THORSTEN KOCH

Rapid Mathematical Programming or How to Solve Sudoku Puzzles in a few Seconds

ZIB-Report 05-51 (Dec 2005)

1 Introduction

Using the popular puzzle game of Sudoku, this article highlights some of the ideas and topics covered in the author's PhD thesis [8]. The thesis deals with the implementation and application of out-of-the-box tools in linear and mixed integer programming. It documents the lessons learned and conclusions drawn from five years of implementing, maintaining, extending, and using several computer codes to model and solve real-world industrial problems.

By means of several examples it is demonstrated how to apply a modeling language to rapidly devise mathematical models of real-world problems. It is shown that today's MIP-solvers are often capable of solving the resulting mixed integer programs, leading to an approach that delivers results very quickly, even on problems that required the implementation of specialized branch-and-cut algorithms a few years ago.

2 The modeling language Zimpl

The presentation is centered around the newly developed algebraic modeling language Zimpl [8], which is similar in concept to well known languages like GAMS [2] or AMPL [6]. 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.

If AMPL could do this in 1989 why would one bother writing a new program to do the same in 1999? One reason is that all major modeling languages for linear and mixed integer programs are commercial products [7]. None of these languages is available as source code. None can be given to colleagues or used in classes for free, apart from very limited "student editions". Usually, only a limited number of operating systems and architectures are supported. The situation has improved somewhat since 1999 when the development of Zimpl started. Today at least one other open source modeling system is available; the GNU MATHPROG language [13].

What Zimpl distinguishes from other modelling 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 language can be found in [8]. Also details of the implementation are described. Both theoretical and practical considerations are discussed. Aspects of software engineering, error prevention, and detection are addressed. ZIMPL is still under active development and available from the author's website at www.zib.de/koch/zimpl.

3 Real-world projects

In the second part of the thesis, several real-world projects that employed the methodology and the tools developed in the first part of the thesis are examined. Figure 1 shows a typical solution process cycle. In our experience customers, from industry share a few attributes. They

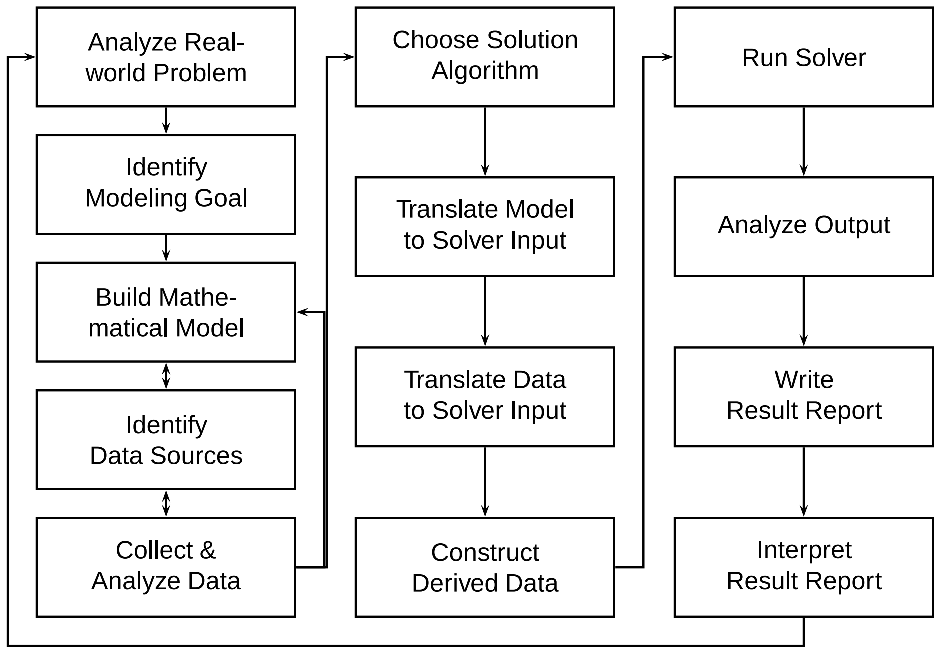
- ▶ do not know exactly what they want,
- ▶ need it next week,
- ▶ have not yet collected the data necessary or do not even have all the data,
- ▶ often need only one shot studies,
- ▶ are convinced *"our problem is unique"*.

This mandates an approach that is fast and flexible. And this is what general tools are all about: Rapid prototyping of mathematical models, quick integration of data, and a fast way to check whether the approach is getting to be feasible. Due to the ongoing advances in hardware and software, the number of problems that can be successfully tackled with this approach is steadily increasing.

While most of the research is aimed at improving solution techniques, we focus on mathematical model building and on how to conveniently translate the model and the problem data into solver input.

The benefits of this approach are demonstrated in [8] by a detailed presentation of four projects from telecommunication industry dealing with facility location and UMTS planning problems. Problems, models, and solutions are discussed. Special emphasis is put on the dependency between the precision of the input data and the results. Possible reasons for unexpected and undesirable solutions are explained. Furthermore, the Steiner tree packing problem in graphs, a well-known hard combinatorial problem,

Figure 1: Modeling cycle according to [10]



is revisited. A formerly known, but not yet used model is applied to combine switchbox wire routing and via minimization in VLSI design. All instances known from the literature are solved by this approach, as well as some newly generated bigger problem instances. The results show that the improvements in solver technology, as claimed in [4, 3], allow our rapid prototyping strategy to succeed even on difficult problems, provided a suitable model is chosen.

4 Sudoku

To give an impression of how to use ZIMPL, we demonstrate the approach on the popular puzzle game Sudoku [5]. The aim of the puzzle is to enter a numeral 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 numerals. At the end, each row, column and subgrid must contain each numeral exactly once. Figure 3(a) shows an example (for details see, e. g., en.wikipedia.org/wiki/Sudoku).

Obviously, the problem can be stated using constraint programming as a collection of *alldifferent* constraints [11]. But how to formulate this as an integer program? ZIMPL can automatically generate IP's for certain constructs such as the absolute value of the difference of two variables (`vabs`). Using 81 integer variables in the range $\{1..9\}$ 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 Listing 1.

Figure 2: Sudoku puzzle and solution

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

(a) Sudoku puzzle

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

(b) Solution

How does the `vabs` construct work? Given a bounded integer variable $l_x \leq x \leq u_x$, where $l_x, x, u_x \in \mathbb{Z}$, two additional binary variables b^+ and b^- are introduced as indicators for whether x is positive or negative, i. e., $b^+ = 1$ if and only if $x > 0$ and $b^- = 1$ if and only if $x < 0$. In case of $x = 0$, both b^+ and b^- are zero. Two additional non-negative variables x^+ and x^- are introduced to hold the positive and negative part of x. This can be formulated as an integer program as follows:

$$\begin{array}{rcccl} & & x^+ - x^- & = & x \\ b^+ & \leq & x^+ & \leq & \max(0, u_x) b^+ \\ b^- & \leq & x^- & \leq & |\min(0, l_x)| b^- \\ & & b^+ + b^- & \leq & 1 \\ & & b^+, b^- & \in & \{0, 1\} \end{array} \tag{1}$$

Note that the polyhedron described by the linear relaxation of System (1) has only integral vertices (see [8] for details).

Listing 1: A Zimpl model to solve Sudoku using integer variables

```
param p         := 3;
set J           := { 0 .. p*p-1 };
set KK          := { 0 .. p-1} * { 0 .. p-1};
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*p+i,n*p+j] - x[m*p+k,n*p+l]) >= 1;
subto fixed: forall <i,j> in F do x[i,j] == fixed[i,j];
```

Using System (1), the following functions and relations can be expressed using $x = v - w$, where $v, w \in \mathbb{Z}$ with $l_x = l_v - u_w$ and $u_x = u_v - l_w$ whenever two operands are involved:

$$\begin{array}{lcl@{\qquad}rcl}
\text{abs}(x) & = & x^+ + x^- & v \neq w & \Leftrightarrow & b^+ + b^- = 1 \\
\text{sgn}(x) & = & b^+ - b^- & v = w & \Leftrightarrow & b^+ + b^- = 0 \\
\min(v,w) & = & w - x^- & v \leq w & \Leftrightarrow & b^+ = 0 \\
\max(v,w) & = & x^+ + w & v < w & \Leftrightarrow & b^- = 1 \\
 & & & v \geq w & \Leftrightarrow & b^- = 0 \\
 & & & v > w & \Leftrightarrow & b^+ = 1
\end{array}$$

More information on this topic can be found, for example, in [12, 9] or at the GAMS website http://www.gams.com/modlib/libhtml/absmip.htm.

Unfortunately, the IP resulting from Listing 1 is hard to solve. CPLEX 9.03 was not able to find a feasible solution after more than six hours and a million branch-and-cut nodes. As an alternative we modeled the problem using binary variables as shown in Listing 2. With this formulation all Sudoku puzzles we have tried so far were solved either by preprocessing or at the root node of the branch-and-bound tree.

Choosing the right formulation is often more important than having the best solver algorithm. Especially with real-world problems, having the ability to experiment swiftly with different formulations is essential.

Listing 2: A Zimpl model to solve Sudoku using binary variables

```
param p := 3;
set J   := { 0 .. p*p-1 };
set KK  := { 0 .. p-1} * { 0 .. p-1 };
set F   := { read "fixed.dat" as "<1n,2n,3n>" };
var x    [J*J*J] binary;

subto rows:  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 nums:  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*p+i,n*p+j,k]==1;
```

5 Conclusion and outlook

It turned out that regarding real-world problems understanding the problem itself and the limitations presented by the available data are often a bigger obstacle than building and solving the resulting mathematical model. The ability to easily experiment with formulations is a key factor for success.

The use of automatically generated constructs in modeling languages makes it even easier to turn complex problems into models. It seems likely that future solvers will "understand" these extended functions directly and either convert them into whatever suits them best or handle them directly [1].

The free availability and the simplicity of use make Zimpl a well suited tool for teaching modeling linear and mixed integer programs.

References

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

[2] 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.

[3] Robert E. Bixby. Solving real-world linear programs: A decade and more of progress. *Operations Research*, 50(1):3–15, 2002.

[4] Robert E. Bixby, Marc Fenelon, Zonghao Gu, Ed Rothberg, and Roland Wunderling. MIP: Theory and practice – closing the gap. In M. J. D. Powell and S. Scholtes, editors, *System Modelling and Optimization: Methods, Theory and Applications*. Kluwer, 2000.

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

[8] Thorsten Koch. *Rapid Mathematical Programming*. PhD thesis, Technische Universität Berlin, 2004. Corrected version available as ZIB technical report 04-58. Regarding software see www.zib.de/koch/zimpl.

[9] Frank Plastria. Formulating logical implications in combinatorial optimization. *European Journal of Operational Research*, 140:338–353, 2002.

[10] Hermann Schichl. Models and the history of modeling. In Josef Kallrath, editor, *Modeling Languages in Mathematical Optimization*, pages 25–36. Kluwer, 2004.

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

[12] H. Paul Williams and Sally C. Brailsford. Computational logic and integer programming. In J. E. Beasley, editor, *Advances in Linear and Integer Programming*, pages 249–281. Oxford University Press, 1996.

[13] GNU linear programming toolkit glpsol version 4.7. www.gnu.org/software/glpk.