

Getting Started with MINION

Ian P. Gent,
Christopher A. Jefferson,
Ian Miguel
Karen E. Petrie and
Andrea M. Rendl

MINION Version 0.4
MINION Input Language 1

February 19, 2007

Contents

1	Overview	1
2	Obtaining and Installing MINION	1
3	Matrix Models in MINION	2
4	Variables	3
5	Constraints	3
5.1	Reification	4
5.2	Reifiable constraints	4
5.3	Non-reifiable constraints	5
6	Input format	7
6.1	Notation	7
6.2	Model	7
6.3	Declarations	8
6.4	Constraints	9
7	Important Notes and Changes	10
8	Example	11

1 Overview

This manual introduces MINION, a general-purpose solver for CSP/COP instances. It will cover installation issues (chapter 2) and give detailed information on how to model problems demonstrated by an example (chapter 3, 8). The variable types and available constraints are presented in chapter 4 and 5, respectively. The BNF of the input language can be found in chapter 6. Important notes (containing bugs) about the current version are given in chapter 7 and updated on a regularly basis.

2 Obtaining and Installing MINION

MINION can be obtained from:

- <http://sourceforge.net/projects/minion>

To compile MINION:

On Linux: Make sure g++ is installed, type `./build-all.sh`

On Mac: Install the latest version of xcode, type `./build-all.sh`

On Windows: Install cygwin with g++, type `./build-all.sh`

Warning: Compilation can take around 30 minutes. It requires g++ version 3.2 or above (type `g++ -v` to check version).

In the `bin` directory, there are two executables:

minion-debug Debugging version. Does a range of extra consistency checks and can be run through a debugger.

minion Optimised version.

Command-line arguments: the filename must appear last, but the other arguments may appear in any order. Behaviour if contradictory arguments are given (e.g. quiet and verbose) is not defined. When run with no arguments, a brief help message is displayed.

- `[-findallsols]`. Find all solutions and count them. This option is ignored if the problem contains any minimising or maximising objective.
- `[-timelimit] N`. Stop after N seconds.
- `[-sollimit] N`. Stop after finding N solutions.
- `[-nodelimit] N`. Stop after N nodes are searched.
- `[-quiet]`. Switch off output from instance parser. (Default depends on compile-time options).
- `[-verbose]`. Switch on output from instance parser. (Default again depends on compile-time options.)
- `[-printsols]`. Print each solution when it is found, including each improved solution when optimising. This is the default.
- `[-noprimsols]`. Do not print solutions.
- `[-dumptree]`. Output the search tree. If you want details on how to use this, or if you want more information, please ask.

- [-test]. A test option for checking and regression testing. Example test instances which this can be run on are in the directory `test_instances`.
- [-fullprop]. Disable incremental propagation. This should always slow down search while producing exactly the same search tree.
- [-varorder = ORDER]. Enables variable order ORDER. Please note that this flag is still in an experimental state, so MINION's default variable ordering might be faster. ORDER can be any of the following :
 - [sdf]. Smallest domain first, then break ties with lex
 - [sdf-random]. Smallest domain first, break ties randomly
 - [ldf]. Largest domain first
 - [ldf-random]. Largest domain first, break ties randomly
 - [random]. Random variable ordering
 - [static]. Normal ordering without caching
- [-sac-root]. Perform SGAC at the first node.
- [-ssac-root]. Perform SSGAC at the first node.
- filename.

3 Matrix Models in MINION

MINION is a general-purpose solver for CSP/COP *instances*, with an expressive input language based on the common constraint modelling device of matrix models. In this context a matrix is a n -dimensional object, which can be used to store CSP variables, matrices allow ease of reference to these variables. In CSP a matrix formulation employs one or more matrices of decision variables, with constraints typically imposed on the rows, columns and planes of the matrices.

To illustrate, consider the *Balanced Incomplete Block Design* (BIBD, CSPLib problem 28), which is defined as follows: Given a 5-tuple of positive integers, $\langle v, b, r, k, \lambda \rangle$, assign each of v objects to b blocks such that each block contains k distinct objects, each object occurs in exactly r different blocks and every two distinct objects occur together in exactly λ blocks. Despite its simplicity, the BIBD has important practical applications, such as cryptography and experimental design.

The matrix model for BIBD has b columns and v rows of 0/1 decision variables. A '1' entry in row i , column j represents the decision to assign the i th object to the j th block. Each row is constrained to sum to r , each column is constrained to sum to k and the scalar product of each pair of rows is constrained to equal λ . A solution to the instance $\langle 7, 7, 3, 3, 1 \rangle$ is given below:

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Matrix models such as this have been identified as a very common pattern in constraint modelling and support, for example, the straightforward modelling of problems that involve

finding a function or relation — indeed, one can view the BIBD as finding a relation between objects and blocks.

MINION's input language supports the definition of one, two, and three-dimensional matrices of decision variables (higher dimensions can easily be created by using multiple matrices of smaller dimension). Furthermore, it provides direct access to matrix rows and columns in recognition of the fact that most matrix models impose constraints on them.

By focusing on matrix models MINION is a lean, highly-optimised constraint programming solver.

4 Variables

MINION supports 5 variable types. These are:

1. *0/1* variables, which are used very commonly for logical expressions, and for encoding the characteristic functions of sets and relations. Note that wherever a 01 variable can appear, the negation of that variable can also appear. For instance, the first Boolean variable (if any) is always x_0 . Its negation is identified by $\neg x_0$.
2. *Bounds* variables, where only the upper and lower bounds of the domain are maintained. These domains must be continuous ranges of integers i.e. holes can not be put in the domains of the variables.
3. *Sparse Bounds* variables, where the domain is composed of discrete values (e.g. $\{1, 5, 36, 92\}$), but only the upper and lower bounds of the domain may be updated during search. Although the domain of these variables is not a continuous range, any holes in the domains must be there at time of specification, as they can not be added during the solving process.
4. *Discrete* variables, where the domain ranges from the lower bound to the upper bound specified, but the deletion of any domain element in this range is permitted. This means that holes can be put in the domain of these variables.
5. *Discrete Sparse* variables, where the domain is composed of discrete values, and any domain element may be removed. This is the most general variable type, it allows any integer value to be in the domain at specification time, and it allows any variable to be removed during the search process. *Unfortunately this type of variable is not yet implemented.*

Sub-dividing the variable types in this manner affords the greatest opportunity for optimisation. In general, we recommend thinking of the variable types as a hierarchy, where 1 (0/1 variables) is the most efficient type, and 5 (Discrete Sparse Variables) is the least. The user should use the variable which is the highest in the hierarchy, yet encompasses enough information to provide a full model for the problem they are attempting to solve.

5 Constraints

MINION supports the following constraints. Note that it does NOT support nesting of constraints. In all cases, a variable may be replaced with a constant. MINION supports a variety of expressions on matrices (e.g. row, and column), and will automatically flatten matrices of higher arity. See BNF in next section for details.

5.1 Reification

There are two different constraints imposing reification.

Reify states that the assignment of a 0/1 variable is 1 iff the constraint is satisfied:
`reify(<constraint>, <01var>).`

Guide states that the reified constraint is entailed if the assignment to the 0/1 variable is 1. If otherwise, there will be no propagation done on the constraint. Please note that in upcoming versions of Minion the guide-constraint will be known as `guide`.
`reifyimply(<constraint>, <01var>).`

5.2 Reifiable constraints

The following list presents constraints that are reifiable - mostly with `reifyimply`.

All-Different states that all the variables in a matrix are assigned different variables/values.
`alldiff(<matrix of variables>).`

\neq states that a variable (`var1`) is not equal to a another variable (`var2`):
`diseq(<var1>, <var2>).`

$=$ states that a variable (`var1`) is equal to another variable (`var2`):
`eq(<var1>, <var2>).`

Element states that the variable at the index of the matrix specified by the assignment to `var1` is equal to the assignment of `var2`:
`element(<matrix of variables>, <var1>, <var2>).`

GACElement states that the variable at the index of the matrix specified by the assignment to `var1` is equal to the assignment of `var2` preserving GAC:
`element(<matrix of variables>, <var1>, <var2>).`
Note that you may find that the constraint `watchelement` works faster: these two constraints should always perform the same propagations as each other.

\leq states that a variable (`var1`) is less than or equal to another variable (`var2`) plus a constant (to obtain $<$ use -1 for the constant):
`ineq(<var1>, <var2>, <const>).`

Lexicographically \leq states that a matrix of variables (`mat1`) is lexicographically less than or equal to another matrix of variables (`mat2`):
`lexleq(<mat1>, <mat2>).`

Lexicographically $<$ states that a matrix of variables (`mat1`) is less than another matrix of variables (`mat2`):
`lexless(<mat1>, <mat2>).`

Maximum states that the maximum assignment among a matrix of variables is equal to the assignment of a variable, (`var`):
`max(<matrix of variables>, <var>).`

Minimum states that the minimum assignment among a matrix of variables is equal to the assignment of a variable, (`var`):
`min(<matrix of variables>, <var>).`

Occurrence states that a given value is assigned to a specified number of variables in a matrix:
`occurrence(<matrix of variables>, <value>, <count>).`

Product states that a given variable (`var1`) multiplied by another variable (`var2`) is equal to a third variable (`var3`):

`Product(<var1>, <var2>, <var3>).`

Sum \geq states that the sum of the variables in a matrix is greater than or equal to the assignment of a single variable:

`sumgeq(<matrix of variables>, <var>).`

Sum \leq states that the sum of the variables in a matrix is less than or equal to the assignment of a single variable:

`sumleq(<matrix of variables>, <var>).`

Note: In the current version of MINION, there is no constraint for Sum=.
This has to be achieved with a Sum \leq and a Sum \geq .

Weighted Sum \leq states that the scalar product of a matrix of variables and a matrix of constants is less than or equal to the assignment of a single variable:

`weightedsumleq(<matrix of constants>, <matrix of variables>, <variable>).`

Weighted Sum \geq states that the scalar product of a matrix of variables and a matrix of constants is greater than or equal to the assignment of a single variable:

`weightedsumgeq(<matrix of constants>, <matrix of variables>, <variable>).`

5.3 Non-reifiable constraints

MinusEq states that a variable equals the negation of another variable.

`minuseq(<variable>, <variable>)`

Element (watched literals) states that the variable at the index of the matrix specified by the assignment to `var1` is equal to the assignment of `var2` using watched literals. Like `gacelement`, this constraint preserves GAC:

`watchelement(<matrix of variables>, <var1>, <var2>).`

Sum = (watched literals) states that the sum of the variables in a matrix is equal to the assignment of a single variable, using watched literals.

`litsumeq(<matrix of variables>, <matrix of constants>)`

Sum \geq (watched literals) states that the sum of the variables in a matrix is greater than or equal to the assignment of a single variable, using watched literals:

`watchsumgeq(<matrix of variables>, <var>).`

Sum \leq (watched literals) states that the sum of the variables in a matrix is less than or equal to the assignment of a single variable, using watched literals:

`watchsumleq(<matrix of variables>, <var>).`

Table allows the specification of an extensional constraint. The set of tuples should be given in strictly increasing lexicographic order.

Note: The implementation has been changed and details can be found in the CPPOD-technical report¹.

`table(<matrix of variables>, <tuples>)`

Vector \neq (watched literals) states that a vector (`vector1`) is not equal to another vector (`vector2`), using watched literals:

`watchvecneq(<vector1>, <vector2>).`

¹<http://www.dcs.st-and.ac.uk/cppod/publications/reports/cppod-19-2006.pdf>

Power the power constraint, with variables in the following order $x^y = z$. It is only available for positive domains.

```
pow( <variable>, <variable>, <variable> )
```


6 Input format

Here are some important notes concerning modelling in the input format.

Variable names variable names start with a specific letter (*x* for simple variables, *v* for 1-dimensional matrices (vectors), *m* for 2-dimensional matrices and *t* for tuples), followed by the order in which they were defined (starting with 0). For example, if there are 4 boolean variables, 5 bounds variables, and 3 1-dimensional matrices(vectors) declared, the first boolean variable has the name *x0* and the second bounds variable has the name *x5* and the third vector the name *v2*.

Constraint order the order in which constraints are stated does not matter

Printing variables the expression to be printed in the `print` statement has to be a defined 1- or 2-dimensional matrix.

Constants a variable can always be substituted by a constant (in constraints)

Variable orders when applying a variable order, the variables defined in the *variableOrder* will be put under the order that was imposed in the *textitvalueOrder* (a for ascending and d for descending). Empty lists [] [] correspond to no ordering.

Comments comments are only allowed in the header

Tuplelists defined tuplelists can be referenced by their name which is composed by *t* followed by the order (beginning from 0) in which they were defined. For example *t2* stands for the third tuple.

6.1 Notation

This is the grammar of the input format of Minion. Expressions written in *italic* font are non-terminals and those written in *typesetter* are terminals. Any expression written in neither of these fonts correspond to integer values whose purpose are described by the expressions. The non-terminal *Variable* stands for a variable name as described above.

- *a** stands for zero or more occurrences of *a*
- *a+* stands for one or more occurrences of *a*
- *a | b* stands for *a* or *b*

6.2 Model

```
MinionInput ::= MinionHeader
              VariableDeclarations
              VariableOrder
              ValueOrder
              MatrixDeclarations
              TupleLists
              objective ObjectiveExpression
              print PrintExpression
              Constraints
```

6.3 Declarations

<i>MinionHeader</i>	::=	MINION inputVersionNumber # <i>comments</i>
<i>VariableDeclarations</i>	::=	amountOf01Variables amountOfBoundsVariables <i>Bounds</i> * amountOfSparseBoundsVariables <i>SparseBounds</i> * amountOfDiscreteVariables <i>Bounds</i> * amountOfSparseDiscreteVariables <i>SparseBounds</i> *
<i>Bounds</i>	::=	lowerBound upperBound amountOfVariables
<i>SparseBounds</i>	::=	{ <i>DomainElements</i> } amountOfVariables
<i>DomainElements</i>	::=	value <i>DomainElements</i> , value
<i>VariableOrder</i>	::=	[<i>VariableList</i>] []
<i>ValueOrder</i>	::=	[<i>OrderList</i>] []
<i>OrderList</i>	::=	<i>Order</i> <i>OrderList</i> , <i>Order</i>
<i>Order</i>	::=	a d
<i>MatrixDeclarations</i>	::=	amountOf1DimensionalMatrices <i>1DimensionalMatrix</i> + amountOf2DimensionalMatrices <i>2DimensionalMatrix</i> + amountOf3DimensionalMatrices <i>3DimensionalMatrix</i> +
<i>TupleListsDefintion</i>	::=	tuplelists amountOfTupleLists <i>TupleDefinitions</i> + ϵ
<i>TupleDefinitions</i>	::=	amountOfTuples tupleLength tupleValues+
<i>ObjectiveExpression</i>	::=	none minimising <i>Variable</i> maximising <i>Variable</i>
<i>PrintExpression</i>	::=	none <i>Defined1DimensionalMatrixName</i> <i>Defined2DimensionalMatrixName</i>
<i>ConstantList</i>	::=	constant <i>ConstantList</i> , constant
<i>VariableList</i>	::=	<i>Variable</i> <i>VariableList</i> , <i>Variable</i>
<i>1DimensionalMatrix</i>	::=	[<i>ConstantList</i>] []
<i>2DimensionalMatrix</i>	::=	[<i>1DimensionalMatrix</i> +] []
<i>3DimensionalMatrix</i>	::=	[<i>2DimensionalMatrix</i> +] []

6.4 Constraints

```

Constraints ::= ReifiableConstraint |
               reify( ReifiableConstraint , Variable ) |
               reifyImplies( ReifiableConstraint , Variable ) |
               table( VariableVector , tuples ) |
               watchsumleq( VariableVector , VariableVector ) |
               watchsumgeq( VariableVector , VariableVector ) |
               watchvecneq( VariableVector , VariableVector ) |
               watchelement( VariableVector , Variable , Variable ) |
               minusequal( Variable , Variable ) |
               litsumequal( VariableVector , [ ConstantList ] ) |
               pow( Variable , Variable , Variable ) |
ReifiableConstraints ::= allDiff( VariableVector ) |
                        eq( Variable , Variable ) |
                        diseq( Variable , Variable ) |
                        ineq( Variable , Variable , constant ) |
                        element( VariableVector , Variable , Variable ) |
                        gacelement( VariableVector , Variable , Variable ) |
                        min( VariableVector , Variable ) |
                        max( VariableVector , Variable ) |
                        lexleq( VariableVector , VariableVector ) |
                        lexless( VariableVector , VariableVector ) |
                        occurence( VariableVector , constant , constant ) |
                        product( VariableVector , Variable ) |
                        product( VariableVector , [ ConstantList ] , Variable ) |
                        sum( VariableVector , Variable ) |
                        weightedsumleq( VariableVector , [ ConstantList ] , Variable ) |
                        weightedsumgeq( VariableVector , [ ConstantList ] , Variable )
VariableVector ::= [ ConstantList ] |
                  [ VariableList ] |
                  1DimensionalMatrix |
                  2DimensionalMatrix |
                  3DimensionalMatrix |
                  row( 2DimensionalMatrix , index ) |
                  col( 2DimensionalMatrix , index ) |
                  rowX( 3DimensionalMatrix , index , index ) |
                  rowY( 3DimensionalMatrix , index , index ) |
                  colX( 3DimensionalMatrix , index , index ) |
                  colY( 3DimensionalMatrix , index , index )

```

7 Important Notes and Changes

Changes

The following changes have been made since the last version:

1. **power constraint:** the `pow` constraint has been added.
2. **table constraint:** there is a new implementation of the `table` constraint where new data structures are used. Details can be found in the CPROD-technical report².
3. **flags for variable ordering and preprocessing** Some flags have been added for preprocessing and variable ordering. Please note, that these flags are still in an experimental stage and are not optimised yet, so the default settings might solve your problems faster.

- `-varorder = <order>`

MINION's normal variable ordering uses information from previous search nodes to avoid rechecking the same variable at later search nodes. The variable orderings generated by `varorder` do not do this, so are likely to be much slower for large numbers of variables. This effect can be measured using the `static` `varorder`, which is the same as the normal ordering except not using state from previous nodes. `<order>` can be:

<code>sdf</code>	smallest domain first, then break ties with lex
<code>sdf-random</code>	smallest domain first, break ties randomly
<code>ldf</code>	Largest domain first
<code>ldf-random</code>	largest domain first, break ties randomly
<code>random</code>	random variable ordering
<code>static</code>	normal ordering without caching

- `-sac-root`
Perform SGAC at the first node.
- `-ssac-root`
Perform SSGAC at the first node.

Notes

1. **Sparse discrete variables** are not implemented yet.
2. **Reification using `reify`**
There are problems with the reification of the constraints: `diseq`, `element`, `gaclement`, `ineq`, `max`, `min`, `occurence`, `product`

²<http://www.dcs.st-and.ac.uk/cpro/publications/reports/cpro-19-2006.pdf>

8 Example

In Section 3 we introduced, the $\langle 7, 7, 3, 3, 1 \rangle$ BIBD instance. In this section we give a Minion specification for this problem. All the text in *italics* are added for explanation, and are not part of the actual specification.

Note that the input starts with the fixed line “MINION 1”. This is the version number of the *input format* and not of the version of MINION. This means that a given version of MINION itself can immediately reject input if in a format intended for a later version. We intend that the counter 1 will be incremented when a new format not readable is introduced, e.g. when a new constraint is introduced into the system. Normally, after such a change, MINION will be able to read in instances defined under older versions, though this is not guaranteed.

```
MINION 1
# Comments appear in consecutive lines after the first, starting
# with # character. Each comment line starts with a #
# You can have as many as you like (including 0)
# But as soon as a line does not start with #, no more comments
# may appear in the input
196
```

there are 196 Boolean variables

0

there are 0 bounds variables

0

there are 0 bounds sparse variables

0

there are 0 discrete variables

0

there are 0 discrete sparse variables

these are the variables to be searched upon...

```
[x0,x1,x2,x3,x4,x5,x6,
x7,x8,x9,x10,x11,x12,x13,
x14,x15,x16,x17,x18,x19,x20,
x21,x22,x23,x24,x25,x26,x27,
x28,x29,x30,x31,x32,x33,x34,
x35,x36,x37,x38,x39,x40,x41,
x42,x43,x44,x45,x46,x47,x48
]
```

...in the order in which the should be searched

```
[a,a,a,a,a,a,a,
a,a,a,a,a,a,a,
a,a,a,a,a,a,a,
a,a,a,a,a,a,a,
```

```
a,a,a,a,a,a,a,
a,a,a,a,a,a,a,
a,a,a,a,a,a,a
]
```

there are twenty-one 1-Dimensional matrices

21

this is the specification of which variables are in each matrix

```
[x49, x50, x51, x52, x53, x54, x55]
[x56, x57, x58, x59, x60, x61, x62]
[x63, x64, x65, x66, x67, x68, x69]
[x70, x71, x72, x73, x74, x75, x76]
[x77, x78, x79, x80, x81, x82, x83]
[x84, x85, x86, x87, x88, x89, x90]
[x91, x92, x93, x94, x95, x96, x97]
[x98, x99, x100, x101, x102, x103, x104]
[x105, x106, x107, x108, x109, x110, x111]
[x112, x113, x114, x115, x116, x117, x118]
[x119, x120, x121, x122, x123, x124, x125]
[x126, x127, x128, x129, x130, x131, x132]
[x133, x134, x135, x136, x137, x138, x139]
[x140, x141, x142, x143, x144, x145, x146]
[x147, x148, x149, x150, x151, x152, x153]
[x154, x155, x156, x157, x158, x159, x160]
[x161, x162, x163, x164, x165, x166, x167]
[x168, x169, x170, x171, x172, x173, x174]
[x175, x176, x177, x178, x179, x180, x181]
[x182, x183, x184, x185, x186, x187, x188]
[x189, x190, x191, x192, x193, x194, x195]
```

there is one 2-dimensional matrix

1

the structure of the 2-dimensional matrix, and the variables involved

```
[[x0, x1, x2, x3, x4, x5, x6],
[x7, x8, x9, x10, x11, x12, x13],
[x14, x15, x16, x17, x18, x19, x20],
[x21, x22, x23, x24, x25, x26, x27],
[x28, x29, x30, x31, x32, x33, x34],
[x35, x36, x37, x38, x39, x40, x41],
[x42, x43, x44, x45, x46, x47, x48]]
```

there are 0 3-dimensional matrices

0

there is no objective function

objective none

display the BIBD when a solution is found. NB Generally, an arbitrary 'display' matrix can be defined to display the output in the manner of the user's choice.

```
print m0
```

these are all the problem constraints

```
sumleq(row(m0, 0), 3)
sumgeq(row(m0, 0), 3)
sumleq(row(m0, 1), 3)
sumgeq(row(m0, 1), 3)
sumleq(row(m0, 2), 3)
sumgeq(row(m0, 2), 3)
sumleq(row(m0, 3), 3)
sumgeq(row(m0, 3), 3)
sumleq(row(m0, 4), 3)
sumgeq(row(m0, 4), 3)
sumleq(row(m0, 5), 3)
sumgeq(row(m0, 5), 3)
sumleq(row(m0, 6), 3)
sumgeq(row(m0, 6), 3)
sumleq(col(m0, 0), 3)
sumgeq(col(m0, 0), 3)
sumleq(col(m0, 1), 3)
sumgeq(col(m0, 1), 3)
sumleq(col(m0, 2), 3)
sumgeq(col(m0, 2), 3)
sumleq(col(m0, 3), 3)
sumgeq(col(m0, 3), 3)
sumleq(col(m0, 4), 3)
sumgeq(col(m0, 4), 3)
sumleq(col(m0, 5), 3)
sumgeq(col(m0, 5), 3)
sumleq(col(m0, 6), 3)
sumgeq(col(m0, 6), 3)
product(x0, x7, x49)
product(x1, x8, x50)
product(x2, x9, x51)
product(x3, x10, x52)
product(x4, x11, x53)
product(x5, x12, x54)
product(x6, x13, x55)
sumleq(v0, 1)
sumgeq(v0, 1)
product(x0, x14, x56)
product(x1, x15, x57)
product(x2, x16, x58)
product(x3, x17, x59)
product(x4, x18, x60)
product(x5, x19, x61)
product(x6, x20, x62)
```

```

sumleq(v1, 1)
sumgeq(v1, 1)
product(x0, x21, x63)
product(x1, x22, x64)
product(x2, x23, x65)
product(x3, x24, x66)
product(x4, x25, x67)
product(x5, x26, x68)
product(x6, x27, x69)
sumleq(v2, 1)
sumgeq(v2, 1)
product(x0, x28, x70)
product(x1, x29, x71)
product(x2, x30, x72)
product(x3, x31, x73)
product(x4, x32, x74)
product(x5, x33, x75)
product(x6, x34, x76)
sumleq(v3, 1)
sumgeq(v3, 1)
product(x0, x35, x77)
product(x1, x36, x78)
product(x2, x37, x79)
product(x3, x38, x80)
product(x4, x39, x81)
product(x5, x40, x82)
product(x6, x41, x83)
sumleq(v4, 1)
sumgeq(v4, 1)
product(x0, x42, x84)
product(x1, x43, x85)
product(x2, x44, x86)
product(x3, x45, x87)
product(x4, x46, x88)
product(x5, x47, x89)
product(x6, x48, x90)
sumleq(v5, 1)
sumgeq(v5, 1)
product(x7, x14, x91)
product(x8, x15, x92)
product(x9, x16, x93)
product(x10, x17, x94)
product(x11, x18, x95)
product(x12, x19, x96)
product(x13, x20, x97)
sumleq(v6, 1)
sumgeq(v6, 1)
product(x7, x21, x98)
product(x8, x22, x99)
product(x9, x23, x100)

```



```

product(x10, x24, x101)
product(x11, x25, x102)
product(x12, x26, x103)
product(x13, x27, x104)
sumleq(v7, 1)
sumgeq(v7, 1)
product(x7, x28, x105)
product(x8, x29, x106)
product(x9, x30, x107)
product(x10, x31, x108)
product(x11, x32, x109)
product(x12, x33, x110)
product(x13, x34, x111)
sumleq(v8, 1)
sumgeq(v8, 1)
product(x7, x35, x112)
product(x8, x36, x113)
product(x9, x37, x114)
product(x10, x38, x115)
product(x11, x39, x116)
product(x12, x40, x117)
product(x13, x41, x118)
sumleq(v9, 1)
sumgeq(v9, 1)
product(x7, x42, x119)
product(x8, x43, x120)
product(x9, x44, x121)
product(x10, x45, x122)
product(x11, x46, x123)
product(x12, x47, x124)
product(x13, x48, x125)
sumleq(v10, 1)
sumgeq(v10, 1)
product(x14, x21, x126)
product(x15, x22, x127)
product(x16, x23, x128)
product(x17, x24, x129)
product(x18, x25, x130)
product(x19, x26, x131)
product(x20, x27, x132)
sumleq(v11, 1)
sumgeq(v11, 1)
product(x14, x28, x133)
product(x15, x29, x134)
product(x16, x30, x135)
product(x17, x31, x136)
product(x18, x32, x137)
product(x19, x33, x138)
product(x20, x34, x139)
sumleq(v12, 1)

```

```

sumgeq(v12, 1)
product(x14, x35, x140)
product(x15, x36, x141)
product(x16, x37, x142)
product(x17, x38, x143)
product(x18, x39, x144)
product(x19, x40, x145)
product(x20, x41, x146)
sumleq(v13, 1)
sumgeq(v13, 1)
product(x14, x42, x147)
product(x15, x43, x148)
product(x16, x44, x149)
product(x17, x45, x150)
product(x18, x46, x151)
product(x19, x47, x152)
product(x20, x48, x153)
sumleq(v14, 1)
sumgeq(v14, 1)
product(x21, x28, x154)
product(x22, x29, x155)
product(x23, x30, x156)
product(x24, x31, x157)
product(x25, x32, x158)
product(x26, x33, x159)
product(x27, x34, x160)
sumleq(v15, 1)
sumgeq(v15, 1)
product(x21, x35, x161)
product(x22, x36, x162)
product(x23, x37, x163)
product(x24, x38, x164)
product(x25, x39, x165)
product(x26, x40, x166)
product(x27, x41, x167)
sumleq(v16, 1)
sumgeq(v16, 1)
product(x21, x42, x168)
product(x22, x43, x169)
product(x23, x44, x170)
product(x24, x45, x171)
product(x25, x46, x172)
product(x26, x47, x173)
product(x27, x48, x174)
sumleq(v17, 1)
sumgeq(v17, 1)
product(x28, x35, x175)
product(x29, x36, x176)
product(x30, x37, x177)
product(x31, x38, x178)

```

```

product(x32, x39, x179)
product(x33, x40, x180)
product(x34, x41, x181)
sumleq(v18, 1)
sumgeq(v18, 1)
product(x28, x42, x182)
product(x29, x43, x183)
product(x30, x44, x184)
product(x31, x45, x185)
product(x32, x46, x186)
product(x33, x47, x187)
product(x34, x48, x188)
sumleq(v19, 1)
sumgeq(v19, 1)
product(x35, x42, x189)
product(x36, x43, x190)
product(x37, x44, x191)
product(x38, x45, x192)
product(x39, x46, x193)
product(x40, x47, x194)
product(x41, x48, x195)
sumleq(v20, 1)
sumgeq(v20, 1)

```

these are symmetry breaking constraints - double lex

```

lexleq(row(m0, 0), row(m0, 1))
lexleq(row(m0, 1), row(m0, 2))
lexleq(row(m0, 2), row(m0, 3))
lexleq(row(m0, 3), row(m0, 4))
lexleq(row(m0, 4), row(m0, 5))
lexleq(row(m0, 5), row(m0, 6))
lexleq(col(m0, 0), col(m0, 1))
lexleq(col(m0, 1), col(m0, 2))
lexleq(col(m0, 2), col(m0, 3))
lexleq(col(m0, 3), col(m0, 4))
lexleq(col(m0, 4), col(m0, 5))
lexleq(col(m0, 5), col(m0, 6))

```