Getting Started with MINION

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

MINION Version 0.2.1

MINION Input Language 1

18 March 2006

Contents

1 Matrix Modelling 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 ith object to the jth 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.

2 Obtaining and Installing MINION

MINION can be obtained from:

- `http://sourceforge.net/projects/minion`

To compile MINION:

```
make all
```

Warning: Compilation can take around 30 minutes. It requires g++ version 4.0 or above (type `g++ -v` to check version). It further requires a program called upx (http://upx.sourceforge.net/) to minimise the size of executables: however, if you do not have this the resulting executables will still be fine (but a bit larger and you will get a make error.)

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 oder. 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.
- [-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.
- [-noprintsols]. Do not print solutions.
- [-test]. A test option for checking and regression testing. Example test instances which this can be run on are in the directory `test_instances`. When run with this option, no other flags are allowed.
- filename.

3 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 $x0$. Its negation is identified by $nx0$.
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.

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.

4 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.

All-Different states that all the variables in a matrix are assigned different variables:
`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⟩)`.

$\leq$ 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 variabl:
`weightedsumleq(⟨matrix of variables⟩, ⟨matrix constants⟩, ⟨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 variables⟩, ⟨matrix of constants⟩, ⟨variable⟩).`

Strong Reification states that the assignment of a 0/1 variable is 1 iff the reified constraint is entailed:
`reify(⟨constraint⟩, ⟨01var⟩).`

Weak Reification states that the assignment of a 0/1 variable is 1 if the reified constraint is entailed:
`reifyimplies(⟨constraint⟩, ⟨01var⟩).`

Table allows the specification of an extensional constraint. The set of tuples should be given in strictly increasing lexicographic order:
`table(⟨matrix of variables⟩, ⟨tuples⟩)`

5 Input format

A formal outline of the input format for Input Format Version 1 is shown below.

<MinionInput> ::=
MINION 1
<comments>
<noOf01Vars>
<noOfBoundsVars> {<lb> <ub> <number>}
<noOfSparseBoundsVars> {’{’ <elem>{,<elem>}’}’<number>}
<noOfDiscreteVars> {<lb> <ub> <number>}
<noOfSparseDiscreteVars> {’{’ <elem>{,<elem>}’}’<number>}
<variableOrder>
<valueOrder>
<noOf1dMatrices> {<literalVar1dMatrix>}
<noOf2dMatrices> {<literalVar2dMatrix>}
<noOf3dMatrices> {<literalVar3dMatrix>}
objective <objectiveExpression>
print <printExpression>
{<constraint>}

<objectiveExpression> ::=
’none’ | ’minimising’ <var> | ’maximising’ <var>

<objectiveExpression> ::=
’none’ | ’minimising’ <var> | ’maximising’ <var>

<printExpression> ::=
’none’ | <2dMatrixId>

<constraint> ::=
<reifiableConstraint> |
allDiff(<varVectorExpression>) |
reify(<reifiableConstraint>, <var>) |
reifyimplies(<reifiableConstraint>, <var>) |
table(<varVectorExpression>, <tuples>)

<reifiableConstraint> ::=
<eqOrDiseqConstraint> (<var>, <var>) |
element(<varVectorExpression>, <var>, <var>) |
ineq(<var>, <var>, <const>) |
<lexConstraint> (<varVectorExpression>, <varVectorExpression>) |
<MinOrMaxConstraint> (<varVectorExpression>, <var>) |
occurrence(<varVectorExpression>, <const>, <var>) |
product(<varVectorExpression>, <var>) |
product(<varVectorExpression>, <literalConstVector>, <var>) |
sum(<varVectorExpression>, <var>)

<varVectorExpression> ::=
<literalVarVector> | <1dMatrixId> | <2dMatrixId> |
<3dMatrixId> | <rowOrCol>(<2dMatrixId>, <index>) |
<colOrRowXOrRowY>(<3dMatrixId>, <index>, <index>)

6 Example

In Section 1 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

```
[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
]
```

these are the variables to be searched upon, 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
]
```

this means that the value ordering to be applied to each search variable is ascending. The options are a, ascending and d, descending

```
21
```

there are 21 1-Dimensional matrices

```
[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]
```

this is the specification of which variables are in each matrix

```
1
```

there is 1 2-dimensional matrix

```
[[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]]
```

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

```
0
```

there are 0 3-dimensional matrices

```
objective none
```

there is no objective function

```
print m0
```

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.

```
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 all the problem constraints

```
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))
```

these are symmetry breaking constraints - double lex