ZIMPL User Guide

(Zuse Institute Mathematical Programming Language)

Thorsten Koch

for Version 3.8.0
September, 2026

Contents

1 Preface **2**

2 Introduction **3**

3 Invocation **5**

4 Format **5**
4.1 Expressions 7
4.2 Tuples and sets 8
4.3 Parameters 10
4.4 Initializing sets and parameters from a file 12
4.5 *sum*-expressions 14
4.6 *forall*-statements 14
4.7 Function definitions 14
4.8 The *do print* and *do check* commands 15

5 Models **15**
5.1 Variables 15
5.2 Objective 16
5.3 Constraints 16

6 Modeling examples **20**
6.1 The diet problem 20
6.2 The traveling salesman problem 21
6.3 The capacitated facility location problem 22
6.4 The n-queens problem 24

7 Error messages **28**

Abstract

ZIMPL is a little language in order to translate the mathematical model of a problem into a linear or (mixed-)integer mathematical program, expressed in LP or MPS file format which can be read and (hopefully) solved by a LP or MIP solver.

1 Preface

May the source be with you, Luke!

Many of the things in ZIMPL (and a lot more) can be found in the excellent book about the modeling language AMPL from Robert Fourer, David N. Gay and Brian W. Kernighan [FGK03]. Those interested in an overview of the current state-of-the-art in (commercial) modeling languages might have a look at [Kal04b]. On the other hand, having the source code of a program has its advantages. The possibility to run it regardless of architecture and operating system, the ability to modify it to suit the needs, and not having to hassle with license managers may make a much less powerful program the better choice. And so ZIMPL came into being.

By now ZIMPL has grown up and matured. It has been used in several industrial projects and university lectures, showing that it is able to cope with large-scale models and also with students. This would not have been possible without my early adopters Armin Fügenschuh, Marc Pfetsch, Sascha Lukac, Daniel Junglas, Jörg Rambau, and Tobias Achterberg. Thanks for their comments and bug reports. Special thanks to Tuomo Takkula for revising this manual.

ZIMPL is licensed under the GNU Lesser General Public License Version 3. For more information on free software see `http://www.gnu.org`. The latest version of ZIMPL can be found at `http://zimpl.zib.de`. If you find any bugs, then please send an email to `mailto:koch@zib.de`, and do not forget to include an example that shows the problem. If somebody extends ZIMPL, then I am interested in getting patches to include them into the main distribution.

The best way to refer to ZIMPL in a publication is to cite my PhD thesis [Koc04]

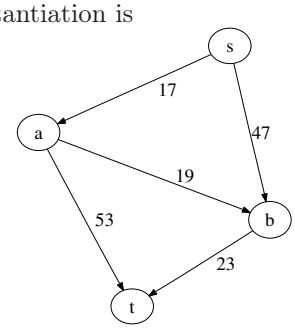
```
@PHDTHESIS{Koch2004,
  author      = "Thorsten Koch",
  title       = "Rapid Mathematical Programming",
  school      = "Technische {Universit\"at} Berlin",
  year        = "2004",
  url         = "http://www.zib.de/Publications/abstracts/ZR-04-58/",
  note        = "ZIB-Report 04-58"
}
```

2 Introduction

Consider the LP formulation of the shortest s,t-path problem, applied to some directed graph (V,A) with cost coefficient c_{ij} for all $(i,j) \in A$:

$$\begin{aligned} \min \quad & \sum_{(i,j)\in A} c_{ij} x_{ij} \\ & \sum_{(iv)\in\delta^-(v)} x_{iv} = \sum_{(vi)\in\delta^+(v)} x_{vi} \quad \text{for all } v \in V \setminus \{s,t\} \\ & x_{ij} \in \{0,1\}, \text{for all i,j in A} \end{aligned} \tag{1}$$

where $\delta^+(v) := \{(v,i) \in A\}$, $\delta^-(v) := \{(i,v) \in A\}$ for $v \in V$. For a given graph the instantiation is



$$\begin{aligned} \min \quad & 17x_{sa} + 47x_{sb} + 19x_{ab} + 53x_{at} + 23x_{bt} \\ \text{subject to} \quad & x_{sa} = x_{ab} + x_{at} \\ & x_{sb} + x_{ab} = x_{bt} \\ & x_{ij} \in \{0,1\}, \text{for all i,j} \end{aligned}$$

The standard format used to feed such a problem into a solver is called MPS which was invented by IBM for the Mathematical Programming System/360 in the sixties [Kal04a, Spi04]. Nearly all available LP and MIP solvers can read this format. While MPS is a nice format to punch into a punch card and at least a reasonable format to read for a computer, it is quite unreadable for humans. For instance, the MPS file of the above linear program looks as follows:

```
NAME          shortestpath.lp
ROWS
 N  Obj
 E  c0
 E  c1
 E  c2
COLUMNS
    INTSTART  'MARKER'        'INTORG'
    x0        Obj         17  c0           1
    x0        c2           1
    x1        Obj         47  c1           1
    x1        c2           1
    x2        c1           1  c0          -1
    x2        Obj         19
    x3        Obj         53  c0          -1
    x4        c1          -1  Obj         23
RHS
    RHS       c2           1
BOUNDS
 UP Bound     x0           1
 UP Bound     x1           1
 UP Bound     x2           1
 UP Bound     x3           1
 UP Bound     x4           1
ENDATA
```

Another possibility is the LP format [ILO02] which is more readable[1], quite similar to the instantiation, but only supported by a few solvers:

```
Minimize
 Obj: +17 x0 +47 x1 +19 x2 +53 x3 +23 x4
Subject to
 c0: -1 x3 -1 x2 +1 x0 = +0
 c1: -1 x4 +1 x2 +1 x1 = +0
 c2: +1 x1 +1 x0 = +1
Bounds
 0 <= x0 <= 1
 0 <= x1 <= 1
 0 <= x2 <= 1
 0 <= x3 <= 1
 0 <= x4 <= 1
Generals
 x0 x1 x2 x3 x4
End
```

Since each coefficient of the matrix A must be stated explicitly it is also not a desirable choice for the development of a mathematical model.

Abstract formulation

Now, with ZIMPL it is possible to write

```
set V      :={"a","b","s","t"};
set A      :={<"s","a">, <"s","b">, <"a","b">, <"a","t">, <"b","t">};
param c[A] := <"s","a"> 17, <"s","b"> 47, <"a","b"> 19, <"a","t"> 53,
              <"b","t"> 23;
defset dminus(v) := {<i,v> in A};
defset dplus(v)  := {<v,j> in A};
var x[A] binary;
minimize cost: sum<i,j> in A: c[i,j] * x[i,j];
subto fc:
      forall <v> in V - {"s","t"}:
        sum<i,v> in dminus(v): x[i,v] == sum<v,i> in dplus(v): x[v,i];
subto uf:
      sum<s,i> in dplus("s"): x[s,i] == 1;
```

– compare this with (1). Feeding the script into ZIMPL will automatically generate MPS or LP files.

The value of modeling languages like ZIMPL lies in their ability to work directly on the mathematical model in lieu of dealing merely with coefficients. Furthermore, more often than not instantiations (read LP or MPS-files) of models are generated from some external data. In some sense, these are the result of the model applied to this external data. In our example above, the external data is the graph together with the cost coefficients and the specification of `s` and `t`, and the model is the formulation of the shortest `st`-path problem. Of course, ZIMPL supports the initialization of models from files. For instance, the first lines from the ZIMPL script above can be replaced by

[1]The LP format has also some idiosyncratic restrictions. For example variables should not be named `e12` or the like. and it is not possible to specify ranged constraints.

```
set       V:= {read "nodes.txt" as "<1s>"};
set       A:= {read "arcs.txt"  as "<1s,2s>"};
param c[A] := read "arcs.txt"  as "<1s,2s>3n";
```

and the ZIMPL script is ready produce instances of any shortest path problem defined by the files "nodes.txt" and "arcs.txt". The format of those files will be described in Subsection 4.4.

3 Invocation

In order to run ZIMPL on a model given in the file `ex1.zpl` type the command:

```
zimpl ex1.zpl
```

In general terms the command is:

```
zimpl [options] <input-files>
```

It is possible to give more than one input file. They are read one after the other as if they were all one big file. If any error occurs while processing, ZIMPL prints out an error message and aborts. In case everything goes well, the results are written into two or more files, depending on the specified options.

The first output file is the problem generated from the model in either CPLEXLP, MPS, or a "human readable" format, with extensions *.lp*, *.mps*, or *.hum*, respectively. The next one is the *table* file which has the extension *.tbl*. The table file lists all variable and constraint names used in the model and their corresponding names in the problem file. The reason for this name translation is the limitation of the length of names in the MPS format to eight characters. Also the LP format restricts the length of names. The precise limit is depending on the version. CPLEX 7.0 has a limit of 16 characters, and ignores silently the rest of the name, while CPLEX 9.0 has a limit of 255 characters, but will for some commands only show the first 20 characters in the output.

A complete list of all options understood by ZIMPL can be found in Table 1. A typical invocation of ZIMPL is for example:

```
zimpl -o solveme -t mps data.zpl model.zpl
```

This reads the files data.zpl and model.zpl as input and produces as output the files solveme.mps and solveme.tbl. Note that in case MPS-output is specified for a maximization problem, the objective function will be inverted, because the MPS format has no provision for stating the sense of the objective function. The default is to assume minimization.

4 Format

Each ZPL-file consists of six types of statements:

- Sets
- Parameters
- Variables
- Objective
- Constraints
- Function definitions

Each statement ends with a semicolon. Everything from a hash-sign `#`, provided it is not part of a string, to the end of the line is treated as a comment and is ignored. If a line starts with the word `include` followed by a filename in double quotation marks, then this file is read and processed instead of the line.

-t *format*	Selects the output format. Can be either `lp`, which is default, or `mps`, or `hum`, which is only human readable, or `rlp`, which is the same as `lp` but with rows and columns randomly permuted. The permutation is depending on the *seed*. Also possible is `pip` which means Polynomial IP, or `q` *x* which means QUBO. *x* are format options. Available are: *0* for zero based matrix indexing (1 is default), *c* for using 'c' as comment line indicator in the file (default is '#'), and *p* for writing an 'p' in the first line of the instance.
-o *name*	Sets the base-name for the output files. Defaults to the name of the first input file with its path and extension stripped off.
-F *filter*	The output is piped through a filter. A %s in the string is replaced by the output filename. For example `-F "gzip -c >%s.gz"` would compress all the output files.
-l *length*	Sets maximal length of variables and constraints for `lp` format files to *length*.
-n *cform*	Select the format for the generation of constraint names. Can be `cm` which will number them `1...n` with a 'c' in front. `cn` will use the name supplied in the `subto` statement and number them `1...n` within the statement. `cf` will use the name given with the `subto`, then a `1...n` number like in `cm` and then append all the local variables from the forall statements.
-P *filter*	The input is piped through a filter. A %s in the string is replaced by the input filename. For example `-P "cpp -DWITH_C1 %s"` would pass the input file through the C-preprocessor.
-s *seed*	Positive `seed` number for the random number generator. For example, `-s ‘date +%N‘` will produce changing random numbers.
-v *0..5*	Set the verbosity level. 0 is quiet, 1 is default, 2 is verbose, 3 and 4 are chatter, and 5 is debug.
-D *name=val*	Sets the parameter *name* to the specified value. This is equivalent to having this line in the ZIMPL program: `param name:=val`. If there is a declaration in the ZIMPL file and a `-D` setting for the same name, the latter takes precedent.
-b	Enables bison debug output.
-f	Enables flex debug output.
-h	Prints a help message.
-m	Writes a CPLEX`mst` (Mip STart) file.
-O	Tries to reduce the generated LP by doing some presolve analysis.
-r	Writes a CPLEX`ord` branching order file.
-V	Prints the version number.

Table 1: ZIMPL options

4.1 Expressions

ZIMPL works on its lowest level with two types of data: Strings and numbers. Wherever a number or string is required it is also possible to use a parameter of the corresponding value type. In most cases, expressions are allowed instead of just a number or a string. The precedence of operators is the usual one, but parentheses can always be used to specify the evaluation order explicitly.

Numeric expressions

A number in ZIMPL can be given in the usual format, e. g. as 2, -6.5 or 5.234e-12. Numeric expressions consist of numbers, numerically valued parameters, and any of the operators and functions listed in Table 2. Additionally the functions shown in Table 3 can be used. Note that those functions are only computed with normal double precision floating-point arithmetic and therefore have limited accuracy. Examples on how to use the `min` and `max` functions can be found in Section 4.3 on page 10.

`a^b`, `a**b`	a to the power of b	a^b, b must be integer
`a+b`	addition	$a + b$
`a-b`	subtraction	$a - b$
`a*b`	multiplication	$a \cdot b$
`a/b`	division	a/b
`a mod b`	modulo	$a \mod b$
`abs(a)`	absolute value	$\lvert a \rvert$
`sgn(a)`	sign	$x > 0 \Rightarrow 1, x < 0 \Rightarrow -1$, else 0
`floor(a)`	round down	$\lfloor a \rfloor$
`ceil(a)`	round up	$\lceil a \rceil$
`round(a)`	round towards zero	$\lfloor a \rceil$
`a!`	factorial	$a!$, a must be nonnegative integer
`min(S)`	minimum of a set	$\min_{s \in S}$
`min <s> in S: e(s)`	minimum over a set	$\min_{s \in S} e(s)$
`max(S)`	maximum of a set	$\max_{s \in S}$
`max <s> in S: e(s)`	maximum over a set	$\max_{s \in S} e(s)$
`min(a,b,c,...,n)`	minimum of a list	$\min(a, b, c, \ldots, n)$
`max(a,b,c,...,n)`	maximum of a list	$\max(a, b, c, \ldots, n)$
`sum(s in S) e(s)`	sum over a set	$\sum_{s \in S} e(s)$
`prod(s in S) e(s)`	product over a set	$\prod_{s \in S} e(s)$
`card(S)`	cardinality of a set	$\lvert S \rvert$
`random(m,n)`	pseudo random number	$\in [m, n]$, rational
`ord(A,n,c)`	ordinal	c-th component of the n-th element of set A.
`length(s)`	length of a string	number of characters in s
`if a then b else c end`	conditional	$\begin{cases} b, & \text{if } a = \text{true} \\ c, & \text{if } a = \text{false} \end{cases}$

Table 2: Rational arithmetic functions

String expressions

A string is delimited by double quotation marks ", e. g. `"Hallo Keiken"`. Two strings can be concatenated using the + operator, i. e., `"Hallo " + "Keiken"` gives `"Hallo Keiken"`. The function `substr(string, begin, length)` can be used to extract parts

`sqrt(a)`	square root	$\sqrt{a}$
`log(a)`	logarithm to base 10	$\log_{10} a$
`ln(a)`	natural logarithm	$\ln a$
`exp(a)`	exponential function	e^a

Table 3: Double precision functions

of a string. `begin` is the first character to be used, and counting starts with zero at first character. If `begin` is negative, then counting starts at the end of the string. `length` is the number of charaters to extract starting at `begin`. The length of a string can be determined using the `length` function.

Boolean expressions

These evaluate either to *true* or to *false*. For numbers and strings the relational operators <, <=, ==, !=, >=, and > are defined. Combinations of Boolean expressions with `and`, `or`, and `xor`[2] and negation with `not` are possible. The expression *tuple* `in` *set-expression* (explained in the next section) can be used to test set membership of a tuple. Boolean expressions can also be in the *then* or *else* part of an *if* expression.

Variant expressions

The following is either a numeric, string or boolean expression, depending on whether *expression* is a string, boolean, or numeric expression:

`if` *boolean-expression* `then` *expression* `else` *expression* `end`

The same is true for the `ord(`*set, tuple-number, component-number*`)` function which evaluates to a specific element of a set (details about sets are covered below).

4.2 Tuples and sets

A tuple is an ordered vector of fixed dimension where each component is either a number or a string. Sets consist of (a finite number of) tuples. Each tuple is unique within a set. All sets in ZIMPL are internally ordered, but there is no particular order. Sets are delimited by braces, `{` and `}`, respectively. All tuples of a specific set have the same number of components. The type of the n-th component for all tuples of a set must be the same, i. e., they have to be either all numbers or all strings. The definition of a tuple is enclosed in angle brackets < and >, e. g. `<1,2,"x">`. The components are separated by commas. If tuples are one-dimensional, it is possible to omit the tuple delimiters in a list of elements, but in this case they must be omitted from all tuples in the definition, e. g. `{1,2,3}` is valid while `{1,2,<3>}` is not.

Sets can be defined with the set statement. It consists of the keyword `set`, the name of the set, an assignment operator `:=`, and a valid set expression.

Sets are referenced by the use of a *template* tuple, consisting of placeholders which are replaced by the values of the components of the respective tuple. For example, a set `S` consisting of two-dimensional tuples could be referenced by `<a,b> in S`. If any of the placeholders are actual values, only those tuples matching these values will be extracted. For example, `<1,b> in S` will only get those tuples whose first component is `1`. Please note that if one of the placeholders is the name of an already defined parameter, set or variable, it will be substituted. This will result either in an error or an actual value.

[2] a xor $b := a \wedge \neg b \vee \neg a \wedge b$

Examples

```
set A := { 1, 2, 3 };
set B := { "hi", "ha", "ho" };
set C := { <1,2,"x">, <6,5,"y">, <787,12.6,"oh"> };
```

For set expressions the functions and operators given in Table 4 are defined.

An example for the use of the `if` *boolean-expression* `then` *set-expression* `else` *set-expression* `end` can be found on page 9 together with the examples for indexed sets.

Examples

```
set D := A cross B;
set E := { 6 to 9 } union A without { 2, 3 };
set F := { 1 to 9 } * { 10 to 19 } * { "A", "B" };
set G := proj(F, <3,1>);
# will give: { <"A",1>, <"A",2"> ... <"B",9> }
```

Conditional sets

It is possible to restrict a set to tuples that satisfy a Boolean expression. The expression given by the `with` clause is evaluated for each tuple in the set and only tuples for which the expression evaluates to *true* are included in the new set.

Examples

```
set F := { <i,j> in Q with i > j and i < 5 };
set A := { "a", "b", "c" };
set B := { 1, 2, 3 };
set V := { <a,2> in A*B with a == "a" or a == "b" };
# will give: { <"a",2>, <"b",2> }
set W := argmin(3) <i,j> in B*B : i+j;
# will give: { <1,1>, <1,2>, <2,1> }
```

Indexed sets

It is possible to index one set with another set resulting in a set of sets. Indexed sets are accessed by adding the index of the set in brackets `[` and `]`, like `S[7]`. Table 5 lists the available functions. There are three possibilities how to assign to an indexed set:

- The assignment expression is a list of comma-separated pairs, consisting of a tuple from the index set and a set expression to assign.
- If an index tuple is given as part of the index, e. g. `<i> in I`, the assignment is evaluated for each value of the index tuple.
- By use of a function that returns an indexed set.

Examples

```
set I           := { 1..3 };
set A[I]        := <1> {"a","b"}, <2> {"c","e"}, <3> {"f"};
set B[<i> in I] := { 3 * i };
set P[]         := powerset(I);
set J           := indexset(P);
set S[]         := subsets(I, 2);
```

`A*B`, `A cross B`	cross product	$\{(x, y) \mid x \in A \wedge y \in B\}$
`A+B`, `A union B`	union	$\{x \mid x \in A \vee x \in B\}$
`union <i>` ` in I: S`	ditto, of indexed sets	$\bigcup_{i \in I} S_i$
`A inter B`	intersection	$\{x \mid x \in A \wedge x \in B\}$
`inter <i>` ` in I: S`	ditto, of indexed sets	$\bigcap_{i \in I} S_i$
`A\B, A-B`, `A without B`	difference	$\{x \mid x \in A \wedge x \notin B\}$
`A symdiff B`	symmetric difference	$\{x \mid (x \in A \wedge x \notin B) \vee (x \in B \wedge x \notin A)\}$
`{n..m by s}`,	generate, (default $s = 1$)	$\{x \mid x = \min(n, m) + i\lvert s\rvert \leqslant \max(n, m),$ $i \in \mathbb{N}_0, x, n, m, s \in \mathbb{Z}\}$
`{n to m by s}`	generate	$\{x \mid x = n + is \leqslant m, i \in \mathbb{N}_0, x, n, m, s \in \mathbb{Z}\}$
`proj(A, t)`	projection $t = (e_1, \ldots, e_n)$	The new set will consist of n-tuples, with the i-th component being the e_i-th component of A.
`argmin <i>` ` in I : e(i)`	minimum argument	$\operatorname{argmin}_{i \in I} e(i)$
`argmin(n) <i>` ` in I : e(i)`	n minimum arguments	The new set consists of those n elements of i for which $e(i)$ was smallest. The result can be ambiguous.
`argmax <i>` ` in I : e(i)`	maximum argument	$\operatorname{argmax}_{i \in I} e(i)$
`argmax(n) <i>` ` in I : e(i)`	n maximum arguments	The new set consists of those n elements of i for which $e(i)$ was biggest. The result can be ambiguous.
`if a then b` `else c end`	conditional	$\begin{cases} b, & \text{if } a = \text{true} \\ c, & \text{if } a = \text{false} \end{cases}$
`permute(A)`	permutations	generate a set consisting of tuples with all permutations of the elements of A

Table 4: Set related functions

```
set T[]         := subsets(I, 1, 2);
set K[<i> in I] := if i mod 2 == 0 then { i } else { -i } end;
set U           := union <i> in I : A[i];
set IN          := inter <j> in J : P[j]; # empty!
```

4.3 Parameters

Parameters are the way to define constants within ZIMPL. They can be declared with or without an index set. Without index a parameter is just a single value which is either a number or a string. For indexed parameters there is one value for each member of the set. It is possible to declare a *default* value.

`powerset(A)`	generates all subsets of A	$\{X \mid X \subseteq A\}$
`subsets(A,n)`	generates all subsets of A with n elements	$\{X \mid X \subseteq A \wedge \lvert X\rvert = n\}$
`subsets(A,n,m)`	generates all subsets of A with between n and m elements	$\{X \mid X \subseteq A \wedge n \leqslant \lvert X\rvert \leqslant m\}$
`indexset(A)`	the index set of A	$\{1 \ldots \lvert A\rvert\}$

Table 5: Indexed set functions

Parameters are declared in the following way: The keyword `param` is followed by the name of the parameter optionally followed by the index set in square brackets. Then, after the assignment operator, there is a list of pairs. The first element of each pair is a tuple from the index set, while the second element is the value of the parameter for this index. If there is only a single value given, it will be assigned to all members of the parameter.

Examples

```
set A := { 12 .. 30 };
set C := { <1,2,"x">, <6,5,"y">, <3,7,"z"> };
param q := 5;
param r[C] := 7;          # all members are 7
param r[C] := default 7; # same as line above
param u[A] := <13> 17, <17> 29, <23> 14 default 99;
param str[A] := <13> "hallo", <17> "tach" default "moin";
param amin := min A;                # = 12
param umin := min <a> in A : u[a]; # = 14
param mmax := max <i> in { 1 .. 10 } : i mod 5;
param w[C] := <1,2,"x"> 1/2, <6,5,"y"> 2/3;
param x[<i> in { 1 .. 8 } with i mod 2 == 0] := 3 * i;
```

Assignments do not need to be complete. In the example, no value is given for index `<3,7,"z">` of parameter `w`. This is correct as long as it is never referenced.

Parameter tables

It is possible to initialize multi-dimensional indexed parameters from tables. This is especially useful for two-dimensional parameters. The data is put in a table structure with | signs on each margin. Then a headline with column indices has to be added, and one index for each row of the table is needed. The column index has to be one-dimensional, but the row index can be multi-dimensional. The complete index for the entry is built by appending the column index to the row index. The entries are separated by commas. Any valid expression is allowed here. As can be seen in the third example below, it is possible to add a list of entries after the table.

Examples

```
set I := { 1 .. 10 };
set J := { "a", "b", "c", "x", "y", "z" };

param h[I*J] :=   | "a", "c", "x", "z"   |
                |1|  12,  17, 99,     23 |
```

```
                |3|   4,   3,-17, 66*5.5 |
                |5| 2/3, -.4,  3, abs(-4)|
                |9|   1,   2,  0,      3 | default -99;

param g[I*I*I] :=     | 1, 2, 3 |
                    |1,3| 0, 0, 1 |
                    |2,1| 1, 0, 1 |;

param k[I*I] :=  |  7,  8,  9 |
               |4| 89, 67, 55 |
               |5| 12, 13, 14 |, <1,2> 17, <3,4> 99;
```

The last example is equivalent to:

```
param k[I*I] := <4,7> 89, <4,8> 67, <4,9> 55, <5,7> 12,
                <5,8> 13, <5,9> 14, <1,2> 17, <3,4> 99;
```

4.4 Initializing sets and parameters from a file

It is possible to load the values for a set or a parameter from a file. The syntax is:

`read` *filename* `as` *template* [`skip` *n*] [`use` *n*] [`match` *s*] [`comment` *s*]

filename is the name of the file to read. *template* is a string with a template for the tuples to generate. Each input line from the file is split into fields. The splitting is done according to the following rules: All leading and trailing space and tab characters in the line are removed. Whenever a space, tab, comma, semicolon or colon is encountered a new field is started. Text that is enclosed in double quotes is not split and the quotes are always removed. When a field is split all space and tab characters around the splitting point are removed. If the split is due to a comma, semicolon or colon, each occurrence of these characters starts a new field.

Examples

All these lines have three fields:

```
Hallo;12;3
Moin   7  2
"Hallo, Peter"; "Nice to meet you" 77
,,2
```

For each component of the tuple, the number of the field to use for the value is given, followed by either `n` if the field should be interpreted as a number or `s` for a string. After the template, some optional modifiers can be given. The order does not matter. `match` *s* compares the regular expression *s* against the line read from the file. Only if the expression matches the line, it is processed further. POSIX extended regular expression syntax is used. `comment` *s* sets a list of characters that start comments in the file. Each line is ended when any of the comment characters is found. `skip` *n* instructs to skip the first *n* lines of the file. `use` *n* limits the number of lines to use to *n*. When a file is read, empty lines, comment lines, and unmatched lines are skipped and not counted for the `use` and `skip` clauses.

Examples

```
set P := { read "nodes.txt" as "<1s>" };
```

```
nodes.txt:
   Hamburg              → <"Hamburg">
   München              → <"München">
   Berlin               → <"Berlin">

set Q := { read "blabla.txt" as "<1s,5n,2n>" skip 1 use 2 };
blabla.txt:
   Name;Nr;X;Y;No       → skip
   Hamburg;12;x;y;7     → <"Hamburg",7,12>
   Bremen;4;x;y;5       → <"Bremen",5,4>
   Berlin;2;x;y;8       → skip

param cost[P] := read "cost.txt" as "<1s> 2n" comment "#";
cost.txt:
   # Name Price         → skip
   Hamburg 1000         → <"Hamburg"> 1000
   München 1200         → <"München"> 1200
   Berlin  1400         → <"Berlin"> 1400

param cost[Q] := read "haha.txt" as "<3s,1n,2n> 4s";
haha.txt:
      1:2:ab:con1       → <"ab",1,2> "con1"
      2:3:bc:con2       → <"bc",2,3> "con2"
      4:5:de:con3       → <"de",4,5> "con3"
```

As with table format input, it is possible to add a list of tuples or parameter entries after a read statement.

Examples

```
set A := { read "test.txt" as "<2n>", <5>, <6> };
param winniepoh[X] :=
   read "values.txt" as "<1n,2n> 3n", <1,2> 17, <3,4> 29;
```

It is also possible to read a single value into a parameter. In this case, either the file should contain only a single line, or the read statement should be instructed by means of a `use 1` parameter only to read a single line.

Examples

```
# Read the fourth value in the fifth line
param n := read "huhu.dat" as "4n" skip 4 use 1;
```

If all values in a file should be read into a set, this is possible by use of the `"<s+>"` template for string values, and `"<n+>"` for numerical values. Note, that currently at most 65536 values are allowed in a single line.

Examples

```
# Read all values into a set
set X := { read "stream.txt" as "<n+>" };
```

```
stream.txt:
1 2 3 7 9 5 6 23
63 37 88
4
87 27

# X := { 1, 2, 3, 7, 9, 5, 6, 23, 63, 37, 88, 4, 87, 27 };
```

4.5 *sum*-expressions

Sums are stated in the form:

`sum` *index* `do` *term*

It is possible to nest several sum instructions, but it is probably more convenient to simply merge the indices. The general form of *index* is:

tuple `in` *set* `with` *boolean-expression*

It is allowed to write a colon `:` instead of `do` and a vertical bar `|` instead of `with`. The number of components in the *tuple* and in the members of the *set* must match. The `with` part of an *index* is optional. The *set* can be any expression giving a set. Examples are given in the next section.

4.6 *forall*-statements

The general forms are:

`forall` *index* `do` *term*

It is possible to nest several forall instructions. The general form of *index* equals that of *sum*-expressions. Examples are given in the next section.

Caveat! Scope rules for index variables in *forall* and *sum*

Index variables which are already defined outside the *forall* and *sum* statement are replaced by their current value. This also happens for variables inside functions definitions which are already defined outside.

Examples

```
k = 5; sum <i,k> in {1..10}*{1..50}: i; # == 55 as k is fixed
forall <k> in K: sum <k,b> in K*{1..3}: z[k,b]; # |K| * sum of 3 elements
```

4.7 Function definitions

It is possible to define functions within ZIMPL. The value a function returns has to be either a number, a string, a boolean, or a set. The arguments of a function can only be numbers or strings, but within the function definition it is possible to access all otherwise declared sets and parameters.

The definition of a function has to start with `defnumb`, `defstrg`, `defbool`, or `defset`, depending on the return value. Next is the name of the function and a list of argument names put in parentheses. An assignment operator `:=` has to follow and a valid expression or set expression.

Examples

```
defnumb dist(a,b)     := sqrt(a*a + b*b);
defstrg huehott(a)    := if a < 0 then "hue" else "hott" end;
defbool wirklich(a,b) := a < b and a >= 10 or b < 5;
defset  bigger(i)     := { <j> in K with j > i };
K = 5; defnum(a) := sum <i,k> in {1..3}*{1..5}: a; # == 3*a
```

4.8 The *do print* and *do check* commands

The `do` command is special. It has two possible incarnations: `print` and `check`. `print` will print to the standard output stream whatever numerical, string, Boolean or set expression, or tuple follows it. This can be used for example to check if a set has the expected members, or if some computation has the anticipated result. `check` always precedes a Boolean expression. If this expression does not evaluate to *true*, the program is aborted with an appropriate error message. This can be used to assert that specific conditions are met. It is possible to use a `forall` clause before a `print` or `check` statement. For string and numeric values, it is possible to give a comma-separated list to the print statement.

Examples

```
set I := { 1..10 };
do print I;
do print "Cardinality of I:", card(I);
do forall <i> in I with i > 5 do print sqrt(i);
do forall <p> in P do check sum <p,i> in PI : 1 >= 1;
```

5 Models

In this section, we use the machinery developed up to now to formulate mathematical programs. Apart from the usual syntax to declare variables, objective functions and constraints there is special syntax available that permits the easy formulation of special constraints, such as special ordered sets, conditional constraints, and extended functions.

5.1 Variables

Like parameters, variables can be indexed. A variable has to be one out of three possible types: Continuous (called `real`), `binary` or `integer`. The default type is real. Variables may have lower and upper bounds. Defaults are zero as the lower and infinity as the upper bound. Binary variables are always bounded between zero and one. It is possible to compute the value of the lower or upper bounds depending on the index of the variable (see the last declaration in the example). Bounds can also be set to `infinity` and `-infinity`. Binary and integer variables can be declared `implicit`, i. e., the variable can be assumed to have an integral value in any optimal solution to the integer program, even if it is declared continuous. Use of *implicit* is only useful if used together with a solver that take advantage of this information. As of this writing, only SCIP (`http://scipopt.org`) with linked in ZIMPL can do so. In all other cases, the variable is treated as a continuous variable. It is possible to specify initial values for integer variables by use of the `startval` keyword. Furthermore, the branching priority can be given using the `priority` keyword. Currently, these values are written to a CPLEX`ord` branching order file if the `-r` command line switch is given.

Examples

```
var x1;
var x2 binary;
var x3 integer >= -infinity      # free variable
var y[A] real >= 2 <= 18;
var z[<a,b> in C] integer
      >= a * 10 <= if b <= 3 then p[b] else infinity end;
var w implicit binary;
var t[<k> in K] integer >= 1 <= 3 * k priority 50 startval 2 * k;
```

5.2 Objective

There must be at most one objective statement in a model. The objective can be either `minimize` or `maximize`. Following the keyword is a name, a colon : and then a linear term expressing the objective function.

If there is an objective offset, i. e., a constant value in the objective function, ZIMPL automatically generates an internal variable `@@ObjOffset` set to one. This variable is put into the objective function with the appropriate coefficient.[3]

Example

```
minimize cost: 12 * x1 -4.4 * x2 + 5
   + sum <a> in A : u[a] * y[a]
   + sum <a,b,c> in C with a in E and b > 3 : -a/2 * z[a,b,c];
maximize profit: sum <i> in I : c[i] * x[i];
```

5.3 Constraints

The general format for a constraint is:

```
subto name: term sense term
```

Alternatively, it is also possible to define *ranged* constraints which have the form:

```
subto name: expr sense term sense expr
```

`name` can be any name starting with a letter. `term` is defined as in the objective. `sense` is one of `<=`, `>=` and `==`. In case of ranged constraints both senses have to be equal and may not be `==`. `expr` is any valid expression that evaluates to a number.

Additional to linear constraints as in the objective it is also possible to state terms with higher degree for constraints.

Many constraints can be generated with one statement by the use of the `forall` instruction, as shown below.

Examples

```
subto time:  3 * x1 + 4 * x2 <= 7;
subto space: 50 >= sum <a> in A: 2 * u[a] * y[a] >= 5;
subto weird: forall <a> in A: sum <a,b,c> in C: z[a,b,c]==55;
subto c21: 6*(sum <i> in A: x[i] + sum <j> in B : y[j]) >= 2;
subto c40: x[1] == a[1] + 2 * sum <i> in A do 2*a[i]*x[i]*3+4;
subto frown: forall <i,j> in X cross { 1 to 5 } without { <2,3> }
              with i > 5 and j < 2 do
```

[3]The reason for this is that there is no portable way to put an offset into the objective function in neither LP nor MPS-format.

```
                  sum <i,j,k> in X cross { 1 to 3 } cross Z do
                     p[i] * q[j] * w[j,k] >= if i == 2 then 17 else 53 end;
subto nonlin: 3 * x * y * z + 6 <= x^3 + z * y^2 + 3;
```

Note that in the example i and j are set by the `forall` instruction. So they are fixed in all invocations of `sum`.

if in terms

Part of a constraint can be put into an `if-then-else-end`. In this particular case, the else part is mandatory and both parts need to include some variables in the term.

Examples

```
subto c2: sum <i> in I :
     if (i mod 2 == 0) then 3 * x[i] else -2 * y[i] end <= 3;
```

if in constraints

It is possible to put two variants of a constraint into an `if`-statement. A `forall` statement inside the result part of an `if` is also possible. The `else` part is optional.

Examples

```
subto c1: forall <i> in I do
  if (i mod 2 == 0) then  3 * x[i] >= 4
                    else -2 * y[i] <= 3 end;
```

Combining constraints with *and*

It is possible to group constraints by concatenating them with `and`. This group will then always be processed together. This operator is particularly useful in combination with `forall` and `if`.

Examples

```
subto c1: forall <i> in I:
  if (i > 3)
     then if (i == 4)
        then z[1] + z[2] == i
        else z[2] + z[3] == i
     end and
     z[3] + z[4] == i and
     z[4] + z[5] == i
  else
     if (i == 2)
        then z[1] + z[2] == i + 10
        else z[2] + z[3] == i + 10
     end and
     z[3] + z[4] == i + 10 and
     z[4] + z[5] == i + 10
  end;
```

Constraint attributes

It is possible to specify special attributes for a constraint regarding how it should be handled later on.

scale Before the constraint is written to a file, it is scaled by $1/\max|c|$ with c being the coefficients of the constraint.

separate Do not include the constraint in the initial LP, but separate it later on. The constraint need not to be checked for feasibility. When written to an LP file it will be written into a `USER CUTS` section, in SCIP separate will be set to true.

checkonly Do not include the constraint in the initial LP, but separate only to check for feasibility. When written to an LP file it will be written into a `LAZY CUTS` section, in SCIP check will be set to true.

indicator If vif (see below) is part of the constraint then it is modeled as an indicator constraint and not by a big-M formulation. If you use this attribute, you can no longer write MPS-format, as indicator variables are not supported by this format.

qubo Transform the constraint into an qudratic objective. Note that neccessary binary slacks are created automatically. However, integer variables are currently not automatically converted into binary variables.

penaltyX Only to be used together with the *qubo* attribute. Multiply the constraint that is put into the objective by a factor of $P = 10^X$, $X \in \{1, \ldots, 6\}$.

The attributes are given after the constraint, before the semi-colon and are separated by commas.

Examples

```
subto c1: 1000 * x + 0.3 * y <= 5, scale, checkonly;
subto c2: x + y + z == 7, separate;
subto c3: vif x == 1 then y == 7 end, indicator;
subto c4: sum <i> in I : x[i] <= 15, qubo, penalty3;
```

Special ordered sets

ZIMPL can be used to specify special ordered sets (SOS) for an integer program. If a model contains any SOS a `sos` file is written together with the `lp` or `mps` file. The general format of a special ordered set is:

```
sos name: [type1|type2] priority expr : term
```

`name` can be any name starting with a letter. SOS use the same namespace as constraints. `term` is defined as in the objective. `type1` or `type2` indicate whether a type-1 or type-2 special ordered set is declared. The priority is optional and equal to the priority setting for variables. Many SOS can be generated with one statement by the use of the `forall` instruction, as shown above.

Examples

```
sos s1: type1: 100 * x[1] + 200 * x[2] + 400 * x[3];
sos s2: type2 priority 100 : sum <i> in I: a[i] * x[i];
sos s3: forall <i> in I with i > 2:
          type1: (100 + i) * x[i] + i * x[i-1];
```

Extended constraints

ZIMPL can generate systems of constraints that mimic conditional constraints. The general syntax is as follows (note that the `else` part is optional):

`vif` *boolean-constraint* `then` *constraint* [`else` *constraint*] `end`

where *boolean-constraint* consists of a linear expression involving variables. All these variables have to be bounded integer or binary variables. It is not possible to use any continuous variables or integer variables with infinite bounds in a *boolean-constraint*. All comparison operators ($<$, $\leqslant$, $==$, $!=$, $\geqslant$, $>$) are allowed. Also, the combination of several terms with `and`, `or`, and `xor` and negation with `not` is possible. The conditional constraints (those which follow after `then` or `else`) may include bounded continuous variables. Be aware that using this construct will lead to the generation of several additional constraints and variables.

Examples

```
var x[I] integer >= 0 <= 20;
subto c1: vif 3 * x[1] + x[2] != 7
   then sum <i> in I : y[i] <= 17
   else sum <k> in K : z[k] >= 5 end;
subto c2: vif x[1] == 1 and x[2] > 5 then x[3] == 7 end;
subto c3: forall <i> in I with i < max(I) :
   vif x[i] >= 2 then x[i + 1] <= 4 end;
```

Extended functions

It is possible to use special functions on terms with variables that will automatically be converted into a system of inequalities. The arguments of these functions have to be linear terms consisting of bounded integer or binary variables. At the moment only the function `vabs(t)` that computes the absolute value of the term `t` is implemented, but functions like the minimum or the maximum of two terms, or the sign of a term can be implemented in a similar manner. Again, using this construct will lead to the generation of several additional constraints and variables.

Examples

```
var x[I] integer >= -5 <= 5;
subto c1: vabs(sum <i> in I : x[i]) <= 15;
subto c2: vif vabs(x[1] + x[2]) > 2 then x[3] == 2 end;
```

6 Modeling examples

In this section we show some examples of well-known problems translated into ZIMPL format.

6.1 The diet problem

This is the first example in [Chv83, Chapter 1, page 3]. It is a classic so-called *diet* problem, see for example [Dan90] about its implications in practice.

Given a set of foods F and a set of nutrients N, we have a table π_{fn} of the amount of nutrient n in food f. Now Π_n defines how much intake of each nutrient is needed. Δ_f denotes for each food the maximum number of servings acceptable. Given prices c_f for each food, we have to find a selection of foods that obeys the restrictions and has minimal cost. An integer variable x_f is introduced for each $f \in F$ indicating the number of servings of food f. Integer variables are used, because only complete servings can be obtained, i. e., half an egg is not an option. The problem may be stated as:

$$\begin{aligned} \min \sum_{f \in F} c_f x_f & \qquad \text{subject to} \\ \sum_{f \in F} \pi_{fn} x_f \geqslant \Pi_n & \qquad \text{for all } n \in N \\ 0 \leqslant x_f \leqslant \Delta_f & \qquad \text{for all } f \in F \\ x_f \in \mathbb{N}_0 & \qquad \text{for all } f \in F \end{aligned}$$

This translates into ZIMPL as follows:

```
set Food      := { "Oatmeal", "Chicken", "Eggs",
                   "Milk",    "Pie",     "Pork" };
set Nutrients := { "Energy",  "Protein", "Calcium" };
set Attr      := Nutrients + { "Servings", "Price" };

param needed[Nutrients] :=
   <"Energy"> 2000, <"Protein"> 55, <"Calcium"> 800;

param data[Food * Attr] :=
            |"Servings","Energy","Protein","Calcium","Price"|
|"Oatmeal"|         4 ,     110 ,       4 ,       2 ,     3 |
|"Chicken"|         3 ,     205 ,      32 ,      12 ,    24 |
|"Eggs"   |         2 ,     160 ,      13 ,      54 ,    13 |
|"Milk"   |         8 ,     160 ,       8 ,     284 ,     9 |
|"Pie"    |         2 ,     420 ,       4 ,      22 ,    20 |
|"Pork"   |         2 ,     260 ,      14 ,      80 ,    19 |;
#                       (kcal)       (g)      (mg) (cents)

var x[<f> in Food] integer >= 0 <= data[f, "Servings"];

minimize cost: sum <f> in Food : data[f, "Price"] * x[f];

subto need: forall <n> in Nutrients do
   sum <f> in Food : data[f, n] * x[f] >= needed[n];
```

The cheapest meal satisfying all requirements costs 97 cents and consists of four servings of oatmeal, five servings of milk and two servings of pie.

6.2 The traveling salesman problem

In this example, we show how to generate an exponential description of the *symmetric traveling salesman problem* (TSP) as given for example in [Sch03, Section 58.5].

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

$$\min \sum_{(i,j)\in E} d_{ij}x_{ij} \qquad \text{subject to}$$

$$\sum_{(i,j)\in\delta_v} x_{ij} = 2 \qquad \text{for all } v \in V$$

$$\sum_{(i,j)\in E(U)} x_{ij} \leqslant |U| - 1 \qquad \text{for all } U \subseteq V, \emptyset \neq U \neq V$$

$$x_{ij} \in \{0,1\} \qquad \text{for all } (i,j) \in E$$

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

```
# City        X    Y
Berlin      5251 1340
Frankfurt   5011  864
Leipzig     5133 1237
Heidelberg  4941  867
Karlsruhe   4901  840
Hamburg     5356  998
Bayreuth    4993 1159
Trier       4974  668
Hannover    5237  972
Stuttgart   4874  909
Passau      4856 1344
Augsburg    4833 1089
Koblenz     5033  759
Dortmund    5148  741
Bochum      5145  728
Duisburg    5142  679
Wuppertal   5124  715
Essen       5145  701
Jena        5093 1158
```

The formulation in ZIMPL follows below. Please note that `P[]` holds all subsets of the cities. As a result 19 cities is about as far as one can get with this approach. Information on how to solve much larger instances can be found on the CONCORDE website[4].

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

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

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

var x[E] binary;

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

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

subto no_subtour:
   forall <k> in K with
```

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

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

The resulting LP has 171 variables, 239,925 constraints, and 22,387,149 non-zero entries in the constraint matrix, giving an MPS-file size of 936 MB. CPLEX solves this to optimality without branching in less than a minute.[5]

An optimal tour for the data above is Berlin, Hamburg, Hannover, Dortmund, Bochum, Wuppertal, Essen, Duisburg, Trier, Koblenz, Frankfurt, Heidelberg, Karlsruhe, Stuttgart, Augsburg, Passau, Bayreuth, Jena, Leipzig, Berlin.

6.3 The capacitated facility location problem

Here we give a formulation of the *capacitated facility location* problem. It may also be considered as a kind of *bin packing* problem with packing costs and variable sized bins, or as a *cutting stock* problem with cutting costs.

Given a set of possible plants P to build, and a set of stores S with a certain demand δ_s that has to be satisfied, we have to decide which plant should serve which store. We have costs c_p for building plant p and c_{ps} for transporting the goods from plant p to store s. Each plant has only a limited capacity κ_p. We insist that each store is served by exactly one plant. Of course we are looking for the cheapest solution:

$$\min \sum_{p\in P} c_p z_p + \sum_{p\in P, s\in S} c_{ps} x_{ps} \qquad \text{subject to}$$

$$\sum_{p\in P} x_{ps} = 1 \qquad \text{for all } s \in S \tag{2}$$

$$x_{ps} \leqslant z_p \qquad \text{for all } s \in S, p \in P \tag{3}$$

$$\sum_{s\in S} \delta_s x_{ps} \leqslant \kappa_p \qquad \text{for all } p \in P \tag{4}$$

$$x_{ps}, z_p \in \{0,1\} \qquad \text{for all } p \in P, s \in S$$

We use binary variables z_p which are set to one, if and only if plant p is to be built. Additionally, we have binary variables x_{ps} which are set to one if and only if plant p serves shop s. Equation (2) demands that each store is assigned to exactly one plant. Inequality (3) makes sure that a plant that serves a shop is built. Inequality (4) assures that the shops are served by a plant which does not exceed its capacity. Putting this into ZIMPL yields the program shown on the next page. The optimal solution for the instance described by the program is to build plants `A` and `C`. Stores 2, 3, and 4 are served by plant `A` and the others by plant `C`. The total cost is 1457.

[5] Only 40 simplex iterations are needed to reach the optimal solution.

```
set PLANTS := { "A", "B", "C", "D" };
set STORES := { 1 .. 9 };
set PS     := PLANTS * STORES;

# How much does it cost to build a plant and what capacity
# will it then have?
param building[PLANTS]:= <"A"> 500, <"B"> 600, <"C"> 700, <"D"> 800;
param capacity[PLANTS]:= <"A">  40, <"B">  55, <"C">  73, <"D">  90;

# The demand for each store
param demand  [STORES]:= <1> 10, <2> 14,
                         <3> 17, <4>  8,
                         <5>  9, <6> 12,
                         <7> 11, <8> 15,
                         <9> 16;

# Transportation cost from each plant to each store
param transport[PS] :=
      |  1,  2,  3,  4,  5,  6,  7,  8,  9 |
  |"A"| 55,  4, 17, 33, 47, 98, 19, 10,  6 |
  |"B"| 42, 12,  4, 23, 16, 78, 47,  9, 82 |
  |"C"| 17, 34, 65, 25,  7, 67, 45, 13, 54 |
  |"D"| 60,  8, 79, 24, 28, 19, 62, 18, 45 |;

var x[PS]     binary;  # Is plant p supplying store s ?
var z[PLANTS] binary;  # Is plant p built ?

# We want it cheap
minimize cost: sum <p> in PLANTS : building[p] * z[p]
             + sum <p,s> in PS : transport[p,s] * x[p,s];

# Each store is supplied by exactly one plant
subto assign:
  forall <s> in STORES do
    sum <p> in PLANTS : x[p,s] == 1;

# To be able to supply a store, a plant must be built
subto build:
   forall <p,s> in PS do
     x[p,s] <= z[p];

# The plant must be able to meet the demands from all stores
# that are assigned to it
subto limit:
   forall <p> in PLANTS do
     sum <s> in S : demand[s] * x[p,s] <= capacity[p];
```

6.4 The n-queens problem

The problem is to place n queens on a $n \times n$ chessboard so that no two queens are on the same row, column or diagonal. The n-queens problem is a classic combinatorial search problem often used to test the performance of algorithms that solve satisfiability problems. Note though, that there are algorithms available which need linear time in practice, like, for example, those of [SG91]. We will show four different models for the problem and compare their performance.

The integer model

The first formulation uses one general integer variable for each row of the board. Each variable can assume the value of a column, i. e., we have n variables with bounds $1 \ldots n$. Next, we use the `vabs` extended function to model an *all different* constraint on the variables (see constraint c1). This makes sure that no queen is located in the same column than any other queen. The second constraint (c2) is used to block all the diagonals of a queen by demanding that the absolute value of the row distance and the column distance of each pair of queens are different. We model $a \neq b$ by $\text{abs}(a-b) \geqslant 1$.

Note that this formulation only works if a queen can be placed in each row, i. e., if the size of the board is at least 4×4.

```
param queens := 8;

set C := { 1 .. queens };
set P := { <i,j> in C * C with i < j };

var x[C] integer >= 1 <= queens;

subto c1: forall <i,j> in P do vabs(x[i] - x[j]) >= 1;
subto c2: forall <i,j> in P do
             vabs(vabs(x[i] - x[j]) - abs(i - j)) >= 1;
```

The following table shows the performance of the model. Since the problem is modeled as a pure satisfiability problem, the solution time depends only on how long it takes to find a feasible solution.[6] The columns titled *Vars*, *Cons*, and *NZ* denote the number of variables, constraints and non-zero entries in the constraint matrix of the generated integer program. *Nodes* lists the number of branch-and-bound nodes evaluated by the solver, and *time* gives the solution time in CPU seconds.

Queens	Vars	Cons	NZ	Nodes	Time [s]
8	344	392	951	1,324	<1
12	804	924	2,243	122,394	120
16	1,456	1,680	4,079	>1 mill.	>1,700

As we can see, between 12 and 16 queens is the maximum instance size we can expect to solve with this model. Neither changing the CPLEX parameters to aggressive cut generation nor setting emphasis on integer feasibility improves the performance significantly.

[6] Which is, in fact, rather random.

The binary models

Another approach to model the problem is to have one binary variable for each square of the board. The variable is one if and only if a queen is in this square and we maximize the number of queens on the board.

For each square we compute in advance which other squares are blocked if a queen is placed on this particular square. Then the extended `vif` constraint is used to set the variables of the blocked squares to zero if a queen is placed.

```
param columns := 8;

set C   := { 1 .. columns };
set CxC := C * C;

set TABU[<i,j> in CxC] := { <m,n> in CxC with (m != i or n != j)
   and (m == i or n == j or abs(m - i) == abs(n - j)) };

var x[CxC] binary;

maximize queens: sum <i,j> in CxC : x[i,j];

subto c1: forall <i,j> in CxC do vif x[i,j] == 1 then
             sum <m,n> in TABU[i,j] : x[m,n] <= 0 end;
```

Using extended formulations can make the models more comprehensible. For example, replacing constraint c1 in line 13 with an equivalent one that does not use `vif` as shown below, leads to a formulation that is much harder to understand.

```
subto c2: forall <i,j> in CxC do
            card(TABU[i,j]) * x[i,j]
          + sum <m,n> in TABU[i,j] : x[m,n] <= card(TABU[i,j]);
```

After the application of the CPLEX presolve procedure both formulations result in identical integer programs. The performance of the model is shown in the following table. *S* indicates the CPLEX settings used: Either *(D)efault*, *(C)uts*[7], or *(F)easibility*[8]. *Root Node* indicates the objective function value of the LP relaxation of the root node.

Queens	S	Vars	Cons	NZ	Root Node	Nodes	Time [s]
8	D	384	448	2,352	13.4301	241	<1
	C				8.0000	0	<1
12	D	864	1,008	7,208	23.4463	20,911	4
	C				12.0000	0	<1
16	D	1,536	1,792	16,224	35.1807	281,030	1,662
	C				16.0000	54	8
24	C	3,456	4,032	51,856	24.0000	38	42
32	C	6,144	7,168	119,488	56.4756	>5,500	>2,000

This approach solves instances with more than 24 queens. The use of aggressive cut generation improves the upper bound on the objective function significantly, though it can be observed that for values of `n` larger than 24 CPLEX is not able to deduce the

[7]Cuts: `mip cuts all 2` and `mip strategy probing 3`.
[8]Feasibility: `mip cuts all -1` and `mip emph 1`

trivial upper bound of n.[9] If we use the following formulation instead of constraint c2, this changes:

```
subto c3: forall <i,j> in CxC do
            forall <m,n> in TABU[i,j] do x[i,j] + x[m,n] <= 1;
```

As shown in the table below, the optimal upper bound on the objective function is always found in the root node. This leads to a similar situation as in the integer formulation, i. e., the solution time depends mainly on the time it needs to find the optimal solution. While reducing the number of branch-and-bound nodes evaluated, aggressive cut generation increases the total solution time.

With this approach instances, up to 96 queens can be solved. At this point, the integer program gets too large to be generated. Even though the CPLEX presolve routine is able to aggregate the constraints again, ZIMPL needs too much memory to generate the IP. The column labeled *Pres. NZ* lists the number of non-zero entries after the presolve procedure.

Queens	S	Vars	Cons	NZ	Pres. NZ	Root Node	Nodes	Time [s]
16	D	256	12,640	25,280	1,594	16.0	0	<1
32	D	1,024	105,152	210,304	6,060	32.0	58	5
64	D	4,096	857,472	1,714,944	23,970	64.0	110	60
64	C					64.0	30	89
96	D	9,216	2,912,320	5,824,640	53,829	96.0	70	193
96	C					96.0	30	410
96	F					96.0	69	66

Finally, we will try the following set packing formulation:

```
subto row: forall <i> in C do
   sum <i,j> in CxC : x[i,j] <= 1;

subto col: forall <j> in C do
   sum <i,j> in CxC : x[i,j] <= 1;

subto diag_row_do: forall <i> in C do
   sum <m,n> in CxC with m - i == n - 1: x[m,n] <= 1;

subto diag_row_up: forall <i> in C do
   sum <m,n> in CxC with m - i == 1 - n: x[m,n] <= 1;

subto diag_col_do: forall <j> in C do
   sum <m,n> in CxC with m - 1 == n - j: x[m,n] <= 1;

subto diag_col_up: forall <j> in C do
   sum <m,n> in CxC with card(C) - m == n - j: x[m,n] <= 1;
```

Here again, the upper bound on the objective function is always optimal. The size of the generated IP is even smaller than that of the former model after presolve. The results for different instances size are shown in the following table:

[9] For the 32 queens instance the optimal solution is found after 800 nodes, but the upper bound is still 56.1678.

Queens	S	Vars	Cons	NZ	Root Node	Nodes	Time [s]
64	D	4,096	384	16,512	64.0	0	<1
96	D	9,216	576	37,056	96.0	1680	331
96	C				96.0	1200	338
96	F				96.0	121	15
128	D	16,384	768	65,792	128.0	>7000	>3600
128	F				128.0	309	90

In case of the 128 queens instance with default settings, a solution with 127 queens is found after 90 branch-and-bound nodes, but CPLEX was not able to find the optimal solution within an hour. From the performance of the Feasible setting, it can be presumed that generating cuts is not beneficial for this model.

7 Error messages

Here is a (hopefully) complete list of the incomprehensible error messages ZIMPL can produce:

101 Bad filename
The name given with the `-o` option is either missing, a directory name, or starts with a dot.

102 File write error
Some error occurred when writing to an output file. A description of the error follows on the next line. For the meaning consult your OS documentation.

103 Output format not supported, using LP format
You tried to select another format than `lp`, `mps`, `hum`, `rlp`, or `pip`.

104 File open failed
Some error occurred when trying to open a file for writing. A description of the error follows on the next line. For the meaning consult your OS documentation.

105 Duplicate constraint name "xxx"
Two `subto` statements have the same name.

106 Empty LHS, constraint trivially violated
One side of your constraint is empty and the other not equal to zero. Most frequently this happens, when a set to be summed up is empty.

107 Range must be $l \leqslant x \leqslant u$, or $u \geqslant x \geqslant l$
If you specify a range you must have the same comparison operators on both sides.

108 Empty Term with nonempty LHS/RHS, constraint trivially violated
The middle of your constraint is empty and either the left- or right-hand side of the range is not zero. This most frequently happens, when a set to be summed up is empty.

109 LHS/RHS contradiction, constraint trivially violated
The lower side of your range is bigger than the upper side, e.g. $15 \leqslant x \leqslant 2$.

110 Division by zero
You tried to divide by zero. This is not a good idea.

111 Modulo by zero
You tried to compute a number modulo zero. This does not work well.

112 Exponent value xxx is too big or not an integer
It is only allowed to raise a number to the power of integers. Also trying to raise a number to the power of more than two billion is prohibited.[10]

113 Factorial value xxx is too big or not an integer
You can only compute the factorial of integers. Also computing the factorial of a number bigger then two billion is generally a bad idea. See also Error 115.

114 Negative factorial value
To compute the factorial of a number it has to be positive. In case you need it for a negative number, remember that for all even numbers the outcome will be positive and for all odd number negative.

[10]The behavior of this operation could easily be implemented as `for(;;)` or in a more elaborate way as `void f(){f();}`.

115 Timeout!
You tried to compute a number bigger than 1000!. See also the footnote to Error 112.

116 Illegal value type in min: `xxx` only numbers are possible
You tried to build the minimum of some strings.

117 Illegal value type in max: `xxx` only numbers are possible
You tried to build the maximum of some strings.

118 Comparison of different types
You tried to compare apples with oranges, i.e, numbers with strings. Note that the use of an undefined parameter can also lead to this message.

119 `xxx` of sets with different dimension
To apply Operation `xxx` (union, minus, intersection, symmetric difference) on two sets, both must have the same dimension tuples,i. e., the tuples must have the same number of components.

120 Minus of incompatible sets
To apply Operation `xxx` (union, minus, intersection, symmetric difference) on two sets, both must have tuples of the same type,i. e., the components of the tuples must have the same type (number, string).

121 Negative exponent on variable
The exponent to a variable was negative. This is not supported.

123 "from" value `xxx` is too big or not an integer
To generate a set, the "from" number must be an integer with an absolute value of less than two billion.

124 "upto" value `xxx` is too big or not an integer
To generate a set, the "upto" number must be an integer with an absolute value of less than two billion.

125 "step" value `xxx` is too big or not an integer
To generate a set, the "step" number must be an integer with an absolute value of less than two billion.

126 Zero "step" value in range
The given "step" value for the generation of a set is zero. So the "upto" value can never be reached.

127 Illegal value type in tuple: `xxx` only numbers are possible
The selection tuple in a call to the `proj` function can only contain numbers.

128 Index value `xxx` in proj too big or not an integer
The value given in a selection tuple of a `proj` function is not an integer or bigger than two billion.

129 Illegal index `xxx`, set has only dimension yyy
The index value given in a selection tuple is bigger than the dimension of the tuples in the set.

131 Illegal element `xxx` for symbol
The index tuple used in the initialization list of a index set, is not member of the index set of the set. E.g, `set A[{ 1 to 5 }] := <1> { 1 }, <6> { 2 };`

132 Values in parameter list missing, probably wrong read template
Probably the template of a read statement looks like `"<1n>"` only having a tuple, instead of `"<1n> 2n"`.

133 Unknown symbol `xxx`
A name was used that is not defined anywhere in scope.

134 Illegal element `xxx` for symbol
The index tuple given in the initialization is not member of the index set of the parameter.

135 Index set for parameter `xxx` is empty
The attempt was made to declare an indexed parameter with the empty set as index set. Most likely the index set has a `with` clause which has rejected all elements.

136 Permutation on non one-dimensional set
Only sets with one dimension elements can be permuted.

137 Set to permute is empty
The set to be permuted is empty.

138 Set size `xxx` > 12 is too big
Since the resulting set will have number of elements of input set factorial elements, a set size of 12 is the maximum allowed.

139 Lower bound for integral var `xxx` truncated to `yyy` (warning)
An integral variable can only have an integral bound. So the given non integral bound was adjusted.

140 Upper bound for integral var `xxx` truncated to `yyy` (warning)
An integral variable can only have an integral bound. So the given non integral bound was adjusted.

141 Infeasible due to conflicting bounds for var `xxx`
The upper bound given for a variable was smaller than the lower bound.

142 Unknown index `xxx` for symbol `yyy`
The index tuple given is not member of the index set of the symbol.

143 Size for subsets `xxx` is too big or not an integer
The cardinality for the subsets to generate must be given as an integer smaller than two billion.

144 Tried to build subsets of empty set
The set given to build the subsets of, was the empty set.

145 Illegal size for subsets `xxx`, should be between 1 and `yyy`
The cardinality for the subsets to generate must be between 1 and the cardinality of the base set.

146 Tried to build powerset of empty set
The set given to build the powerset of, was the empty set.

147 use value `xxx` is too big or not an integer
The use value must be given as an integer smaller than two billion.

148 use value `xxx` is not positive
Negative or zero values for the use parameter are not allowed.

149 skip value `xxx` is too big or not an integer
The skip value must be given as an integer smaller than two billion.

150 skip value `xxx` is not positive
Negative or zero values for the skip parameter are not allowed.

151 Not a valid read template
A read template must look something like `"<1n,2n>"`. There have to be a < and a > in this order.

152 Invalid read template syntax
Apart from any delimiters like <, >, and commas a template must consists of number character pairs like `1n`, `3s`.

153 Invalid field number `xxx`
The field numbers in a template have to be between 1 and 255.

154 Invalid field type `xxx`
The only possible field types are `n` and `s`.

155 Invalid read template, not enough fields
There has to be at least one field inside the delimiters.

156 Not enough fields in data
The template specified a field number that is higher than the actual number of field found in the data.

157 Not enough fields in data (value)
The template specified a field number that is higher than the actual number of field found in the data. The error occurred after the index tuple in the value field.

159 Type error, expected `xxx` **got** `yyy`
The type found was not the expected one, e.g. subtracting a string from a number would result in this message.

160 Comparison of elements with different types `xxx` / `yyy`
Two elements from different tuples were compared and found to be of different types.

161 Line `xxx`**: Unterminated string**
This line has an odd number of `"` characters. A String was started, but not ended.

162 Line `xxx`**: Trailing** `"yyy"` **ignored** (warning)
Something was found after the last semicolon in the file.

163 Line `xxx`**: Syntax Error**
A new statement was not started with one of the keywords: `set`, `param`, `var`, `minimize`, `maximize`, `subto`, or `do`.

164 Duplicate element `xxx` **for set rejected** (warning)
An element was added to a set that was already in it.

165 Comparison of different dimension sets (warning)
Two sets were compared, but have different dimension tuples. (This means they never had a chance to be equal, other than being empty sets.)

166 Duplicate element `xxx` **for symbol** `yyy` **rejected** (warning)
An element that was already there was added to a symbol.

167 Comparison of different dimension tuples (warning)
Two tuples with different dimensions were compared.

168 No program statements to execute
No ZIMPL statements were found in the files loaded.

169 Execute must return void element
This should not happen. If you encounter this error please email the `.zpl` file to `mailto:koch@zib.de`.

170 Uninitialized local parameter `xxx` in call of define `yyy`
A define was called and one of the arguments was a "name" (of a variable) for which no value was defined.

171 Wrong number of arguments (`xxx` instead of `yyy`) for call of define `zzz`
A define was called with a different number of arguments than in its definition.

172 Wrong number of entries (`xxx`) in table line, expected `yyy` entries
Each line of a parameter initialization table must have exactly the same number of entries as the index (first) line of the table.

173 Illegal type in element `xxx` for symbol
A parameter can only have a single value type. Either numbers or strings. In the initialization both types were present.

174 Numeric field `xxx` read as `"yyy"`. This is not a number
It was tried to read a field with an 'n' designation in the template, but what was read is not a valid number.

175 Illegal syntax for command line define `"xxx"` – ignored (warning)
A parameter definition using the command line `-D` flag, must have the form `name=value`. The `name` must be a legal identifier, i. e., it has to start with a letter and may consist only out of letters and numbers including the underscore.

176 Empty LHS, in Boolean constraint (warning)
The left hand side, i. e., the term with the variables, is empty.

177 Boolean constraint not all integer
No continuous (real) variables are allowed in a Boolean constraint.

178 Conditional always true or false due to bounds (warning)
All or part of a Boolean constraint are always either true or false, due to the bounds of variables.

179 Conditional only possible on bounded constraints
A Boolean constraint has at least one variable without finite bounds.

180 Conditional constraint always true due to bounds (warning)
The result part of a conditional constraint is always true anyway. This is due to the bounds of the variables involved.

181 Empty LHS, not allowed in conditional constraint
The result part of a conditional constraint may not be empty.

182 Empty LHS, in variable vabs
There are no variables in the argument to a `vabs` function. Either everything is zero, or just use `abs`.

183 vabs term not all integer
There are non integer variables in the argument to a `vabs` function. Due to numerical reasons continuous variables are not allowed as arguments to `vabs`.

184 vabs term not bounded
The term inside a `vabs` has at least one unbounded variable.

185 Term in Boolean constraint not bounded
The term inside a `vif` has at least one unbounded variable.

186 Minimizing over empty set – zero assumed (warning)
The index expression for the minimization was empty. The result used for this expression was zero.

187 Maximizing over empty set – zero assumed (warning)
The index expression for the maximization was empty. The result used for this expression was zero.

188 Index tuple has wrong dimension
The number of elements in an index tuple is different from the dimension of the tuples in the set that is indexed. This might occur, when the index tuple of an entry in a parameter initialization list has not the same dimension as the indexing set of the parameter. This might occur, when the index tuple of an entry in a set initialization list has not the same dimension as the indexing set of the set. If you use a `powerset` or `subset` instruction, the index set has to be one dimension.

189 Tuple number `xxx` is too big or not an integer
The tuple number must be given as an integer smaller than two billion.

190 Component number `xxx` is too big or not an integer
The component number must be given as an integer smaller than two billion.

191 Tuple number `xxx` is not a valid value between 1..`yyy`
The tuple number must be between one and the cardinality of the set.

192 Component number `xxx` is not a valid value between 1..`yyy`
The component number must be between one and the dimension of the set.

193 Different dimension tuples in set initialization
The tuples that should be part of the list have different dimension.

195 Genuine empty set as index set (warning)
The set of an index set is always the empty set.

197 Empty index set for set
The index set for a set is empty.

198 Incompatible index tuple
The index tuple given had fixed components. The type of such a component was not the same as the type of the same component of tuples from the set.

199 Constants are not allowed in SOS declarations
When declaring an SOS, weights are only allowed together with variables. A weight alone does not make sense.

200 Weights are not unique for SOS `xxx` (warning)
All weights assigned to variables in an special ordered set have to be unique.

201 Invalid read template, only one field allowed
When reading a single parameter value, the read template must consist of a single field specification.

202 Indexing over empty set (warning)
The indexing set turns out to be empty.

203 Indexing tuple is fixed (warning)
The indexing tuple of an index expression is completely fixed. As a result only this one element will be searched for.

204 Random function parameter minimum= xxx >= maximum= yyy
The second parameter to the function `random` has to be strictly greater than the first parameter.

205 xxx excess entries for symbol yyy ignored (warning)
When reading the data for symbol `yyy` there were `xxx` more entries in the file than indices for the symbol. The excess entries were ignored.

206 argmin/argmax over empty set (warning)
The index expression for the `argmin` or `argmax` was empty. The result is the empty set.

207 "size" value xxx is too big or not an integer
The size argument for an `argmin` or `argmax` function must be an integer with an absolute value of less than two billion.

208 "size" value xxx not >= 1
The size argument for an `argmin` or `argmax` function must be at least one, since it represents the maximum cardinality of the resulting set.

209 MIN of set with more than one dimension
The expressions `min(A)` is only allowed if the elements of set A consist of 1-tuples containing numbers.

210 MAX of set with more than one dimension
The expressions `max(A)` is only allowed if the elements of set A consist of 1-tuples containing numbers.

211 MIN of set containing non number elements
The expressions `min(A)` is only allowed if the elements of set A consist of 1-tuples containing numbers.

212 MAX of set containing non number elements
The expressions `max(A)` is only allowed if the elements of set A consist of 1-tuples containing numbers.

213 More than 65535 input fields in line xxx of yyy (warning)
Input data beyond field number 65535 in line `xxx` of file `yyy` are ignored. Insert some newlines into your data!

214 Wrong type of set elements – wrong read template?
Most likely you have tried read in a set from a stream using `"n+"` instead of `"<n+>"` in the template.

215 Startvals violate constraint, ... (warning)
If the given startvals are summed up, they violate the constraint. Details about the sum of the LHS and the RHS are given in the message.

216 Redefinition of parameter xxx ignored
A parameter was declared a second time with the same name. The typical use would be to declare default values for a parameter in the ZIMPL file and override them by command-line defined.

217 begin value xxx in substr too big or not an integer
The begin argument for an `substr` function must be an integer with an absolute value of less than two billion.

218 length value xxx in substr too big or not an integer
The length argument for an `substr` function must be an integer with an absolute value of less than two billion.

219 length value `xxx` in substr is negative
The length argument for an `substr` function must be greater or equal to zero.

220 Illegal size for subsets `xxx`, should be between `yyy` and `zzz`
The cardinality of the subsets to generate must be between the given lower bound and the cardinality of the base set.

222 Term inside a then or else constraint not linear
The term inside a `then` or `else` is not linear.

223 Objective function `xxx` overwrites existing one (warning)
Another objective function declaration has been encountered, superceeding the previous one.

301 variable priority has to be integral (warning)
If branching priorities for variables are given, these have to be integral.

302 SOS priority has to be integral (warning)
If SOS priorities are given, these have to be integral.

401 Slack too large (`xxx`) for QUBO conversion The upper bound of the slack needed for this constraint is more then 2^{30}.

403 Non linear term can't be converted to QUBO
Only linear terms can be automatically converted into a QUBO.

404 Non linear expressions can't be converted to QUBO
Only linear expressions without indicator constraints can be automatically converted into a QUBO.

600 File format can only handle linear and quadratic constraints (warning)

The chosen file format can currently only handle linear and quadratic constraints. Higher degree constraints were ignored.

601 File format can only handle binary variables (warning)]
The chosen file format can only handle binary variables. Variables of other types will be ignored.

602 QUBO file format can only handle linear and quadratic term (warning)

By definition the QUBO file format can only handle linear and quadratic terms. No file can be written.

700 log(): `OS specific domain or range error message`
Function `log` was called with a zero or negative argument, or the argument was too small to be represented as a `double`.

701 sqrt(): `OS specific domain error message`
Function `sqrt` was called with a negative argument.

702 ln(): `OS specific domain or range error message`
Function `ln` was called with a zero or negative argument, or the argument was too small to be represented as a `double`.

703 exp(): `OS specific domain or range error message`
Function `exp` was called with the magnitude of the argument too large.

704 asin(): `OS specific domain error message`
Function `asin` was called with an argument outside the domain [-1,1].

705 acos(): `OS specific domain error message`
Function `acos` was called with an argument outside the domain [-1,1].

706 atan(): `OS specific domain or range error message`
Function `atan` was called with an argument causing overflow.

800 parse error: expecting `xxx` (or `yyy`)
Parsing error. What was found was not what was expected. The statement you entered is not valid.

801 Parser failed
The parsing routine failed. This should not happen. If you encounter this error, please email the `.zpl` file to `mailto:koch@zib.de`.

802 Regular expression error
A regular expression given to the `match` parameter of a `read` statement was not valid. See error messages for details.

803 String too long `xxx` > `yyy`
The program encountered a string which is larger than 1 GB.

900 Check failed!
A `check` instruction did not evaluate to true.

References

[Chv83] Vašek Chvátal. *Linear Programming*. H.W. Freeman and Company, New York, 1983.

[Dan90] Georg B. Dantzig. The diet problem. *Interfaces*, 20:43–47, 1990.

[FGK03] R. Fourier, D. M. Gay, and B. W. Kernighan. *AMPL: A Modelling Language for Mathematical Programming*. Brooks/Cole—Thomson Learning, 2nd edition, 2003.

[GNU03] GNU multiple precision arithmetic library (GMP), version 4.1.2., 2003. Code and documentation available at `http://gmplib.org`.

[IBM97] IBM optimization library guide and reference, 1997.

[ILO02] ILOG CPLEX Division, 889 Alder Avenue, Suite 200, Incline Village, NV 89451, USA. *ILOG CPLEX 8.0 Reference Manual*, 2002. Information available at `http://www.ilog.com/products/cplex`.

[Kal04a] Josef Kallrath. Mathematical optimization and the role of modeling languages. In Josef Kallrath, editor, *Modeling Languages in Mathematical Optimization*, pages 3–24. Kluwer, 2004.

[Kal04b] Josef Kallrath, editor. *Modeling Languages in Mathematical Optimization*. Kluwer, 2004.

[Koc04] Thorsten Koch. *Rapid Mathematical Programming*. PhD thesis, Technische Universität Berlin, 2004.

[Sch03] Alexander Schrijver. *Combinatorial Optimization*. Springer, 2003.

[Sch04] Hermann Schichl. Models and the history of modeling. In Josef Kallrath, editor, *Modeling Languages in Mathematical Optimization*, pages 25–36. Kluwer, 2004.

[SG91] Rok Sosič and Jun Gu. 3,000,000 million queens in less than a minute. *SIGART Bulletin*, 2(2):22–24, 1991.

[Spi04] Kurt Spielberg. The optimization systems MPSX and OSL. In Josef Kallrath, editor, *Modeling Languages in Mathematical Optimization*, pages 267–278. Kluwer, 2004.

[vH99] Pascal van Hentenryck. *The OPL Optimization Programming Language*. MIT Press, Cambridge, Massachusetts, 1999.

[XPR99] *XPRESS-MP Release 11 Reference Manual*. Dash Associates, 1999. Information available at `http://www.dashoptimization.com`.