

# minion executable documentation

Ian Gent, Chris Jefferson, Ian Miguel, Neil Moore, Karen Petrie, Andrea Rendl

June 25, 2008

You are viewing documentation for minion. The same documentation is available from a minion executable by typing `minion help` at the command line. We intend that the command line help system be the main source of documentation for the system.

Each of the entries below concerns a different aspect of the system, and the entries are arranged hierarchically. For example to view information about the set of available constraints as a whole view “constraints” and to view specific information about the alldiff constraint view “constraints alldiff”.

A good place to start would be viewing the “input example” entry which exhibits a complete example of a minion input file.

Usage: `minion [switches] [minion input file]`

## Contents

<b>1</b>	<b>constraints</b>	<b>1</b>
<b>2</b>	<b>constraints alldiff</b>	<b>1</b>
<b>3</b>	<b>constraints difference</b>	<b>2</b>
<b>4</b>	<b>constraints diseq</b>	<b>2</b>
<b>5</b>	<b>constraints div</b>	<b>3</b>
<b>6</b>	<b>constraints element</b>	<b>3</b>
<b>7</b>	<b>constraints element_one</b>	<b>4</b>
<b>8</b>	<b>constraints eq</b>	<b>5</b>
<b>9</b>	<b>constraints gacalldiff</b>	<b>5</b>
<b>10</b>	<b>constraints gcc</b>	<b>6</b>
<b>11</b>	<b>constraints hamming</b>	<b>6</b>

12 constraints ineq	7
13 constraints lexleq	7
14 constraints lexless	8
15 constraints litsumgeq	8
16 constraints max	9
17 constraints min	9
18 constraints minuseq	9
19 constraints modulo	10
20 constraints occurrence	10
21 constraints occurrencegeq	11
22 constraints occurrenceleq	11
23 constraints pow	11
24 constraints product	12
25 constraints reification	12
26 constraints reify	13
27 constraints reifyimply	13
28 constraints sumgeq	13
29 constraints sumleq	13
30 constraints table	14
31 constraints watchelement	14
32 constraints watchelement_one	15
33 constraints watchsumgeq	15
34 constraints watchsumleq	16
35 constraints watchvecexists_and	16
36 constraints watchvecexists_less	17

37 constraints watchvecneq	17
38 constraints weightedsumgeq	17
39 constraints weightedsumleq	18
40 input	18
41 input constraints	19
42 input example	19
43 input search	21
44 input tuplelist	22
45 input variables	23
46 switches	23
47 switches -check	23
48 switches -dumptree	24
49 switches -findallsols	24
50 switches -fullprop	24
51 switches -nocheck	24
52 switches -odelimit	24
53 switches -noprintsols	25
54 switches -preprocess	25
55 switches -printsols	26
56 switches -printsolonly	26
57 switches -quiet	26
58 switches -randomiseorder	26
59 switches -randomseed	26
60 switches -sollimit	26
61 switches -solsout	27

62 switches -tableout	27
63 switches -timelimit	27
64 switches -varorder	27
65 switches -verbose	28
66 switches -X-prop-node	28
67 variables	28
68 variables 01	29
69 variables alias	29
70 variables bounds	30
71 variables constants	30
72 variables discrete	30
73 variables sparsebounds	31
74 variables vectors	31

## 1 constraints

### Description

Minion supports many constraints and these are regularly being improved and added to. In some cases multiple implementations of the same constraints are provided and we would appreciate additional feedback on their relative merits in your problem.

Minion does not support nesting of constraints, however this can be achieved by auxiliary variables and reification.

Variables can be replaced by constants. You can find out more on expressions for variables, vectors, etc. in the section on variables.

### References

help variables

## 2 constraints alldiff

### Description

Forces the input vector of variables to take distinct values.

## Example

Suppose the input file had the following vector of variables defined:

```
DISCRETE myVec[9] {1..9}
```

To ensure that each variable takes a different value include the following constraint:

```
alldiff(myVec)
```

## Notes

Enforces the same level of consistency as a clique of not equals constraints.

## Reifiability

This constraint is reifiable and reifyimply'able.

## References

See

```
help constraints gacalldiff
```

for the same constraint that enforces GAC.

# 3 constraints difference

## Description

The constraint

```
difference(x,y,z)
```

ensures that  $z=|x-y|$  in any solution.

## Notes

This constraint can be expressed in a much longer form, this form both avoids requiring an extra variable, and also gets better propagation. It gets bounds consistency.

## Reifiability

This constraint is reifyimply'able but not reifiable.

# 4 constraints diseq

## Description

Constrain two variables to take different values.

### Notes

Achieves arc consistency.

### Example

```
diseq(v0,v1)
```

### Reifiability

This constraint is reifiable and reifyimply'able.

## 5 constraints div

### Description

The constraint

```
div(x,y,z)
```

ensures that  $\text{floor}(x/y)=z$ .

### Notes

This constraint is only available for positive domains x, y and z.

### Reifiability

This constraint is reifyimply'able but not reifiable.

### References

help constraints modulo

## 6 constraints element

### Description

The constraint

```
element(vec, i, e)
```

specifies that, in any solution,  $\text{vec}[i] = e$  and i is in the range  $[0 \dots |\text{vec}|-1]$ .

### Reifiability

This constraint is reifyimply'able but not reifiable.

## Notes

Warning: This constraint is not confluent. Depending on the order the propagators are called in Minion, the number of search nodes may vary when using `element`. To avoid this problem, use `watchelement` instead. More details below.

The level of propagation enforced by this constraint is not named, however it works as follows. For constraint `vec[i]=e`:

- After `i` is assigned, ensures that  $\min(\text{vec}[i]) = \min(e)$  and  $\max(\text{vec}[i]) = \max(e)$ .
- When `e` is assigned, removes `idx` from the domain of `i` whenever `e` is not an element of the domain of `vec[idx]`.
- When `m[idx]` is assigned, removes `idx` from `i` when `m[idx]` is not in the domain of `e`.

This level of consistency is designed to avoid the propagator having to scan through `vec`, except when `e` is assigned. It does a quantity of cheap propagation and may work well in practise on certain problems.

Element is not confluent, which may cause the number of search nodes to vary depending on the order in which constraints are listed in the input file, or the order they are called in Minion. For example, the following input causes Minion to search 41 nodes.

```
MINION 3
**VARIABLES**
DISCRETE x[5] {1..5}
**CONSTRAINTS**
element([x[0],x[1],x[2]], x[3], x[4])
alldiff([x])
**EOF**
```

However if the two constraints are swapped over, Minion explores 29 nodes. As a rule of thumb, to get a lower node count, move `element` constraints to the end of the list.

## References

See the entry

`constraints watchelement`

for details of an identical constraint that enforces generalised arc consistency.

## 7 constraints element\_one

### Description

The constraint `element one` is identical to `element`, except that the vector is indexed from 1 rather than from 0.

## References

See

`help constraints element`

for details of the `element` constraint which is almost identical to this one.

## 8 constraints eq

### Description

Constrain two variables to take equal values.

### Example

`eq(x0,x1)`

### Notes

Achieves bounds consistency.

### Reifiability

This constraint is reifiable and reifyimply'able.

### Reference

`help constraints minuseq`

## 9 constraints gacalldiff

### Description

Forces the input vector of variables to take distinct values.

### Example

Suppose the input file had the following vector of variables defined:

`DISCRETE myVec[9] {1..9}`

To ensure that each variable takes a different value include the following constraint:

`gacalldiff(myVec)`

### Reifiability

This constraint is reifiable and reifyimply'able.



## Notes

This constraint enforces generalized arc consistency.

## 10 constraints gcc

### Description

The Generalized Cardinality Constraint (GCC) constrains the number of each value that a set of variables can take.

```
gcc([primary variables], [capacity variables])
```

For each value in the initial domains of the primary variables, there must be a capacity variable.

For example, if the union of the initial domains of the primary variables is  $\{-5, -3, -1, 0, 2, 3, 5\}$  then there would be 11 capacity variables, specifying the number of occurrences of each value in the interval  $[-5 \dots 5]$ .

This constraint is new, and its syntax and implementation are not finalised.

### Example

Suppose the input file had the following vectors of variables defined:

```
DISCRETE myVec[9] {1..9}  
BOUND cap[9] {0..2}
```

The following constraint would restrict the occurrence of values 1..9 in myVec to be at most 2 each initially, and finally equal to the values of the cap vector.

```
gcc(myVec, cap)
```

### Reifiability

This constraint is reifyably able but not reifiable.

## Notes

This constraint enforces a hybrid consistency. It reads the bounds of the capacity variables, then enforces GAC over the primary variables only. Then the bounds of the capacity variables are updated by counting values in the domains of the primary variables.

## 11 constraints hamming

### Description

The constraint

```
hamming(X,Y,c)
```

ensures that the hamming distance between X and Y is c. That is, that c is the size of the set  $\{i \mid X[i] \neq Y[i]\}$

### Reifiability

This constraint is reifyimply'able but not reifiable.

## 12 constraints ineq

### Description

The constraint

```
ineq(x, y, k)
```

ensures that

```
x <= y + k
```

in any solution.

### Notes

Minion has no strict inequality (<) constraints. However  $x < y$  can be achieved by

```
ineq(x, y, -1)
```

### Reifiability

This constraint is reifiable and reifyimply'able.

## 13 constraints lexleq

### Description

The constraint

```
lexleq(vec0, vec1)
```

takes two vectors `vec0` and `vec1` of the same length and ensures that `vec0` is lexicographically less than or equal to `vec1` in any solution.

### Notes

This constraints achieves GAC.

### Reifiability

This constraint is reifiable and reifyimply'able.

### References

See also

```
help constraints lexless
```

for a similar constraint with strict lexicographic inequality.

## 14 constraints lexless

### Description

The constraint

```
lexless(vec0, vec1)
```

takes two vectors `vec0` and `vec1` of the same length and ensures that `vec0` is lexicographically less than `vec1` in any solution.

### Notes

This constraint maintains GAC.

### Reifiability

This constraint is reifiable and reifyimply'able.

### References

See also

```
help constraints lexleq
```

for a similar constraint with non-strict lexicographic inequality.

## 15 constraints litsumgeq

### Description

The constraint `litsumgeq(vec1, vec2, c)` ensures that there exists at least `c` distinct indices `i` such that `vec1[i] = vec2[i]`.

### Notes

A SAT clause `{x,y,z}` can be created using:

```
litsumgeq([x,y,z],[1,1,1],1)
```

Note also that this constraint is more efficient for smaller values of `c`. For large values consider using `watchsumleq`.

### Reifiability

This constraint is reifyimply'able but not reifiable.

### References

See also

```
help constraints watchsumleq  
help constraints watchsumgeq
```

## 16 constraints max

### Description

The constraint

```
max(vec, x)
```

ensures that  $x$  is equal to the maximum value of any variable in  $vec$ .

### Reifiability

This constraint is reifyimply'able but not reifiable.

### References

See

```
help constraints min
```

for the opposite constraint.

## 17 constraints min

### Description

The constraint

```
min(vec, x)
```

ensures that  $x$  is equal to the minimum value of any variable in  $vec$ .

### Reifiability

This constraint is reifyimply'able but not reifiable.

### References

See

```
help constraints max
```

for the opposite constraint.

## 18 constraints minuseq

### Description

Constraint

```
minuseq(x,y)
```

ensures that  $x=-y$ .

### Reifiability

This constraint is reifyimply'able but not reifiable.

### Reference

help constraints eq

## 19 constraints modulo

### Description

The constraint

```
modulo(x,y,z)
```

ensures that  $x\%y=z$  i.e.  $z$  is the remainder of dividing  $x$  by  $y$ .

### Notes

This constraint is only available for positive domains  $x$ ,  $y$  and  $z$ .

### Reifiability

This constraint is reifyimply'able but not reifiable.

### References

help constraints div

## 20 constraints occurrence

### Description

The constraint

```
occurrence(vec, elem, count)
```

ensures that there are  $count$  occurrences of the value  $elem$  in the vector  $vec$ .

### Notes

$elem$  must be a constant, not a variable.

### Reifiability

This constraint is reifyimply'able but not reifiable.

### References

help constraints occurrenceleq

help constraints occurrencegeq

## 21 constraints occurrencegeq

### Description

The constraint

```
occurrencegeq(vec, elem, count)
```

ensures that there are AT LEAST count occurrences of the value elem in the vector vec.

### Notes

elem and count must be constants

### Reifiability

This constraint is reifyimply'able but not reifiable.

### References

help constraints occurrence

help constraints occurrenceleq

## 22 constraints occurrenceleq

### Description

The constraint

```
occurrenceleq(vec, elem, count)
```

ensures that there are AT MOST count occurrences of the value elem in the vector vec.

### Notes

elem and count must be constants

### Reifiability

This constraint is reifyimply'able but not reifiable.

### References

help constraints occurrence

help constraints occurrencegeq

## 23 constraints pow

### Description

The constraint

```
pow(x,y,z)
```

ensures that  $x^y=z$ .

## Notes

This constraint is only available for positive domains  $x$ ,  $y$  and  $z$ .

## Reifiability

This constraint is reifyimply'able but not reifiable.

# 24 constraints product

## Description

The constraint

```
product(x,y,z)
```

ensures that  $z=xy$  in any solution.

## Notes

This constraint can be used for (and, in fact, has a specialised implementation for) achieving boolean AND, i.e.  $x \& y=z$  can be modelled as

```
product(x,y,z)
```

The general constraint achieves bounds generalised arc consistency for positive numbers.

## Reifiability

This constraint is reifyimply'able but not reifiable.

# 25 constraints reification

## Description

Reification is provided in two forms: reify and reifyimply.

```
reify(constraint, r) where r is a 0/1 var
```

ensures that  $r$  is set to 1 if and only if constraint is satisfied. That is, if  $r$  is 0 the constraint must NOT be satisfied; and if  $r$  is 1 it must be satisfied as normal. Conversely, if the constraint is satisfied then  $r$  must be 1, and if not then  $r$  must be 0.

```
reifyimply(constraint, r)
```

only checks that if  $r$  is set to 1 then constraint must be satisfied. If  $r$  is not 1, constraint may be either satisfied or unsatisfied. Furthermore  $r$  is never set by propagation, only by search; that is, satisfaction of constraint does not affect the value of  $r$ .

## Notes

Not all constraints are reifiable. Entries for individual constraints give more information.

## 26 constraints reify

### References

See  
    `help constraints reification`

## 27 constraints reifyimply

### References

See  
    `help constraints reification`

## 28 constraints sumgeq

### Description

The constraint

```
sumgeq(vec, c)
```

ensures that  $\text{sum}(\text{vec}) \geq c$ .

### Reifiability

This constraint is reifiable and `reifyimply`'able.

## 29 constraints sumleq

### Description

The constraint

```
sumleq(vec, c)
```

ensures that  $\text{sum}(\text{vec}) \leq c$ .

### Reifiability

This constraint is reifiable and `reifyimply`'able.



## 30 constraints table

### Description

An extensional constraint that enforces GAC. The constraint is specified via a list of tuples.

### Example

To specify a constraint over 3 variables that allows assignments (0,0,0), (1,0,0), (0,1,0) or (0,0,1) do the following.

1) Add a tuplelist to the **\*\*TUPLELIST\*\*** section, e.g.:

```
**TUPLELIST**
myext 4 3
0 0 0
1 0 0
0 1 0
0 0 1
```

N.B. the number 4 is the number of tuples in the constraint, the number 3 is the -arity.

2) Add a table constraint to the **\*\*CONSTRAINTS\*\*** section, e.g.:

```
**CONSTRAINTS**
table(myvec, myext)
```

and now the variables of myvec will satisfy the constraint myext.

### Example

The constraints extension can also be specified in the constraint definition, e.g.:

```
table(myvec, {<0,0,0>,<1,0,0>,<0,1,0>,<0,0,1>})
```

### Reifiability

This constraint is reifyimply'able but not reifiable.

### References

help input tuplelist

## 31 constraints watchelement

### Description

The constraint

```
watchelement(vec, i, e)
```

specifies that, in any solution,  $vec[i] = e$  and  $i$  is in the range  $[0 \dots |vec|-1]$ .

### Reifiability

This constraint is reifyimply'able but not reifiable.

### Notes

Enforces generalised arc consistency.

### References

See entry

`help constraints element`

for details of an identical constraint that enforces a lower level of consistency.

## 32 constraints watchelement\_one

### Description

This constraint is identical to `watchelement`, except the vector is indexed from 1 rather than from 0.

### References

See entry

`help constraints watchelement`

for details of `watchelement` which `watchelement_one` is based on.

## 33 constraints watchsumgeq

### Description

The constraint `watchsumgeq(vec, c)` ensures that  $\text{sum}(\text{vec}) \geq c$ .

### Notes

For this constraint, small values of  $c$  are more efficient.

Equivalent to `litsumgeq(vec, [1,...,1], c)`, but faster.

This constraint works on 0/1 variables only.

### Reifiability

This constraint is reifyimply'able but not reifiable.

## References

See also

```
help constraints watchsumleq
help constraints litsumgeq
```

## 34 constraints watchsumleq

### Description

The constraint `watchsumleq(vec, c)` ensures that  $\text{sum}(\text{vec}) \leq c$ .

### Notes

Equivalent to `litsumgeq([vec1,...,vecn], [0,...,0], n-c)` but faster.

This constraint works on binary variables only.

For this constraint, large values of  $c$  are more efficient.

### Reifiability

This constraint is reifyimply'able but not reifiable.

## References

See also

```
help constraints watchsumgeq
help constraints litsumgeq
```

## 35 constraints watchvecexists\_and

### Description

The constraint

```
watchvecexists_and(A, B)
```

ensures that there exists some index  $i$  such that  $A[i] > 0$  and  $B[i] > 0$ .

For booleans this is the same as 'exists  $i$  s.t.  $A[i] \ \&\& \ B[i]$ '.

### Reifiability

This constraint is reifyimply'able but not reifiable.

## 36 constraints watchvecexists\_less

### Description

The constraint

```
watchvecexists_less(A, B)
```

ensures that there exists some index  $i$  such that  $A[i] < B[i]$ .

### Reifiability

This constraint is reifyimply'able but not reifiable.

## 37 constraints watchvecneq

### Description

The constraint

```
watchvecneq(A, B)
```

ensures that  $A$  and  $B$  are not the same vector, i.e., there exists some index  $i$  such that  $A[i] \neq B[i]$ .

### Reifiability

This constraint is reifyimply'able but not reifiable.

## 38 constraints weightedsumgeq

### Description

The constraint

```
weightedsumgeq(constantVec, varVec, total)
```

ensures that  $\text{constantVec} \cdot \text{varVec} \geq \text{total}$ , where  $\text{constantVec} \cdot \text{varVec}$  is the scalar dot product of  $\text{constantVec}$  and  $\text{varVec}$ .

### Reifiability

This constraint is reifiable and reifyimply'able.

### References

```
help constraints weightedsumleq
help constraints sumleq
help constraints sumgeq
```

## 39 constraints weightedsumleq

### Description

The constraint

```
weightedsumleq(constantVec, varVec, total)
```

ensures that  $\text{constantVec} \cdot \text{varVec} \leq \text{total}$ , where  $\text{constantVec} \cdot \text{varVec}$  is the scalar dot product of  $\text{constantVec}$  and  $\text{varVec}$ .

### Reifiability

This constraint is reifiable and reifyimply'able.

### References

```
help constraints weightedsumgeq
help constraints sumleq
help constraints sumgeq
```

## 40 input

### Description

Minion expects to be provided with the name of an input file as an argument. This file contains a specification of the CSP to be solved as well as settings that the search process should use. The format is

```
Minion3Input ::= MINION 3
                <InputSection>+
                **EOF**

InputSection ::= <VariablesSection>
                | <SearchSection>
                | <ConstraintsSection>
                | <TupleListSection>
```

i.e. 'MINION 3' followed by any number of variable, search, constraints and tuplelists sections (can repeat) followed by '\*\*EOF\*\*', the end of file marker.

All text from a '#' character to the end of the line is ignored.

See the associated help entries below for information on each section.

### Notes

You can give an input file via standard input by specifying '--' as the file name, this might help when minion is being used as a tool in a shell script or for compressed input, e.g.,

```
gunzip -c myinput.minion.gz | minion
```

## 41 input constraints

### Description

The constraints section consists of any number of constraint declarations on separate lines.

```
ConstraintsSection ::= **CONSTRAINTS**
                    <ConstraintDeclaration>*
```

### Example

```
**CONSTRAINTS**
eq(bool,0)
alldiff(d)
```

### References

See help entries for individual constraints under

```
    help constraints
```

for details on constraint declarations.

## 42 input example

### Example

Below is a complete minion input file with commentary, as an example.

```
MINION 3

# While the variable section doesn't have to come first, you can't
# really do anything until
# You have one...
**VARIABLES**

# There are 4 type of variables
BOOL bool          # Boolean don't need a domain
BOUND b {1..3}     # Bound vars need a domain given as a range
DISCRETE d {1..3}  # So do discrete vars

#Note: Names are case sensitive!

# Internally, Bound variables are stored only as a lower and upper bound
# Whereas discrete variables allow any sub-domain

SPARSEBOUND s {1,3,6,7} # Sparse bound variables take a sorted list of values

# We can also declare matrices of variables!

DISCRETE q[3] {0..5} # This is a matrix with 3 variables: q[0],q[1] and q[2]
BOOL bm[2,2] # A 2d matrix, variables bm[0,0], bm[0,1], bm[1,0], bm[1,1]
BOOL bn[2,2,2,2] # You can have as many indices as you like!
```

```

#The search section is entirely optional
**SEARCH**

# Note that everything in SEARCH is optional, and can only be given at
# most once!

# If you don't give an explicit variable ordering, one is generated.
# These can take matrices in interesting ways like constraints, see below.
VARORDER [bool,b,d]

# If you don't give a value ordering, 'ascending' is used
#VALORDER [a,a,a,a]

# You can have one objective function, or none at all.
MAXIMISING bool
# MINIMISING x3

# both (MAX/MIN)IMISING and (MAX/MIN)IMIZING are accepted...

# Print statement takes a vector of things to print

PRINT [bool, q]

# You can also give:
# PRINT ALL (the default)
# PRINT NONE

# Declare constraints in this section!
**CONSTRAINTS**

# Constraints are defined in exactly the same way as in MINION input
# formats 1 & 2
eq(bool, 0)
eq(b,d)

# To get a single variable from a matrix, just index it
eq(q[1],0)
eq(bn[0,1,1,1], bm[1,1])

# It's easy to get a row or column from a matrix. Just use _ in the
# indices you want
# to vary. Just giving a matrix gives all the variables in that matrix.

#The following shows how flattening occurs...

# [bm] == [ bm[_,_] ] == [ bm[0,0], bm[0,1], bm[1,0], bm[1,1] ]
# [ bm[_,1] ] = [ bm[0,1], bm[1,1] ]
# [ bn[1,_,0,_] ] = [ bn[1,0,0,0], b[1,0,0,1], b[1,1,0,0], b[1,1,0,1] ]

# You can string together a list of such expressions!

lexleq( [bn[1,_,0,_], bool, q[0]] , [b, bm, d] )

# One minor problem.. you must always put [ ] around any matrix expression, so
# lexleq(bm, bm) is invalid

```

```

lexleq( [bm], [bm] ) # This is OK!

# Can give tuplelists, which can have names!
# The input is: <name> <num_of_tuples> <tuple_length> <numbers...>
# The formatting can be about anything..

**TUPLELIST**

Fred 3 3
0 2 3
2 0 3
3 1 3

Bob 2 2 1 2 3 4

#No need to put everything in one section! All sections can be reopened..
**VARIABLES**

# You can even have empty sections.. if you want

**CONSTRAINTS**

#Specify tables by their names..

table([q], Fred)

# Can still list tuples explicitly in the constraint if you want at
# the moment.
# On the other hand, I might remove this altogether, as it's worse than giving
# Tuplelists

table([q],{ <0,2,3>,<2,0,3>,<3,1,3> })

#Must end with the **EOF** marker!

**EOF**

Any text down here is ignored, so you can write whatever you like (or
nothing at all...)

```

## 43 input search

### Description

Inside the search section one can specify

- variable orderings,
- value orderings,
- optimisation function, and
- details of how to print out solutions.

```

SearchSection::= <VariableOrdering>?
                 <ValueOrdering>?
                 <OptimisationFn>?

```



<PrintFormat>?

In the variable ordering a fixed ordering can be specified on any subset of variables. These are the search variables that will be instantiated in every solution. If none is specified some other fixed ordering of all the variables will be used.

```
VariableOrdering ::= VARORDER[ <varname>+ ]
```

The value ordering allows the user to specify an instantiation order for the variables involved in the variable order, either ascending (a) or descending (d) for each. When no value ordering is specified, the default is to use ascending order for every search variable.

```
ValueOrdering ::= VALORDER[ (a|d)+ ]
```

To model an optimisation problem the user can specify to minimise or maximise a variable's value.

```
OptimisationFn ::= MAXIMISING <varname>  
                  | MINIMISING <varname>
```

Finally, the user can control some aspects of the way solutions are printed. By default (no PrintFormat specified) all the variables are printed in declaration order. Alternatively a custom vector, or ALL variables, or no (NONE) variables can be printed. If a matrix or, more generally, a tensor is given instead of a vector, it is automatically flattened into a vector as described in 'help variables vectors'.

```
PrintFormat ::= PRINT <vector>  
               | PRINT ALL  
               | PRINT NONE
```

## 44 input tuplelist

### Description

In a tuplelist section lists of allowed tuples for table constraints can be specified. This technique is preferable to specifying the tuples in the constraint declaration, since the tuplelists can be shared between constraints and named for readability.

The required format is

```
TuplelistSection ::= **TUPLELIST**  
                   <Tuplelist>*
```

```
Tuplelist ::= <name> <num_tuples> <tuple_length> <numbers>+
```

### Example

```
**TUPLELIST**  
AtMostOne 4 3  
0 0 0  
0 0 1  
0 1 0  
1 0 0
```

## References

help constraints table

## 45 input variables

### Description

The variables section consists of any number of variable declarations on separate lines.

```
VariablesSection ::= **VARIABLES**  
                  <VarDeclaration>*
```

### Example

```
**VARIABLES**  
  
BOOL bool                                #boolean var  
BOUND b {1..3}                          #bounds var  
SPARSEBOUND myvar {1,3,4,6,7,9,11}      #sparse bounds var  
DISCRETE d[3] {1..3}                   #array of discrete vars
```

## References

See the help section

help variables

for detailed information on variable declarations.

## 46 switches

### Description

Minion supports a number of switches to augment default behaviour. To see more information on any switch, use the help system. The list below contains all available switches. For example to see help on -quiet type something similar to

```
minion help switches -quiet
```

replacing 'minion' by the name of the executable you're using.

## 47 switches -check

### Description

Check solutions for correctness before printing them out.

### Notes

This option is the default for DEBUG executables.

## 48 switches -dumptree

### Description

Print out the branching decisions and variable states at each node.

## 49 switches -findallsols

### Description

Find all solutions and count them. This option is ignored if the problem contains any minimising or maximising objective.

## 50 switches -fullprop

### Description

Disable incremental propagation.

### Notes

This should always slow down search while producing exactly the same search tree.

Only available in a DEBUG executable.

## 51 switches -nocheck

### Description

Do not check solutions for correctness before printing them out.

### Notes

This option is the default on non-DEBUG executables.

## 52 switches -nodelimit

### Description

To stop search after N nodes, do

```
minion -nodelimit N myinput.minion
```

### References

```
help switches -timelimit  
help switches -sollimit
```

## 53 switches -noprintsols

### Description

Do not print solutions.

## 54 switches -preprocess

This switch allows the user to choose what level of preprocess is applied to their model before search commences.

The choices are:

- GAC
- generalised arc consistency (default)
- all propagators are run to a fixed point
- if some propagators enforce less than GAC then the model will not necessarily be fully GAC at the outset
  
- SACBounds
- singleton arc consistency on the bounds of each variable
- AC can be achieved when any variable lower or upper bound is a singleton in its own domain
  
- SAC
- singleton arc consistency
- AC can be achieved in the model if any value is a singleton in its own domain
  
- SSACBounds
- singleton singleton bounds arc consistency
- SAC can be achieved in the model when domains are replaced by either the singleton containing their upper bound, or the singleton containing their lower bound
  
- SSAC
- singleton singleton arc consistency
- SAC can be achieved when any value is a singleton in its own domain

These are listed in order of roughly how long they take to achieve. Preprocessing is a one off cost at the start of search. The success of higher levels of preprocessing is problem specific; SAC preprocesses may take a long time to complete, but may reduce search time enough to justify the cost.

### Example

To enforce SAC before search:

```
minion -preprocess SAC myinputfile.minion
```

### References

help switches -X-prop-node

## 55 switches -printsols

### Description

Print solutions.

## 56 switches -printsolonly

### Description

Print only solutions and a summary at the end.

## 57 switches -quiet

### Description

Do not print parser progress.

### References

help switches -verbose

## 58 switches -randomiseorder

### Description

Randomises the ordering of the decision variables. If the input file specifies an ordering it will randomly permute this. If no ordering is specified a random permutation of all the variables is used.

## 59 switches -randomseed

### Description

Set the pseudorandom seed to N. This allows 'random' behaviour to be repeated in different runs of minion.

## 60 switches -sollimit

### Description

To stop search after N solutions have been found, do

```
minion -sollimit N myinput.minion
```

### References

help switches -odelimit

help switches -timelimit

## 61 switches -solsout

### Description

Append all solutions to a named file.  
Each solution is placed on a line, with no extra formatting.

### Example

To add the solutions of myproblem.minion to mysols.txt do

```
minion -solsout mysols.txt myproblem.minion
```

## 62 switches -tableout

### Description

Append a line of data about the current run of minion to a named file.  
This data includes minion version information, arguments to the executable, build and solve time statistics, etc. See the file itself for a precise schema of the supplied information.

### Example

To add statistics about solving myproblem.minion to mystats.txt do

```
minion -tableout mystats.txt myproblem.minion
```

## 63 switches -timelimit

### Description

To stop search after N seconds, do

```
minion -timelimit N myinput.minion
```

### References

```
help switches -odelimit  
help switches -sollimit
```

## 64 switches -varorder

### Description

Enable a particular variable ordering for the search process. This flag is experimental and minion's default ordering might be faster.

The available orders are:

```
- sdf - smallest domain first, break ties lexicographically
```

- sdf-random - sdf, but break ties randomly
- srf - smallest ratio first, chooses unassigned variable with smallest percentage of its initial values remaining, break ties lexicographically
- srf-random - srf, but break ties randomly
- ldf - largest domain first, break ties lexicographically
- ldf-random - ldf, but break ties randomly
- random - random variable ordering
- static - lexicographical ordering

## 65 switches -verbose

### Description

Print parser progress.

### References

help switches -quiet

## 66 switches -X-prop-node

### Description

Allows the user to choose the level of consistency to be enforced during search.

See entry 'help switches -preprocess' for details of the available levels of consistency.

### Example

To enforce SSAC during search:

```
minion -X-prop-node SSAC input.minion
```

### References

help switches -preprocess

## 67 variables

### General

Minion supports 4 different variable types, namely

- 0/1 variables,
- bounds variables,
- sparse bounds variables, and
- discrete variables.

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 4 (Discrete 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.

Minion also supports use of constants in place of variables, and constant vectors in place of vectors of variables. Using constants will be at least as efficient as using variables when the variable has a singleton domain.

See the entry on vectors for information on how vectors, matrices and, more generally, tensors are handled in minion input. See also the alias entry for information on how to multiply name variables for convenience.

## 68 variables 01

### Description

01 variables 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. A boolean variable  $x$ 's negation is identified by  $!x$ .

### Example

Declaration of a 01 variable called `bool` in input file:

```
BOOL bool
```

Use of this variable in a constraint:

```
eq(bool, 0) #variable bool equals 0
```

## 69 variables alias

### Description

Specifying an alias is a way to give a variable another name. Aliases appear in the **\*\*VARIABLES\*\*** section of an input file. It is best described using some examples:

```
ALIAS c = a
```

```
ALIAS c[2,2] = [[myvar,b[2]], [b[1],anothervar]]
```



## 70 variables bounds

### Description

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 cannot be put in the domains of the variables.

### Example

Declaration of a bound variable called myvar with domain between 1 and 7 in input file:

```
BOUND myvar {1..7}
```

Use of this variable in a constraint:

```
eq(myvar, 4) #variable myvar equals 4
```

## 71 variables constants

### Description

Minion supports the use of constants anywhere where a variable can be used. For example, in a constraint as a replacement for a single variable, or a vector of constants as a replacement for a vector of variables.

### Examples

Use of a constant:

```
eq(x,1)
```

Use of a constant vector:

```
element([10,9,8,7,6,5,4,3,2,1],idx,e)
```

## 72 variables discrete

### Description

In discrete variables, the domain ranges between the specified lower and upper bounds, but during search any domain value may be pruned, i.e., propagation and search may punch arbitrary holes in the domain.

### Example

Declaration of a discrete variable x with domain {1,2,3,4} in input file:

```
DISCRETE x {1..4}
```

Use of this variable in a constraint:

```
eq(x, 2) #variable x equals 2
```

## 73 variables sparsebounds

### Description

In sparse bounds variables 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.

### Notes

Declaration of a sparse bounds variable called myvar containing values {1,3,4,6,7,9,11} in input file:

```
SPARSEBOUND myvar {1,3,4,6,7,9,11}
```

Use of this variable in a constraint:

```
eq(myvar, 3) #myvar equals 3
```

## 74 variables vectors

### Description

Vectors, matrices and tensors can be declared in minion input. Matrices and tensors are for convenience, as constraints do not take these as input; they must first undergo a flattening process to convert them to a vector before use.

### Examples

A vector of 0/1 variables:

```
BOOL myvec[5]
```

A matrix of discrete variables:

```
DISCRETE sudoku[9,9] {1..9}
```

A 3D tensor of 0/1s:

```
BOOL mycube[3,3,2]
```

One can create a vector from scalars and elements of vectors, etc.:

```
alldiff([x,y,myvec[1],mymatrix[3,4]])
```

When a matrix or tensor is constrained, it is treated as a vector whose entries have been strung out into a vector in index order with the rightmost index changing most quickly, e.g.

```
alldiff(sudoku)
```

is equivalent to

```
alldiff([sudoku[0,0],...,sudoku[0,8],...,sudoku[8,0],...,sudoku[8,8]])
```

Furthermore, with indices filled selectively and the remainder filled with underscores (\_) the flattening applies only to the underscore indices:

```
alldiff(sudoku[4,_])
```

is equivalent to

```
alldiff([sudoku[4,0],...,sudoku[4,8]])
```

Lastly, one can optionally add square brackets ([]) around an expression to be flattened to make it look more like a vector:

```
alldiff([sudoku[4,_]])
```

is equivalent to

```
alldiff(sudoku[4,_])
```