Package: DoE.MIParray (via r-universe)

September 10, 2024

Type Package

Title Creation of Arrays by Mixed Integer Programming

Date 2023-08-21

Version 1.0-1

Author Ulrike Groemping

Maintainer Ulrike Groemping <ulrike.groemping@bht-berlin.de>

Description 'CRAN' packages 'DoE.base' and 'Rmosek' and non-'CRAN' package 'gurobi' are enhanced with functionality for the creation of optimized arrays for experimentation, where optimization is in terms of generalized minimum aberration. It is also possible to optimally extend existing arrays to larger run size. The package writes 'MPS' (Mathematical Programming System) files for use with any mixed integer optimization software that can process such files. If at least one of the commercial products 'Gurobi' or 'Mosek' (free academic licenses available for both) is available, the package also creates arrays by optimization. For installing 'Gurobi' and its R package 'gurobi', follow instructions at <https://www.gurobi.com/products/gurobi-optimizer/> and <https://www.gurobi.com/documentation/7.5/refman/r_api_overview.html> (or higher version). For installing 'Mosek' and its R package 'Rmosek', follow instructions at <https://www.mosek.com/downloads/> and <https://docs.mosek.com/8.1/rmosek/install-interface.html>, or use the functionality in the stump CRAN R package 'Rmosek'.

Imports stats, methods, combinat, DoE.base

Enhances gurobi, Rmosek (>= 8.0)

Suggests slam (>= 0.1-9), Matrix (>= 1.1.0)

License GPL (>= 2)

Encoding UTF-8

NeedsCompilation no

Date/Publication 2023-08-21 12:10:10 UTC

Repository https://ugroempi.r-universe.dev

RemoteUrl https://github.com/cran/DoE.MIParray

RemoteRef HEAD

RemoteSha 9204f888d8dee2dfb746d2db78757440465662d6

Contents

`DoE.MIParray-package` *Package to Create a MIP Based Array*

Description

'CRAN' packages 'DoE.base' and 'Rmosek' and non-'CRAN' package 'gurobi' are enhanced with functionality for the creation of optimized arrays for experimentation, where optimization is in terms of generalized minimum aberration. It is also possible to optimally extend existing arrays to larger run size. The package writes 'MPS' (Mathematical Programming System) files for use with any mixed integer optimization software that can process such files. If at least one of the commercial products 'Gurobi' or 'Mosek' (free academic licenses available for both) is available, the package also creates arrays by optimization. For installing 'Gurobi' and its R package 'gurobi', follow instructions at <https://www.gurobi.com/products/gurobi-optimizer/> and <https://www.gurobi.com/documentation/7.5/refman/r_api_overview.html> (or higher version). For installing 'Mosek' and its R package 'Rmosek', follow instructions at <https://www.mosek.com/downloads/> and <https://docs.mosek.com/8.1/rmosek/install-interface.html>, or use the functionality in the stump CRAN R package 'Rmosek'.

Details

The package creates an array with specified minimum resolution and optimized word length pattern, using mixed integer programming (MIP). This is a step in generalized minimum aberration (GMA) according to Xu and Wu (2001), which is a way to minimize the confounding potential in a factorial design, as measured by the generalized word length pattern (GWLP). Reduction of short words has priority over reduction of long words, because it pertains to confounding of lower order interactions, which is assumed to be more severe than confounding of higher order interactions.

There is always one word of length zero. An array is said to have strength t (resolution R=t+1), if it has no words of lengths 1,...,t, but has words of length R=t+1; in that case, t-factor interactions are not confounded with the overall mean, (t-1)-factor interactions are not confounded with main effects, and so forth. Groemping and Xu (2014) provided an interpretation of the number of shortest words (i.e. words of length R) in terms of coefficients of determination of linear models with main effects model matrix columns on the LHS and full factorial t-factor models on the right-hand side; for example, in an array of strength 2 with three 3-level factors, the number of words of length 3 is the sum of the two R^2 values obtained from regressing the two main effects model matrix columns of one of the factors on a full model in the other two factors, provided main effect model matrix columns are coded orthogonally (to each other and the overall mean). GMA considers a design better than another one, if it has larger strength or resolution. In case of ties, a design is better if it has fewer shortest words; further ties are resolved by comparing words of lengths t+2, t+3, ...

Any array found by a function of this package will have the requested resolution (if not possible, an error will be thrown). If desired (default, but can be suppressed), optimization of the number of shortest words will be attempted. If `kmax` is chosen larger than the resolution, optimization of longer words will also be attempted. Mixed integer optimization is very resource-intensive and often fails to provide a confirmed optimum (see also below). Choosing `kmax` larger than the resolution is therefore advisable for very small problems only; in most cases, one should attempt an optimization of the number of shortest words only, or even suppress that by `find.only=TRUE`. Only after this has been achieved, possibly with several sequential attempts, a subsequent attempt to improve the number of longer words should be undertaken.

Functions `gurobi_MIParray` and `mosek_MIParray` create an array from scratch (`start=NULL` and `forced=NULL`), by trying to improve a starting array specified with the argument `start`, or by restricting a portion of the array to correspond to a pre-existing array with the argument `forced`. If no starting array is given, the functions initially obtain one by solving a linear optimization problem for obtaining a design with the requested resolution. Subsequently, the number of shortest words of the starting array is optimized, followed by the numbers of words of length up to `kmax`.

Where functions `gurobi_MIParray` and `mosek_MIParray` do not easily find an optimal array, it may be worthwhile to consider using functions `gurobi_MIPsearch` and `mosek_MIPsearch`, which can search over different orderings of the factor levels; these have been provided because a brief search for a fortunate level orderings may be successful where a very long search for an unfortunate level ordering fails.

The algorithm implemented in the package is explained in Groemping and Fontana (2019); it is a modification of Fontana (2017). Modifications include enforcing the requested resolution via a linear optimization step (much faster than sequentially optimizing all those word lengths until they are zeroes), and using a result on coding invariance from Groemping (2018) for a more parsimonious formulation of constraints. Mixed integer programming can use a lot of time and resources; in particular, the confirmation of the optimality of a solution that has been found can take prohibitively long, even if the optimal solution itself has been found fast (which is also not necessarily so). Functions `gurobi_MIPcontinue` and `mosek_MIPcontinue` can be used to continue optimization for larger problems, where optimization was previously aborted, e.g. due to a time limit (however, a lot of effort is lost and has to be repeated when continuing a previous attempt). Note that it is possible to continue an optimization effort started with Gurobi using Mosek and vice versa, because the functions `gurobi_MIPcontinue` and `mosek_MIPcontinue` can convert problems using the internal functions `mosek2gurobi` and `gurobi2mosek`. Groemping (2020) explains the package itself.

The solver functions in the package use the commercial solvers Mosek and/or Gurobi, which provide free academic licenses. These solvers and their corresponding vendor-provided R packages

have to be installed for using the package's solver functions. For users who do not have access to these but to other solvers, the package provides export functions to mps files that can be read by other solvers, for example by IBM CPLEX (`write_MPSMIQP`, `write_MPSILPlist`).

Warning

Escaping from a Gurobi run will most likely be unsuccessful, might leave R in an unstable state, and usually fails to release the entire CPU usage; thus, one should think carefully about the affordable run times.
Mosek can usually be escaped using the <ESC> key, and one can even hope to get a valid output. However, after such escapes, it is also advisable to use **RMosek**'s internal clean function (`Rmosek:::mosek_clean()`), since computer instabilities after repeated escapes from Mosek have been observed (the package functions execute the `mosek_clean` command after conducting Mosek runs).

Installation

Gurobi and Mosek need to be separately installed; please follow vendors' instructions; it is necessary to obtain a license; for academic use, free academic licenses are available in both cases.

Both Gurobi and Mosek provide R packages (**gurobi** and **Rmosek**) for accessing the software, and they also provide instructions on their installation.

For Gurobi, the R package **gurobi** is provided with the software installation files and can simply be installed like usual for a non-Web package.

For Mosek, there is a CRAN package **Rmosek**, which is a stump only and can be used for installing the suitable package **Rmosek** from the Mosek website. Its installation requires compilation from source. Several prerequisites are needed, basically the same ones needed for compiling packages. Make sure to set up the R environment for this purpose (for Windows, see https://cran.r-project.org/bin/windows/Rtools/; you may have to set some paths yourself; I am not familiar with the proper process for other platforms). If this is accomplished and package **Matrix** is at the latest level (as recommended in the **Rmosek** online documentation), install the CRAN package **Rmosek**, whose purpose it is to support installation of the appropriate **Rmosek** package for your platform and Mosek version. Once this CRAN package is available, run function `mosek_attachbuilder`, specifying the appropriate path as pointed out in the function's documentation, and subsequently run the custom-made function `install.rmosek`. After this activity (if all went well), the CRAN package **Rmosek** will have been replaced by a working version of package **Rmosek** whose version number depends on your version of Mosek. In case of problems, running the `install.packages` command provided in the **Rmosek** online documentation for your version of Mosek (Mosek ApS 2017b) may also be worth a try; if all else fails, you may have to contact **Rmosek** support.

Don't be confused by Mosek ApS's somewhat strange communication regarding the package **Rmosek**: the CRAN version (stump for the purpose of supporting installation of the working version) has its own separate numbering that currently starts with "1". The working **Rmosek** packages have version numbers whose first digit is kept in sync with the Mosek major version number; apart from that, version numbers of **Rmosek** versions and Mosek versions are not kept in sync; for example, the current version (July 13 2019) of package Rmosek for Mosek version 8 is 8.0.69, while the Mosek version is 8.1.0.81 (the revision versions of Mosek change quite frequently). Whenever Mosek itself is re-installed (at least with a change of minor version like 8.0 to 8.1), the **Rmosek** sources must be re-compiled, even if their version has not changed; this is presumably why Mosek ApS speak of a

version number for the binary that corresponds to the Mosek version number, while the sources keep their version number (somewhat confusing for the R community). From within R, version numbers can be queried by `packageVersion("Rmosek")` and `Rmosek::mosek_version()`, respectively.

Note

The package is not meant for situations, for which a full factorial design would be huge; the mixed integer problem to be solved has at least `prod(nlevels)` binary or general integer variables and will likely be untractable, if this number is too large. (For extending an existing designs, since some variables are fixed, the limit moves out a bit.)

Author(s)

Ulrike Groemping

References

Fontana, R. (2017). Generalized Minimum Aberration mixed-level orthogonal arrays: a general approach based on sequential integer quadratically constrained quadratic programming. *Communications in Statistics – Theory Methods* **46**, 4275-4284.

Groemping, U. and Xu, H. (2014). Generalized resolution for orthogonal arrays. *The Annals of Statistics* **42**, 918-939.

Groemping, U. (2018). Coding Invariance in Factorial Linear Models and a New Tool for Assessing Combinatorial Equivalence of Factorial Designs. *Journal of Statistical Planning and Inference* **193**, 1-14.

Groemping, U. and Fontana R. (2019). An Algorithm for Generating Good Mixed Level Factorial Designs. *Computational Statistics & Data Analysis* **137**, 101-114.

Groemping, U. (2020). DoE.MIParray: an R package for algorithmic creation of orthogonal arrays. *Journal of Open Research Software*, **8: 24**. DOI: https://doi.org/10.5334/jors.286

Gurobi Optimization Inc. (2018). Gurobi Optimizer Reference Manual. https://www.gurobi.com:443/documentation/.

Mosek ApS (2017a). MOSEK version w.x.y.z documentation. Accessible at: https://www.mosek.com/documentation/. This package has been developed using version 8.1.0.23 (accessed August 29 2017).

Mosek ApS (2017b). MOSEK Rmosek Package 8.1.y.z. https://docs.mosek.com/8.1/rmosek/index.html. *!!! In normal R speak, this is the documentation of the Rmosek package version 8.0.69, when applied on top of the Mosek version 8.1.y.z (this package has been devoloped with Mosek version 8.1.0.23 and will likely not work for Mosek versions before 8.1). !!! (accessed August 29 2017)*

Xu, H. and Wu, C.F.J. (2001). Generalized minimum aberration for asymmetrical fractional factorial designs. *Annals of Statistics* **29**, 549-560.

Examples

```
## Not run:
## ideal sequence of optimization problems
## shown here for Mosek,
```

```
## for Gurobi analogous, if necessary increasing maxtime to e.g. 600 or 3600 or ...

## very small problem
plan <- mosek_MIParray(16, rep(2,6), resolution=4, kmax=6)

## an example approach for a larger problem
## optimize shortest word length
plan3 <- mosek_MIParray(24, c(2,4,3,2,2,2,2), resolution=3, maxtime=20)
## feasible solution was found, no confirmed optimum, 7/3 words of length 3
## try to optimize further or confirm optimality (improve=TRUE does this),
##                      give it 10 minutes
plan3b <- mosek_MIPcontinue(plan3, improve=TRUE, maxtime=600)
##     no improvement has been found, and the gap is still very large
##     (the time limit makes the result non-deterministic, of course,
##      because it depends on the computer's power and availability of its resources)

## For large problems, it cannot be expected that a *confirmed* optimum is found.
## Of course, one can put more effort into the optimization, e.g. by running overnight.
## It is also advisable to compare the outcome to other ways for obtaining a good array,
##    e.g. function oa.design from package DoE.base with optimized column allocation.
require(DoE.base)
show.oas(nruns=24, nlevels=c(2,4,3,2,2,2,2), show=Inf)
GWLP(plan_oad <- oa.design(nruns=24, nlevels=c(2,4,3,2,2,2,2), col="min34"))
## here, plan3b has a better A3 than plan_oad

## one might also try to confirm optimality by switching to the other optimizer
plan3c <- gurobi_MIPcontinue(plan3b, improve=TRUE, maxtime=600, MIPFocus=3)
   ## focus on improved bound with option MIPFocus
   ## still same value with very large gap after running this
   ## thus, now assume this as best practically feasible value

## one might now try to improve words of length 4 (improve=FALSE turns to the next word length)
plan4 <- mosek_MIPcontinue(plan3b, improve=FALSE, maxtime=600)
   ## this does not yield any improvement
   ## working on longer words is not considered worthwhile
   ## thus, plan3 or plan3b are used for pragmatic reasons,
   ## without confirmed optimality

## End(Not run)
```

---

dToCount.Rd *Functions to switch between count and array representation of an array*

Description

dToCount rearranges an array into count vector format. countToDmixed rearranges a count vector representation into an array.

Usage

```
dToCount(d, nlevels=NULL, startfrom1=FALSE)
countToDmixed(nlevels, count)
ff(...)
```

Arguments

d	an array (matrix with runs as rows and factors as columns)
nlevels	vector of integers (>=2); numbers of factor levels
startfrom1	logical; if TRUE, values are from 1 to nlevels; the default is that values are from 0 to nlevel -1
count	vector of prod(nlevels) nonnegative integers; the number of times the respective run of the full factorial in lexicographic order (as produced by ff(nlevels)) occurs in the array
...	vector of integers (>=2) or the integers themselves, separated by commata

Details

dToCount transforms an array into count representation. If all array columns contain all potential factor levels, nlevels does not need to be specified. Otherwise, nlevels is needed.

countToDmixed transforms the count representation of an array (counts refer to the rows of ff(nlevels)) into an array

Value

dToCount produces a vector of length prod(nlevels),
countToDmixed produces a matrix with sum(count) rows and length(nlevels) columns,
countToDmixed produces a matrix with prod(nlevels) rows and length(nlevels) columns.

Note

The size of the full factorial design (produced with ff) is a limiting factor for using the functionality of this package.

Author(s)

Ulrike Groemping

References

Groemping, U. (2020). DoE.MIParray: an R package for algorithmic creation of orthogonal arrays. *Journal of Open Research Software*, **8: 24**. DOI: https://doi.org/10.5334/jors.286

Examples

```
d <- ff(c(2,2,4))[1:6,]   ## first six rows of the full factorial only
d
## the count vector must have 2*2*4=16 elements,
## the first six must be 1, the last ten must be zero
dToCount(d)               ## does not produce the desired result,
                          ## because the first column of d
                          ## does not contain both levels
(d_as_count <- dToCount(d, nlevels=c(2,2,4)))
                          ## corresponds to the above way of creating d
dToCount(d, nlevels=c(2,2,5))  ## would correspond to a different reality,
                               ## where the third factor has in fact 5 levels,
                               ## of which only four are in the array
countToDmixed(c(2,2,4), d_as_count)
                          ## creates d from the count representation
```

---

functionsFromDoE.base *Functions from package DoE.base*

Description

These functions from DoE.base are exported from DoE.MIParray, because they are especially important for its use.

Usage

```
oa_feasible(nruns, nlevels, strength = 2, verbose = TRUE, returnbound = FALSE)
lowerbound_AR(nruns, nlevels, R, crit = "total")
length2(design, with.blocks = FALSE, J = FALSE)
length3(design, with.blocks = FALSE, J = FALSE, rela = FALSE)
length4(design, with.blocks = FALSE, separate = FALSE, J = FALSE, rela = FALSE)
length5(design, with.blocks = FALSE, J = FALSE, rela = FALSE)
contr.XuWu(n, contrasts=TRUE)
GWLP(design, ...)
SCFTs(design, digits = 3, all = TRUE, resk.only = TRUE, kmin = NULL, kmax = ncol(design),
   regcheck = FALSE, arft = TRUE, cancors = FALSE, with.blocks = FALSE)
ICFTs(design, digits = 3, resk.only = TRUE, kmin = NULL, kmax = ncol(design),
   detail = FALSE, with.blocks = FALSE, conc = TRUE)
```

Arguments

nruns	see **DoE.base**
nlevels	see **DoE.base**
strength	see **DoE.base**
verbose	see **DoE.base**
returnbound	see **DoE.base**

R	see **DoE.base**
crit	see **DoE.base**
design	see **DoE.base**
with.blocks	see **DoE.base**
J	see **DoE.base**
rela	see **DoE.base**
n	see **DoE.base**
contrasts	see **DoE.base**
separate	see **DoE.base**
digits	see **DoE.base**
all	see **DoE.base**
resk.only	see **DoE.base**
kmin	see **DoE.base**
kmax	see **DoE.base**
regcheck	see **DoE.base**
arft	see **DoE.base**
cancors	see **DoE.base**
detail	see **DoE.base**
conc	see **DoE.base**
...	see **DoE.base**

Details

for documentation of the functions, see the links under "See also"

Value

for documentation of the functions, see the links under "See also"

Author(s)

Ulrike Groemping

References

for documentation of the functions, see the links under "See also"

See Also

See also `oa_feasible`, `lowerbound_AR`, `length2`, `length3`, `length4`, `length5`, `GWLP`, `SCFTs`, `ICFTs`.

Examples

```
oa_feasible(24, c(2,3,4,6),2)
lowerbound_AR(24, c(2,3,4,6),2)
```

mosek2gurobi *Functions to recast quadratically constrained MIP in different format, and class qco*

Description

The functions recast a Mosek model into Gurobi format and vice versa, for use with objects of class qco from package DoE.MIParray. The class is also documented here.

Usage

```
mosek2gurobi(qco, ...)
gurobi2mosek(qco, ...)
```

Arguments

qco	a mixed integer optimization problem of class qco, as generated from package **DoE.MIParray**
...	not used so far

Details

The functions treat the special qco objects created by package **DoE.MIParray**: these are minimization problems with linear equality constraints and possibly conic quadratic constraints, as suitable for the problems treated in **DoE.MIParray**.

Class qco objects on their own only occur as interim results of the optimization functions mosek_MIParray, mosek_MIPcontinue, gurobi_MIParray or gurobi_MIPcontinue. Where it might be useful, the class link[DoE.base]{oa} output objects of the optimization functions contain an attribute MIPinfo of class qco. For reducing the size of an object that is not going to be used for further improvement, the following command can be run for extracting the the useful information content from the qco object and replacing the large MIPinfo attribute with this much smaller object:
`attr(obj, "MIPinfo") <- attr(obj, "MIPinfo")$info`
Make sure to only run this command if MIPinfo attribute is indeed of class qco and further optimization is not intended.

Value

an object of S3 class qco (see Details section)

Author(s)

Ulrike Groemping

See Also

See also as mosek_MIParray, gurobi_MIParray.

mosek_MIParray *Functions to Create a MIP Based Array Using Gurobi or Mosek*

Description

The functions create an array with specified minimum resolution and optionally also optimized word length pattern based on mixed integer programming with the commercial software Gurobi (free academic license available) or Mosek (free academic license available). Creation is done from scratch, or using a user-specified starting value, or extending an existing array. Important: Installation of Gurobi and/of Mosek as well as the corresponding R packages is necessary. The R package gurobi comes with the software, the current version of the R package Rmosek has to be obtained from vendor's website (CRAN version is outdated!).

Usage

```
mosek_MIParray(nruns, nlevels, resolution = 3, kmax = max(resolution, 2),
   distinct = TRUE, detailed = 0, start=NULL, forced=NULL, find.only = FALSE,
   maxtime = Inf, nthread=2, mosek.opts = list(verbose = 10, soldetail = 1),
   mosek.params = list(dparam = list(LOWER_OBJ_CUT = 0.5, MIO_TOL_ABS_GAP = 0.2,
      INTPNT_CO_TOL_PFEAS = 1e-05, INTPNT_CO_TOL_INFEAS = 1e-07),
      iparam = list(PRESOLVE_LINDEP_USE="OFF", LOG_MIO_FREQ=100)))
gurobi_MIParray(nruns, nlevels, resolution = 3, kmax = max(resolution, 2),
  distinct = TRUE, detailed = 0, start=NULL, forced=NULL, find.only = FALSE,
  maxtime = 60, nthread = 2, heurist=0.5, MIQCPMethod=0, MIPFocus=1,
  gurobi.params = list(BestObjStop = 0.5, LogFile=""))
```

Arguments

nruns	positive integer; number of runs
nlevels	vector of integers (>=2); numbers of factor levels
resolution	positive integer; the minimum resolution requested
kmax	integer, kmax >= resolution and kmax >= 2 are required; the largest number of words to be optimized (default: kmax = resolution). (If find.only=TRUE, optimization of numbers of words is suppressed.)
distinct	logical; if TRUE (default), restricts counting vector to 0/1 entries, which means that the resulting array is requested to have distinct rows; otherwise, duplicate rows are permitted, i.e. the counting vector can have arbitrary non-negative integers. Designs with distinct runs are usually better; in addition, binary variables are easier to handle by the optimization algorithm. Nevertheless, there are occasions where a better array is found faster with option distinct=FALSE, even if it has distinct rows.
detailed	integer (default 0); determines the output detail: positive values imply inclusion of a problem and solution history (attribute history), values of at least 3 add the lists of optimization matrices (Us and Hs, attribute matrices).

`start` for `resolution` > 1 only;
a starting value for the algorithm: can be a array matrix with entries 1 to number of levels for each column, or a counting vector for the full factorial in lexicographic order; if specified, `start` must specify an array with the appropriate number of rows and columns, the requested resolution and, if `distinct` = `TRUE`, also contain distinct rows (matrix) or 0/1 elements only.

`forced` for `resolution` > 1 only;
runs to force into the solution design; can be given as an array matrix with the appropriate number of columns and less than `nruns` rows or a counting vector for the full factorial in lexicographic order with sum smaller than `nruns`; if `distinct=TRUE`, `forced` must have distinct rows (matrix) or 0/1 elements only.

`find.only` logical; if FALSE (default), a design of the requested resolution is found, which is subsequently improved in terms of its word lengths up to words of length `kmax`; otherwise, the function only attempts to find an array of the requested resolution, without optimizing word lenghts.

`maxtime` the maximum run time in seconds per Gurobi or Mosek optimization request (the overall run time may become (much) larger); in case of conflict between `maxtime` and an explicit timing request in `gurobi.params$TimeLimit` or `mosek.params$dparam$$MIO_M` the stricter request prevails; the default values differ between Gurobi (60) and Mosek (Inf), because Mosek runs can be easily escaped, while Gurobi runs cannot.

`nthread` the number of threads (=cores) to use; there are also the Mosek parameter `NUM_THREADS` and the Gurobi parameter `Threads`; in case of conflict, the smaller request prevails. For using Gurobi's or Mosek's default (which is in most cases the use of all available cores), choose `nthread=0`.
CAUTION: `nthread` should not be chosen larger than the available number of cores. Gurobi warns that performance will deteriorate, but was observed to perform OK. For Mosek, performance will strongly deteriorate, and for extreme choices the R session might even crash (even for small problems)!

`mosek.opts` list of Mosek options; these have to be looked up in Mosek documentation

`mosek.params` list of mosek parameters, which can have the list-valued elements `dparam`, `iparam` and/or `sparam`; their use has to be looked up in the RMosek documentation. The arguments `maxtime` and `nthread` correspond to the `dparam$MIO_MAX_TIME` and `iparam$NUM_THREADS` specifications. Conflicts are resolved as stated in their documentation.

The element `dparam$LOWER_OBJ_CUT` can be used to incorporate a best bound found in an earlier successless optimization attempt; per default, it is set to 0.5, since the target function can take on integer values only and cannot be negative.

If a valid starting value is not accepted by Mosek, it may be worthwhile to increase `dparam$INTPNT_CO_TOL_PFEAS`.

Users of Mosek versions 9 and higher may want to play with `iparam$MIO_SEED`, which was introduced as a new parameter with Mosek version 9 (default: 42); different seeds modify the path taken through the search space. Varying the seed may be an alternative to searching over different level orderings with function `mosek_MIPsearch`.

Note that a user specified `mosek.params` should always contain the specifications shown under Usage. Exceptions: `LOWER_OBJ_CUT` is always specified to

be at least 0.5, i.e. this option can be safely omitted without loosing anything, and intentional changes can of course be made.

`heurist` the proportion heuristics time used by Gurobi in quadratic objective optimization (default 0.5; Gurobi default is 0.05); there is also the Gurobi parameter Heuristics; in case of conflict, the larger request prevails; the setting for heurist is deactivated for the initial linear problem which is always run with the Gurobi default. It can be worthwhile playing with this option for improving the run time for certain settings; for example, with `nruns=48` and `nlevels=c(2,2,3,4,4)`, `heurist=0.05` performs better than the default 0.5.

`MIQCPMethod` the method used by Gurobi for quadratically constrained optimization (default 0; other possibilities -1 (Gurobi decides) or 1); there is also the Gurobi parameter MIQCPMethod; in case of conflict, the method is set to "0"; this choice is made because it proved beneficial in many cases explored (although there also were a few cases which fared better with Gurobi's default).

`MIPFocus` the strategy used by Gurobi for quadratically constrained optimization (default 1: focus on finding good feasible solutions fast; other possibilities: 0 (Gurobi decides/compromise), 2 or 3 (focus on increasing the lower bound fast)); there is also the Gurobi parameter MIPFocus; in case of conflict, MIPFocus is set to "0"; the setting for MIPFocus is deactivated for the initial linear problem which is always run with the Gurobi default.

`gurobi.params` list of gurobi parameters; these have to be looked up in Gurobi documentation; the arguments `maxtime`, `heurist`, `MIQCPMethod` and `MIPFocus` refer to the Gurobi parameters "TimeLimit", "Heuristics", "MIQCPMethod" and "MIPFocus", respectively. See their documentation for what happens in case of conflict.

The Gurobi parameter `BestObjStop` can be used to incorporate a best bound found in an earlier successless optimization attempt; per default, it is set to 0.5, since the objective function can take on integer values only and cannot be negative.

Details

The functions initially solve a linear optimization problem for obtaining a design with the requested resolution; if a start value is provided, this step is skipped; if `find.only=TRUE`, the functions stop after this step. If `find.only=FALSE`, the number of shortest words is optimized, followed by the numbers of words of length up to `kmax`. The argument `forced` allows to specify an existing array that is to be extended (e.g. to double or triple size; extension by a small number of runs will usually not be possible) in an optimized way.

For all but very small problems, it is likely advisable to choose `kmax` equal to the requested resolution (the default), and to proceed to longer words only after it has been made sure that the shortest word length has been optimized (as far as possible with reasonable effort). Further improvements of the same can be attempted by applying `gurobi_MIPcontinue` or `mosek_MIPcontinue` to the result object returned by the function. Note that it is possible to switch from using Mosek to using Gurobi or vice versa. An example for an optimization sequence can be found in the package overview at `DoE.MIParray`.

In case of long run times, escaping from the gurobi run will most likely be unsuccessful and might even leave R in an unstable state; thus, one should think carefully about the affordable run times. On

the contrary, it should usually be doable to escape a Mosek run; the remaining code of the function will still be executed and will return the final state.

Besides the run time, the number of threads is a very important resource control parameter. The default assumes that the user wants to use two of the computer's multiple cores. For using Gurobi's or Mosek's default (which is in most cases the use of all available cores), choose `nthread=0`.

The default Gurobi parameters have been chosen after systematic experimentation with a limited set of scenarii und Gurobi version 7.5.1. The choice of `MIQCPMethod=0` was instrumental in many cases; changing it to -1 may occasionally be tried. The MIPFocus parameter also appears beneficial in many cases; changing it to 0 (leave choice to Gurobi) can be an option; it was slightly better than choice 1 for mixed level designs with relatively small run sizes, while choice "1" was substantially better for the other cases. The heuristics proportion has been chosen as 0.5, because this choice seemed the best compromise for the situations considered. Note, however, that these parameters deteriorate performance for very simple cases, e.g. the test cases of Fontana (2017). For such cases, using `MIQCPMethod=-1`, `MIPFocus=0` and `heurist=0.05` will be preferrable; the defaults were chosen in this way, since doubling or even tripling very short run times was decided to be less detrimental than making more difficult problems completely intractable.

For Gurobi, several optimization parameters are switched off for the initial linear optimization step: the parameters `Heuristics` and `MIPFocus` are reset to their defaults (0.05 and 0).

Gurobi always stores the file "gurobi.log" in the working directory; even if storage of the log is suppressed with the default option `LogFile=""` or directed to another location by specifying a path, the default file "gurobi.log" is created and filled with a small amount of content. Thus, make sure to use a different file name when intentionally storing some log.

For Mosek, storing log output can be accomplished by directing the printed output to a suitable storing location. Note that the setting `iparam$LOG_MIO_FREQ = 100` reduces the frequency of printing a log line for branch-and-cut optimization by the factor 10 versus the default. Parallelization in Mosek is not well-protected against interference from screen activity (for example). Thus, one should switch off logging to screen or otherwise, when working with many (all available) threads in parallel (`LOG=0` instead of `LOG_MIO_FREQ = 100` in the list `iparam`).

Value

an array of class `oa`, possibly with the following attributes: `MIPinfo`, which is either an object of class `qco` or a simple list with information (which would be the `info` element of the object of class `qco` in case the last optimization was not successful), `history` as a list of problem and solution lists, and `matrices` as a list of matrix lists. Presence or absence of `history` and `matrices` is controlled by option `detailed`, while `MIPInfo` is present if the optimization can be potentially improved by improving the last step (stop because of time limit and not because of optimal value) or by improving the number of longer words.

Installation

Gurobi and Mosek need to be separately installed; please follow vendors' instructions; see also `mosek_MIPsearch` for more comments.

Note

The functions are not meant for situations, for which a full factorial design would be huge; the mixed integer problem to be solved has at least `prod(nlevels)` binary or general integer variables

and will likely be untractable, if this number is too large. (For extending an existing designs, since some variables are fixed, the limit moves out a bit.)

Please be aware that escaping a Gurobi run will be not unlikely to leave the computer in an unstable situation. If function `gurobi_MIParray` is successfully interrupted by the <ESC> key or <Ctrl>-<C>, it will usually be necessary to restart R in order to free all CPU usage.

If a Mosek run is interrupted by the <ESC> key, it is advised to execute the command `Rmosek::mosek_clean()` afterwards; this may help prevent problems from unclean closes of mosek runs.

Author(s)

Ulrike Groemping

References

Fontana, R. (2017). Generalized Minimum Aberration mixed-level orthogonal arrays: a general approach based on sequential integer quadratically constrained quadratic programming. *Communications in Statistics – Theory Methods* **46**, 4275-4284.

Groemping, U. and Fontana R. (2019). An Algorithm for Generating Good Mixed Level Factorial Designs. *Computational Statistics & Data Analysis* **137**, 101-114.

Groemping, U. (2020). DoE.MIParray: an R package for algorithmic creation of orthogonal arrays. *Journal of Open Research Software*, **8: 24**. DOI: https://doi.org/10.5334/jors.286

Gurobi Optimization Inc. (2017). Gurobi Optimizer Reference Manual. https://www.gurobi.com:443/documentation/.

Mosek ApS (2017a). MOSEK version w.x.y.z documentation. Accessible at: https://www.mosek.com/documentation/. This package has been developed using version 8.1.0.23 (accessed August 29 2017).

Mosek ApS (2017b). MOSEK Rmosek Package 8.1.y.z. https://docs.mosek.com/8.1/rmosek/index.html. *!!! In normal R speak, this is the documentation of the Rmosek package version 8.0.69 (or whatever comes next), when applied on top of the Mosek version 8.1.y.z (this package has been devoloped with Mosek version 8.1.0.23 and will likely not work for Mosek versions before 8.1). !!! (accessed August 29 2017)*

See Also

See also `mosek_MIPsearch` and `gurobi_MIPsearch` for searching over `nlevels` orderings, `mosek_MIPcontinue` and `gurobi_MIPcontinue` for continuing an uncompleted optimization, `show.oas` from package **DoE.base** for catalogued orthogonal arrays, and `oa_feasible` from package **DoE.base** for checking feasibility of requested array strength (resolution - 1) for combinations of `nruns` and `nlevels`.

Examples

```
## Not run:
## can also be run with gurobi_MIParray instead of mosek_MIParray
## there are of course better ways to obtain good arrays for these parameters
## (e.g. function FrF2 from package FrF2)
feld <- mosek_MIParray(16, rep(2,7), resolution=3, kmax=4)
feld
names(attributes(feld))
```

```
attr(feld, "MIPinfo")$info

## using a start value
start <- DoE.base::L16.2.8.8.1[,1:5]
feld <- mosek_MIParray(16, rep(2,5), resolution=4, start=start)

## counting vector representation of the start value could also be used
DoE.MIParray:::dToCount(start-1)
   ## "-1", because the function requires values starting with 0
   ## 32 elements for the full factorial in lexicographic order, 16 ones for the runs

## extending an existing array
force <- matrix(as.numeric(as.matrix(DoE.base::undesign(DoE.base::oa.design(L8.2.7)))), nrow=8)
feld <- mosek_MIParray(16, rep(2,7), resolution=3, kmax=4, forced=force)
attr(feld, "MIPinfo")$info

## End(Not run)
```

---

mosek_MIPcontinue *Functions to Continue Optimization from Stored State*

Description

These functions continue optimization for a MIP-based array from a stored state.

Usage

```
mosek_MIPcontinue(qco, improve = TRUE, maxtime = Inf, nthread = 2,
      mosek.opts = list(verbose = 10, soldetail = 1),
      mosek.params = list(dparam = list(LOWER_OBJ_CUT = 0.5,
      MIO_TOL_ABS_GAP = 0.2, INTPNT_CO_TOL_PFEAS = 1e-05,
      INTPNT_CO_TOL_INFEAS = 1e-07),
      iparam = list(PRESOLVE_LINDEP_USE="OFF", LOG_MIO_FREQ=100)))
gurobi_MIPcontinue(qco, improve = TRUE, maxtime = 60, nthread = 2,
      heurist = 0.05, MIQCPMethod = 0, MIPFocus = 0,
      gurobi.params =list(BestObjStop = 0.5, LogFile=""))
```

Arguments

qco	object of class qco, created by a function in the package; or object of class oa that has a qco object as its MIPinfo attribute
improve	logical; if TRUE (default), try to improve the already obtained solution for word length qco$info$last.k, otherwise optimize the next word length
maxtime	time in seconds for the optimization call; defaults differ for Mosek (Inf) and Gurobi (60), because a Mosek run can be easily escaped (<ESC>-key), contrary to a Gurobi run

nthread	number of cores to use (0=all cores) CAUTION: nthread should not exceed the available number of cores. Gurobi warns that performance might deteriorate. For Mosek, performance WILL strongly deteriorate, and for extreme choices the R session might even crash (even for small problems)!
heurist	for gurobi_MIPcontinue only: the percentage of time (number between 0 and 1) spent on heuristics
MIQCPMethod	for gurobi_MIPcontinue only: the choice of optimization method; the default "0" has been observed to be better than Gurobi's default for most cases (Gurobi version 7.5.1); "-1" leaves the choice to Gurobi, "1" chooses the other method
MIPFocus	for gurobi_MIPcontinue only: the choice of strategy; the default "0" leaves this choice to Gurobi; for finding better feasible solutions, "1" is recommended; for improving the speed of increasing the lower bound for eventually proving optimality, "3" can be tried
mosek.opts	mosek options
mosek.params	Mosek parameters
gurobi.params	Gurobi parameters

Details

Note that it is possible to continue optimization with Gurobi, if it was started with Mosek, and vice versa. The tool will transform the problem into the respective other format.

Usage of options is analogous to functions mosek_MIParray and gurobi_MIParray, respectively, where these are described in more detail.

For some applications, usability of mosek_MIPcontinue is hampered in Mosek versions up to 8 by the fact that Mosek's presolve routines identify additional integer variables and fail to recognise user-specified starting values for these that are not exactly integer-valued. According to Mosek ApS, this is scheduled to be remedied with Mosek Version 9 (version 9 is now available; I have not checked whether this was indeed fixed).

Value

an array of class link[DoE.base]{oa}, if not optimized to GMA with info for further continuation (see documentation of mosek_MIParray or gurobi_MIParray)

Author(s)

Ulrike Groemping

See Also

See also DoE.MIParray for examples of the role of the MIPcontinue functions, mosek_MIParray and gurobi_MIParray for more detail on the optimization arguments, mosek_MIPsearch and gurobi_MIPsearch for searching over nlevels orderings (which may be a very successful alternative to trying to improve an initial optimization based on a fixed nlevels vector).

mosek_MIPsearch *Functions to Search for optimum MIP Based Array Using Gurobi or Mosek*

Description

The functions search through different orderings of the nlevels vector with the goal to create an array with minimum resolution and optimized shortest word length. They create the orders and call gurobi_MIParray or mosek_MIParray for each order.

Usage

```
mosek_MIPsearch(nruns, nlevels, resolution = 3, maxtime = 60,
   stopearly=TRUE, listout=FALSE, orders=NULL,
   distinct = TRUE, detailed = 0, forced=NULL, find.only=TRUE,
   nthread=2, mosek.opts = list(verbose = 1, soldetail = 1),
   mosek.params = list(dparam = list(LOWER_OBJ_CUT = 0.5, MIO_TOL_ABS_GAP = 0.2,
      INTPNT_CO_TOL_PFEAS = 1e-05, INTPNT_CO_TOL_INFEAS = 1e-07),
      iparam = list(PRESOLVE_LINDEP_USE="OFF", LOG_MIO_FREQ=100)))
gurobi_MIPsearch(nruns, nlevels, resolution = 3, maxtime = 60,
   stopearly=TRUE, listout=FALSE, orders=NULL,
   distinct = TRUE, detailed = 0, forced=NULL, find.only=TRUE,
   nthread = 2, heurist=0.5, MIQCPMethod=0, MIPFocus=1,
   gurobi.params = list(BestObjStop = 0.5, OutputFlag=0))
```

Arguments

nruns	positive integer; number of runs
nlevels	vector of integers (>=2); numbers of factor levels
resolution	positive integer; the minimum resolution requested
maxtime	the maximum run time in seconds per Gurobi or Mosek optimization request (the overall run time may become (much) larger); in case of conflict between maxtime and an explicit timing request in gurobi.params$TimeLimit or mosek.params$dparam$$MIO_ the stricter request prevails; the default values differ between Gurobi (60) and Mosek (Inf), because Mosek runs can be easily escaped, while Gurobi runs cannot.
stopearly	logical; if TRUE, the search stops if the shortest word length hits the lower bound; set to FALSE if you want longer word lengths to be optimized among several choices with the same shortest word length
listout	logical; if TRUE, all experimental plans are stored, instead of only the best one; if stopearly=TRUE, listout=TRUE does not make sense
orders	NULL (in which case distinct level orders are automatically determined) or a list of level orders to be searched

`distinct` logical; if TRUE (default), restricts counting vector to 0/1 entries, which means that the resulting array is requested to have distinct rows; otherwise, duplicate rows are permitted, i.e. the counting vector can have arbitrary non-negative integers. Designs with distinct runs are usually better; in addition, binary variables are easier to handle by the optimization algorithm. Nevertheless, there are occasions where a better array is found faster with option `distinct=FALSE`, even if it has distinct rows.

`detailed` integer (default 0); determines the output detail: positive values imply inclusion of a problem and solution history (attribute `history`), values of at least 3 add the lists of optimization matrices (Us and Hs, attribute `matrices`).

`forced` for `resolution > 1` only;
runs to force into the solution design; can be given as an array matrix with the appropriate number of columns and less than `nruns` rows or a counting vector for the full factorial in lexicographic order with sum smaller than `nruns`; if `distinct=TRUE`, `forced` must have distinct rows (matrix) or 0/1 elements only.

`find.only` logical; if TRUE (default), the function only attempts to find an array of the requested resolution, without optimizing word lenghts; otherwise, a design of the requested resolution is found, which is subsequently improved in terms of its word lengths up to words of length `kmax`.

`nthread` the number of threads (=cores) to use; there are also the Mosek parameter `NUM_THREADS` and the Gurobi parameter `Threads`; in case of conflict, the smaller request prevails. For using Gurobi's or Mosek's default (which is in most cases the use of all available cores), choose `nthread=0`.
CAUTION: `nthread` should not be chosen larger than the available number of cores. Gurobi warns that performance will deteriorate, but was observed to perform OK. For Mosek, performance will strongly deteriorate, and for extreme choices the R session might even crash (even for small problems)!

`mosek.opts` list of Mosek options; these have to be looked up in Mosek documentation

`mosek.params` list of mosek parameters, which can have the list-valued elements `dparam`, `iparam` and/or `sparam`; their use has to be looked up in the RMosek documentation. The arguments `maxtime` and `nthread` correspond to the `dparam$MIO_MAX_TIME` and `iparam$NUM_THREADS` specifications. Conflicts are resolved as stated in their documentation.

The element `dparam$LOWER_OBJ_CUT` can be used to incorporate a best bound found in an earlier successless optimization attempt; per default, it is set to 0.5, since the target function can take on integer values only and cannot be negative.

If a valid starting value is not accepted by Mosek, it may be worthwhile to increase `dparam$INTPNT_CO_TOL_PFEAS`.

Users of Mosek versions 9 and higher may want to play with `iparam$MIO_SEED`, which was introduced as a new parameter with Mosek version 9 (default: 42); different seeds modify the path taken through the search space for a given level ordering; thus, varying seeds can also be the route to choose where searching over level orderings is not feasible.

Note that a user specified `mosek.params` should always contain the specifications shown under Usage. Exceptions: `LOWER_OBJ_CUT` is always specified to be at least 0.5, i.e. this option can be safely omitted without loosing anything, and intentional changes can of course be made.

`heurist` the proportion heuristics time used by Gurobi in quadratic objective optimization (default 0.5; Gurobi default is 0.05); there is also the Gurobi parameter Heuristics; in case of conflict, the larger request prevails; the setting for heurist is deactivated for the initial linear problem which is always run with the Gurobi default. It can be worthwhile playing with this option for improving the run time for certain settings; for example, with `nruns=48` and `nlevels=c(2,2,3,4,4)`, `heurist=0.05` performs better than the default 0.5.

`MIQCPMethod` the method used by Gurobi for quadratically constrained optimization (default 0; other possibilities -1 (Gurobi decides) or 1); there is also the Gurobi parameter MIQCPMethod; in case of conflict, the method is set to "0"; this choice is made because it proved beneficial in many cases explored (although there also were a few cases which fared better with Gurobi's default).

`MIPFocus` the strategy used by Gurobi for quadratically constrained optimization (default 1: focus on finding good feasible solutions fast; other possibilities: 0 (Gurobi decides/compromise), 2 or 3 (focus on increasing the lower bound fast)); there is also the Gurobi parameter MIPFocus; in case of conflict, MIPFocus is set to "0"; the setting for MIPFocus is deactivated for the initial linear problem which is always run with the Gurobi default.

`gurobi.params` list of gurobi parameters; these have to be looked up in Gurobi documentation; the arguments `maxtime`, `heurist`, `MIQCPMethod` and `MIPFocus` refer to the Gurobi parameters "TimeLimit", "Heuristics", "MIQCPMethod" and "MIPFocus", respectively. See their documentation for what happens in case of conflict.

The Gurobi parameter `BestObjStop` can be used to incorporate a best bound found in an earlier successless optimization attempt; per default, it is set to 0.5, since the objective function can take on integer values only and cannot be negative.

Details

The search functions have been implemented, because the algorithm's behavior may strongly depend on the order of factors in case of mixed level arrays. In many examples, Mosek quickly improved the objective function which then stayed constant for a long time; thus, it may be promising to run `mosek_MIPsearch` with `maxtime=60` (or even less). See also Groemping and Fontana (2019) for examples of successful applications of the search functionality.

Even though Gurobi was less successful as a search tool in the examples that were examined so far, it may be helpful for other examples.

The options suppress printed output from the optimizers themselves.

Mosek Version 9 has gained a seed argument (`iparam$MIO_SEED`, which implements the Mosek parameter `MSK_IPAR_MIO_SEED`). Playing with seeds in `mosek_MIParray` may be an alternative to using the search approach, because it may lead to different paths through the search space for a fixed ordering of the `nlevels` vector. So far, I have only very little experience with using seeds; user reports are very welcome.

Value

an array of class `oa` with the attributes added by `mosek_MIParray` or `gurobi_MIParray`, resp.
In addition, the attribute `optorder` contains the vector of level orders that yielded the best design; if `listout=TRUE`, also the attributes `orders` and `allplans`.

Objects with the attribute `allplans` are quite large. If the attribute is no longer needed, it can be removed from an object named `obj` (replace with the name of your object) by the command `attr(obj, "allplans") <- NULL`

Author(s)

Ulrike Groemping

References

Groemping, U. and Fontana R. (2019). An Algorithm for Generating Good Mixed Level Factorial Designs. *Computational Statistics & Data Analysis* **137**, 101-114.

Groemping, U. (2020). DoE.MIParray: an R package for algorithmic creation of orthogonal arrays. *Journal of Open Research Software*, **8: 24**. DOI: https://doi.org/10.5334/jors.286

See Also

See also `mosek_MIParray` and `gurobi_MIParray`, `oa_feasible` from package **DoE.base** for checking feasibility of requested array strength (resolution - 1) for combinations of `nruns` and `nlevels`, and `lowerbound_AR` from package **DoE.base** for a lower bound for the length R words in a resolution R array.

Examples

```
## Not run:
## can also be run with gurobi_MIParray instead of mosek_MIParray
## there are of course better ways to obtain good arrays for these parameters
## (e.g. function FrF2 from package FrF2)
oa_feasible(18, c(2,3,3,3,3), 2)  ## strength 2 array feasible
lowerbound_AR(18, c(2,3,3,3,3), 3)  ## lower bound for A3
## of course not necessary here, the design is found fast
feld <- mosek_MIPsearch(18, c(2,3,3,3,3), stopearly=FALSE, listout=TRUE, maxtime=30)
## even stopearly=TRUE would not stop, because the lower bound 2 is not achievable
feld
names(attributes(feld))
attr(feld, "optorder")
## even for this simple case, running optimization until confirmed optimality
## would be very slow

## End(Not run)
```

`print.oa` *Function to Print oa Objects with a Lot of Added Info*

Description

The function suppresses printing of voluminous info attached as attributes to oa objects.

Usage

```
## S3 method for class 'oa'
print(x, ...)
```

Arguments

x	the oa object to be printed
...	further arguments for default print function

Details

The function currently removes all attributes except origin, class, dim, dimnames before printing. If available, status information from the MIPinfo attribute is printed.
Additionally, the names of unusual attributes are printed. They can also be printed separately by running names(attributes(x)); to access an attribute, run attr(x, "MIPinfo"), for example.

Value

The function is used for its side effects and does not return anything.

Author(s)

Ulrike Groemping

See Also

See also print.default and str

---

write_MPSILPlist.Rd *Functions to create and write lists of (mixed) integer quadratic or linear problems related to orthogonal arrays*

Description

create_ILPlist creates a list of problems in Mosek formatting. write_MPSILPlist saves a list of integer linear problems as separate MPS files that are accompanied by a table of content txt file. write_MPSMIQP creates and writes a single quadratic mixed integer problem in MPS format. All functions work, even if neither Mosek nor Gurobi is available.

Usage

```
create_ILPlist(nruns, nlevels, resolution=3, distinct=TRUE,
              search.orders=TRUE, start=NULL, forced=NULL, orders=NULL)
write_MPSILPlist(prefix, qcolist, toc=TRUE)
write_MPSMIQP(prefix, nruns, nlevels, resolution=3, distinct=TRUE, start=NULL,
              forced=NULL, name="ImproveAR", commentline="* quadratic problem")
```

Arguments

nruns	positive integer; number of runs
nlevels	vector of integers (>=2); numbers of factor levels
resolution	positive integer; the minimum resolution requested
distinct	logical; if TRUE (default), restricts counting vector to 0/1 entries, which means that the resulting array is requested to have distinct rows; otherwise, duplicate rows are permitted, i.e. the counting vector can have arbitrary non-negative integers. Designs with distinct runs are usually better; in addition, binary variables are easier to handle by the optimization algorithm. Nevertheless, there are occasions where a better array is found faster with option distinct=FALSE, even if it has distinct rows.
search.orders	logical (default TRUE); determines whether a list of arrays for different level orderings is produced or a single array is output only
start	a starting value for the algorithm: can be a matrix with entries 1 to number of levels for each column, or a counting vector for the full factorial in lexicographic order; if specified, start must specify an array with the appropriate number of rows and columns, the requested resolution and, if distinct = TRUE, also contain distinct rows (matrix) or 0/1 elements only. start cannot be combined with search.orders=TRUE.
forced	for resolution > 1 only; runs to force into the solution design; can be given as an array matrix with the appropriate number of columns and less than nruns rows or a counting vector for the full factorial in lexicographic order with sum smaller than nruns; if distinct=TRUE, forced must have distinct rows (matrix) or 0/1 elements only.
orders	a list of level orderings to be considered; if orders is not specified but search.orders=TRUE, all distinct level orderings
prefix	file name prefix to be supplemented with a number and the .mps suffix later. In addition, a further file with a different suffix may be created (see details).
qcolist	list of ILP objects in Mosek notation (as e.g. produced by function create_ILPlist)
toc	logical (default TRUE) that indicates whether a table of content should be produced (txt file)
name	name to be written into the name field of the mps file
commentline	comment to be written directly underneath the name (comments have to start with a *)

Details

The functions do *not* do any problem solving, they serve the sole purpose of exporting problems so that they can be addressed by solvers other than Mosek or Gurobi. The target solver must be able to process the MPS format.

create_ILPlist creates a list with one or more integer linear problems for creating designs of at least the specified resolution. The problems are in Mosek formatting.

write_MPSILPlist writes a list produced by create_ILPlist to one or several MPS files, one file per list element. If not suppressed, a toc file is also created (suffix _toc.txt), which contains the number of runs, the target resolution and the elements of nlevels for each MPS file.

`write_MPSMIQP` creates and writes a problem for improving the number of shortest words, in a form that corresponds to general MPS format, without extensions by Mosek or Gurobi. The problem has a quadratic objective like in Groemping and Fontana (2019) Eq.(2Q) (instead of Eq.(2L), which is solved by Gurobi and Mosek).

The example section demonstrates on a small example how these functions can be used.

For `write_MPSMIQP`, a start array can be provided from a previous linear optimization (this is not enforced). If an admissible start array is provided, `write_MPSMIQP` initially prints the GWLP of that start array. Otherwise, it warns of inadmissibility. The start value cannot be stored in the MPS file. Instead, for a non-NULL start array, a separate file (suffix `.start`) is created, and users have to work out how they can make their solver use that start solution (which is stored in the form of a counting vector, see example section). Note that the availability of a start array can improve the ability of a solver to find an optimum solution. However, this is not always the case, there are also instances for which a better solution is found without providing a start solution.

For `write_MPSMIQP`, a lower bound for the objective value can substantially improve the run time, if the solution achieves that lower bound. The lower bound is not provided in the MPS file, and its detail depends on the optimizer's way to implement quadratic problems (see example section for how to obtain it). Users have to work out how to provide that bound to their optimizer.

Note that it can take a long time to write the mps files, if the problem has many variables (the number of variables is `prod(nlevels)`).

Value

Function `create_ILPlist` creates a list of one or more optimization problems, which can be used in a call to function `write_MPSILPlist`.

Function `write_MPSILPlist` returns a table of contents matrix for the written files. If `toc=TRUE`, that value is also written to a separate file with suffix `toc`.

Function `write_MPSMIQP` returns the start vector (in counting vector representation), if `start` is not NULL; that vector is also written to a separate file with suffix `start`.

Note

It can take an optimizer a long time to confirm optimality after finding an optimum. For the quadratic problem, it is therefore very beneficial to provide an optimal value to the algorithm, where possible (see example section).

The functions are not meant for situations, for which a full factorial design would be huge. Even though the functions do not solve anything, MPS files will be very large and writing them will be quite slow for such cases.

Author(s)

Ulrike Groemping

References

Groemping, U. and Fontana, R. (2019). An Algorithm for Generating Good Mixed Level Factorial Designs. *Computational Statistics & Data Analysis* **137**, 101-114. doi:10.1016/j.csda.2019.01.020.

Mosek ApS (2017a). MOSEK version w.x.y.z documentation. Accessible at: https://www.mosek.com/documentation/. This package has been developed using version 8.1.0.23 (accessed August 29 2017).

Mosek ApS (2017b). MOSEK Rmosek Package 8.1.y.z. https://docs.mosek.com/8.1/rmosek/index.html. *!!! In normal R speak, this is the documentation of the Rmosek package version 8.0.69 (or whatever comes next), when applied on top of the Mosek version 8.1.y.z (this package has been devoloped with Mosek version 8.1.0.23 and will likely not work for Mosek versions before 8.1). !!! (accessed August 29 2017)*

See Also

See also `create_MIQP` and `write_MPSILP` for internal functions that support these exported functions, and `dToCount` and `countToDmixed` for switching back and forth between an array and its counting vector representation.

Examples

```
#####################################################################
## an array and its counting vector

## arrays (starting the coding with 1)
## and their counting vectors can be used interchangeably
myarr <- cbind(c(1,1,1,1,1,1,1,2,2,2,2,2,2,2,2),
               c(1,1,1,1,2,2,2,2,1,1,1,1,2,2,2,2),
               c(1,2,3,4,1,2,3,4,1,2,3,4,1,2,3,4),
               c(1,5,3,7,2,6,4,8,8,4,6,2,7,3,5,1))

## we want to see it w.r.t. a 2,2,4,8 level full factorial
## determine the counting vector representation of this array
##     nlevels is needed,
##     because the third column of myarr has only 2 levels
(myarr_cv <- dToCount(myarr, nlevels=c(2,2,4,8), startfrom1=TRUE))

#####################################################################
## demo: counting vector represents the array runs
#####################################################################
## full factorial in lexicographic order
fullfac <- ff(2,2,4,8) +  1   ### ff levels start with 0
##
## pick the selected runs from fullfac
selfac <- fullfac[which(myarr_cv==1),]
##
## order both variants in the same way and compare them
## (in this case, they are equal without reordering)
ord1 <- DoE.base::ord(selfac)  ## order them
ord2 <- DoE.base::ord(myarr)   ## order them
selfac[ord1,] == myarr[ord2,]
#####################################################

#####################################################
## We go for an array in 16 runs with four factors in
```

```
## 2,2,4,8 levels.

## Is a strength 2 oa feasible?
##
oa_feasible(16, c(2,2,4,8), 2)  ## FALSE
##
## consequence: use resolution 2 (=strength 1),
##              minimize number of words of length 2

problemlist <- create_ILPlist(16, nlevels = c(2,2,4,8), resolution = 2)
length(problemlist) ## 12 distinct search orders
names(problemlist[[3]])
problemlist[[3]][-2]  ## ILP is too long for printing
problemlist1 <- create_ILPlist(16, nlevels = c(2,2,4,8),
                  resolution = 2, search.orders = FALSE)
                  ## only the pre-specified search order
problemlist2 <- create_ILPlist(16, nlevels = c(2,2,4,8),
                  resolution = 2, orders = list(c(2,2,4,8),
                                                c(8,2,4,2)))
                  ## the two specified search orders
## Not run:
write_MPSILPlist(prefix="miniprob", problemlist)
## writes miniprob01.mps, ..., miniprob12.mps and miniprob_toc.txt
write_MPSILPlist(prefix="miniprob", problemlist1, toc=FALSE)
## writes miniprob1.mps

## End(Not run)

## The MPS files can be read by various optimizers.
## The ILP problems aim for a feasible solution.
## Start values are possible, but usually not useful.
## The best solution (lowest target value) can be imported into R.

## the solution  a counting vector
## its format depends on the optimizer
## import it into R and calculated array from it
importedsol <- myarr_cv # for demo only
solarray <- countToDmixed(myarr_cv, nlevels=c(2,2,4,8))
##
## it is crucial to use the order of the levels
## that corresponds to the problem that the solver solved

GWLP(solarray)

#########################################################
## providing a lower bound for the number of
## length 2 words in a strength 1 (resolution 2) array
#########################################################
##
lowerbound_AR(nruns = 16, nlevels = c(2,2,4,8), R = 2)  # 1
##
## In this example, we have immediately hit on a solution
## with optimum A2-value (see GWLP)
```

```
#########################################################
## using a quadratic problem for optimizing A2
##
## Not run:
write_MPSMIQP("quadprob", 16, c(2,2,4,8), resolution=2)
## writes quadprob.mps

## End(Not run)

## Run time for solving the quadratic problem exported by write_MPSMIQP
## may substantially (!) benefit from providing the lower bound of the
## objective function, if that bound is attained.
##
## The lower bound for the minimum of the quadratic problem
##     created by write_MPSMIQP
##   is the lower bound for the word length, multiplied with n^2,
##     here 16 ^ 2 * 1 = 256,
##   or half that value,
##     depending on how the optimizer handles quadratic objectives.
#########################################################

## Depending on the optimizer, it is useful or even crucial to provide a
## starting value to write_MPSMIQP. This starting value can be obtained
## as the solution to a linear problem (that was exported using functions
## create_ILPlist and write_MPSILPlist).
```

Index