Konrad-Zuse-Zentrum
für Informationstechnik Berlin

Takustraße 7
D-14195 Berlin-Dahlem
Germany

TOBIAS ACHTERBERG
MARTIN GRÖTSCHEL
THORSTEN KOCH

Software for Teaching Modeling of Integer Programming Problems

ZIB-Report 06-23 (May 2006)

Software for Teaching Modeling of Integer Programming Problems*

Tobias Achterberg† Martin Grötschel† Thorsten Koch†

June 1, 2006

Abstract

Modern applications of mathematical programming must take into account a multitude of technical details, business demands, and legal requirements. Teaching the mathematical modeling of such issues and their interrelations requires real-world examples that are well beyond the toy sizes that can be tackled with the student editions of most commercial software packages.

We present a new tool, which is freely available for academic use including complete source code. It consists of an algebraic modeling language and a linear mixed integer programming solver. The performance and features of the tool are in the range of current state-of-the-art commercial tools, though not in all aspects as good as the best ones. Our tool does allow the execution and analysis of large real-world instances in the classroom and can therefore enhance the teaching of problem solving issues.

Teaching experience has been gathered and practical usability was tested in classes at several universities and a two week intensive block course at TU Berlin. The feedback from students and teachers has been very positive.

MSC 97U70 97-04 90-04 90C11 68N99

1 Introduction

In almost all courses where problem solving is addressed an implicit assumption is made that every problem investigated is "given". Most mathematical theorems are of the form *"If the following is given, then something else holds"*. The execution of algorithms also requires that data are given. In the "real world" it is not so clear what really is given. In fact, one is usually aware that there is some problem but it is often not completely clear what the problem exactly is. Such issues are rarely taught, and students do not necessarily learn how to handle situations of this type.

A way to teach real-world problem solving is to provide software that helps students (and practitioners as well) to model the problem at hand fast and make codes available that solve the mathematical models quickly. They may realize from the initial shot at the problem that the model they came up with does not deliver proper solutions. They find out that certain constraints have been forgotten, others may have been incorrectly stated, etc. Problem solving requires the execution of three basic steps: modeling, solver run, solution analysis. And these

*Work supported by the DFG Research Center MATHEON in Berlin

†Zuse Institute Berlin, Takustr. 7, 14195 Berlin, Germany, eMail: koch@zib.de

three steps have to be iterated (often many times) until all important side constraints are adequately modeled, some irrelevant side conditions are neglected, and the solution appears satisfactory. In complex cases, the solution software may have to be modified as well so that that production runs terminate in an acceptable time frame. How existing software can be adapted to run faster on particular models is another topic that needs to be taught.

Since *large scale* is a common attribute of real-world operations research, one difficulty from the teaching perspective is that modeling pitfalls and puzzling observations (e. g., running time abnormalities, strange model behaviors) can rarely be made when toy models are employed. In fact, the issues raised above are abound in practice, and do not only arise in teaching. Modern applications of mathematical programming must take into account a multitude of technical details, business demands, and legal requirements. In order to show how to design algorithms that solve practical problems, suitable data sets and application specific software from industry projects are needed. Otherwise the students will not be adequately prepared for practical work and do not acquire a feeling for the instance sizes that can be successfully tackled.

That is why there is a basic need to have both, large-scale real-world instances and flexible software that can handle such instances, available in the classroom and on the students' laptops. In this paper we describe a freely available software tool that closes this gap for the area of mixed integer programming, and we indicate where large-scale data sets can be found (at present and in the future). Of course there exists a large set of commercial products which are suitable for the task, see, for example, [7, 12] and especially [8] for surveys. But using commercial systems for teaching has several drawbacks:

- None of the systems is free to use and the price for a class license may not be affordable for many institutions, in particular outside Europe and North America. Often free or at least cheap student editions are available, but they are usually restricted to only very small problem sizes.
- Using a commercial tool makes it difficult for the students to use the software at home, not only because of the price, but also because they are usually only available on specific platforms. This conflicts with the current trend away from using computer labs towards the use of individual laptops.
- All of the systems come as black boxes. There is no possibility for the students to look behind the scenes and learn how the tools are working.
- Since no source code is available it is not possible to change or extend the tools either as an exercise or to facilitate teaching of techniques to deal with problems of special structure.
- Finally, using black box software is not helpful in teaching students scientific methodology. Just believing the result of some unknown software without even the possibility to check what actually happened will not qualify for a scientific experiment.

Over the years high performance codes in the area of linear programming (LP) and mixed integer programming (MIP) have been developed by several researchers at the Zuse Institute Berlin (ZIB). To improve the usability of this software, we have spent some effort to combine the codes. Now we provide a complete tool consisting of the algebraic modeling language

ZIMPL[1], the linear programming solver SOPLEX[2], and the mixed integer programming framework SCIP[3].

This combination of software allows to easily build and solve linear mixed integer programming models. Models written in the ZIMPL language can be directly read by the MIP-solver SCIP, which automatically calls SOPLEX as a subroutine to solve the resulting LP subproblems. The software is highly portable, free for academic use, and the complete source code as well as ready-to-run binaries for several platforms are available for download.

As recent benchmarks show[4] the performance is at least on par with the best other freely available tools and, depending on the instance, even comparable to the top commercial tools. Two other initiatives that provide free tools with source code should be mentioned: The GNU Linear Programming Toolkit GLPK[5] and the Computational Infrastructure for Operations Research COIN-OR[6]. For an overview, see also [11] and [12].

In October 2005, a two week block course consisting of over 80 hours of lectures and exercises was held at the Zuse Institute Berlin with more than 100 participants from 10 countries. One aim of the course was to test the ideas and concepts of the the MATHEON[7] project *Combinatorial Optimization at Work*[8], which starts out to provide a *digital textbook* describing case studies from practice, commenting on difficulties encountered, discussing the mathematical models and theories, and explaining the algorithmic approaches successfully employed in practice. When finished, the textbook together with data sets, programs, pictures, and visualizations will be made available on the Internet. The software we describe in this article was used on most of the exercises in the course.

In the next three sections we will shortly present the programs. In Section 5 a small example on what students can experience when working with our tool will be given.

2 The modeling language: Zimpl

Algebraic modeling languages allow to describe a mathematical model in terms of sets depending on parameters. This description is translated automatically into a mixed integer program, which can be fed into any out-of-the-box MIP-solver. ZIMPL [9] is a newly developed algebraic modeling language, which is similar in concept to well-known languages such as GAMS [4] or AMPL [6]. While ZIMPL implements only (the most important) 20 percent of the functionality of AMPL, this proved to be sufficient for many real-world projects [9].

What ZIMPL distinguishes from other modeling languages is the use of rational arithmetic. With a few exceptions, all computations in ZIMPL are done with infinite precision rational arithmetic. This ensures that no rounding errors can occur. One might think that the use of rational arithmetic results in a huge increase of computation time and memory. But experience shows that this seems not to be relevant with current hardware. ZIMPL has been successfully used to generate integer programs with more than 30 million non zero coefficients.

An introduction into modeling with ZIMPL together with a complete description of the

[1] http://www.zib.de/koch/zimpl
[2] http://www.zib.de/Optimization/Software/Soplex
[3] http://scip.zib.de
[4] http://plato.asu.edu/bench.html
[5] http://www.gnu.org/software/glpk
[6] http://www.coin-or.org
[7] http://www.matheon.de
[8] http://co-at-work.zib.de

language can be found in [9]. The reference discusses both theoretical and practical considerations of the implementation. Aspects of software engineering, error prevention, and detection are also addressed. ZIMPL is under active development and the standalone version is available from the web site at http://www.zib.de/koch/zimpl.

3 The mixed integer programming framework: SCIP

SCIP is a framework for *constraint integer programming* that aims at integrating constraint programming and mixed integer programming into a single solver [1]. It is designed to support the implementation of CP and MIP solver components like branching rules, primal heuristics, cutting plane separators, or domain propagators. It allows for a very flexible integration of user-defined non-linear constraints, and it is a fully-fledged branch-cut-and-price framework.

SCIP comes with a large number of components that suffice to turn the basic framework into a sophisticated mixed integer programming solver. It is able to directly read ZIMPL files. *Preprocessors* transform the problem instance into an equivalent instance that is easier to solve, *constraint handlers* provide specialized algorithms and data structures to handle specific constraint classes, *separators* add cutting planes to tighten the LP relaxation, *primal heuristics* search for feasible solutions, *branching rules* define the search tree, and *node selectors* define the way the search tree is processed. To solve the LP relaxations, SCIP calls an externally linked LP solver through a basic LP solver interface.

In a course on mixed integer programming, SCIP can be used to demonstrate and compare the usefulness of certain components, e. g., node selection strategies, branching rules, cut separation algorithms, or primal heuristics. In particular, the search tree can be visualized by using VBCTOOL [10] which helps to analyze the impact of the employed branching rule and node selection. An example is given in Figure 7, which shows two MIP-instances from the MIPLIB [3] solved with the same parameter settings, but producing vastly different branching trees.

With the source code at hand, a focus can be set on specific implementation issues, e. g., how one can quickly generate violated clique cuts, or what can be done to ensure *numerical stability* of Gomory mixed integer cuts. In a more advanced course, students can even implement their own components like a specialized primal heuristic for a given problem class such as the traveling salesman problem.

In a course that is primarily focused on modeling, like our workshop *Combinatorial Optimization at Work*, SCIP can be used in conjunction with ZIMPL and SOPLEX to evaluate ones own mixed integer programming models. SCIP as a stand-alone MIP solver is easy to use, provides lots of parameters to adjust the solver components, and reports a large number of statistics that show the impact of the different components on the given problem instance.

4 The linear programming solver: SoPlex

SOPLEX [14] is an implementation of the revised simplex algorithm for the solution of linear programs. It features primal and dual solving routines and is implemented as a C++ class library that can be used within other programs. SoPlex is under continuous development and the current version as of March 2006 is 1.3.0.

An example program to solve standalone linear programs given in MPS or LP format files is included. But other than to experiment with parameter settings or for programming exercises there is not much need to use SoPlex as a standalone LP-Solver if Scip is available. Its main use in this context is to act as a solver engine for LP-based mixed integer programming.

5 A puzzling puzzle model

In the following, we use the popular puzzle game named Sudoku [5] to give an impression of how to use Zimpl and Scip to model and solve a problem. The aim of the puzzle is to enter an integer from 1 through 9 in each cell of a 9×9 grid made up of 3×3 subgrids. At the beginning several cells are already given preset integers. At the end, each row, column and subgrid must contain each of the nine integers exactly once. Figure 1 shows an example. For details see, e. g., http://en.wikipedia.org/wiki/Sudoku.

		5						3
				4	6			
		7						2
	1				3		6	9
	4		6		9		5	
9	8		2				7	
2						9		
			8	1				
6						4		

4	6	5	7	2	8	1	9	3
1	2	9	3	4	6	7	8	5
8	3	7	1	9	5	6	4	2
5	1	2	4	7	3	8	6	9
7	4	3	6	8	9	2	5	1
9	8	6	2	5	1	3	7	4
2	7	1	5	6	4	9	3	8
3	9	4	8	1	7	5	2	6
6	5	8	9	3	2	4	1	7

Figure 1: Sudoku puzzle and solution

There are several possibilities to model this puzzle. A popular choice is to state the problem as a constraint program using a collection of *alldifferent* constraints [13]. But how can this be formulated as an integer program? Zimpl can automatically generate IPs for certain constructs such as the absolute value of the difference of two variables (`vabs`). Using 81 integer variables in the range $\{1, \ldots, 9\}$ the *alldifferent* constraint can be formulated by demanding that the absolute difference of all pairs of relevant variables is greater than or equal to one. This leads to the Zimpl program shown in Figure 2. Note that lines beginning with *set* define sets, lines with *param* define parameters, i. e., data, lines starting with *var* declare variables and lines starting with *subto* define constraints.

Figure 3 shows a Scip session solving the Sudoku instance shown in Figure 1, which is modeled by the Zimpl program of Figure 2. We stripped or slightly relocated parts of the output such that it fits to the size of the paper. As one can see in line 4, parameters can easily be changed with the "set" command. In this case, we impose a time limit of 600 seconds on the solution process. The Zimpl model is read in line 6, generating a MIP instance with 3969 variables and 4884 constraints. The large size of the instance arises from auxiliary variables and constraints produced by Zimpl's automatic modeling of the `vabs` function. The command "optimize" in line 8 lets Scip solve the instance. Between lines 9 and 29 one can see the solving progress. Presolving is conducted in consecutive rounds, in this case terminating after 66 rounds. The presolving modifications can be seen in lines 13 and 14. After presolving is finished, Scip tries to solve the remaining problem with 1424 variables and 2291 constraints, but the process is interrupted due to the time limit and stops without finding a feasible solution[9].

[9] We checked this result with Cplex 9.03, one of the top-of-the-line commercial MIP-solvers. After six hours

```
param p         := 3;
set J           := { 1 .. p*p };
set KK          := { 1 .. p } * { 1 .. p };
set F           := { read "fixed.dat" as "<1n,2n>" };
param fixed[F] := read "fixed.dat" as "<1n,2n>3n";
var x           [J * J] integer >= 1 <= 9;

subto rows:     forall <i,j,k> in J*J*J with j < k do
                   vabs(x[i,j]-x[i,k]) >= 1;
subto cols:     forall <i,j,k> in J*J*J with j < k do
                   vabs(x[j,i]-x[k,i]) >= 1;
subto squares: forall <m,n> in KK do
                   forall <i,j,k,l> in KK*KK with p*i+j < p*k+l do
                      vabs(x[(m-1)*p+i,(n-1)*p+j] - x[(m-1)*p+k,(n-1)*p+l]) >= 1;
subto fixed:    forall <i,j> in F do x[i,j] == fixed[i,j];
```

Figure 2: A Zimpl model to solve Sudoku using integer variables

The command "display statistics" at line 31 causes the output of a large table with statistical data on the different components and solving steps. One can see that most of the solving time is spent to solve LP relaxations.

Two reasons to solve the LP relaxation in a branch-and-bound algorithm are to compute a lower bound for the objective function and to detect infeasibility early. Since Sudoku is a pure feasibility problem it is not possible to compute a bound for the objective function. Also detecting infeasibility seems not to work very effective either, given the huge amount of unprocessed nodes left to compute. Therefore, an idea to improve the performance would be to find better parameter settings for the solver.

Figure 4 shows a solving run on the same instance as before, but with different parameter settings. This time we set Scip to perform more like a pure constraint programming solver. In line 4 we select depth-first-search node selection, in line 6 we enable conflict analysis [2] on propagation conflicts, and in line 8 we disable the solving of the LP relaxations. Since we no longer solve the LP relaxations, the branching nodes are processed much faster. The previous run of Figure 3 needed three minutes to process the first 10000 nodes, while this quantity is now processed in 7.4 seconds, as one can see in line 20 of Figure 3 and line 21 of Figure 4. Additionally, mainly due to conflict analysis, a solution is found in 8.2 seconds and 11534 nodes. After displaying the statistics at line 28, the command "display solution" is issued at line 59 to report the optimal solution. The objective value of the solution and all non-zero elements in the solution vector are displayed. In the listing, we only show the output for the first and last model variable. One can see that the Zimpl model generated variable names "x#i#j" with (i, j) being the position in the Sudoku grid. The solution states to use numeral 4 in cell $(1, 1)$ and numeral 7 in cell $(9, 9)$.

Choosing the right model is often more important (and more effective) than having the best solver implementation. Especially with real-world problems, having the ability to experiment swiftly with different formulations is essential. Conveying this idea to the students has been a major point in our workshop. Zimpl has proven to be a valuable tool in this regard by letting the students experiment which model works best. For practical work it is important to experience the fact that mathematical equivalent formulations might behave totally different

and a million nodes there was still no feasible solution available.

```
SCIP version 0.81f [precision: 8 byte] [mode: optimized] [LP solver: SOPLEX 1.3.0]
Copyright (c) 2002-2006 Konrad-Zuse-Zentrum fuer Informationstechnik Berlin (ZIB)

SCIP> set limits time 600
parameter <limits/time> set to 600
SCIP> read sudoku_int.zpl
original problem has 3969 variables (972 bin, 2997 int, 0 cont) and 4884 constraints
SCIP> optimize
presolving:
(round 1) 24 del vars, 996 del conss, 1014 chg bounds, 0 chg sides, 0 chg coeffs
(round 65) 2543 del vars, 2593 del conss, 3311 chg bounds, 20 chg sides, 40 chg coeffs
presolving (66 rounds):
 2543 deleted vars, 2593 deleted constraints, 3311 tightened bounds, 20 changed sides,
 40 changed coefficients, 23285 implications
presolved problem has 1424 variables (392 bin, 1032 int, 0 cont) and 2291 constraints

 time | node  | left  |LP iter|frac |rows |cuts | dualbound    | primalbound | gap
  2.9s|     1 |     0 |  1101 | 518 |2123 |   0 | 0.000000e+00 |     --      | Inf
 11.7s|     1 |     2 | 13077 | 764 |3036 | 913 | 0.000000e+00 |     --      | Inf
  176s| 10000 |  4013 |119005 | 413 |2644 | 913 | 0.000000e+00 |     --      | Inf
  272s| 20000 |  7523 |231183 | 588 |2644 | 913 | 0.000000e+00 |     --      | Inf
  352s| 30000 |  9779 |320146 | 498 |2644 | 913 | 0.000000e+00 |     --      | Inf
  426s| 40000 | 12171 |402351 | 327 |2644 | 913 | 0.000000e+00 |     --      | Inf
  513s| 50000 | 14549 |504125 | 141 |2644 | 913 | 0.000000e+00 |     --      | Inf
  596s| 60000 | 17171 |600208 |   - |2644 | 913 | 0.000000e+00 |     --      | Inf

SCIP Status        : solving was interrupted [time limit reached]
Solving Time (sec) : 600.00
Solving Nodes      : 60572

SCIP> display statistics

Original Problem   :
  Variables        : 3969 (972 binary, 2997 integer, 0 continuous)
  Constraints      : 4884 initial, 4884 maximal
Presolved Problem  :
  Variables        : 1424 (392 binary, 1032 integer, 0 continuous)
  Constraints      : 2291 initial, 2291 maximal
Presolvers         :   Time   Fixed    Aggr   ChgBds  DelCons
  trivial          :   0.01      38       0        0        0
  dualfix          :   0.00      35       0        0        0
  implics          :   0.06       0     402        0        0
  probing          :   1.77     444     114     1726        0
  varbound         :   0.00       0       0       83      989
  linear           :   0.85     215    1295     1502     1604
Constraints        :   Time  Number Cutoffs  DomReds     Cuts
  integral         : 171.60       0    1400     5675        0
  varbound         :  14.68    1723    2619   507022        0
  linear           :   9.23     568    3770  1221137        0
Separators         :   Time   Calls Cutoffs  DomReds     Cuts
  cut pool         :   0.00       6       -        -        3
  impliedbounds    :   0.03       7       0        0      950
  cmir             :   0.92       7       0        0        3
  clique           :   0.00       7       0        0        2
Branching Rules    :   Time   Calls Cutoffs  DomReds     Cuts
  relpscost        : 171.40   43452    1400     5675        0
LP                 :   Time   Calls   Iters  It/call   It/sec
  dual LP          : 248.57   57531  554595     9.64  2231.14
  diving/probing LP:  16.67    3126   50303    16.09  3017.58
  strong branching : 160.34   17903  202879    11.33  1265.30
B&B Tree           :
  nodes            :  60572
  max depth        :    107
  backtracks       :  10778 (17.8%)
```

number of unprocessed subproblems in search tree

number of generated implied bound cutting planes

reliable pseudo cost branching was applied

time spent in dual simplex to solve relaxations

maximal depth of search tree

Figure 3: A SCIP session solving a Sudoku instance modeled by Figure 2

```
SCIP version 0.81f [precision: 8 byte] [mode: optimized] [LP solver: SOPLEX 1.3.0]
Copyright (c) 2002-2006 Konrad-Zuse-Zentrum fuer Informationstechnik Berlin (ZIB)

SCIP> set nodeselection dfs stdpriority 1000000
parameter <nodeselection/dfs/stdpriority> set to 1000000
SCIP> set conflict useprop TRUE
parameter <conflict/useprop> set to TRUE
SCIP> set lp solvefreq -1
parameter <lp/solvefreq> set to -1
SCIP> read sudoku_int.zpl
original problem has 3969 variables (972 bin, 2997 int, 0 cont) and 4884 constraints
SCIP> optimize
presolving:
...
 time | node  | left  |vars |cons |ccons|confs| dualbound    | primalbound  | gap
  2.0s|     1 |     2 |1424 |2291 |2291 |    0 | 0.000000e+00 |      --      | Inf
  3.2s|  2000 |    51 |1424 |2484 | 965 |  288 | 0.000000e+00 |      --      | Inf
  4.4s|  4000 |    41 |1424 |2607 | 457 |  642 | 0.000000e+00 |      --      | Inf
  5.4s|  6000 |    50 |1424 |2549 | 210 |  959 | 0.000000e+00 |      --      | Inf
  6.4s|  8000 |    22 |1424 |2547 | 928 |1313 | 0.000000e+00 |      --      | Inf
  7.4s| 10000 |    11 |1424 |2569 | 942 |1663 | 0.000000e+00 |      --      | Inf
* 8.2s| 11534 |     0 |1424 |2598 | 157 |1955 | 0.000000e+00 | 0.000000e+00 | 0.00%

SCIP Status        : problem is solved [optimal solution found]
Solving Time (sec) : 8.17
Solving Nodes      : 11534

SCIP> display statistics

Original Problem   :
  Variables        : 3969 (972 binary, 2997 integer, 0 continuous)
  Constraints      : 4884 initial, 4884 maximal
Presolved Problem  :
  Variables        : 1424 (392 binary, 1032 integer, 0 continuous)
  Constraints      : 2291 initial, 2615 maximal
Presolvers         :   Time   Fixed     Aggr   ChgBds  DelCons
  trivial          :   0.02      38        0        0        0
  dualfix          :   0.00      35        0        0        0
  implics          :   0.01       0      402        0        0
  probing          :   1.28     444      114     1726        0
  varbound         :   0.00       0        0       83      989
  linear           :   0.64     215     1295     1502     1604
Constraints        :   Time  Number  Cutoffs  DomReds     Cuts
  varbound         :   1.57    1723      393   119198        0
  linear           :   1.64     568     1959   285006        0
  logicor          :   0.00      0+        0        8        0
  bounddisjunction :   0.17      0+       61    17496        0
Conflict Analysis  :   Time   Calls  Success   Confls     Lits
  propagation      :   0.01    1880     1781     2011      3.0
Branching Rules    :   Time   Calls  Cutoffs  DomReds     Cuts
  inference        :   1.50    9560        0        0        0
B&B Tree           :
  nodes            :  11534
  max depth        :    100
  backtracks       :   1634 (14.2%)
  delayed cutoffs  :   7567
  repropagations   :  11380 (45088 domain reductions, 298 cutoffs)

SCIP> display solution

objective value:                              0
x#1#1                                         4   (obj:0)
...
x#9#9                                         7   (obj:0)
```

number of domain reductions found by linear constraints

constraints added by conflict analysis

number of generated conflict constraints

used inference branching

Figure 4: Solving the Sudoku model of Figure 2 with CP/SAT techniques

in practice.

As an alternative we modeled the Sudoku problem using 729 binary variables as shown in Figure 5, instead of 81 integer variables. Figure 6 shows a SCIP session for this model on the same data set as before. In this case, presolving managed to fix all variables and to remove all constraints after six rounds. In fact after solving several thousand instances we never encountered a Sudoku instance that could not be solved in preprocessing or in the root node with this model. After presolving has been finished, the remaining trivial empty problem is solved.

During the workshop, a new problem area was usually presented each morning in a lecture. In the afternoon, we provided the students with data and a description of the problem setting and let them work out and solve the mathematical modeling on their own. In a course taking place at the University of Bayreuth, we set out a competition for the best performing formulation for the n-Queens problem. And even though the students had only few hours experience with modeling and solving integer programs, they were able to model the problem on their own and beat the formulation of the lecturer.

6 Conclusion

We tested the combination of SCIP, ZIMPL and SOPLEX as the main software environment for the exercises in our two week block course. The problems we posed to the students covered a wide range of applications, including computing the tour of a wielding robot, modeling Sudoku puzzles, solving Steiner tree problems, optimize pizza production, computing OSPF routing weights, modeling telecom network design problems, chip verification, bus routing, driver scheduling, line planning, and repairman tour scheduling problems.

The first exercise was to find the minimal number of students that have to change their seats such that everybody without a laptop has a neighbor with a laptop. For the real topology of the lecture hall, this turned out to be much more difficult than initially anticipated.

Having the students directly sit down and practically address problems by building a mathematical model and trying to solve it was a great experience. The students developed practical skills and advanced their knowledge both in theory and practice. We received enthusiastic responses within the evaluation after the course.

For the course the students had to bring their own laptops. This was very successful for two reasons: first, everybody could work in a familiar environment since the software presented is available for Linux, Windows and Mac-OS alike. And second, after the course the students took home not only their newly acquired knowledge but also their working environment, keeping the ability to actually model and solve problems.

All available files from the course including most of the slides and exercises can be found at http://co-at-work.zib.de/download/CD. Further material on using ZIMPL and SCIP in a course was prepared by Jörg Rambau at the University of Bayreuth and is available at http://www.uni-bayreuth.de/departments/wirtschaftsmathematik/rambau/Teaching/Uni_Bayreuth/WS_2004/Diskrete_Optimierung_Anwendungen.

Today, companies are forced to rethink their planning methods due to the high innovation pressure. Knowledge in the practical application of mathematical modeling is becoming a key skill in industry, since the solution of mixed integer programs is one of the very few areas which can provide globally optimal answers to discrete (yes or no, choose an option) questions. To teach this knowledge state-of-the-art software as described in this article is required.

References

[1] Tobias Achterberg. SCIP - a framework to integrate constraint and mixed integer programming. Technical Report 04-19, Zuse Institute Berlin, 2004.

[2] Tobias Achterberg. Conflict analysis in mixed integer programming. Technical Report 05-19, Zuse Institute Berlin, 2005.

[3] Tobias Achterberg, Thorsten Koch, and Alexander Martin. MIPLIB 2003. *Operations Research Letters*, 34(4):1–12, 2006.

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

[5] David Eppstein. Nonrepetitive paths and cycles in graphs with application to Sudoku. ACM Computing Research Repository, 2005.

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

[7] Robert Fourer. Software survey: Linear programming. *MS/OR Today*, 32, jun 2005.

[8] Josef Kallrath, editor. *Modeling Languages in Mathematical Optimization*. Kluwer, 2004.

[9] Thorsten Koch. *Rapid Mathematical Programming*. PhD thesis, Technische Universität Berlin, 2004.

[10] Sebastian Leipert. The tree interface – version 1.0 user manual. Technical Report 96.242, Institut für Informatik, Universität zu Köln, 1996.

[11] J. Linderoth and T. Ralphs. Noncommercial software for mixed-integer linear programming. *Optimization Online*, 2005. http://www.optimization-online.org/DB_FILE/2004/12/1028.pdf.

[12] H. D. Mittelmann and P. Spellucci. Decision tree for optimization software, 2006. See http://plato.asu.edu/guide.html.

[13] W.J. van Hoeve. The alldifferent constraint: A survey. In *6th Annual Workshop of the ERCIM Working Group on Constraints*. Prague, June 2001.

[14] R. Wunderling. Paralleler und objektorientierter Simplex. Technical Report TR 96-09, Konrad-Zuse-Zentrum Berlin, 1996.

```
param p  := 3;
set J    := { 1 .. p*p };
set KK   := { 1 .. p } * { 1 .. p };
set F    := { read "fixed.dat" as "<1n,2n,3n>" };
var x    [J*J*J] binary;

subto nums:     forall <i,j> in J*J do sum <k> in J : x[i,j,k] == 1;
subto cols:     forall <j,k> in J*J do sum <i> in J : x[i,j,k] == 1;
subto rows:     forall <i,k> in J*J do sum <j> in J : x[i,j,k] == 1;
subto fixed:    forall <i,j,k> in F do                x[i,j,k] == 1;
subto squares:  forall <m,n,k> in KK*J do
                   sum <i,j> in KK : x[(m-1)*p+i,(n-1)*p+j,k] == 1;
```

Figure 5: A ZIMPL model to solve Sudoku using binary variables

```
SCIP version 0.81f [precision: 8 byte] [mode: optimized] [LP solver: SOPLEX 1.3.0]
Copyright (c) 2002-2006 Konrad-Zuse-Zentrum fuer Informationstechnik Berlin (ZIB)

SCIP> read sudoku_bin.zpl
original problem has 729 variables (729 bin, 0 int, 0 cont) and 348 constraints
SCIP> optimize
presolving:
(round 1) 24 del vars, 24 del conss, 24 chg bounds, 0 impls, 324 clqs
(round 5) 723 del vars, 324 del conss, 614 chg bounds, 236 impls, 0 clqs
presolving (6 rounds):
 729 deleted vars, 348 deleted constraints, 619 tightened bounds, 238 implications
presolved problem has 0 variables (0 bin, 0 int, 0 cont) and 0 constraints

  time | node  | left  |LP iter|frac |rows |cuts |  dualbound   | primalbound  |  gap
  0.0s|     1 |     0 |     0 |   0 |   0 |   0 | 0.000000e+00 |      --      |  Inf
* 0.0s|     1 |     0 |     0 |   - |   0 |   0 | 0.000000e+00 | 0.000000e+00 | 0.00%

SCIP Status        : problem is solved [optimal solution found]
Solving Time (sec) : 0.03
Solving Nodes      : 1

SCIP> display statistics

Original Problem   :
  Variables        : 729 (729 binary, 0 integer, 0 continuous)
  Constraints      : 348 initial, 348 maximal
Presolved Problem  :
  Variables        : 0 (0 binary, 0 integer, 0 continuous)
  Constraints      : 0 initial, 0 maximal
Presolvers         :     Time  FixedVars   AggrVars  ChgBounds    DelCons
  trivial          :     0.00          9          0          0          0
  linear           :     0.01        646         74        619        348
B&B Tree           :
  nodes            :        1

SCIP> display solution

objective value:                                0
x#1#1#4                                         1    (obj:0)
...
x#9#9#7                                         1    (obj:0)

SCIP> quit
```

number of variables that were fixed by linear constraints

number of variable aggregations due to linear constraints

number of deleted linear constraints

Figure 6: A SCIP session solving a Sudoku instance modeled by Figure 5

Figure 7: Comparison of node trees resulting **vpm2** and **neos3**.