions may be very difficult.

Dual bounds Finding upper bounds for a maximization problem (or lower bounds for a minimization problem) raises a different challenge. The most important approach is by relaxation, the idea being to replace a "difficult" max (min) IP by a simpler optimization problem whose optimal value is at least as large (small) as z. For the relaxed problem to have this property, there are two obvious possibilities:

- to enlarge the set of feasible solutions so that one optimizes over a larger set;
- to replace the max (min) objective function by a function that has the same or a larger (smaller) value everywhere.

1.3.3 Floating point arithmetic and tolerances for MIP solvers

Most MIP solvers are based on floating-point arithmetic and work with tolerances to check solutions for feasibility and to decide on optimality. In their feasibility tests, solvers typically consider absolute tolerances for the integrality constraints and relative ones for linear constraints. Some of them normalize the activity of linear constraints individually, others scale directly the constraint matrix.

The tolerances affect solution times and solution accuracy, normally in opposite ways, and the solvers apply different strategies here. Typically it can happen that for a given instance different solvers compute different optimal objective values. If one fixes all integer variables from the reported solution to the closest integer value and recomputes the continuous variables by solving the resulting LP with exact arithmetic, some of these post-processed solutions turn out to be infeasible compared to exact arithmetic and zero tolerances.

It is worth to note that this does not mean that any of the solvers made a mistake. It only means that the computed solution lies outside the feasible area described by the input file, but inside the extended feasible area created by reading in the problem and introducing tolerances. It is only solutions that are feasible in the latter sense that solvers attempt to deliver, and those are the solutions the checker checks. More precisely, the operation which rounds the reported value of the integer variables to the closest integer is only applied to compute fully reliable primal bounds for the MIPs.

As introduced in section 1.3.1, floating-point computations can be performed quickly on computers but the limited size of this representation has its disadvantages. The error incurred by a single operation is usually small but algorithms requiring many operations may accumulate and propagate these small errors, leading to errors of significant magnitude.

Let's now consider the phases of the process that precedes the actual solution of the problem.

The MPS file format, which is used as a standard to define MIP instances, requires the input numbers to be written in base 10 ASCII representation. Furthermore, the definition of the MPS file format specifies that each entry uses only 12 characters, thereby if a problem cannot be expressed exactly in this format, even the input file will be an approximation of the intended problem. Suppose a problem is defined in an MPS file as having the feasible region

$$\{x\colon Ax \le b,\, x \ge 0,\, x \in \mathbb{Z}^n\}. \tag{1.13}$$

As this problem is read in by the solver, the entries in A, b will be transformed to a binary representation, possibly modifying their values and changing the feasible region to

$$\{x\colon \tilde{A}x \le \tilde{b},\, x \ge 0,\, x \in \mathbb{Z}^n\}. \tag{1.14}$$

In addition, due to inexact floating-point computation, the solvers need to introduce tolerances, hence relaxing the feasible region. Typically relative tolerances are used. In order to do this efficiently and to improve the numerical properties of the model, the constraint matrix is usually scaled. As a result, solvers operate on something similar to

$$\{x\colon (\widetilde{Q\tilde{A}})x \le \widetilde{Q\tilde{b}} + \mathbf{1}\varepsilon,\, x \ge -\mathbf{1}\varepsilon,\, x \in (\mathbb{Z} + [-\delta, \delta])^n\}, \tag{1.15}$$

where ε and δ are tolerances for feasibility and integrality and $\mathbf{1}$ is the vector of all ones. As we can see, even the steps of parsing and scaling the problem can change its description. The entire solution procedure is then applied to this transformed problem.

Furthermore, other preprocessing techniques are usually applied by the solver in order to simplify the problem, such as removing redundant constraints and variables, tightening bounds and coefficients, or aggregating variables. All of these issues may lead to further modifications of the problem before the branch-and-bound and cutting plane phases actually start. [24]

Chapter 2

SCIP: Solving Constraint Integer Programs

In this chapter will be discussed the framework underlying the work treated in the course of the current dissertation. The main framework is SCIP, a CIP solver that has been developed for 13 years at Zuse Institute Berlin, in cooperation with a few academic partners.

Section 2.1 provides a general introduction to SCIP. An overview of its history, as well as its performances and features are described.

Section 2.2 focuses on practical details concerning what working with SCIP means. In particular, I will consider my work experience with SCIP, by giving an overview on code organization and documentation.

In section 2.3 constraint handlers are introduced. Their key role in SCIP and their structure will be discussed there. Also, we might consider this section as an introduction to the following chapter, which will mainly focus the attention on xprim constraint handler.

Section 2.4 presents a brief discussion about exact LP solvers. SCIP needs to interface with external LP solvers, however it can support many. For our application, it is important to have an exact LP solver at disposal, thereby we are going to explore a few solutions in this sense.

2.1 Introduction to SCIP

2.1.1 What is SCIP

SCIP is a framework for Constraint Integer Programming oriented towards the needs of mathematical programming experts who wants to have total control of the solution process and access detailed information. SCIP provides the infrastructure to implement very flexible branch-and-bound based search algorithms. In addition, it includes a large library of default algorithms to control the search. These main algorithms of SCIP are part of external plugins, which are user defined callback objects that interact with the framework through a very detailed interface.

A similar technique is used for solving both Integer Programs and Constraint Programs: the problem is successively divided into smaller subproblems (branching) that are solved recursively. On the other hand, Integer Programming and Constraint Programming have different strengths: Integer Programming uses LP relaxations and cutting planes to provide strong dual bounds, while Constraint Programming can handle arbitrary (non-linear) constraints and uses propagation to tighten variables' domain. SCIP can also be used as a pure MIP solver or as a framework for branch-cut-and-price.

It is worth to point out that CIPs inherit from CPs the possibility of a single constraint to represent a whole set of inequalities and not only a single one. From this idea it follows that SCIP is constraint based. This approach provides high flexibility and the capability to manage differently each kind of constraint. This allows in many cases to consider constraints as a unique entity, without separating the inequalities it is composed by. The disadvantage of the constraint based approach is the limited global view of the problem, since a constraint knows its variable but a variable does not know the constraints it appears in.

2.1.2 History and performances

SCIP development started in 2002 and since then many developers contributed to this project. The headquarter of SCIP is Zuse Institute Berlin, an interdisciplinary research institute for applied mathematics and data-intensive high-performance computing. Its research focuses on modeling, simulation and optimization with scientific cooperation partners from academia and industry. During the years many people have given their contribute to SCIP and it counts more than 500 000 lines of source code. Nowadays SCIP development is still very active and the number of contributors is growing and growing.

SCIP is one of the fastest non-commercial solvers for mixed integer programming (MIP) and mixed integer nonlinear programming (MINLP). It is distributed under the ZIB Academic License, that guarantees freedom to

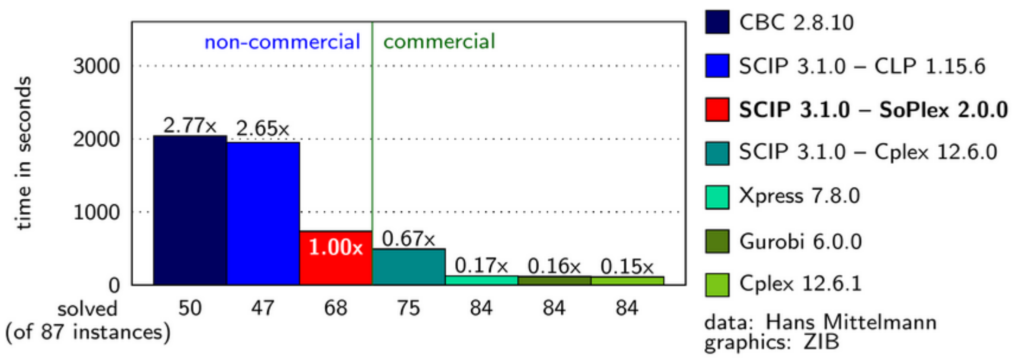

Figure 2.1: SCIP performances compared to some commercial and non-commercial solvers [37].

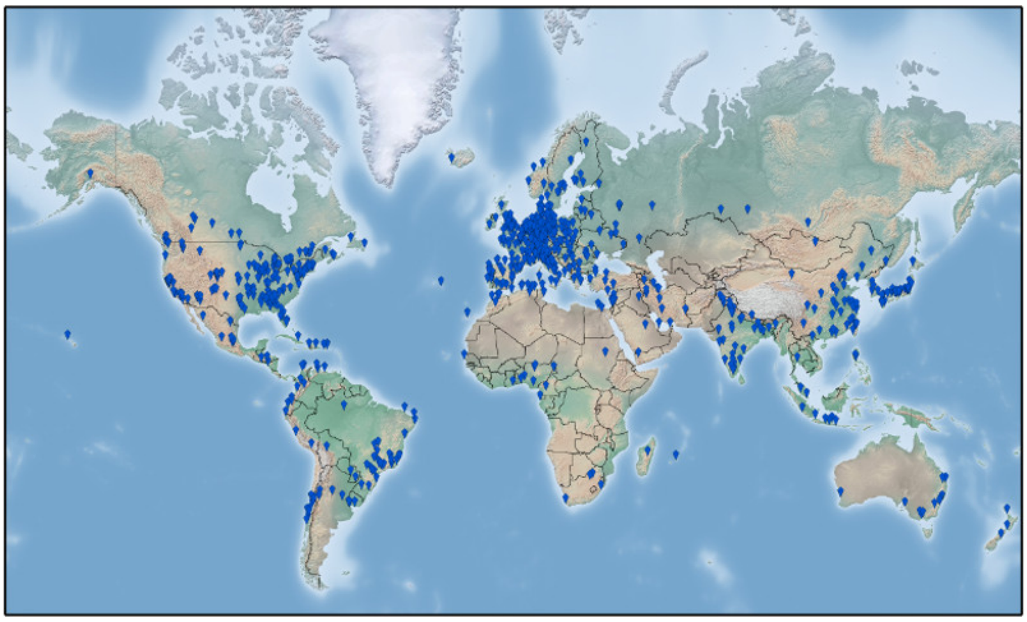

Figure 2.2: Locations of registered SCIP downloads [37].

share and change software for academic use. SCIP is then freely retrievable for research purposes, moreover its code is open source. Its fame is widespread to all over the world as it has been downloaded from all the continents (see Figure 2.2).

SCIP can be used alone, but the SCIP Optimization Suite is available. It is a complete source code bundle of SCIP, SoPlex, ZIMPL, GCG and UG.

SoPlex (Sequential object-oriented simPlex) is a Linear Programming (LP) solver based on the revised simplex algorithm. It features preprocessing techniques, exploits sparsity, and also offers primal and dual solving routines. It can be used as both a standalone solver and embedded into other programs.

ZIMPL (Zuse Institut Mathematical Programming Language) is a little language to translate the mathematical model of a problem into a linear or nonlinear mixed integer mathematical program such that it can be read by a LP or MIP solver.

UG (Ubiquity Generator framework) is a generic framework to parallelize branch-and-bound based solvers in a distributed or shared memory computing environment.

GCG (Generic Column Generation) is a generic branch-cut-and-price solver for mixed integer programs

2.1.3 Features

SCIP is characterized by several features. First, it is a very fast standalone solver for LPs, MIPs and MINLPs, as well as a framework for branching, cutting, pricing and propagation.

Every existing unit is implemented as a plugin, leading to a very flexible interface. Users can add many different plugins:

- constraint handlers to implement arbitrary constraints,
- variable pricers to dynamically create problem variables,
- domain propagators to apply constraint dependent propagations on the variables' domains,
- 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,
- 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.

Interfaces to other applications and programming languages are provided. In particular, SCIP is compatible with Python, Java, AMPL, GAMS and MATLAB.

SCIP can be supported by many LP solvers. SoPlex is a natural choice, since it is part of SCIP Optimization Suite. However, other LP solvers are supported, e.g., CPLEX, Gurobi, XPress, Mosek, QSopt and CLP.

Different relaxations can be included, as well as conflict analysis can be applied to learn from infeasible subproblems. In addition, the dynamic memory management reduces the number of system calls with automatic memory leakage detection in debug mode.

2.2 Working with SCIP

Implementing a SCIP plugin at ZIB has not only been a matter of code. I experienced a well organized working method, which is necessary according to SCIP dimension. In addition, I learned how to use many tools in order to produce, organize and debug the code. In the current section we will examine all of these aspects.

2.2.1 Code structure and documentation

SCIP is composed of many structured files, containing many callable methods. Furthermore, variables are properly masked and often only accessible via dedicated methods, as in object-oriented programming concept. Although SCIP is written in C-language, implementing a SCIP plugin requires to write just a few lines of pure C-language. Indeed, it is crucial to use an already implemented method to perform any operation one desires, if it exists. This is due to the importance to have optimized code, therefore, a good working method is to specifically focus on writing efficient methods and call them as much as possible. To write efficient code is demanding, but code optimization is essential as well as algorithms optimization.

Since exploiting existing methods is required, it is extremely important to have an easy way to find them. The code has always to be well documented, so that everybody can quickly understand what every method achieves. Documentation is then generated by doxygen [14], a tool that extracts comments directly from the source and creates well structured documents.

In order to have robust code, as well as to simplify debugging phase, some precautions are adopted. Asserts and debug messages are heavily used, and many methods are called by using a *SCIP_CALL()* function, which allows to have a check on method execution success.

2.2.2 File system organization

In the file system, SCIP is stored in a folder containing several files and directories with different aims. In the current paragraph we will present a few of them.

As already introduced, SCIP is very customizable by tuning properly its parameters. This represents a strength, as it is possible to obtain a countless amount of different behaviors. On the other hand, it might be a limit, according to the complexity of setting up so many values. In order to partially overcome this problem, it is possible to create files containing a set of parameters with their correspondent desired values. These files have extension *.set* and are included in a directory called */settings*. It is then possible to load a settings file while running SCIP, accelerating and simplifying parameters tuning. Parameters not considered in the loaded settings file keep their default values.

The directory */check* contains many useful files and directories. In */instances* are included a lot of files representing instances used to test the software. These files can have many formats: most used are *.mps*, *.lp* and *.cip*, the last one is also the format SCIP uses to print problems. Instances can be run either singularly or by means of a test set. In folder */testset* files with format *.test* and *.solu* can be found. *Test* files list a set of instances that can be run by simply invoking the file. Instances are identified by their relative path. *Solu* files include information about the best known solution for every instance listed in the corresponding *test* file. Best solution can be compared with the solution found by SCIP, automatically obtaining a measure of the quality of such solution.

In */check* is also contained */results* folder. In such a folder a few files are stored for every tested instance or set of instances. A file *.err* includes all the messages printed in the standard error device, allowing to obtain a log of debug issues arisen during the execution. Similarly, a file *.out* includes all the messages printed in the standard output device. A file *.res* displays execution results organized in a table. Results information is acquired by examining the *output* file. This operation is performed by an AWK small program, which performs the task of looking for information in the *.out* file and to properly build a table with desired data. In order to obtain the results displayed in chapter 4, I modified the standard *evalcheck.awk* file by adding some columns which were interesting for our purposes and excluding information that could be neglected.

2.3 Constraint Handlers

2.3.1 Role of constraint handlers

Since 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 suffices to turn SCIP into an algorithm which correctly solves CIPs with constraints of the supported type, at least if no continuous variables are involved. However, the resulting procedure would be a complete enumeration of all potential solutions, because no additional information about the problem structure would be available. To improve the performance of the solving process constraint handlers may provide additional algorithms and information about their constraints to the framework, namely:

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

2.3.2 Constraint handlers implementation

In the current section we are going to introduce which features a constraint handler is composed of. In section 3.3 a more detailed description is provided, focusing on the implementation of xprim constraint handler.

In general the main items of a constraint handler are:

- properties;
- additional parameters;
- data structures;
- interface methods;

- callback methods.

Some of the listed components have to be compulsorily adjusted or implemented, while many of them are optional. This aspect follows from the policy of SCIP of providing many possibilities and parameters to developers and users to cover the widest possible needs.

Constraint handler properties are given as compiler defines and are useful to properly tune the constraint handler itself. Such properties are used to identify the constraint handler and to relate it with others constraint handlers. The most important properties are name, description and priorities.

Additional parameters related to the constraint handler can be added to SCIP. Default values of such parameters are defined among the properties. Parameters can be tuned in the interactive shell in order to exploit constraint handler with a wider range of possibilities.

Two important data structures can be defined:

- struct SCIP_ConsData (constraint data);
- struct SCIP_ConshdlrData (constraint handler data).

Constraint data structure records information about every constraint managed by the constraint handler. Constraint handler data structure record all the information related to the constraint handler.

Two interface methods have to be implemented and are used to interface the constraint handler with SCIP.

The first interface method is used to include the constraint handler into SCIP, i.e. to make the constraint handler available to the model. In such method the memory for constraint handler data is allocated, and data are initialized. Also, in this case it is possible to add parameters and make them available from the interactive shell.

The second interface method is called to create a single constraint of the constraint handler's class. It looks for the already existent constraint handler, allocates and creates constraint data, and finally creates the constraint.

In conclusion, the fundamental aspect of implementation of constraint handlers is the use of callback methods. A callback is a piece of executable code that is passed as an argument to other code, which is expected to execute the argument at some convenient time.

The implementation of callbacks has far been the most demanding job to create a constraint handler.

2.4 Exact LP solvers

One of the features SCIP provides is to support many LP solvers. The complete list has been presented in section 2.1.3. SoPlex is part of the SCIP Optimization Suite, and can be a natural choice when more efficient commercial LP solvers are not available. SCIP and SoPlex guarantee good performances together, since SoPlex is optimized to work with SCIP.

However, standard version of SoPlex as well as all the other cited LP solvers, uses floating-point arithmetic. On the contrary, we would need to use an exact LP solver for the purpose of our application. Fortunately, in the last years exact LP solvers have been developed.

2.4.1 QSopt_ex

QSopt_ex is an exact LP solver, which provides an exact implementation of simplex algorithm. It has been developed by Applegate, Cook, Dash and Espinoza [3]. In the following paragraph a brief presentation of their work is introduced.

A first approach to obtain exact LP solutions has been to implement a solver that computed entirely in rational arithmetic. To achieve this, the authors began with the source code for the QSopt implementation of the simplex algorithm, then, changed every floating-point type into the rational type provided by the GNU-MP (GMP) library [16], and also changed every operation in the original code into use GMP operations. They tested the code, and results highlighted some factors leading to highly unpredictable behavior for the overall code, making this naive method impractical for most applications.

Dhiflaoui et al. [12] pioneered an alternative approach for obtaining exact LP solutions, by using the output of a floating-point solver as a starting point for rational computations. They found that in many cases the rational solutions is indeed optimal, testing a subset of NETLIB instances.

Koch [23] modified this approach to compute optimal solutions for the full set of NETLIB instances. Koch explained that in these calculations he employed the long double type, thus changing the representation from 64 to 128 bit floating-point arithmetic.

In order to obtain an exact solver, Applegate at al. [3] extended Koch's methodology with an implementation which dynamically increases the precision of the floating-point computations. The GMP library allows to perform floating-point calculations with arbitrary precision, moreover, to adjust this precision at running time. Finally, they ended up with a solver capable of achieving fast computation times on average and solving general LPs exactly over the rational numbers.

2.4.2 Iterative refinement

Gleixner, Steffy, and Wolter [18] further developed the concept used for QSopt_ex implementation. They observed that QSopt_ex is often very effective at finding quickly exact solutions quickly, but in some cases solution times can increase significantly.

Iterative refinement is a commonly applied technique for finding accurate solutions to linear system of equations. The authors applied this technique to Linear Programming.

The main idea of their algorithm is as follows. First, the LP solves approximately, producing a primal-dual solution x^*, y^*. Then, based on the error in x^*, y^*, a modified problem is created by shifting and scaling the primal and dual feasible regions of the original instance; a solution to this newly constructed problem gives a correction that is used to refine the accuracy of x^*, y^*. This process is iterated, correcting the candidate solution repeatedly, until it meets a required accuracy.

Recently, iterative refinement has been implemented for SoPlex together with others improvements, therefore an exact beta version of SoPlex is now available.

Chapter 3

Xprim Constraint Handler

In this chapter we will discuss about the constraint handler which has been implemented and plugged into SCIP during my internship at Zuse-Institute Berlin.

Section 3.1 presents aim and motivation of the current dissertation. In order to highlight the relevance of an exact precision in certain situations, dangers of floating-point arithmetic are treated. Follows the introduction to the wireless network design problem, which contributed to inspire this research. Furthermore, a few approaches developed in the last years are presented. Finally, some hypotheses regarding our implementation are justified.

In section 3.2 a detailed description of xprim constraint handler is provided. Main settings for the tool are discussed, as well as its initialization. Algorithm's behavior is then explored, and, at the end, used tolerances are taken into account.

In section 3.3 many aspects concerning xprim constraint handler implementation, already introduced in section 2.3, are deeply investigated.

3.1 Aim and motivation

3.1.1 Dangers of numerical computation

In section 1.3 we have observed that using floating-point calculations leads to imprecise results. However, numbers in SCIP are represented by using the double precision floating-point format. Due to a number of reasons, for many industrial MIP applications near optimal solutions are sufficient. Moreover, when data describing a problem arises from imprecise sources, exact feasibility is usually not necessary.

Nonetheless, accuracy is important in many settings. Direct examples arise in the use of MIP models to establish fundamental theoretical results and in subroutines for the construction of provably accurate cutting planes. Furthermore, industrial customers of MIP software request modules for exact solutions in critical applications.

There are several documented tragic errors involving floating-point computation. During the first US Gulf War patriot missiles were used to intercept SCUD missiles and the software controlling missiles was based on floating-point computations. Repeated use of the number 1/10 in the code, which is not representable exactly as a base-2 floating-point number, led to miscalculations that accumulated to form significant errors. On February 25, 1991, as a direct result of this miscalculation, a patriot missile failed to intercept an incoming Iraqi SCUD missile; it was off target by more than 0.6 kilometers and resulted in the death of 28 US soldiers [6]. In a later incident, the 1996 launch of the European Ariane 5 Rocket ended in failure when it went out of control and exploded 37 seconds into its flight path. The explosion was due to a software error caused by improper handling of a floating-point calculation; the software converted a 64-bit floating-point number to a 16-bit signed integer causing an overflow and system crash. The rocket and its cargo were worth an estimated 360 million USD [13, 27].

In the inexact setting, errors in the branch-and-bound process can be introduced at several different places: while reading in the instance, in the bounding step and feasibility test (because of the floating-point arithmetic and the consequent usage of tolerances), and also because of inaccurate LP solutions.

3.1.2 Wireless Network Design Problem

Wireless network design problem belong to the subset of problems requiring exact precision. It is interesting to use it as example since many difficulties and peculiarities of wireless network design problems are common to others imprecisely solved problems.

For modeling purposes, a wireless network can be described as a set of transmitters T that provide a telecommunication service to a set of receivers R. Transmitters and receivers are characterized by a location and a number of radio-electrical parameters (e.g., power emission and transmission frequency). The *Wireless Network Design Problem* (WND) consists in establishing the location and suitable values for the parameters of the transmitters with the goal of optimizing an objective function that points out the interest of the decision maker: common objectives are the maximization of a revenue function associated with wireless service coverage or the minimization of the total power emission of the network transmitter [9]. For an exhaustive introduction to the WND see [8, 10, 21].

Two main issues cause errors in the solutions returned by state-of-the-art MIP solvers [8, 9]:

- coefficients may vary in a wide range leading to very ill-conditioned coefficient matrices that make the solution process numerically unstable;
- natural formulations make use of big-M coefficients, leading to extremely weak bounds and linear relaxations.

What actually happens is that, for many WND instances, MIP solvers return solutions which are not feasible at all. Therefore, it is necessary to improve solver's precision.

3.1.3 Advances in exact MIP solving

Recently, different approaches have been developed in order to try to tackle the lack of precision of some provided solutions.

One straightforward strategy to exactly solve MIPs would be to implement the standard solution procedures entirely in exact arithmetic. Unfortunately, as introduced in section 2.4, it has been observed that optimization software relying exclusively on exact arithmetic can be prohibitively slow [3]. That motivates the development of more sophisticated algorithms to compute exact solutions.

Cook, Koch, Steffy, and Wolter [7] achieved a hybrid symbolic/numeric implementation of LP-based branch-and-bound, by using numerically-safe methods for all binding computations in the search tree. Since the authors exclusively focused on the branch-and-bound procedure, the exact solver they implemented is still not directly competitive with the full version of SCIP. However, it is realistic to think that the future inclusion of additional MIP machinery such as cutting planes, presolving, and primal heuristics into this exact framework, could lead to a full featured exact MIP solver that is not prohibitively slower than its inexact counterparts.

Gleixner and D'Andreagiovanni [9] proved that coefficient scaling, a practice consisting in multiplying coefficients for a given factor to avoid very small values, is useful to tighten the feasible region. This leads to better results, i.e. smaller constraint violations, but still is not sufficient to guarantee accurate feasibility of solutions returned by floating-point solvers. Authors also showed that advances in exact LP solving can be of help and suggested to integrate exact arithmetic into the branch-and-bound process. The current dissertation further develops such idea, by presenting an implementation aimed at the computation of exact primal solutions within the tree.

3.1.4 Integer and continuous parts of a solution

In the following paragraph a detailed tool description is provided, but there is also an additional point we need to highlight. That is the explanation about some choices we did while implementing the constraint handler is then presented.

In particular we focused on a practical issue of WND that is common to many other problems characterized by feasibility troubles. As example let's consider the following specific problem related to WND [21]. Given a set L of candidate locations for transmitters and a set of receivers R, select a subset of location where placing transmitters in order to serve the maximum number of receivers. Binary variables can be associated to candidate locations stating whether the location is used or not. Every transmitter can be tuned to different power levels, represented by continuous variables. These two families of variables, binary and continuous, are both necessary to find an optimal solution. However it can be noticed that, in a practical viewpoint, the implementation of the binary part of the solution, i.e. the construction of transmitters in proper places, comes first than the continuous part, i.e. tuning constructed transmitters. This means that we are strongly interested in having a correct binary part of the solution.

Previous example can be extended to many other problems where the integer part of the solution has to be implemented first, while the continuous part can be considered in a further step. In other words, what we want to do is to check whether, given a fixing of integer variables, a feasible solution exists.

3.2 Tool description

3.2.1 Analysis and enforce modes

The tool we implemented is a constraint handler plugged into SCIP, where floating-point solutions are checked and exact primal solutions provided. From here on we will call such tool *xprim constraint handler*, standing *xprim* for *exact primal*.

Although in section 3.1.2 we focused on wireless network design problem, xprim constraint handler is a very general tool that can be useful every time we want to improve solution's accuracy. It is also intended to provide information about floating-point solver's precision.

Many parameters are associated to xprim constraint handler, however we can mainly distinguish two different modes: *analysis mode* and *enforce mode*. In analysis mode the tool is passive since it simply checks the best solution found by SCIP and store information about it. This mode is used to collect statistics about how the actual version of SCIP works using floating-point numbers, since it doesn't influence at all its evolution. The main functionality provided is to check whether SCIP best solution is or not an exactly feasible solution. It is also possible to know whether a possible infeasibility is due to either the integer or the continuous part of the solution. Moreover, constraint violations are computed to singularly check every constraint. Since SCIP computations are performed up to a certain tolerance (10^{-6} by default), when feasible solutions are obtained we can expect to find out violations less than the tolerance, but in general different from zero.

In enforce mode the tool is invoked for every feasible solution found by SCIP and it returns feedback to SCIP influencing its behavior within the tree. This mode's target is to improve the quality of the solutions found by SCIP, discarding infeasible solutions and suggesting exact feasible solutions, when available.

The two modes share the same structure and many features. However, they clearly differ in some aspects. Since analysis mode can be considered a subset of enforce mode where only the last check is performed, in the following paragraphs the latter will be mainly considered.

3.2.2 Initialization

Every candidate solution found by SCIP solver needs to be checked by all the constraint handlers in a sequential way. Xprim constraint handler is set to be the last invoked constraint handler, so that the input solution has already been confirmed to be feasible for all the other constraint handlers. There is a couple of explanations for that. Since the exact check is very time consuming it would be better to have the least possible number of solutions to be checked by xprim constraint handler, i.e. every solution that is infeasible for other reasons has to be discarded by previous and faster constraint handlers. Moreover, one of the purposes of xprim constraint handler is to return some statistics about the behavior of actual standard SCIP version.

Xprim constraint handler is initialized by storing an exact copy of the original problem. Constraints are obtained by means of a translation from floating-point original problem data to rational data. They are stored in the structure called *SCIP_ConsData*, already introduced in chapter 2.3, where constraints are characterized by sides, variables and coefficients. The structure also records some other useful information related to single constraints. Problem variables are characterized by bounds and objective value, i.e. the coefficient associated with the variable in the objective function. In the first tool version these values were translated from floating-point values, but in successive updates the possibility of reading rational values directly has been implemented. This improvement enhances data precision, since it removes errors due to the translation from floating-point to rationals.

The first time xprim constraint handler is invoked, an exact LP problem is set up. Originally fixed variables are marked for optimality purposes, i.e. they will not be further checked to save computation time. Constraints are divided into two subsets: integer constraints, having only integer variables, and continuous constraints, having at least one continuous variable. For optimality reasons, continuous constraints are sorted in order to have non-fixed continuous variables first. Coefficients and sides are updated by fixing values of originally fixed variables. A hash-map table is set up in order to create a mapping between original problem variables and column indexes in the exact LP.

3.2.3 Algorithm

After the set up, xprim constraint handler behavior can be summarized as follows:

- it is invoked every time a feasible solution is found by SCIP;

- solution's integer part is rounded to the nearest integer;
- integer constraints, i.e. constraints containing only integer variables, are locally checked for feasibility;
- an exact LP is created and solved by an exact LP solver;
- exact solution and/or conflicts may be added to SCIP.

The first operation to perform every time the exact check is requested is to verify the correctness of fixings. Values of integer variables obtained by SCIP incumbent solution are rounded to the nearest integer in order to remove possible approximation errors. In general, the value of an integer variable that is supposed to be n, can be a value in the interval $[n-\varepsilon, n+\varepsilon]$, $\varepsilon < t$, where t is the tolerance used by SCIP.

Integer constraints are then checked considering rounded solution. In case all constraints result to be feasible, all integer values are assumed to be correct. Consequently, left hand and right hand sides of continuous constraints are updated, leaving only continuous variables as unknowns. In the event that at least one infeasibility is observed, a conflict representing the integer part of candidate solution is added to the original problem. The new constraint is a set covering in case all involved variables are binary, otherwise is a bound disjunction. This operation allows to avoid to consider again a solution having such (infeasible) integer part as candidate.

The second step consists in solving the exact LP. QSopt_ex and SoPlex can be used as exact solvers. It is important to note that the solution returned by the exact LP solver, if it exists, and the continuous part of floating-point SCIP solution are independent to each other. This follows from their different formulations, which lead, in general, to two completely different feasible solutions. This means that finding an exact LP solution does not imply that candidate solution is feasible. However, it is guaranteed that an exact feasible solution having rounded fixings as integer part and exact LP solution as continuous part exists.

Another possible approach to use in future improvements may be to solve a different exact LP aimed at minimizing the distance from the floating-point SCIP solution. With such formulation we could also estimate the magnitude of the errors on each variable.

The third and last step consists in returning feedback to SCIP. Let's consider the case exact LP admits a feasible solution. In analysis mode it is simply possible to conclude that a feasible solution exists, and such solution is represented by the union of rounded integer part of SCIP solution and exact LP solution.

On the other hand, in enforce mode we can add the exact solution to SCIP. This guarantees that all feasible solutions in SCIP are exactly feasible, and consequently also the best solution will be exactly feasible. A conflict is also created after having passed the exact solution. This avoids to check candidate solutions which only differ in the continuous part. Indeed, in theory, an infinite number of candidate solutions having identical integer part can be generated, but for each of them xprim constraint handler would return the same result. If no exact feasible solutions are found, only the conflict is added indicating that the search for exact feasible solutions has failed.

3.2.4 Tolerances

It is possible to perform the previously described computations with an arbitrary degree of tolerance. This option has been introduced to provide a wider use of this tool.

Sometimes it is hard to perfectly achieve exactness. In our application it can be due to small errors in the conversion from floating-point to rational numbers. Moreover, for some application it may be sufficient to achieve a certain degree of precision.

It is possible to relax variables' bounds and constraints' sides. Bounds can be relaxed while the exact LP problem is constructed, since it would make no sense to apply tolerances to integer variables. Sides tolerances are implemented differently according to the chosen exact LP solver. SoPlex supports tolerances, in the sense that it is possible to set directly the degree of tolerance inside the LP solver. Vice-versa, when one uses QSopt_ex it is necessary to apply tolerances in the constraint handler before the exact LP is constructed. Then the solver will perform exact computations on a larger feasible region.

3.3 Implementation details

In section 2.3.2 constraint handlers implementation has been introduced. Starting from the sketch depicted there, we will now go deeper into the details of the implementation of xprim constraint handler.

3.3.1 Properties

There are many properties to tune for every constraint handler. The following list shows the main properties that have been set up for xprim constraint handler, neglecting the less relevant ones.

Name

CONSHDLR_NAME defines the name of the constraint handler. The name is used in the interactive shell to address the constraint handler and for this reason it has to be unique. The name assigned to xprim constraint handler is *xprim*.

Description

CONSHDLR_DESC provides a description of the constraint handler. Such description is displayed in the interactive shell of SCIP. Xprim constraint handler is briefly described as constraint handler for exact primal solutions.

Enforce and Check Priority

CONSHDLR_ENFOPRIORITY and *CONSHDLR_CHECKPRIORITY* represent the priorities of the constraint handler for constraint enforcing and checking feasibility. These two numerical values define the order constraint handlers are called with, either during the constraint enforcement or to check the feasibility of a given primal solution candidate. Constraint handlers are called in order of non decreasing priority, i.e. high values for these parameters correspond to high priorities.

The integrality constraint handler has an enforcement priority of 0. That means, if a constraint handler has negative priorities, it has only to deal with integral solutions. The priority should be set according to the complexity of the algorithm and the impact of the results. Constraint handlers that provide fast algorithms, which have usually a high impact, should have higher priority.

Since xprim constraint handler is very time consuming and restrictive, it has been assigned to very low priority. This is consistent with the aim of studying how SCIP worked before the implementation of this tool. In fact, xprim constraint handler is the last constraint handler to check feasibility, and therefore statistics about floating-point SCIP behavior can be collected.

Eager Frequency

CONSHDLR_EAGERFREQ indicates the default frequency applied for separation, propagation and enforcement in order to use all the constraints instead of the useful ones only. When constraint aging is activated, some constraints which were not useful in the past for propagation or separation are marked to be obsolete and they are not used anymore. However, every n'th call, with n being the *EAGERFREQ* of the constraint handler, also obsolete constraints are presented to the separation and propagation methods of the constraint handler.

Needs Constraints

CONSHDLR_NEEDSCONS indicates whether the constraint handler should be skipped, in case no constraints of its class are available. For xprim constraint handler this property is set to *TRUE*. That means, the constraint handler is only executed if there are constraints of its corresponding class in the model. In fact, it would make no sense to invoke xprim constraint handler if rational constraints have not been created.

3.3.2 Additional parameters

Some parameters have been added to xprim constraint handler to properly set up it depending on what we want to obtain. Default values have been defined among the properties but they can be changed in the interactive shell. The list of such parameters follows.

Use Tolerances

USETOL is set to *TRUE* if we want to relax bounds and sides, to *FALSE* otherwise. The *TOLERANCE* to apply is a given constant.

The default value is set to *TRUE*, and the *TOLERANCE* is 10^{-15}, much smaller than SCIP tolerance, which default value is 10^{-6}. This allows to obtain very precise solutions, but taking into account small inaccuracies.

Use Time Limit

USETIMELIMIT is set to *TRUE* if a time limit should be passed to the exact LP solver, to *FALSE* otherwise. It is useful to kill the execution of the LP solver when it exceeds a certain time limit.

Add Solution

ADDSOL is set to *TRUE* if the exact solution has to be added to the problem, to *FALSE* otherwise. Its default value is *TRUE*, since we usually want to return some feedback to SCIP when we use enforce mode.

Return Feasible

RETURNFEAS is set to *TRUE* if, in enforce mode, we always want to return to SCIP that the candidate solution is feasible. Its default value is *FALSE*. It can be used to set up a hybrid mode between analysis mode and enforce mode. Indeed, if *RETURNFEAS* is *TRUE* and we are in enforce mode, all the primal solution candidates are checked without influencing SCIP behavior, just collecting data about candidates.

Relative Difference

RELDIFF indicates how much the candidate solution should be better compared to the incumbent solution in order to be checked. If *RELDIFF* is equal to 0 it means that xprim constraint handler only checks improving solutions. Negative values allow to consider also suboptimal solutions. It can be interesting to check also slightly suboptimal solutions, because after an exact check they can come out to be actually better than incumbent solution. Positive values can be used when we do not want to waste time in checking solutions that does not improve the objective value enough.

Use SoPlex

USESOPLEX is a boolean value indicating the exact LP solver to use. SoPlex is used when this parameter is set to *TRUE*, QSopt_ex is used otherwise.

3.3.3 Data structures

Constraint Data

The constraint data are the information needed to define a single constraint of the constraint handler's class. In xprim constraint handler, constraints are defined by the following fields.

- Left hand side and right hand side. The peculiarity of these two fields in xprim constraint handler is that they are rational values.

- Array of variables. It contains a pointer to each of the variables having nonzero coefficients.
- Array of coefficients. It contains all the nonzero coefficients related to constraint variables.
- Number of variables. It is the number of elements in the array of variables.
- Number of continuous non-fixed variables. This integer value records the number of continuous variables that have not been fixed in the original problem.
- Row index. This integer value records the index of the row corresponding to the constraint in the LP solver.
- Two boolean flags used to record whether operations of sorting variables and applying tolerances have already been executed for the constraint.

Constraint Handler Data

The constraint handler data are additional variables, that belong to the constraint handler itself and which are not specific to a single constraint. Although the implementation of this structure is optional, it has been strongly used in xprim constraint handler. The most important fields of constraint handler data are illustrated below.

- A pointer to the exact LP solver.
- A pointer to trysol heuristic. It is the heuristic exact solutions are passed to when enforce mode is used and *ADDSOL* parameter is set to *TRUE*.
- An hash-map table. It is used to store the mapping between original variables and column indexes in the exact LP.
- An array containing a copy of all the original variables.
- An array of boolean values associated to variables and indicating whether a variable is fixed or not.
- Integer values indicating the number of variables. In particular we record the total number of variables in the original problem, the number

of binary variables, the number of integer variables, the number of implicit variables and the number of continuous variables.

- All the original constraints are copied, divided into two different arrays: the first containing integer constraints, i.e. constraints containing only integer variables; the second containing continuous constraints, i.e. constraints having at least one continuous variable.
- Integer values indicating the number of constraints. In particular we record the total number of constraints in the original problem, as well as the number of integer constraints and the number of continuous constraints.
- Pointers to last and best solutions' integer part. Every time the constraint handler has to check a primal solution candidate, such solution's integer part is compared to last and best ones. This operation is useful to avoid multiple checking of solutions having the same integer part. In fact, it can happen in SCIP to have candidates having the same integer part and a different continuous part, but it makes no sense to check them all, since if all the integer variables have the same value, then the exact LP will return the same solution for continuous variables every time.
- Parameters (see section 3.3.2).
- Statistics (see chapter 4).

3.3.4 Interface methods

Interface methods described in section 2.3.2 have been implemented for xprim constraint handler.

The only peculiarity of xprim constraint handler is that two ways of creating a new constraint are possible. In fact we have the possibility to create an xprim constraint starting from floating-point data or from rational data. In the first case data need to be translated from floating-point to rationals.

3.3.5 Callback methods

The implementation of callback methods have been the most important activity to perform.

Fundamental callbacks

The most important callbacks are the ones dealing with the feasibility of a given solution. There are three different methods doing that, with slightly different meaning. They are called *CONSCHECK*, *CONSENFOLP* and *CONSENFOPS*.

Since their functions are similar, in xprim constraint handler they all call the same method, whose behavior is thoroughly illustrated in chapter 3.2.

The difference among these methods lies in the temporal moment they are called and the kind of solution they have to check.

The *CONSCHECK* callback gets a primal solution candidate in the form of a pointer to *SCIP_SOL*, i.e. the data structure used in SCIP to store solutions. Then such solution has to be checked for global feasibility. The *CONSCHECK* method has to return a result *SCIP_FEASIBLE*, if the solution satisfies all the constraints of the constraint handler, and a result *SCIP_INFEASIBLE*, if there is at least one constraint that is violated.

The *CONSENFOLP* callback is called after the price-and-cut loop has finished and an LP solution is available. That means, solution is not given as a pointer to *SCIP_SOL* data structure, but variables can be accessed by other methods. Like *CONSCHECK* call, *CONSENFOLP* method should return a result *SCIP_FEASIBLE* if the solution satisfies all the constraints. However, the behavior should be different if the solution violates one or more constraints. The constraint handler may return a result *SCIP_INFEASIBLE* in this situation, but this is not the best what one can do because the *CONSENFOLP* method has the possibility of resolving the infeasibility by:

- stating that the current subproblem is infeasible (result *SCIP_CUTOFF*);
- adding an additional constraint that resolves the infeasibility (result *SCIP_CONSADDED*);
- reducing the domain of a variable (result *SCIP_REDUCEDDOM*);
- adding a cutting plane (result *SCIP_SEPARATED*);
- performing a branching (result *SCIP_BRANCHED*).

The *CONSENFOPS* callback is similar to the *CONSENFOLP*, as the main difference lies in the fact the former is invoked with pseudo solutions instead of LP solutions. Pseudo solutions are used when the LP is not solved at the current subproblem, and they can be thought as the solution to the LP relaxation with all constraints except the bounds.

Unlike the *CONSENFOLP* callback, the *CONSENFOPS* callback must not add cutting planes. However, it can force the solving of the LP by returning the result *SCIP_SOLVELP*.

As can be deducted from the description of how xprim constraint handler works (chapter 3.2) and considering what previously discussed, *CONS-ADDED* is returned every time a conflict is created and therefore a new constraint is added. Despite the impossibility of *CONSCHECK* callback to return such result, it is possible to add conflicts also while checking primal solution.

Additional callbacks There are many other callbacks that can be implemented for a constraint handler. A short description of the main additional callbacks implemented in xprim constraint handler is provided below.

The *CONSFREE* callback is the destructor of constraint handler to free constraint handler data. Furthermore, in *CONSFREE* all the allocated memory for fields in constraint handler data has to be freed. Also, in xprim constraint handler, the execution of the exact solver is terminated. *CONSFREE* is called when SCIP is exiting.

The *CONSPRINT* callback is used when the user asks SCIP to display the problem on the screen or save the problem into a file. The output format that is defined by *CONSPRINT* is called CIP format.

The *CONSPARSE* callback is the counter part to *CONSPRINT*. This method allows the constraint handler to parse and consequently to read problems in CIP format.

Finally, *CONSGETVARS* and *CONSGETNVARS* callbacks return the variables and the number of variables, respectively.

3.3.6 GMP Library

The GNU Multiple Precision Arithmetic Library (GMP) [16] offers routines for infinite-precision rational arithmetic. In contrast to the commonly used finite-precision arithmetic systems, GMP dynamically allocates as much memory as is necessary to exactly represent numbers and is limited only by the available system memory.

For xprim constraint handler implementation, high-level rational arithmetic functions have been exploited. In GMP every operation with rational precision is identified by the prefix *mpq_*. Rational numbers are stored as a couple of integer values of arbitrary length, representing the numerator and denominator. The object allowing such representation is the data type *mpq_t*. All rational arithmetic functions assume operands have a canonical form, and canonicalize their result. The canonical form means that the denominator and the numerator have no common factors, and that the denominator is positive.

A total of 35 methods are then available in order to initialize and assign rationals, to convert from and to other data types, to perform arithmetic computations, to compare rational numbers and to perform input from a standard I/O stream and output to a standard I/O stream.

Chapter 4

Computational experiments

The last chapter presents the computational experiments we conducted in order to test xprim constraint handler.

The goal of our experiments was twofold: first, in order to test the quality of the solutions currently provided by floating-point SCIP, we analyzed the accuracy of such solutions by means of exact checking; second, we investigated whether the use of xprim constraint handler within the tree can lead to a significant improvement of the solutions' accuracy, without lowering performances too much.

Section 4.1 briefly presents the instances we tested, subdivided into three sets.

In section 4.2 we discuss the first experiment, performed in analysis mode. We will focus on the setting chosen for the test, as well as on the tables and the results we can deduce.

In section 4.3 we discuss the second experiment, performed in enforce mode, and we try to investigate the effects of xprim constraint handler in SCIP execution.

4.1 Instances

The first test set includes instances belonging to the MIPLIB 2010 benchmark set [25]. Such instances are classified to be easy, i.e. they can be solved by one hour using a commercial solver.

The second test set is the MIPLIB 2010 unstable set [26]. Since such instances are numerically unstable, they provide a more challenging test for our tool. In the unstable set we have easy instances, as well as hard and open instances. Hard instances have been somehow solved, although not in an easy way. Open instances are the most challenging, since the optimal solution is still unknown.

The third test set is composed by realistic instances derived from several problems. Five instances are a subset of the scaled instances exploited by D'Andreagiovanni and Gleixner in [9], and represent a WiMAX network. Such instances constitute a valid example of WND problem, introduced in section 3.1.2. Nine instances come from the Radio Resource Assignment problem and the last one from OFDM mobile systems [28].

4.2 Analysis mode

In the first experiment we ran SCIP in analysis mode for the three sets of instances introduced in section 4.1. Settings included a time limit of one hour and a tolerance of 10^{-15}. SoPlex has been used as exact LP solver, since a few experiments showed better performances compared to QSopt_ex, at least when coupled with SCIP. However, no proofs of this statement are provided, since a comparison between the two exact solvers is not part of this dissertation: we consider this empirical result to exclusively justify our choice.

As provided by analysis mode, standard floating-point SCIP ran freely, and eventually the best solution was exactly checked.

4.2.1 Tables

Table 4.1 shows the results for MIPLIB benchmark set of instances, Table 4.2 shows the results for MIPLIB unstable set of instances, Table 4.3 shows the results for the set of realistic instances.

The first column gives either the name of each instance or a contraction of the name. The second and third columns characterize the instance with the number of constraints and variables in the original formulation.

The fourth column, named *cand*, indicates the number of candidate solutions checked by xprim constraint handler. Since we are using analysis mode, only two values are available: *1* means that the best solution has been checked, while *0* means that no solution were found within the time limit.

The fifth, sixth and seventh columns provide a count of checking results. The sum of such three columns must necessarily correspond to the number of candidates. The column named *feas* displays the number of feasible solutions; the one named *int* displays the number of solutions for which an infeasibility has been detected while checking the fixings, i.e. an integer constraint was violated; the last column, named *LP*, displays the number of solutions for which, although the fixings were correct, the exact LP solver returned *infeasible*. Again, analysis mode impose a restriction to these values. When a best solution is found and checked, one of these columns is marked with *1*. When no solutions have been found, all of these columns are valued as *0*.

The eighth, ninth and tenth columns show statistics about relative violations in the continuous constraints. Such statistics are collected by means of a check performed on each constraint, after integer variables have been fixed. Let's consider a constraint in the general form $l \leq ax \leq r$, where a is a vector of coefficients, x is a vector of variables, l and r are the left hand side and right hand side, respectively. The absolute violation v_{abs} of a constraint is obtained as

$$v_{abs} = \max\{ax - l, r - ax\}. \tag{4.1}$$

If $v_{abs} \leq 0$, the inequality is respected for both sides and the constraint is not violated. On the contrary, if $v_{abs} > 0$, the constraint is violated and such value is recorded. In order to have comparable results among all the constraints, we transform the absolute violation v_{abs} into a relative violation v_{rel} by normalizing the former. Therefore, we obtain the relative violation as

$$v_{rel} = \frac{v_{abs}}{\max\{ax, l, r, 1\}}. \tag{4.2}$$

In the following the description of the three columns containing information about relative violations. The column named *high* displays the number of constraints presenting a violation larger than the defined tolerance. The column named *tot* displays the total number of constraints affected by a violation. The column named Δmax displays the largest recorded violation.

The eleventh column, named Δobj, represents the difference between the objective value computed for the floating-point SCIP solution and the one computed for the exact solution.

The twelfth and last column gives the execution time of each instance expressed in seconds. Since the time limit is set to one hour, a time equal to $3600s$ indicates that such instance stopped its execution because of time limit reached.

4.2.2 Results

In this section we are going to analyze data retrieved in analysis mode and shown by tables 4.1, 4.2 and 4.3.

Before starting the discussion, a general observation is necessary for all the tables of both analysis mode and enforce mode. Not all the instances of respective sets are present in the tables. This is due to the fact that the solver did not return any result for such instances. That can be explained by considering that the exact SoPlex does not provide a time limit after which the execution is stopped. It may be happened that for some instances the SCIP time limit has been reached while the exact LP was executing, and there were no ways to communicate that. Therefore, the exact solver may have continued its execution indefinitely, with no possibilities to return results.

The entity of such phenomenon is more evident in enforce mode, while in analysis mode is quite rare. Although this problem should be fixed and has to be taken into account, it does not limit our discussion. It can be observed that the instances never returned are quite hard to solve, showing complexities or reaching the time limit in analysis mode.

Looking at the tables, the first thing we can note is that the number of original constraints for each instance is doubled compared to the values we would obtain with standard SCIP. This can be explained by the fact that every original constraint of each instance is copied with an exact representation and stored in the constraint handler data (see section 3.3.3).

Let's start considering table 4.1. Only 5 instances have no feasible solutions found within the time limit. Instances *ash608gpia-3col*, *enlight14* and *ns1766074* terminate in less than one hour, but it is known they are infeasible. Instances *mspp16* and *neos-1601936* are known to be feasible, but they both exceed the time limit and no feasible solution is found.

More than a half on the instances (47 out of 87) presents neither a violation nor a difference in the objective value, indicating that floating-point SCIP solutions were exact. Since the instances in the MIPLIB 2010 benchmark set are generally very stable, this result could be expected.

However, there are several instances with significantly inexact solutions. Most evident inaccuracies affect instances *ns1208400* and *rocII-4-11*. The former's best solution is infeasible since an integer constraint is violated. Vice-versa, in the latter fixings are correct, but the obtained LP is not exactly feasible. In the other instances, we have maximum violations which go from 10^{-7} to 10^{-16}. This result is consistent to the feasibility tolerance imposed by SCIP, which is equal to 10^{-6}. Maximum difference between floating-point and exact objective values are even smaller, going from 10^{-11} to 10^{-16}.

The results of the MIPLIB unstable set are shown in table 4.2. Although we would expect more troubles compared to the benchmark set of instances, none of unstable ones return an infeasible best solution. However, all but one instances do not terminate their execution within the time limit, and many of them do not return any solution. Violations and differences in the objective values are similar to the previous.

Realistic instances are shown in table 4.3. Although all of the instances return a feasible solution, they also exhibit more relevant violations and differences in the objective values compared to MIPLIB instances. Violations larger than a factor of 10^{-10}, which were infrequent for MIPLIB instances, are in this case quite common. Furthermore, the difference between the objective values of floating-point and correspondent exact solutions reach a magnitude of 10^{-5}.

All of these factors lead to a first observation. The realistic instances we tested are not enough hard to solve for a floating-point MIP solver like SCIP. This conclusion is supported by an empirical fact. Indeed, it has been observed that WND instances having a magnitude around 10^{22} between the largest and the lowest coefficients were numerically difficult, in the sense that usually the best solution returned by a floating-point MIP solver was actually infeasible. WiMax instances, for example, are characterized by a maximum magnitude of 10^{12} between largest and lowest coefficients, while radio resource assignment instances have a peak of 10^{14}.

Table 4.1: MIPLIB 2010 benchmark set of instances in analysis mode

	Original				Infeas.		Violations				
name	conss	vars	cand	feas	int	LP	high	tot	Δmax	Δobj	time
30n20b8	1152	18380	1	1	0	0	0	0	0	0	193.0
acc-tight5	6104	1339	1	1	0	0	0	0	0	0	108.0
aflow40b	2884	2728	1	1	0	0	0	0	0	0	2246.9
air04	1646	8904	1	1	0	0	0	0	0	$7 \cdot 10^{-11}$	106.5
app1-2	106934	26871	1	1	0	0	1252	5367	$4 \cdot 10^{-15}$	0	3600.0
ash608gpia.	49496	3651	0	0	0	0	0	0	0	0	36.0
bab5	9928	21600	1	1	0	0	0	0	0	0	3600.0

continued on next page

continued from previous page

	Original				Infeas.		Violations				
name	conss	vars	cand	feas	int	LP	high	tot	Δmax	Δobj	time
beasleyC3	3500	2500	1	1	0	0	0	0	0	0	3600.0
biella1	2406	7328	1	1	0	0	2	2	$2 \cdot 10^{-14}$	$3 \cdot 10^{-14}$	1126.3
bienst2	1152	505	1	1	0	0	12	19	$1 \cdot 10^{-14}$	$2 \cdot 10^{-15}$	499.7
binkar10_1	2052	2298	1	1	0	0	0	4	$6 \cdot 10^{-16}$	$3 \cdot 10^{-15}$	296.0
bley_xl1	351240	5831	1	1	0	0	0	0	0	0	477.6
bnatt350	9846	3150	1	1	0	0	0	0	0	0	1556.0
core2536-691	5078	15293	1	1	0	0	0	0	0	0	871.5
cov1075	1274	120	1	1	0	0	0	0	0	0	3600.0
csched010	702	1758	1	1	0	0	0	0	0	$3 \cdot 10^{-16}$	3600.0
danoint	1328	521	1	1	0	0	95	104	$2 \cdot 10^{-12}$	$5 \cdot 10^{-15}$	3600.0
dfn-gwin-UUM	316	938	1	1	0	0	0	0	0	0	194.7
eil33-2	64	4516	1	1	0	0	0	0	0	$2 \cdot 10^{-14}$	100.2
eilB101	200	2818	1	1	0	0	0	0	0	$2 \cdot 10^{-16}$	456.5
enlight13	338	338	1	1	0	0	0	0	0	0	68.1
enlight14	392	392	0	0	0	0	0	0	0	0	0.0
ex9	81924	10404	1	1	0	0	0	0	0	0	126.7
glass4	792	322	1	1	0	0	6	10	$1 \cdot 10^{-7}$	$2 \cdot 10^{-12}$	3600.0
gmu-35-40	848	1205	1	1	0	0	5	5	$2 \cdot 10^{-13}$	0	3600.0
iis-100-0-cov	7662	100	1	1	0	0	0	0	0	$1 \cdot 10^{-16}$	915.7
iis-bupa-cov	9606	345	1	1	0	0	0	0	0	0	3600.0
iis-pima-cov	14402	768	1	1	0	0	0	0	0	0	474.9
lectsched-4.	28326	7901	1	1	0	0	0	0	0	0	937.8
m100n500k4r1	200	500	1	1	0	0	0	0	0	$4 \cdot 10^{-13}$	3600.0
macrophage	6328	2260	1	1	0	0	0	0	0	0	3600.0
map18	657636	164547	1	1	0	0	0	0	0	$6 \cdot 10^{-15}$	792.0
map20	657636	164547	1	1	0	0	0	0	0	$5 \cdot 10^{-15}$	541.6
mcsched	4214	1747	1	1	0	0	0	0	0	$1 \cdot 10^{-16}$	292.1
mik-250-1.	302	251	1	1	0	0	0	0	0	0	614.1
mine-166-5	16858	830	1	1	0	0	0	0	0	0	38.1
mine-90-10	12540	900	1	1	0	0	0	0	0	0	470.5
msc98-ip	31700	21143	1	1	0	0	0	0	0	0	3600.0
mspp16	1123314	29280	0	0	0	0	0	0	0	0	3600.0
mzzv11	18998	10240	1	1	0	0	0	0	0	0	688.9
n3seq24	12088	119856	1	1	0	0	0	0	0	0	3600.0
n4-3	2472	3596	1	1	0	0	24	34	$9 \cdot 10^{-13}$	0	820.8
neos-1109824	57958	1520	1	1	0	0	0	0	0	0	389.7
neos-1337307	11374	2840	1	1	0	0	0	0	0	0	3600.0
neos-1396125	2988	1161	1	1	0	0	0	0	0	$2 \cdot 10^{-16}$	726.3
neos13	41704	1827	1	1	0	0	0	0	0	$2 \cdot 10^{-16}$	817.2
neos-1601936	6262	4446	0	0	0	0	0	0	0	0	3600.0

continued on next page

continued from previous page

	Original				Infeas.		Violations				
name	conss	vars	cand	feas	int	LP	high	tot	Δmax	Δobj	time
neos18	22804	3312	1	1	0	0	0	0	0	0	50.6
neos-476283	20030	11915	1	1	0	0	6	15	$1 \cdot 10^{-14}$	$3 \cdot 10^{-16}$	357.7
neos-686190	7328	3660	1	1	0	0	0	0	0	0	101.6
neos-849702	2082	1737	1	1	0	0	0	0	0	0	1838.9
neos-916792	3818	1474	1	1	0	0	57	208	$2 \cdot 10^{-13}$	$2 \cdot 10^{-15}$	3600.0
neos-934278	22990	23123	1	1	0	0	0	0	0	0	3600.0
net12	28042	14115	1	1	0	0	0	0	0	0	3600.0
netdiversion	239178	129180	1	1	0	0	0	0	0	0	3600.0
newdano	1152	505	1	1	0	0	2	2	$3 \cdot 10^{-14}$	$4 \cdot 10^{-16}$	3600.0
noswot	364	128	1	1	0	0	1	1	$7 \cdot 10^{-15}$	0	814.6
ns1208400	8578	2883	1	0	1	0	0	0	0	-	2119.4
ns1688347	8382	2685	1	1	0	0	0	0	0	0	465.7
ns1758913	1248332	17956	1	1	0	0	120	120	$3 \cdot 10^{-7}$	$7 \cdot 10^{-12}$	3600.0
ns1766074	364	100	0	0	0	0	0	0	0	0	1891.3
ns1830653	5864	1629	1	1	0	0	0	0	0	$7 \cdot 10^{-15}$	999.5
opm2-z7-s2	63596	2023	1	1	0	0	0	0	0	0	954.3
pg5_34	450	2600	1	1	0	0	17	31	$3 \cdot 10^{-12}$	0	2249.8
pigeon-10	1862	490	1	1	0	0	0	0	0	0	3600.0
pw-myciel4	16328	1059	1	1	0	0	0	0	0	0	3600.0
qiu	2384	840	1	1	0	0	0	1	$2 \cdot 10^{-16}$	$5 \cdot 10^{-14}$	123.9
rail507	1018	63019	1	1	0	0	0	0	0	0	365.5
ran16x16	576	512	1	1	0	0	4	5	$4 \cdot 10^{-14}$	0	482.1
reblock67	5046	670	1	1	0	0	0	0	0	0	253.3
rmatr100-p10	14520	7359	1	1	0	0	0	0	0	$4 \cdot 10^{-15}$	192.0
rmatr100-p5	17370	8784	1	1	0	0	0	0	0	$9 \cdot 10^{-15}$	332.5
rmine6	14156	1096	1	1	0	0	0	0	0	0	3600.0
rocII-4-11	43476	9234	1	0	0	1	3	6	$1 \cdot 10^{-6}$	-	3542.2
rococoC10.	2586	3117	1	1	0	0	0	0	0	0	2337.5
roll3000	4590	1166	1	1	0	0	12	14	$3 \cdot 10^{-13}$	$1 \cdot 10^{-16}$	3600.0
sat.1-25	11992	9013	1	1	0	0	16	142	$1 \cdot 10^{-10}$	$5 \cdot 10^{-16}$	2158.6
sp98ic	1650	10894	1	1	0	0	0	0	0	0	3600.0
sp98ir	3062	1680	1	1	0	0	0	0	0	$2 \cdot 10^{-15}$	92.4
tanglegram1	136684	34759	1	1	0	0	0	0	0	0	1846.5
tanglegram2	17960	4714	1	1	0	0	0	0	0	0	21.3
timtab1	342	397	1	1	0	0	0	0	0	$2 \cdot 10^{-16}$	628.9
triptim1	31412	30055	1	1	0	0	2	2	$2 \cdot 10^{-11}$	$4 \cdot 10^{-15}$	3580.1
unitcal_7	97878	25755	1	1	0	0	58	97	$3 \cdot 10^{-11}$	$2 \cdot 10^{-15}$	3159.4
vpphard	94560	51471	1	1	0	0	0	0	0	0	3600.0
zib54-UUE	3618	5150	1	1	0	0	1	4	$3 \cdot 10^{-14}$	0	3600.0

Table 4.2: MIPLIB 2010 unstable set of instances in analysis mode

	Original				Infeas.		Violations				
name	conss	vars	cand	feas	int	LP	high	tot	Δmax	Δobj	time
cdma	18190	7891	0	0	0	0	0	0	0	0	3600.0
ger50_17_trans	998	22414	1	1	0	0	26	90	$4 \cdot 10^{-11}$	0	3600.0
harp2	224	2993	1	1	0	0	0	0	0	0	3600.0
momentum2	48474	3732	0	0	0	0	0	0	0	0	3600.0
nb10tb	300990	73340	0	0	0	0	0	0	0	0	3600.0
neos-1112782	4230	4140	1	1	0	0	0	4	$8 \cdot 10^{-16}$	$2 \cdot 10^{-16}$	3600.0
neos-1112787	3360	3280	1	1	0	0	0	3	$6 \cdot 10^{-16}$	0	3600.0
neos-1140050	7590	40320	0	0	0	0	0	0	0	0	3600.0
neos-1225589	1350	1300	1	1	0	0	0	0	0	$2 \cdot 10^{-16}$	3600.0
neos-520729	62356	91149	1	1	0	0	0	0	0	$2 \cdot 10^{-15}$	3600.0
neos-799711	118436	41998	1	1	0	0	964	975	$3 \cdot 10^{-9}$	$3 \cdot 10^{-16}$	1144.4
ns2017839	109020	55224	0	0	0	0	0	0	0	0	3600.0
ns2122603	49508	19300	1	1	0	0	0	99	$5 \cdot 10^{-16}$	$2 \cdot 10^{-15}$	3600.0
ofi	845174	420434	0	0	0	0	0	0	0	0	3600.0
sat.2-60	41832	35378	1	1	0	0	2	153	$4 \cdot 10^{-11}$	0	3600.0
sat.3-40-fs	71106	81681	1	1	0	0	2	253	$4 \cdot 10^{-11}$	0	3600.0
sat.3-40	89608	81681	1	1	0	0	2	253	$4 \cdot 10^{-11}$	0	3600.0
splan1	1145600	1317382	0	0	0	0	0	0	0	0	3600.0
transport.	19232	9685	1	1	0	0	2158	2192	$1 \cdot 10^{-7}$	$8 \cdot 10^{-13}$	3600.0

Table 4.3: Realistic instances in analysis mode

	Original				Infeas.		Violations				
name	conss	vars	cand	feas	int	LP	high	tot	Δmax	Δobj	time
w100R8T	1800	809	1	1	0	0	1	1	$1 \cdot 10^{-12}$	0	1215.2
w400R25T	32800	16041	1	1	0	0	5	13	$2 \cdot 10^{-13}$	0	3600.0
w529R40T	43378	21201	1	1	0	0	6	13	$6 \cdot 10^{-14}$	0	3600.0
w625R25T	32500	15651	1	1	0	0	1	12	$3 \cdot 10^{-14}$	0	3600.0
w900R36T	66600	32437	1	1	0	0	11	18	$8 \cdot 10^{-13}$	0	3600.0
rra01	2940	897	1	1	0	0	6	7	$1 \cdot 10^{-9}$	$2 \cdot 10^{-8}$	700.0
rra02	2940	897	1	1	0	0	2	6	$5 \cdot 10^{-9}$	$3 \cdot 10^{-7}$	3600.0
rra03	2940	897	1	1	0	0	1	4	$1 \cdot 10^{-13}$	$2 \cdot 10^{-14}$	2243.9
rra04	2940	897	1	1	0	0	2	6	$8 \cdot 10^{-10}$	$1 \cdot 10^{-7}$	3600.0
rra05	2940	897	1	1	0	0	7	10	$6 \cdot 10^{-9}$	$3 \cdot 10^{-5}$	628.8
rra07	5656	1793	1	1	0	0	0	5	$7 \cdot 10^{-18}$	$2 \cdot 10^{-14}$	3600.0
rra08	5656	1793	1	1	0	0	4	9	$8 \cdot 10^{-9}$	$8 \cdot 10^{-6}$	3600.0
rra09	5656	1793	1	1	0	0	6	13	$7 \cdot 10^{-9}$	$1 \cdot 10^{-6}$	3600.0
rra10	5656	1793	1	1	0	0	0	7	$1 \cdot 10^{-16}$	$2 \cdot 10^{-14}$	3600.0
ODTM	11088	3585	1	1	0	0	2	9	$1 \cdot 10^{-8}$	$5 \cdot 10^{-9}$	2124.4

4.3 Enforce mode

In the second experiment we ran SCIP in enforce mode for the same three sets of instances. Our purpose was to verify whether it is possible to improve the results obtained by floating-point SCIP, and to evaluate performances of xprim constraint handler. In this experiment the time limit is set to twelve hours, since we considered a factor of 12 between the time limit in analysis mode and in enforce mode as appropriate. No tolerances are used, and SoPlex is exploited as exact LP solver.

4.3.1 Tables

Table 4.4 shows the results for MIPLIB benchmark set of instances, Table 4.5 shows the results for MIPLIB unstable set of instances, Table 4.6 shows the results for the set of realistic instances.

Tables are quite similar to analysis mode ones. The first column gives the name of the instance. The number of constraints and variables associated to each instance can be seen in the corresponding row of the analysis mode tables. The number of candidate solutions, displayed in the column named *cand*, gives a measure of how many solutions have been checked by xprim constraint handler within the tree. Columns *feas*, *int* and *LP* represent the results of such checks, as already explained in section 4.2.1. Columns *high*, *tot* and Δmax are the statistics about relative violations, while column Δobj gives the difference between the objective values of floating-point and exact solutions; the latter four values are based only on the analysis of the best solution found.

The tenth column, named *xprim*, displays the execution time spent running xprim constraint handler, therefore providing a measure of the performances. The last column displays the total execution time, and can be compared to the correspondent column of the analysis mode table to have further performances indications. Since in this case the time limit is set to twelve hours, instances displaying $43200s$ have not completely been solved.

4.3.2 Results

Tables 4.4, 4.5 and 4.6 show the computational results obtained in enforce mode. For each table we are going to analyze solutions' accuracy first, and subsequently also performances.

A consideration has to be pointed out before the detailed discussion of results. A comparison can be performed between analysis mode and enforce mode results, but remembering that the SCIP execution is completely different in the two cases. Indeed, we can not simply think that in enforce mode every candidate solution also found in analysis mode is checked. Since xprim constraint handler returns a feedback to SCIP, the tree exploration the solver varies, therefore different nodes are solved and different decisions are taken.

In table 4.4, 77 instances of the MIPLIB benchmark set are shown. The number of candidate solutions checked by xprim constraint handler is variable in a range from 1 to some decades. The only exception is instance *ns1208400*, which almost produced a solution per second, but each of them were marked as infeasible because of violations in the integer constraints, as also happened in analysis mode. Also *rocII-4-11* still shows the same issues discussed in section 4.2.2, i.e. inaccuracies in the exact LP, but in this experiment also feasible solutions are found. All the other instances always produced exactly feasible candidate solutions, confirming that floating-point SCIP often produces very good results. This observation is strengthened by the fact that no tolerances are used.

It is remarkable that 61 out of 77 instances show neither a violation nor a difference in the objective value for their best solution. Moreover, the magnitude of such indicators is much less compared to what obtained in analysis mode. Maximum violations in the last solutions go from 10^{-12} to 10^{-16}, while maximum difference between floating-point and exact objective values are included between 10^{-15} and 10^{-16}. These results reveal that feedback returned by xprim constraint handler to SCIP significantly improved candidate solutions provided.

A comparison between the total execution times obtained in analysis mode and enforce mode determines a variable behavior on different instances, since sometimes we get better results with one mode and other times with the other one. Apart from a random component that can slightly affect times, differences can be due to several factors. First, as already stated, the tree is explored differently, therefore the best solution can be reached by following a different path. Second, on the one side xprim constraint handler can be time consuming, on the other side may suggest very good solutions which floating-point SCIP was not able to find.

Results on MIPLIB benchmark set show that in many cases the time spent by xprim constraint handler is negligible. For a few instances, instead, is extremely relevant, resulting in a peak of about 90% of the run time for instances *map18* and *map20*. In addition, we should also consider never returned instances, which may have very large execution times due to the exact check.

Table 4.5 show results for MIPLIB unstable instances in enforce mode. In this case the phenomenon of never returned instances is much relevant, since for only a few instances statistics are available. Apart from that, all the observations previously done are confirmed. Floating-point SCIP solutions are enough accurate, since all of the candidate solutions result to be feasible. The comparison with the execution times of analysis mode gives the same indications, too. As example, we can consider the instance *harp2*, which is solved more than 4 times faster in enforce mode, and the instance *neos-799711*, which is solved about 5 times slower, in enforce mode. Another interesting instance is *npmv07*, which has very high violations and very large exact checking time (about 96%). Note that such instance does not appear in table 4.2, that proves the theory of time consuming exact checking for instances having no results.

Table 4.6 confirms what already discussed, too. There are no infeasible candidate solutions, violations and differences between objective values are generally smaller than in table 4.3, and the time spent for the exact check is very variable.

It is worth to observe instance *ODTM*. Although it spent about 70% of its run time checking candidate solutions, its total execution time is more than 10 times lower than in the analysis mode experiment. Moreover, by calling $ODTM_a$ the result of such instance in analysis mode, and $ODTM_e$ the result in enforce mode, if we only consider the floating-point SCIP time, we obtain

$$t_{SCIP}(ODTM_e) = t_{tot}(ODTM_e) - t_{xprim}(ODTM_e) = 65.8s \tag{4.3}$$

and, therefore,

$$\frac{t_{SCIP}(ODTM_a)}{t_{SCIP}(ODTM_e)} = \frac{2124.4s}{65.8s} = 32.29. \tag{4.4}$$

This result suggests that if we would be able to improve performances of xprim constraint handler and of the exact LP solver, for some instances we may obtain the best solution more than 30 times faster.

Table 4.4: MIPLIB 2010 benchmark set of instances in enforce mode

			Infeas.		Violations				Time	
name	cand	feas	int	LP	high	tot	Δmax	Δobj	xprim	tot
30n20b8	1	1	0	0	0	0	0	0	0.0	180.2
acc-tight5	1	1	0	0	0	0	0	0	0.0	84.1
aflow40b	10	10	0	0	0	0	0	0	0.5	1550.6
air04	4	4	0	0	0	0	0	0	0.1	96.8
app1-2	2	2	0	0	0	0	0	0	1242.2	4299.9
ash608gpia.	0	0	0	0	0	0	0	0	0.0	31.8
biella1	14	14	0	0	0	0	0	0	0.8	675.3
bienst2	7	7	0	0	14	64	$5 \cdot 10^{-15}$	0	0.3	661.3
binkar10_1	11	11	0	0	0	0	0	$7 \cdot 10^{-16}$	0.6	263.3
bley_xl1	3	3	0	0	0	0	0	0	0.8	404.1
bnatt350	1	1	0	0	0	0	0	0	0.0	526.6
core2536-691	8	8	0	0	0	0	0	0	1.3	1076.6
cov1075	7	7	0	0	0	0	0	0	0.1	8329.4
csched010	20	20	0	0	0	0	0	0	0.1	11453.4
danoint	5	5	0	0	22	48	$1 \cdot 10^{-14}$	0	0.3	8544.7
dfn-gwin-UUM	19	19	0	0	0	0	0	0	0.2	167.7
eil33-2	14	14	0	0	0	0	0	0	0.2	149.7
eilB101	9	9	0	0	0	0	0	0	0.1	1058.5
enlight13	1	1	0	0	0	0	0	0	0.0	126.8
enlight14	0	0	0	0	0	0	0	0	0.0	742.4
ex9	1	1	0	0	0	0	0	0	0.2	91.8
gmu-35-40	18	18	0	0	5	5	$2 \cdot 10^{-13}$	0	0.1	43200.0
iis-100-0-cov	19	19	0	0	0	0	0	0	0.2	988.6
iis-bupa-cov	13	13	0	0	0	0	0	0	0.2	4509.1
iis-pima-cov	16	16	0	0	0	0	0	0	0.4	517.3
lectsched-4.	16	16	0	0	0	0	0	0	0.8	1501.0
m100n500k4r1	7	7	0	0	0	0	0	0	0.0	30362.5
macrophage	42	42	0	0	0	0	0	0	0.3	43200.0
map18	3	3	0	0	0	16	$1 \cdot 10^{-16}$	0	16067.7	18377.8
map20	5	5	0	0	0	16	$8 \cdot 10^{-17}$	0	16137.1	17889.0
mcsched	14	14	0	0	0	0	0	0	0.0	612.4
mik-250-1.	4	4	0	0	0	0	0	0	0.0	836.0
mine-166-5	72	72	0	0	0	0	0	0	0.7	72.8
mine-90-10	45	45	0	0	0	0	0	0	0.3	2132.0
msc98-ip	1	1	0	0	0	0	0	0	1.6	43200.0
mspp16	1	1	0	0	0	0	0	0	0.0	43200.0
mzzv11	23	23	0	0	0	0	0	0	0.9	810.0
n4-3	15	15	0	0	0	0	0	0	1.8	1482.9

continued on next page

continued from previous page

			Infeas.		Violations				Time	
name	cand	feas	int	LP	high	tot	Δmax	Δobj	xprim	tot
neos-1109824	6	6	0	0	0	0	0	0	0.2	210.6
neos-1337307	9	9	0	0	0	0	0	0	0.3	21083.6
neos-1396125	14	14	0	0	0	2	$1 \cdot 10^{-16}$	$2 \cdot 10^{-16}$	1.4	1021.3
neos13	57	57	0	0	0	0	0	0	1090.1	1916.5
neos-1601936	8	8	0	0	0	0	0	0	0.5	10356.2
neos18	7	7	0	0	0	0	0	0	0.1	101.3
neos-476283	5	5	0	0	4	228	$1 \cdot 10^{-14}$	$2 \cdot 10^{-15}$	1744.8	2147.2
neos-686190	8	8	0	0	0	0	0	0	0.1	105.4
neos-849702	1	1	0	0	0	0	0	0	0.0	187.3
neos-916792	9	9	0	0	85	263	$9 \cdot 10^{-15}$	$4 \cdot 10^{-16}$	45.6	43200.0
net12	2	2	0	0	0	0	0	0	9.2	2637.3
netdiversion	6	6	0	0	0	0	0	0	2.1	13270.9
newdano	23	23	0	0	12	27	$9 \cdot 10^{-15}$	0	0.8	12193.6
noswot	12	12	0	0	1	2	$1 \cdot 10^{-15}$	$1 \cdot 10^{-15}$	0.0	551.7
ns1208400	41978	0	41978	0	0	0	0	-	364.0	43200.0
ns1688347	4	4	0	0	0	0	0	0	0.1	516.3
ns1766074	0	0	0	0	0	0	0	0	0.0	2019.4
ns1830653	9	9	0	0	0	0	0	0	0.4	420.8
opm2-z7-s2	36	36	0	0	0	0	0	0	1.2	1629.6
pg5_34	13	13	0	0	40	51	$1 \cdot 10^{-14}$	$1 \cdot 10^{-16}$	0.5	2311.6
pigeon-10	2	2	0	0	0	0	0	0	0.0	43200.0
pw-myciel4	5	5	0	0	0	0	0	0	0.1	9955.2
qiu	16	16	0	0	0	2	$1 \cdot 10^{-17}$	$4 \cdot 10^{-15}$	0.7	126.7
rail507	6	6	0	0	0	0	0	0	4.1	4290.4
ran16x16	8	8	0	0	0	0	0	0	0.1	511.9
reblock67	51	51	0	0	0	0	0	0	0.1	455.8
rmatr100-p10	5	5	0	0	0	0	0	0	8.9	267.3
rmatr100-p5	4	4	0	0	0	0	0	0	13.9	452.5
rmine6	13	13	0	0	0	0	0	0	0.1	1941.4
rocII-4-11	31	7	0	24	0	45	$8 \cdot 10^{-17}$	$3 \cdot 10^{-16}$	320.7	5772.7
rococoC10.	17	17	0	0	0	0	0	0	0.1	7171.3
roll3000	31	31	0	0	0	0	0	0	0.4	1291.1
sp98ir	17	17	0	0	0	0	0	0	0.2	136.8
tanglegram1	6	6	0	0	0	0	0	0	0.6	4348.6
tanglegram2	7	7	0	0	0	0	0	0	0.1	14.9
timtab1	34	34	0	0	0	0	0	0	0.1	5931.8
triptim1	3	3	0	0	1	1	$2 \cdot 10^{-15}$	0	0.5	9412.7
unitcal_7	12	12	0	0	69	118	$8 \cdot 10^{-12}$	$3 \cdot 10^{-15}$	703.8	4165.4
zib54-UUE	21	21	0	0	0	0	0	0	3.2	10419.6

Table 4.5: MIPLIB 2010 unstable set of instances in enforce mode

			Infeas.		Violations				Time	
name	cand	feas	int	LP	high	tot	Δmax	Δobj	xprim	tot
ger50_17_trans	2	2	0	0	0	0	0	0	1.9	43200.0
harp2	17	17	0	0	0	0	0	0	0.2	865.4
neos-1112787	32	32	0	0	0	2	$6 \cdot 10^{-16}$	0	1.3	43200.0
neos-1140050	0	0	0	0	0	0	0	0	0.0	43200.0
neos-1225589	22	22	0	0	0	0	0	$2 \cdot 10^{-16}$	0.5	43200.0
neos-799711	7	7	0	0	1007	1027	$2 \cdot 10^{-9}$	0	1483.3	5835.4
npmv07	4	4	0	0	11188	17665	$2 \cdot 10^{0}$	0	14913.5	15492.0
ns2017839	3	3	0	0	1361	2095	$2 \cdot 10^{-10}$	$2 \cdot 10^{-16}$	3300.2	3872.3
splan1	0	0	0	0	0	0	0	0	0.0	43200.0

Table 4.6: Realistic instances in enforce mode

			Infeas.		Violations				Time	
name	cand	feas	int	LP	high	tot	Δmax	Δobj	xprim	tot
w100R8T	5	5	0	0	0	4	$2 \cdot 10^{-17}$	0	0.3	332.1
w400R25T	16	16	0	0	1	22	$1 \cdot 10^{-15}$	0	424.7	43200.0
w900R36T	8	8	0	0	1	28	$1 \cdot 10^{-15}$	0	499.4	43200.0
rra01	14	14	0	0	0	72	$1 \cdot 10^{-17}$	$1 \cdot 10^{-16}$	1.7	731.9
rra04	7	7	0	0	0	77	$1 \cdot 10^{-19}$	$1 \cdot 10^{-16}$	1.3	5501.7
rra05	8	8	0	0	1	41	$2 \cdot 10^{-8}$	$8 \cdot 10^{-5}$	1.4	2529.1
rra08	12	12	0	0	0	76	$3 \cdot 10^{-19}$	0	12.4	6598.4
rra09	4	4	0	0	0	83	$2 \cdot 10^{-19}$	$3 \cdot 10^{-16}$	11.8	12162.5
rra10	9	9	0	0	0	77	$9 \cdot 10^{-20}$	$5 \cdot 10^{-16}$	13.5	43200.0
ODTM	25	25	0	0	2	7	$2 \cdot 10^{-9}$	$3 \cdot 10^{-10}$	145.2	211.0

4.4 Conclusions

In the current dissertation we analyzed the accuracy of SCIP, a floating-point MIP solver, and we investigated a way to improve the quality of the solutions obtained. Many advances have been recently achieved in researching new techniques to develop an exact MIP solver. A few exact LP solvers having acceptable performances were already developed. However, no exact MIP solvers are still available and exploitable for a wide range of instances, although many approaches have been proposed and only need to be refined.

The results we obtained with SCIP confirm that its behavior is satisfying for many instances, therefore the solutions are quite reliable. Moreover, exploitation of floating-point precision is confirmed to provide a good balance between performances and accuracy. However, numerical errors of variable entity are present, and it is well-known that such inaccuracies lead to falsely feasible solutions in some cases.

In summary, the approach we adopted starts from previous knowledge on exact MIP solving, and introduces a new perspective in tackling such problem. Indeed, we tried to plug into SCIP a constraint handler in order to achieve the goal of exactly solving MIPs by deeply exploiting SCIP floating-point computation.

On the one side, xprim constraint handler provided us interesting results, on the other side, its behavior may be improved according to several ways. First, an exact parsing of the problem would be necessary in order to discard possible inaccuracies due to the translation from floating-point to rational data. Second, a slightly different approach can be implemented: we might look for the exact LP solution of a problem that tries to minimize the distance between exact and floating-point solutions instead of simply solving a feasibility problem. Finally, it would be advisable to test xprim constraint handler with more numerically difficult instances and verify whether it can find feasible solutions that can replace falsely feasible ones.

List of Figures

List of Tables

Bibliography

[1] Tobias Achterberg. “Constraint Integer Programming.” Ph.D thesis. Technischen Universität Berlin, 2007.

[2] Tobias Achterberg. “SCIP: Solving constraint integer programs.” In: *Mathematical Programming Computation* 1.1 (2009). `http://mpc.zib.de/index.php/MPC/article/view/4`, pp. 1–41.

[3] David L Applegate et al. “Exact solutions to linear programming problems.” In: *Operations Research Letters* 35.6 (2007), pp. 693–699.

[4] David H Bailey. “High-precision floating-point arithmetic in scientific computation.” In: *Computing in science & engineering* 7.3 (2005), pp. 54–61.

[5] Timo Berthold and Kati Wolter. “Branching, Presolving, and Constraint Handlers.” 2009.

[6] Michael Blair, Sally Obenski, and Paula Bridickas. *Patriot missile defense: Software problem led to system failure at dhahran.* Tech. rep. Saudi Arabia. Technical report, US Government Accountability Office (GAO), 1992.

[7] William Cook et al. “A hybrid branch-and-bound approach for exact rational mixed-integer programming.” In: *Mathematical Programming Computation* 5.3 (2013), pp. 305–344.

[8] Fabio D’Andreagiovanni. “Pure 0-1 programming approaches to wireless network design.” In: *4OR: A Quarterly Journal of Operations Research* 10.2 (2012), pp. 211–212.

[9] Fabio D’Andreagiovanni and Ambros M. Gleixner. “Towards an accurate solution of wireless network design problems.” 2013.

[10] Fabio D’Andreagiovanni, Carlo Mannino, and Antonio Sassano. “GUB covers and power-indexed formulations for wireless network design.” In: *Management Science* 59.1 (2013), pp. 142–156.

[11] George B Dantzig. “Maximization of a Linear Function of Variables Subject to Linear Inequalities.” In: *Activity Analysis of Production and Allocation* (1951). Ed. by Wiley, pp. 339–347.

[12] Marcel Dhiflaoui et al. “Certifying and repairing solutions to large lps how good are lp-solvers?” In: *Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics. 2003, pp. 255–256.

[13] Mark Dowson. “The Ariane 5 software failure.” In: *ACM SIGSOFT Software Engineering Notes* 22.2 (1997), p. 84.

[14] *Doxygen*. URL: `http://www.stack.nl/~dimitri/doxygen/`.

[15] Daniel G Espinoza. “On linear programming, integer programming and cutting planes.” Ph.D thesis. Georgia Institute of Technology, 2006.

[16] GMP. *The GNU Multiple Precision Arithmetic Library*. URL: `https://gmplib.org/`.

[17] *Git*. URL: `http://git-scm.com/`.

[18] Ambros M Gleixner, Daniel E Steffy, and Kati Wolter. “Improving the accuracy of linear programming solvers with iterative refinement.” In: *Proceedings of the 37th International Symposium on Symbolic and Algebraic Computation*. ACM. 2012, pp. 187–194.

[19] David Goldberg. “What every computer scientist should know about floating-point arithmetic.” In: *ACM Computing Surveys (CSUR)* 23.1 (1991), pp. 5–48.

[20] Stefan Heinz. “Basic concepts of SCIP.” 2009.

[21] Jeff Kennington, Eli Olinick, and Dinesh Rajan. *Wireless Network Design: Optimization Models and Solution Procedures*. Springer, 2010.

[22] Leonid G Khachiyan. “A polynomial algorithm in linear programming.” In: *Soviet Mathematics Doklady, 20* (1979), pp. 191–194.

[23] Thorsten Koch. “The final NETLIB-LP results.” In: *Operations Research Letters* 32.2 (2004), pp. 138–142.

[24] Thorsten Koch et al. “MIPLIB 2010.” In: *Mathematical Programming Computation* 3.2 (2011), pp. 103–163.

[25] *MIPLIB 2010 Benchmark set*. URL: `http://miplib.zib.de/miplib2010-benchmark.php`.

[26] *MIPLIB 2010 Unstable set.* URL: `http://miplib.zib.de/miplib2010-unstable.php`.

[27] Bashar Nuseibeh. "Ariane 5: who dunnit?" In: *IEEE Software* 14.3 (1997), pp. 15–16.

[28] Sanam Sadr, Alagan Anpalagan, and Kaamran Raahemifar. "Radio resource allocation algorithms for the downlink of multiuser OFDM communication systems." In: *Communications Surveys & Tutorials, IEEE* 11.3 (2009), pp. 92–106.

[29] Jonathan Richard Shewchuk. "Adaptive precision floating-point arithmetic and fast robust geometric predicates." In: *Discrete & Computational Geometry* 18.3 (1997), pp. 305–363.

[30] Daniel E Steffy. "Topics in exact precision mathematical programming." Ph.D thesis. Georgia Institute of Technology, 2011.

[31] *Valgrind.* URL: `http://valgrind.org/`.

[32] Daniele Vigo. "Resource Optimization - Introduction to ILP." 2014.

[33] Wikipedia. *Double-precision floating point format.* URL: `http://en.wikipedia.org/wiki/Double-precision_floating-point_format`.

[34] Wikipedia. *Integer programming.* URL: `http://en.wikipedia.org/wiki/Integer_programming`.

[35] Wikipedia. *Linear programming.* URL: `http://en.wikipedia.org/wiki/Linear_programming`.

[36] Laurence A. Wolsey. *Integer Programming.* Wiley, 1998.

[37] ZIB. *SCIP: Solving Constraint Integer Programs.* URL: `http://scip.zib.de`.

[38] *ZIB.* URL: `http://www.zib.de`.