Package: CAs (via r-universe)

October 19, 2026

Type Package

Title Creating Covering Arrays

Version 0.24-4

Description Implements covering arrays, as proposed in Groemping (2025) <doi:10.1109/ICSTW64639.2025.10962463>. Uniform and mixed-level covering arrays are created and can be assessed for their coverage properties. The mathematical background of the constructions is provided in the technical reports Groemping (2026) <https://www1.bht-berlin.de/FB_II/reports/Report-2026-001.pdf> for the uniform arrays and Groemping (2025) <https://www1.bht-berlin.de/FB_II/reports/Report-2025-001.pdf> for the mixed-level constructions. The post-processing method by Nayeri, Colbourn, and Konjevod (2013) <doi:10.1016/j.ejc.2012.07.017> is also provided. In addition, there are preliminary tools for the creation of sequence covering arrays.

Depends R (>= 3.5.0)

Imports lhs, DoE.base, caverify (>= 0.2.0), sets, prepplot, primes, igraph, permutations, dplyr, curl, gmp, conf.design, sfsmisc, combinat

Suggests parallel, FrF2, testthat (>= 3.3.2)

License GPL-3

LazyData true

LazyDataCompression xz

Encoding UTF-8

URL https://github.com/ugroempi/CAs

BugReports https://github.com/ugroempi/CAs/issues

Roxygen list(markdown = TRUE)

Config/testthat/edition 3

Config/roxygen2/version 8.0.0

Config/pak/sysreqs libglpk-dev libgmp3-dev libxml2-dev libssl-dev

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

Date/Publication 2026-10-19 20:31:13 UTC

RemoteUrl https://github.com/ugroempi/cas

RemoteRef HEAD

RemoteSha 97741df18c711203e6c4882727a0aa5f4a981ea3

Contents

CAs-package . . . 3
add1CA . . . 10
bestCA . . . 11
ca12.2.8 . . . 14
CAEX . . . 16
CAEX_lineages . . . 19
checkcond1 . . . 21
CK_constrD . . . 22
CK_doubling . . . 23
CK_NRB . . . 25
CKRS_augment . . . 26
CKRScat . . . 28
colbournBigFrame . . . 29
compositCA . . . 30
conference_paley . . . 31
coverage . . . 34
coverageSeqCA . . . 37
cphfCA . . . 38
createDHF . . . 41
crossCAs . . . 42
crossSum . . . 43
CS_CK . . . 45
CS_CMMSSY . . . 46
CS_LCDST . . . 48
CS_MS . . . 50
cyclotomyCA . . . 53
derive . . . 55
DHF2CA . . . 57
DHHF2CA . . . 59
dpCA . . . 61
DWYERcat . . . 63
fuseBoseCA . . . 64
gen.next.cbn . . . 66
greedySeqCA_Kuhn . . . 67
greedySeqCA_naive . . . 69
greedySeqCA_TJ . . . 70
identify_needed . . . 72

int2poly . . . 73
juxtapose . . . 74
lower_bound_CA . . . 75
markflex . . . 76
MCAt . . . 79
MeagherStevensCombis . . . 83
miscCA . . . 86
MTVTRouxtypeCA . . . 87
N_Hartman74 . . . 89
N_KSK . . . 91
NISTcat . . . 92
Ns . . . 93
OD2 . . . 97
ODbasedCA . . . 97
one_to_howmany . . . 99
PALEYcat . . . 100
pcaCA . . . 101
PCAcat . . . 105
permvals . . . 107
powerCA . . . 109
powerCTcat . . . 110
print.ca . . . 112
print.CAcoverage . . . 113
projBoseCA . . . 113
readCA . . . 115
recBoseCA . . . 116
recursiveBoseHartman . . . 119
reduce_rows_complete . . . 121
SCA_Bose . . . 122
scphfCA . . . 123
SeqCA_Levenshtein . . . 125
simulated_annealing_sca . . . 127
SMC_SCPHFs . . . 130
swapvals . . . 131
TJcat . . . 132
WCS_CPHFs . . . 134
WKScat . . . 135

Index **137**

`CAs-package` *Overview of Package CAs*

Description

Implements covering arrays, as proposed in Groemping (2025) doi:10.1109/ICSTW64639.2025.10962463. Uniform and mixed-level covering arrays are created and can be assessed for their coverage properties. The mathematical background of the constructions is provided in the technical reports Groemping (2026) https://www1.bht-berlin.de/FB_II/reports/Report-2026-001.pdf for the uniform arrays and Groemping (2025) https://www1.bht-berlin.de/FB_II/reports/Report-2025-001.pdf for the mixed-level constructions. The post-processing method by Nayeri, Colbourn, and Konjevod (2013) doi:10.1016/j.ejc.2012.07.017 is also provided. In addition, there are preliminary tools for the creation of sequence covering arrays.

Details

This package constructs covering arrays, i.e., arrays that cover all t-ary combinations of a set of factors at least once. The focus is on mathematical constructions.

The goal is to implement as many constructions as possible that yield arrays with small numbers of runs, as evidenced by the Colbourn covering array tables, which do not provide the CAs themselves but pointers to which CA constructions yield the smallest known array for which setting.

All references of the package are listed in this file and referenced from the other documentation files.

Within the package, available CA constructions for specific situations can be queried using the function `Ns`, which provides sizes from available constructions, as well as the current known optimum size; it is not guaranteed that all implemented constructions are incorporated, as this is not trivial for constructions with intricate ingoing quantities. An analogous function `ks` is far less advanced, as the typical application situation presumably starts from a set of variables to be covered.

Function `bestCA` makes use of function `Ns` (with the above-stated limitations) and produces the best currently implemented design, including a download from internet libraries, where applicable.

Besides the construction functions, coverage properties of any array can be checked by function `coverage` and plotted by function `coverplot`.

So far, the following constructions for uniform CAs have been implemented:

- strength 2 2-level CAs by Kleitman and Spencer (1973) and Katona (1973) in function `KSK`,
- strength 3 to 6 2-level constructions via Paley matrices according to Colbourn (2015) in function `paleyCA`,
- orthogonal arrays from a Bose (1938) or Bush (1952) construction in functions `SCA_Bose` and `SCA_Busht`; the latter does one more column for strength 3 with `q` a power of 2, according to the Sherwood, Martirosyan and Colbourn (2006) example for the usage of permutation vectors.
- constructions based on cyclotomy (Colbourn 2010) in function `cyclotomyCA` for practitioners (based on arrays listed in the Colbourn tables), and function `cyc` for experts who know which prime power and construction type to request for which strength.
- Chateauneuf and Kreher (2002) doubling in function `CK_doublingCA` (workhorse function `CK_doubling`)
- strength 3 CAs for v=3,4,5 with up to 2v columns in 33, 88 or 185 runs (`miscCA`)
- strength 3 CAs for v=q+1 with q a prime power, and k<=v, based on augmenting an ordered design with further runs (Cohen, Colbourn and Ling 2003 and 2008)

- the Colbourn and Torres-Jimenez (2010) power construction in function powerCA
- the Sherwood, Martirosyan and Colbourn (2006) construction based on SCPHFs and permutation vectors, including not only the SCPHFs from the Sherwood et al. paper but also those provided by Walker and Colbourn (2009) and by Lanus (based on Colbourn and Lanus 2018 and Colbourn, Lanus and Sarkar 2018) in function scphfCA
- the construction based on CPHFs using CPHFs from Wagner, Colbourn and Simos (2022) in function cphfCA
- Colbourn et al. (2010, CKRS) cross-sum of an N x k code with a strength t-1 CA to yield a strength t CA
- recursive Bose constructions using productPCA or productCA, implemented in functions recursiveBose and recBoseCA; the variant with link{productPCA} is uniformly better than a similar construction of Hartman (2005, Theorem 7.1 and Corollary 7.2).
- a fusion construction based on a Bose matrix, from Colbourn (2008), in function fuseBoseCA
- a fusion construction based on a Bush matrix, from Colbourn (2008), in function fuseBushtCA
- a projection construction based on a Bose matrix, from Colbourn (2008), in functions projectionBose and projBoseCA
- the group-based strength 2 construction by Meagher and Stevens (2005) which uses starter vectors and a group of cycling permutations that leaves one value fixed (function CS_MS)
- the strength 2 cover starter construction by Lobb et al. (2012) which works similarly to Meagher and Stevens (2005) but fix one or more symbols using the additive group on the non-fixed symbols (function CS_LCDST)
- the strength 2 cover starter construction by Colbourn et al. (2006) which is simpler than Meagher and Stevens (2005) and allows a recursive product construction (function CS_CMMSSY)
- the 2-level cover starter construction by Colbourn and Keri (2009), which creates strength 4, 5, and 6 CAs and is also closely related to the Paley type construction of Colbourn (2015)
- direct product construction for strength 2 CAs in function productCA, with generalizations for slightly reducing the number of runs
- product construction for strength 2 PCAs in productPCA, proposed by Colbourn et al. (2006), improved by Colbourn and Torres-Jimenez (2013); automated in function dpCA based on some ingredients; the construction is behind various current best CAs of the Colbourn tables, which are not yet achieved by the package.
- adding a column, as detailed in Theorem 3.2 of Colbourn et al. (2010, CKRS)
- composition of arrays, i.e., cross product of levels for arbitrary strength (e.g., Theorem 2.8 of Zhang et al. 2014), crossCAs; automated in function pcaCA based on some ingredients
- the CA EXtender for strength 2 3-level CAs (Torres-Jimenez, Acevedo-Juarez and Avila-George 2021), which yields the current best arrays for all scenarii (but, contrary to the 2-level case, these arase not theoretically optimal and might improve in the future) in function CAEX.
- identification of flexible values as in Colbourn and Torres-Jimenez (2013), and the Nayeri et al. (2013) postprocessing in function postopNCK

So far, there are three functions for mixed level CAs: MCA2 implements a strength 2 construction, generalized from Sherwood (2008) by Groemping (2025), that is based on expanding some columns of a (mixed or) uniform CA, after reducing their number of levels, by replacing the flexible values

of those columns with suitable small matrices and replicating the rest of those columns the corresponding number of times. Its details can be found in Groemping (2025).
CA_to_MCA takes a uniform (or mixed level) array, removes levels (makes them flexible) as required. Both constructions may benefit from subsequent removal of as many rows as possible via the Nayeri et al. (2013) construction. For CA_to_MCA, this is the crucial step; for MCA2, gains are less dramatic, but still often relevant.
projBoseMCA implements the mixed level construction by Stevens, Ling and Mendelsohn (2002) in the form stated by Colbourn (2008) in his Corollary 2.2, and the construction of Colbourn's Theorem 2.3.

Furthermore, there are functions for the creation and inspection of sequence covering arrays (SeqCAs) (Levenshtein 1991 as detailed in Na 2021, Kuhn et al. 2012, Murray and Colbourn 2015 (not yet implemented), Torres-Jimenez et al. 2022):

- SeqCA_Levenshtein for algebraic creation of a SeqCA of strength t with t+1 columns,
- greedySeqCA_naive, greedySeqCA_Kuhn, greedySeqCA_TJ and their iterative versions,
- coverageSeqCA and identify_needed for assessing coverage and usefulness / redundancy of SeqCA entries,
- reduce_rows_iterative_complete and optimize_SeqCA for reducing the run size of a SeqCA.

These functions are limited to SeqCAs for relatively small numbers of columns and strengths.

Exported objects

The constructions are based on various catalogue objects and occasional arrays that can be inspected by expert users, but are mainly meant to be used by the package functions:

- colbournBigFrame,
- CAEX_CAs and CAEX_lineages,
- TJcat and TJ2level_CAs,
- powerCTcat,
- PCAcat
- DPcat
- SCPHFcat, SMC_SCPHFs, CL_SCPHFs, WC_SCPHFs
- CPHFcat and WCS_CPHFs
- MeagherStevensCombis and MeagherStevensStarters,
- CMMSSYCombis and CMMSSYStarters,
- LCDSTCombis and LCDSTStarters,
- ColbournKeriCombis and ColbournKeriStarters,
- CKRScat and CKRS_CAs,
- WKScat, WKS_CAs,
- PALEYcat, CYCLOTOMYcat, DWYERcat, NISTcat, miscCAcat, and individual arrays named in the latter.

The arrays referenced by DWYERcat and NISTcat can only be accessed with an internet connection.

The R objects are available in the data folder of the package, corresponding raw data and construction code can be found in the extdata folder for the package, which can be located in the directory inst/extdata of the GitHub repo.

Author(s)

Author: Ulrike Groemping, BHT Berlin.

References

Ball, W. W. R. and Coxeter, H. S. M. (1987). Mathematical Recreations and Essays, 13th ed. New York: Dover, pp. 308-309.

Bose, R.C. (1938). On the application of the properties of Galois fields to the problem of construction of hyper-Graeco-Latin squares. Sankhya 3, 323-338.

Bush, K.A. (1952). Orthogonal Arrays of Index Unity. Annals of Mathematical Statistics 23, 426-434.

Chateauneuf, M., Colbourn, C. and Kreher, D.L. (1999). Covering Arrays of Strength Three. Des. Codes Cryptogr. 16, 235-242.

Chateauneuf, M. and Kreher, D.L. (2002). On the state of strength-three covering arrays. Journal of Combinatorial Designs, vol. 10, no. 4, pp. 217-238. doi: 10.1002/jcd.10002.

Cohen, M.B., Colbourn, C.J., Ling, A.C.H. (2003). Augmenting simulated annealing to build interaction test suites, in: 14th International Symposium on Software Reliability Engineering, 2003. ISSRE 2003. IEEE, Denver, Colorado, USA, pp. 394-405. https://doi.org/10.1109/ISSRE.2003.1251061

Cohen, M.B., Colbourn, C.J., Ling, A.C.H. (2008). Constructing strength three covering arrays with augmented annealing. Discrete Mathematics 308, 2709-2722. https://doi.org/10.1016/j.disc.2006.06.036

Colbourn, C.J. (without year). Covering array tables: 2 <= v <= 25, 2 <= t <= 6, t <= k <= 10000, 2005–23. https://www.public.asu.edu/~ccolbou/src/tabby.

Colbourn, C.J., (2004). Combinatorial aspects of covering arrays. Matematiche (Catania) 59, 125-172.

Colbourn, C.J., (2008). Strength two covering arrays: Existence tables and projection. Discrete Math., vol. 308, 772-786. https://doi.org/10.1016/j.disc.2007.07.050

Colbourn, C. J. (2010). Covering arrays from cyclotomy, Des. Codes Cryptogr., vol. 55, no. 2, pp. 201–219. doi: 10.1007/s10623-009-9333-8.

Colbourn, C.J. (2015). Suitable Permutations, Binary Covering Arrays, and Paley Matrices, in: Colbourn, C.J. (Ed.), Algebraic Design Theory and Hadamard Matrices, Springer Proceedings in Mathematics & Statistics. Springer International Publishing, Cham, pp. 29-42. https://doi.org/10.1007/978-3-319-17729-8_3

Colbourn, C.J., Dougherty, R.E., Horsley, D. (2019). Distributing hash families with few rows. Theoretical Computer Science 800, 31-41. doi: 10.1016/j.tcs.2019.10.014

Colbourn, C.J., Kéri, G., (2009). Binary Covering Arrays and Existentially Closed Graphs, in: Chee, Y.M., Li, C., Ling, S., Wang, H., Xing, C. (Eds.), Coding and Cryptology, Lecture Notes in Computer Science. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 22-33. https://doi.org/10.1007/978-3-642-01877-0_3

Colbourn, C., Kéri, G., Rivas Soriano, P.P., Schlage-Puchta, J.-C., (2010). Covering and radius-covering arrays: constructions and classification. Discrete Appl. Math 158, 1158-1180. https://doi.org/10.1016/j.dam.2010.0

Colbourn, C.J., Lanus, E., 2018. Subspace restrictions and affine composition for covering perfect hash families. Art Discrete Appl. Math. 1, #P2.03. https://doi.org/10.26493/2590-9770.1220.3a1

Colbourn, C.J., Lanus, E., Sarkar, K., 2018. Asymptotic and constructive methods for covering perfect hash families and covering arrays. Des. Codes Cryptogr. 86, 907-937. https://doi.org/10.1007/s10623-017-0369-x

Colbourn, C.J., Martirosyan, S.S., Mullen, G.L., Shasha, D., Sherwood, G.B., Yucas, J.L., (2006). Products of mixed covering arrays of strength two. *J of Combinatorial Designs* **14**, 124-138. doi: 10.1002/jcd.20065

Colbourn, C.J., Torres-Jimenez, J. (2010). Heterogeneous hash families and covering arrays, in: Bruen, A.A., Wehlau, D.L. (Eds.), Contemporary Mathematics. American Mathematical Society, Providence, Rhode Island, pp. 3-15. https://doi.org/10.1090/conm/523/10309

Colbourn, C.J., Torres-Jimenez, J. (2013). Profiles of covering arrays of strength two. J. Alg. Comput. 44, 31-59. doi: 10.22059/jac.2013.7914.

Colbourn, C.J., Zhou, J. (2012). Improving Two Recursive Constructions for Covering Arrays. Journal of Statistical Theory and Practice 6, 30-47. https://doi.org/10.1080/15598608.2012.647489

Dwyer, A. (2024). CA Database: Data base of covering arrays and related objects. https://github.com/aadwyer/CA_Database.

Groemping, U. (2025). Generalizing a Sherwood (2008) construction for mixed level covering arrays. Report 01/2025, Reports in Mathematics, Physics and Chemistry. BHT Berlin. https://www1.beuth-hochschule.de/FB_II/reports/Report-2025-001.pdf

Hartmann, A. (2005). Software and Hardware Testing Using Combinatorial Covering Suites. In: Golumbic, M.C., Hartman, I.BA. (eds) *Graph Theory, Combinatorics and Algorithms*. Operations Research/Computer Science Interfaces Series, vol 34. Springer-Verlag, New York.

Ji, L. and Yin, J., 2010. Constructions of new orthogonal arrays and covering arrays of strength three. Journal of Combinatorial Theory, Series A 117, 236-247. https://doi.org/10.1016/j.jcta.2009.06.002

Kleitman, D.J. and Spencer, J. (1973). Families of k-independent sets, Discrete Math., vol. 6, no. 3, pp. 255-262. https://doi.org/10.1016/0012-365X(73)90098-8.

Katona, G. O. H. (1973). Two applications (for search theory and truth functions) of Sperner type theorems, Period. Math. Hung., vol. 3, no. 1, pp. 19-26. https://doi.org/10.1007/BF02018457.

Kokkala, J. I., Meagher, K., Naserasr, R., Nurmela, K. J., Östergård, P. R. J., & Stevens, B. (2018). Classification of small strength-2 covering arrays. Zenodo. https://doi.org/10.5281/zenodo.1476059

Kuhn, D.R., Higdon, J.M., Lawrence, J.F., Kacker, R.N. (2012). Combinatorial Methods for Event Sequence Testing. ICST '12: Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation, 601-609. https://doi.org/10.1109/ICST.2012.147

Levenshtein, V.I. (1991). Perfect codes in the metric of deletions and insertions. Diskretnaya Matematik 3(1), 3-20.

Lobb, J.R., Colbourn, C.J., Danziger, P., Stevens, B., Torres-Jimenez, J., (2012). Cover starters for covering arrays of strength two. Discrete Math. vol. 312, 943-956. https://doi.org/10.1016/j.disc.2011.10.026

Martirosyan, S. and Trung, T.V. (2004). On t-Covering Arrays. Des. Codes Cryptogr., vol. 32, no. 2, pp. 323–339. https://doi.org/10.1023/B:DESI.0000029232.40302.6d.

Meagher, K., Stevens, B., (2005). Group construction of covering arrays. J of Combinatorial Designs 13, 70-77. https://doi.org/10.1002/jcd.20035

Meagher, K., (2005a). Group Construction of Covering Arrays: Part 2. (Unpublished technical report). Ottawa.

Meagher, K., (2005b). Covering Arrays on Graphs: Qualitative Independence Graphs and Extremal Set Partition Theory. PhD thesis. University of Ottawa, Ottawa.

Moura, L., Stardom, J., Stevens, B., Williams, A. (2003). Covering arrays with mixed alphabet sizes. J of Combinatorial Designs 11, 413-432. https://doi.org/10.1002/jcd.10059

Murray, P.C., Colbourn, C.J. (2015). Sequence Covering Arrays and Linear Extensions. In: Jan, K., Miller, M., Froncek, D. (eds) Combinatorial Algorithms. IWOCA 2014. Lecture Notes in Computer Science(), vol 8986. Springer, Cham. https://doi.org/10.1007/978-3-319-19315-1_24

Na, J. (2021). Perfect Sequence Covering Arrays. Master's thesis at Simon Fraser University. https://summit.sfu.ca/_flysystem/fedora/2022-09/etd21570.pdf

Nayeri, P., Colbourn, C.J., Konjevod, G., (2013). Randomized post-optimization of covering arrays. European Journal of Combinatorics 34, 91-103. https://doi.org/10.1016/j.ejc.2012.07.017

NIST Covering Array Tables (last modified 2008, accessed 12 Jan 2025). https://math.nist.gov/coveringarrays/.

Sherwood, G.B., Martirosyan, S.S., Colbourn, C.J. (2006). Covering arrays of higher strength from permutation vectors. J. of Combinatorial Designs 14, 202-213. https://doi.org/10.1002/jcd.20067

Sherwood, G.B. (2008). Optimal and near-optimal mixed covering arrays by column expansion. Discrete Mathematics 308, 6022-6035. https://doi.org/10.1016/j.disc.2007.11.021

Stevens, B., Ling, A. and Mendelsohn, E. (2002). A Direct Construction of Transversal Covers Using Group Divisible Designs. Ars Combinatoria 63, 145-159.

Torres-Jimenez, J. (without year, accessed 10 Feb 2025). Covering arrays. Formerly available at `www . tamps . cinvestav . mx / ~oc /` (blanks must be removed, but no longer functional, no alternative location known).

Torres-Jimenez, J., Acevedo-Juarez, B., Avila-George, H. (2021). Covering array EXtender. Applied Mathematics and Computation 402, 126122. https://doi.org/10.1016/j.amc.2021.126122

Torres-Jimenez, J., Ramirez-Acuna, D.O., Acevedo-Juarez, B., Avila-George, H. (2022). New upper bounds for sequence Covering Arrays using a 3-stage approach. Expert Systems With Applications 207, 118022. https://doi.org/10.1016/j.eswa.2022.118022

Wagner, M., Colbourn, C., Simos, D.E., (2022). In-Parameter-Order strategies for covering perfect hash families. Applied Mathematics and Computation 421, 126952. https://doi.org/10.1016/j.amc.2022.126952

Wagner, M., Kampel, L., Simos, D.E., (2021). Heuristically Enhanced IPO Algorithms for Covering Array Generation, in: Flocchini, P., Moura, L. (Eds.), Combinatorial Algorithms, Lecture Notes in Computer Science. Springer International Publishing, Cham, pp. 571–586. https://doi.org/10.1007/978-3-030-79987-8_40

Walker II, R. and Colbourn, C. (2009). Tabu search for covering arrays using permutation vectors. Journal of Statistical Planning and Inference 139, 69-80.

Zhang, J., Zhang, Z. and Ma, F. (2014). Automatic Generation of Combinatorial Test Data. SpringerBriefs in Computer Science. Springer Verlag, Berlin.

See Also

Useful links:

- https://github.com/ugroempi/CAs
- Report bugs at https://github.com/ugroempi/CAs/issues

---

add1CA *function to add a columns*

Description

increases the number of columns by 1, adding at most bestN(t-2,k-1,v)*v*(v-1) rows.

Usage

```
add1CA(t, k, v, ...)

N_add1CA(t, k, v)

add1Column(D, t, v, ...)
```

Arguments

t	integer: the required strength, or the strength of D (not checked)
k	integer: the required number of columns
v	integer: the required number of levels, or the number of levels of D
...	currently not used
D	a strength t CA with v levels in k-1 columns (integer levels, starting at 0 or 1)

Value

add1Column and add1CA return a CA of strength t in v levels in k columns.
It will have at most bestN(t-2,k-2,v)*v*(v-1) additional rows.
N_add1CA returns the number of runs obtainable by add1CA.

Details

Function add1CA applies function add1Column the the best available strength t CA at v levels with k-1 columns in the role of D.
The construction relies on appending to D a copy of the last column of D, followed by adding rows that consist of
the best strength t-2 CA at v levels with k-2 columns, repeated v*(v-1) times, in the first k-2 columns,
each combined with one of the v(v-1) distinct level pairs for the last two columns.
For t=2, the best CA is simply a constant row of flexible values.

The construction is not interesting for small CAs, but may be useful, where good large CAs, e.g., ones constructed by cyclotomy, have one less column than needed (and a good CA with strength `t-2` for `k-1` columns is available).

References

CKRS, Theorem 3.2

Examples

```
D <- powerCA(3,1331,2)
dim(D)
D2 <- add1Column(D, 3, 2)
dim(D2)
eCAN(3,1331,2)
eCAN(3,1332,2)
N_add1CA(3,1332,2)

dim(add1CA(3,1332,2))


A <- add1CA(6,2504,2)
dim(A)
eCAN(6,2504,2) ## much better
Ns(4,2504,2) ## its so much worse because this ingredient is poor
3047 + (1394 - 272)*2
eCAN(4,2504,2) ## Colbourn and Zhou power construction not implemented
```

`bestCA` *Function to obtain the best uniform CA for a combination of t, k and v*

Description

The function uses function Ns to find the best currently implemented CA for the requested scenario.

Usage

```
bestCA(t, k, v, maxN=1000000, fixNA=TRUE, seed=NULL, ...)

bestN(t, k, v, internet=TRUE, exclude=NULL, ...)

ckrsCA(t, k, v, ...)

dwyerCA(t, k, v, ...)

nistCA(t, k, v, ...)

tjCA(t, k, v, ...)
```

Arguments

`t`	integer: the strength
`k`	integer: the number of columns
`v`	integer: the number of levels for each column
`maxN`	integer: the largest number of runs N for which an array is desired; if the smallest possible N is larger, an error is thrown; default: 1000000 (too large for being created in reasonable time on most computers)
`fixNA`	logical: should flexible values be fixed? If the ingoing CA has flexible values, `fixNA=TRUE` randomly assigns a fixed value to each flexible value. Thus, a seed is needed for reproducibility.
`seed`	`NULL` or an integer seed for making the fixing of NA values reproducible; a non-NULL seed is stored in the attribute `fixNA_seed` of the returned object.
`...`	handed to construction function, currently, only implemented for `fuseBose`, for enabling `fixNA=FALSE`
`internet`	logical; if FALSE, methods whose designs need an internet connection are excluded
`exclude`	`NULL` or a character vector of method(s) to exclude, in support of size calculations for recursive constructions which need to calculate sizes of best ingredients

Value

`bestN` returns a named integer: the first method that yields the smallest implemented size.
The other functions return a matrix of class `ca`, which is a uniform covering array of strength at least `t` with `k` columns at `v` levels.

Details

Function `bestCA` uses function `Ns` for locating the best construction for the given scenario, and subsequently implements the first such construction, where the order is that returned by function `Ns`.

Some constructions (`"DWYER"` and `"NIST"`) require an internet connection, because the respective CAs are not part of the package.

Some constructions provide CAs with flexible values, which are denoted as NA values in package **CAs**. These need to be fixed for actual experimentation, and function `bestCA` fixes them randomly; in such cases, the returned object has an attribute `fixNA_seed`, which is either the seed that was specified by the user or a seed that was created by the function. Specifying the seed in a subsequent call to the function permits to get the same CA again (provided that no new smaller CAs have been implemented).

The attributes of the returned object contain elements `eCAN`, `CAs-version` and `date`: `eCAN` gives the current best array; as this may change, the other attributes provide the context.

See Also

`Ns()`

Examples

```
# example with a single best construction
Ns(2,12,4)
bestCA(2,12,4)
# example with three best constructions
Ns(4,8,2)
# bestCA uses the leftmost
bestCA(4,8,2)

Ns(3, 1500, 2)
## powerCT is best
attributes(bestCA(3, 1500, 2))
## it is, of course, also possible to directly use the
## construction functions, in this case powerCA with type="CT".

# a case that is not implemented (would be huge)
Ns(6, 5000, 10)
try(bestCA(6, 5000, 10))

Ns(3, 50, 2)
## DWYER needs internet connection,
## as earlier in list, TJ is preferred
D <- bestCA(3, 50, 2)
dim(D)
# coverage(D, 3)

Ns(6, 50, 2)
## WKS is the only best
D <- bestCA(6, 50, 2)
dim(D)

## fusing is far better than direct construction

## slow, yields 8187 runs by fusing an 8191 run cphfCA
## in 16 levels twice
Ns(3,37,14)
Ns_fuse(3,37,14,maxfuse=2)
bestN(3,37,14)


Ns(3,9,7)
## CKRS and DWYER are best
D <- bestCA(3, 9, 7)
dim(D)

Ns(3,9,6)
## CKRS is best implemented, but 18 runs worse than eCAN
eCAN(3,9,6)
## Torres-Jimenez designs are currently (Sep 2025) unavailable
D <- bestCA(3, 9, 6)
attributes(D)
```

```
Ns(3,9,5)
## CK_NRB is the unique best
ks(3,185,5)

Ns(2,26,24) ## fuseBose is best-known
dim(bestCA(2,26,24))

Ns(2,26,23) ## fuseBushtCA is best implemented,
            ## projBoseCA is close
dim(D <- bestCA(2,26,23))
attributes(D)
## eCAN from NCK post processing of projBose
## (removes 7 runs)
eCAN(2,26,23)

Ns(3,6,12)
## the optimum Ji and Yin OA is in the package as oa1728.12.6


## this code needs an internet connection
## and is therefore wrapped in \donttest
## and in try
   try(dim(nistCA(3,5,4)))
```

---

ca12.2.8 *Useful arrays for covering array constructions*

Description

lists useful single arrays that are used themselves as covering arrays or as ingredients for covering array constructions

Usage

```
ca12.2.8

ca26.4.15

ca23.4.8

ca30.5.7

ca41.6.6

oa1728.12.6

oa3375.15.6
```

`oa9261.21.6`

`CohenSA`

`miscCAcat`

Format

All arrays from this package that are listed here have their rows arranged such that constant rows are at the top or in PCA format; if there are fewer than `v` constant rows, they are usually arranged in PCA format (see `productPCA`).
`miscCAcat` also contains references to orthogonal arrays from R package **DoE.base**, e.g. `DoE.base::L81.3.5-1`, which is an OA(81,4,5,3) (strength 4; 1 is subtracted, because OAs in that package are coded with levels starting at 1, whereas CAs in this package have levels starting at 0).
`ca26.4.15`, `ca23.4.8`, `ca30.5.7` and `ca41.6.6` are strength 2 CAs.
There is also the list `CohenSA` that contains strength 2 CAs that were created by Cohen using simulated annealing.

`oa1728.12.6`, `3375.15.6` and `oa9261.21.6` are strength 3 orthogonal arrays (classes `ca` and `oa`) and matrices with levels starting at 0. They might eventually reside in the R package **DoE.base**.

`miscCAcat` is a data.frame that collects the properties of all the CAs (including orthogonal arrays and the CAs from within `CohenSA`) that are documented in this file. It has the columns `t`, `k`, `v`, `N`, `fns` (character), `nconst`, `PCAstatus` (numeric, holds `k1`; `v` implies that `nconst=v`, i.e., k1=k) and `hasNA` (logical, at present, FALSE for all entries). .

An object of class `ca` (inherits from `ca`, `matrix`, `array`) with 26 rows and 15 columns.

An object of class `ca` (inherits from `ca`, `ca`, `ca`, `matrix`, `array`) with 23 rows and 8 columns.

An object of class `ca` (inherits from `matrix`, `array`) with 30 rows and 7 columns.

An object of class `ca` (inherits from `matrix`, `array`) with 41 rows and 6 columns.

An object of class `ca` (inherits from `oa`, `matrix`, `array`) with 1728 rows and 6 columns.

An object of class `ca` (inherits from `matrix`, `array`) with 3375 rows and 6 columns.

An object of class `ca` (inherits from `matrix`, `array`) with 9261 rows and 6 columns.

An object of class `list` of length 7.

An object of class `data.frame` with 23 rows and 9 columns.

Source

`oa12.2.8` was created by the package author based on learning about its existence (with two constant rows) from Colbourn and Torres-Jimenez (2010) and Cohen, Colbourn and Ling (2003).
The list `CohenSA` holds arrays created by Cohen via simulated annealing (personal communication).
`oa1728.12.6` was created by the package author using construction 2 of Ji and Yin (2010).
The creation of the objects is documented by the R file `miscCAs.R` in the `extdata` folder for this package, which can be located in the directory `inst/extdata` of the GitHub repo.

References

Cohen, Colbourn and Ling (2003) and Cohen (personal communication), Colbourn and Torres-Jimenez (2010), Ji and Yin (2010), Kokkala et al. (2018)

Examples

```
attributes(ca12.2.8)
attributes(ca26.4.15)
attributes(oa1728.12.6)
attributes(oa3375.15.6)
attributes(oa9261.21.6)
```

CAEX — *CAEX: current optimal CAs of strength 2 with v=3*

Description

Function for CAEX CAs of strength 2 with v=3. These yield current optimal CAs. The functions use stored CAs from Torres-Jimenez, Acevedo-Juarez and Avila-George (2021), where necessary, and product constructions from that paper, where possible.

Usage

```
CAEX(k=NULL, N=NULL, t=2, v=3, maxk1=FALSE, ...)

k_detail_CAEX(t=2, N=NULL, v=3)
```

Arguments

`k`	positive integer or `NULL`; the requested number of factors; if both `k` and `N` are specified, `k` takes precedence; if both `k` and `N` are `NULL`, function `CAEX` throws an error.
`N`	positive integer or `NULL`; the requested number of runs; run sizes up to 50 are implemented; these accommodate up to 112770(!) columns for `k_CAEX`, `N=NULL` implies that k for all possible `N` (9, 11 to 50) `k` and `N` are returned (as part of data.frame that also has `N` and the construction type).
`t`	integer, the requested strength; currently, only 2 is implemented
`v`	integer, the requested number of levels; currently, only 3 is implemented
`maxk1`	logical: should it be attempted to achieve maximum k1 for the output's PCA structure? `TRUE` may substantially increase run time.
`...`	currently not used

Value

Function CAEX returns a CA (matrix of class ca) with levels coded from 0 to v-1, and flexible positions (if any) denoted as NA. The returned object has an attribute PCAstatus that shows the PCA status, with k1 and k2 obtained from applying function CA_to_PCA to the result in case of product calculation. The splits k=k1+k2 are worse than the splits reported in Torres-Jimenez et al. (2021) for the product constructions (see also listing in the Examples section), unless maxk1=TRUE is chosen, at the expense of much longer run time.
Function k_detail_CAEX returns a data frame with columns N, k and method, and sometimes also one and two, where k holds the maximum number of columns for run sizes up to N, and method indicates, whether the design is the Bose array (N=9), a stored array, or is obtained from stored arrays one and two via a CAxCA or a PCAxPCA construction. For N=NULL, a data.frame of all CAEX constructions is returned.

Details

Function CAEX provides only strength 2 CAS, and currently only for 3-level columns. These are the current best ones, i.e., the necessary run size for a desired number of columns k can be obtained from function eCAN, or from function N_TJcat.

Function CAEX implements the arrays from Torres-Jimenez, Acevedo-Juarez and Avila-George (2021). It uses internally-stored CAs (in object CAEX_CAs), where the last construction step is based on numerical column extension (N=11 to 33 and 35), and product constructions as indicated in Table 4 of Torres-Jimenez, Acevedo-Juarez and Avila-George (2021) for N = 34 and 36 to 50. The information on how to proceed comes from the internal list CAEX_lineages. In all cases, the returned strength 2 (P)CAs have the same dimensions as those listed in the paper; where products were calculated by function CAEX, the k1 is per default smaller than that provided in the paper, as it is not undertaken to optimize it beyond moving the already suitable columns to the left.

If the designs are intended for use in further PCAxPCA constructions, maximization of the k1 value in the resulting PCA structure is desirable. Setting maxk1 to TRUE will attempt to achieve the k1 indicated in Torres-Jimenez et al. (2021). This is attempted by using the internal function find_clique, which searches for a subset of v rows with at least k1 columns with distinct levels. If successful, these rows are moved to the top, and function CA_to_PCA yields the split k = k1+k2 with k1 as stated in the paper.
The current implementation is not ideal regarding flexible values (coded as NA) among the possibilities for increasing k1 (a single NA in a v-tuple can be made use of, two or more NAs are ignored, due to R's handling of missing values in assessing uniqueness); There are not many flexible values in the stored ingredient CAs (none for N=11to 15, 17, 18, 20 and 21, 1 for N=23, 24 or 27, 2 for N=16, 4 for N=26, 11 for N=19 or 29, 12 for N=22, 17 for N=30, 48 for N=25 or 32, 50 for N=33, 65 for N=31, 69 for N=35, 197 for N=28). It is thus conceivable, if not likely, that it may be impossible to achieve the k1 value attained in Torres-Jimenez et al. (2021); in that case, the best possible attainable k1 value will be produced, which is likely larger than the default, but may require a long run time. The search may take a long time, if a successful v-tuple of rows is found late in the search path over the choose(N, v) tuples. Thus, this search should only be triggered if a large k1 is really needed; it is not relevant for the coverage properties of the design itself.

References

Torres-Jimenez, Acevedo-Juarez and Avila-George (2021)

Examples

```
## a stored CA
D <- CAEX(12) ## k=12 columns
eCAN(2,12,3)
coverage(D, 2)
k_detail_CAEX(N=9)
k_detail_CAEX(N=10)
k_detail_CAEX(N=11)
k_detail_CAEX(N=34)
k_detail_CAEX(N=50)
k_detail_CAEX(N=55)

##############################################################
## CAs obtained using function productCA as the workhorse

## from two CAs without flexible values
D <- CAEX(N=34)
attributes(D)
is.PCA(D)


## from two CAs with flexible values
## (slightly longish run time)
## NOTE: example() skips this example,
##      unless the user specifies
##      options(run_heavy_examples = TRUE)
D <- CAEX(N=43)
attributes(D)

#############################################################

################################################################
## available constructions from function CAEX
################################################################
#      NOTE: example() skips this construction example,
#      unless the user specifies
#      options(run_heavy_examples = TRUE)
# longest about 07 min for N=49 (using productCA, like two other cases)
# about 13 s for N=50 (using productPCA, like most cases)
allCAEX <- lapply(11:50, function(obj) CAEX(N=obj))

dimsAchieved <- t(sapply(allCAEX, dim))
colnames(dimsAchieved) <- c("N","k")
k1Achieved <- sapply(allCAEX, function(obj) attr(obj, "PCAstatus")$k1)
(CAEXinfos <- cbind(as.data.frame(cbind(dimsAchieved, k1=k1Achieved)),
     method=unlist(lapply(CAEX_lineages[[1]][[1]],
           function(obj) ifelse(is.character(obj), "stored", obj[1]))),
     k1_paper=c(4,4,6,7,14,15,22,31,34,45,69,76,99,144,138,185,294,295,414,
                582,655,872,1327,1426,1938,2918,3210,4163,6262,6842,8838,
                12216,13485,18894,27606,30271,39387,55151,60884,76842)
                ))
## see optimality by comparing with
```

```
## TJcat[which(TJcat[,"t"]==2 & TJcat[,"v"]==3 & TJcat[,"N"]<=50)[-1],c("k","N")])

#     N      k    k1  method k1_paper
# 11 11      5     4  stored        4
# 12 12      7     4  stored        4
# 13 13      9     6  stored        6
# 14 14     10     7  stored        7
# 15 15     20    14  stored       14
# 16 16     21    15  stored       15
# 17 17     29    22  stored       22
# 18 18     46    31  stored       31
# 19 19     49    34  stored       34
# 20 20     63    45  stored       45
# 21 21     93    69  stored       69
# 22 22    107    76  stored       76
# 23 23    138    99  stored       99
# 24 24    199   144  stored      144
# 25 25    216   138  stored      138
# 26 26    288   185  stored      185
# 27 27    435   294  stored      294
# 28 28    449   295  stored      295
# 29 29    610   414  stored      414
# 30 30    878   582  stored      582
# 31 31    964   655  stored      655
# 32 32   1308   872  stored      872
# 33 33   1964  1327  stored     1327
# 34 34   2116   460   CAxCA     1426  ## < 1 second,
                                       ## or 95 seconds with maxk1=TRUE
# 35 35   2700  1938  stored     1938
# 36 36   3918  2139 PCAxPCA     2918  ## < 1 second,
                                       ## or > 8 minutes with maxk1=TRUE
# 37 37   4457  2356 PCAxPCA     3210
# 38 38   5763  3069 PCAxPCA     4163
# 39 39   8329  4464 PCAxPCA     6262
# 40 40   9207  5244 PCAxPCA     6842
# 41 41  11898  6831 PCAxPCA     8838
# 42 42  17895  9114 PCAxPCA    12216  ## approx 1.3 seconds (productPCA),
                                       ##   or approx. 10.5 min with maxk1=TRUE
# 43 43  20010  4350   CAxCA    13485  ## approx 16 seconds (productCA slower),
                                       ##   or approx 11.5 min with maxk1=TRUE
# 44 44  25317 14256 PCAxPCA    18894  ## approx 1.9 seconds (productPCA)
                                       ##   or more than 40 min with maxk1=TRUE
# 45 45  37071 20286 PCAxPCA    27606
# 46 46  42174 22344 PCAxPCA    30271
# 47 47  54531 29106 PCAxPCA    39387
# 48 48  80789 41137 PCAxPCA    55151
# 49 49  90344 19640   CAxCA    60884  ## approx 7 min (productCA slower)
# 50 50 112770 60078 PCAxPCA    76842  ## approx 13 seconds (productPCA)
```

`CAEX_lineages` *Data for function CAEX*

Description

Two stored objects in support of function CAEX

Usage

```
CAEX_lineages

CAEX_CAs
```

Format

`CAEX_CAs` is a list of stored CAs for the character string entries of `CAEX_lineages`, with names corresponding to those character string entries.

`CAEX_lineages` is a list of lists of lists of lists, where the first and second list each have length one only and are named `2` for the strength and `3` for the number of levels of all arrays, whereas the list `CAEX_lineages[['2']][['3']]` has length 40 and names `11` to `50`, for the run sizes of the respective CAs.

An object of class `list` of length 24.

Details

This section provides background information for interested users. Array creation can be easily accomplished via the function `CAEX`, which works for strength 2 with 3-level columns.

Torres-Jimenez et al. (2021) used product constructions and column extensions to create a series of strength 2 CAs with 3-level columns from 11 to 50 runs. Later CAs build on the best earlier CAs. The column extensions were obtained with massive computer search effort. Hence, arrays obtained from those are stored in the list `CAEX_CAs`, and their names are given in the object `CAEX_lineages`. Arrays for larger numbers of runs (34, and 36 to 50) can be obtained from function `productCA`, and the details are given in the respective entry of `CAEX_lineage`.

Source

The Source is Torres-Jimenez et al. (2021), and the arrays area also included in `TJcat`.
The creation of the objects is documented by R files in the `extdata` folder for this package, which can be located in the directory `inst/extdata` of the GitHub repo. The actual raw arrays for `CAEX_CAs` be found there as well.

References

Torres-Jimenez (2021)

Examples

```
lengths(CAEX_lineages[[1]][[1]])
CAEX_lineages[['2']][['3']][['12']]
CAEX_CAs[["ca12.2.3.7"]]
CAEX_lineages[['2']][['3']][['34']]
```

checkcond1 *Functions for checking the Cyclotomy conditions*

Description

Conditions for constructions 1 to 4 and their modifications of Colbourn (2010) can be checked. The checks take a long time for large cases.

Usage

```
checkcond1(t, v, q, iter=FALSE, progress=FALSE)

checkcond2(t, v, q, iter=FALSE, progress=FALSE)

checkcond3(t, v, q, iter=FALSE, progress=FALSE)

checkcond4(t, v, q, iter=FALSE, progress=FALSE)

checkcond3a(t, v, q, iter=FALSE, progress=FALSE, type3=TRUE)

checkcond3b(t, v, q, iter=FALSE, progress=FALSE, type3=TRUE)

checkcond4a(t, v, q, iter=FALSE, progress=FALSE, type4=TRUE)

checkcond4b(t, v, q, iter=FALSE, progress=FALSE, type4=TRUE)
```

Arguments

`t`	CA strength to be checked
`v`	number of levels
`q`	prime or prime power
`iter`	logical (default FALSE): if FALSE, all combinations are calculated at once (slightly faster); otherwise, combinations are iterated through one at a time (works for large cases where memory is not sufficient for the other strategy).
`progress`	logical (default FALSE): if TRUE, show a progress bar. For the a and b types, two progress bars will be shown (second one progresses much faster than first one).
`type3`	Should the type 3 check be run? Set to FALSE, if it is already known that the type 3 condition is fulfilled.
`type4`	Should the type 4 check be run? Set to FALSE, if it is already known that the type 4 condition is fulfilled.

Value

a logical indicating whether the condition is fulfilled.

Details

checkcond1 to checkcond4 do the basic checks. Checks for constructions 3a and 4a consist of checks for constructions 3 and 4, combined with checks for construction 1 for strength t-1,
and checks for constructions 3b and 4b consist of checks for constructions 3 and 4, combined with checks for condition 2 for strength t-1.

Examples

```
checkcond1(3,2,11)  ## FALSE
checkcond2(3,2,11, progress=TRUE)  ## TRUE, example for progress bar
## if construction 4 with strength 4 works,
## then this setting will also fulfill construction 4b
checkcond4b(4,2,11, progress=TRUE)
N_cyc(11,2,"4b")
k_cyc(11,2,"4b")
## this array is the currently best-known
eCAN(4,12,2)
```

CK_constrD — *Function to augment a design with Chateauneuf-Kreher construction D*

Description

A strength 3 CA in k (v-1)-level columns and N runs is extended to k v-level columns in N + kM + k(v-1) runs using a strength 2 CA in M runs for k-1 (v-1)-level columns (construction D)

Usage

```
CK_constrDCA(k, v, ...)

CK_constrD(D3, D2=NULL, check=FALSE, start0=TRUE, ...)

N_CK_constrDCA(t=3, k, v, ...)
```

Arguments

D3	a strength 3 CA with k columns in v-1 levels, coded as integers 0, ..., v-2 or 1,...,v-1
D2	a strength 2 CA with k-1 columns in v-1 levels, coded as integers 0, ..., v-2 or 1,...,v-1, same coding as D3; NULL implies automatic creation with function bestCA, which uses the known optimal function KSK for v=2, and the best implemented construction for larger v.
check	a logical: if TRUE, the strength requirements are verified (may take very long for large arrays, therefore defaults to FALSE)

start0	logical: Do integer values start with 0 ? (D3 and D2 must be compatible)
...	further arguments to function coverage (CK_constrD) or bestN (function N_CK_constrDCA)
k	integer: the target number of columns
v	integer: the number of levels
t	integer: the target strength (always 3)

Details

CK_constrDCA is the interface function that uses the workhorse function CK_constrD.

Value

Function CK_constrDCA returns a matrix of class ca with attributes, which is a strength 3 covering array.
Function CK_constrD returns a matrix without class and attributes,
function N_CK_constrDCA returns the run size of the CA.

Note that the run size may depend on the availability of an internet connection for cases for which the best strength 3 CA for k columns or the best strength 2 CA for k-1 columns is in the Dwyer data base or the NIST library only (these might exist - has not been checked).

References

Chateauneuf and Kreher (2002)

Examples

```
N_CK_constrDCA(3,6,6)
bestN(3,6,6) ## not competitive
dim(CK_constrD(SCA_Busht(5,3)))
dim(CK_constrDCA(6,6)) ## always strength 3
```

CK_doubling *Function to create a strength 3 CA with Chateauneuf-Kreher doubling*

Description

A strength 3 CA in j v-level columns and N runs is extended to k=2j v-level columns in N + M*(v-1) runs by adding cyclic permutations of a strength 2 CA in v levels and M runs (Thms 4.1 and 4.5)

Usage

```
CK_doublingCA(k, v, ...)

CK_doubling(D3, D2=NULL, check=FALSE, start0=TRUE, ...)

N_CK_doublingCA(t=3, k, v, ...)
```

Arguments

D3	a strength 3 CA with j columns in v levels, coded as integers 0, ..., v-1 or 1,...,v
D2	a strength 2 CA with j columns in v levels, coded as integers 0, ..., v-1 or 1,...,v, same coding as D3; NULL implies automatic creation with function bestCA, which uses the known optimal function KSK for v=2, and the best implemented construction for larger v.
check	a logical: if TRUE, the strength requirements are verified (may take very long for large arrays, therefore defaults to FALSE)
start0	logical: Do integer values start with 0 ? (D3 and D2 must be compatible)
...	further arguments to function coverage (CK_doubling) or bestN (function N_CK_doublingCA)
k	integer: the target number of columns
v	integer: the number of levels
t	integer: the target strength (always 3)

Details

CK_doublingCA is the interface function that uses the workhorse function CK_doubling and should normally be used.

When directly using CK_doubling, the user is responsible for ensuring validity of ingoing D3 and, if provided, D2. For check=FALSE, a result with strength less than 3 will be returned in case of inadequate ingoing matrices.

Chateauneuf and Kreher's (2002) doubling contributes a few best-known strength 3 CAs to the Colbourn tables; in particular, where v is a prime or prime power and thus fulfills the conditions for the Bose construction of a strength 2 OA and the Bush construction of a strength 3 OA (as implemented in SCA_Busht), the designs are at present (13 Feb 2025) best-known for v >= 5.

Value

Function CK_doublingCA returns a matrix of class ca with attributes, which is a strength 3 covering array.
Function CK_doubling returns a matrix without class and attributes,
function N_CK_doublingCA returns the run size of the CA.

Note that the run size may depend on the availability of an internet connection for cases for which the best strength 3 CA for ceiling(k/2) columns is in the Dwyer data base or the NIST library only (these might exist - has not been checked).

References

Chateauneuf and Kreher (2002)

Examples

```
(E1 <- CK_doublingCA(8, 2))
coverage(E1, 3)  ## successful
## faster TRUE/FALSE check
```

```
caverify::ca_verify(E1, 3)$coverage
eCAN(3,8,2)   ## not optimal, but close

## the above was achieved with these ingredients:
D3 <- bestCA(3, 4, 2)  ## 8x4, OA
dim(D3)
coverage(D3, 3)
D2 <- bestCA(2,4,2)   ## 5x4, KSK construction
dim(D2)
E2 <- CK_doubling(D3, D2)
dim(E2)
table(E1-E2) ## content is the same, attributes differ

D3 <- paleyCA(3, 11)
coverage(D3, 3)  # suitable
E <- CK_doubling(D3)
dim(E)
coverage(E, 3)  ## successful
eCAN(3,22,2)   ## optimal

E3 <- CK_doublingCA(12, 4)
dim(E3)
## Thm 4.5, 4 levels
## ingredients for E3:
D3 <- DoE.base::L64.4.6 - 1 ## 6 columns are not created from SCA_Busht
  ## bestCA(3, 6, 4) would yield a suitable CA from the DWYER database,
  ## provided that there is an internet connection
D2 <- bestCA(2, 6, 4)        ## 19 runs, optimal
eCAN(2, 6, 4)
## E3 has 64 + (4-1)*19 = 121 rows
eCAN(3, 12, 4)    ## size almost optimal

## not all outcomes are so close to optimal
dim(CK_doublingCA(16, 3))
## 42 + 2*13 = 68 runs
eCAN(3, 16, 3)  ## run size too large by factor 4/3 versus the current optimum 51
```

CK_NRB *Construction Chateauneuf-Kreher NRB of strength 3 for v=3,4,5 with k=2*v*

Description

Function for three very good strength 3 CAs with v=3,4,5 and k=2*v from Chateauneuf, Colbourn and Kreher 1999 and Chateauneuf and Kreher 2002

Usage

```
CK_NRB(k, v)
```

```
N_CK_NRB(t=3, k, v)

k_CK_NRB(t=3, N, v)
```

Arguments

k	integer, number of columns needed (<= 2*v)
v	integer, number of levels for each column
t	integer, strength (must be 3)
N	integer, maximal number of affordable runs

Value

CK_NRB returns a covering array of 33 (v=3), 88 (v=4) or 185 rows (v=5),
N_CK_NRB returns a number for valid entries and NA otherwise.

Details

CA(33,3,6,3) is from Figure 1 of Chateauneuf, Colbourn and Kreher (1999),
CA(88,3,8,4) is from Example 2.3 of the paper with the ingredient M for from Figure 2 (where typos in the last row of M had to be fixed).
CA(185,3,10,5) is from Theorem 3.4 of Chateauneuf and Kreher (2002, with AGL(2,q) as stated above Theorem 3.1 and the ingredient M in Figure 5 of the paper).

References

Chateauneuf, Colbourn and Kreher (1999), Chateauneuf and Kreher (2002)

Examples

```
dim(CK_NRB(6, 3))
attributes(CK_NRB(9,5))
Ns(3, 7, 4)  ## substantially better than other constructions
Ns(3, 9, 4) ## not possible
dim(bestCA(3,8,4))
k_CK_NRB(3, 100, 4) ## with up to 100 runs, 8 columns affordable
```

CKRS_augment *Function to augment a CA with Eqn (5) of CKRS*

Description

increase the number of levels of a CA by 1 in a greedy fashion

Usage

```
CKRS_augmentCA(t, k, v, ...)

CKRS_augment(D, start0=TRUE, ...)

N_CKRS_augmentCA(t, k, v, ...)
```

Arguments

D	a strength t CA with k columns in v-1 levels, coded as integers 0, ..., v-2 (or 1, ..., v-1)
start0	logical: Do integer values start with 0 ?
...	further arguments to function bestN (function N_CKRS_augmentCA)
t	integer: the target strength
k	integer: the target number of columns
v	integer: the number of levels

Details

CKRS_augmentCA is the interface function that uses the workhorse function CKRS_augment.

Value

Function CKRS_augmentCA returns a matrix of class ca with attributes, which is a strength t covering array.
Function CKRS_augment returns a matrix without class and attributes,
function N_CKRS_augmentCA returns the run size of the CA.

Note that the run size may depend on the availability of an internet connection for cases for which the best strength t CA for k columns with v-1 levels is in the Dwyer data base or the NIST library only (these might exist - has not been checked).

References

CKRS (2010)

Examples

```
dim(CKRS_augment(SCA_Busht(5,3))) ## not competitive
bestN(3,6,6)
```

CKRScat *Data for function ckrsCA*

Description

The data.frame CKRScat supports the identification of CAs from the list CKRS_CAs

Usage

```
CKRScat

CKRS_CAs
```

Format

`CKRScat` is a data.frame in 53 rows with columns `t` (strength), `v` (number of levels), `k` (number of CA columns), `N` (number of CA rows) and `fn` (name of entry in list `CKRS_CAs`.

`CKRS_CAs` is a list of 53 covering arrays (class `ca`) corresponding to the rows of `CKRScat`.

An object of class `list` of length 53.

Details

The data.frame `CKRScat` holds information on arrays in 2:7 and 10:12 levels, at strengths 2 to 6. The list `CKRS_CAs` holds the actual arrays.

Source

The reference is Colbourn, Keri, Rivas Soriano and Schlage-Puchta (2010, CKRS). The last available source for the arrays was `http://old.sztaki.hu/~keri/arrays/CA_listings.zip`, which is unfortunately no longer functional.
The creation of the objects is documented by R files in the `extdata` folder for this package, which can be located in the directory `inst/extdata` of the GitHub repo. The actual raw arrays can be found there as well (they are small enough to be included in spite of their availability elsewhere on the web).

References

CKRS (2010)

Examples

```
## an impression of the content
head(CKRScat)
tail(CKRScat)
fivenum(CKRScat$N)
xtabs(~ v + t, CKRScat)

fun <- function(t,k,v) eCAN(t,k,v)$CAN
```

```
fun <- Vectorize(fun)
diff <- CKRScat[,"N"]-fun(CKRScat[,"t"], CKRScat[,"k"], CKRScat[,"v"])
fivenum(diff)
## 0 0 2 6 69
ratio <- CKRScat[,"N"]/fun(CKRScat[,"t"], CKRScat[,"k"], CKRScat[,"v"])
fivenum(ratio)
## 1 1 1.0010 1.0349 1.25
## at least half of the arrays are pretty close to the best-known
```

`colbournBigFrame` *Colbourn tables for upper bounds on CAN / lower bounds on CAK*

Description

data.frame that holds the November 2024 status of the Colbourn tables of upper bounds for covering array numbers

Usage

```
colbournBigFrame
```

Format

`data.frame` with 13641 rows in 5 columns:

t the strength, from 2 to 6

v the number of levels, from 2 to 25

k the maximum number of columns

N the smallest known run size for which a construction is *principally* available

Source the table entry for the source information; sometimes (too) cryptic

Details

Until recently, the Colbourn (without year) tables were an authoritative source of the state of the art run sizes of covering arrays and were kept reasonably up-to-date by Charlie Colbourn. At present (June 2025), the tables are no longer available to the public, and it is unclear, whether they are still updated. The status of the tables in this package is November 2024. If the community finds a way to maintain a database of this information, possibly in improved form, this package needs to find a way to keep up with the changes. This affects in particular the data.frame `colbournBigFrame`, which is being used as the basis behind decisions on which methods to implement.

As the source information in the Colbourn tables is sometimes very terse, it is not always intelligible for uninitiated humans, including the package author. The function `Ns` always includes the `eCAN` value from `colbournBigFrame` without source information. There are some instances for which a better CA has meanwhile been found, but there are also a few instances, for which a claimed size in `colbournBigFrame` is wrong; these have not been corrected, as the table (at present) reflects a historic state of the Colbourn tables.

Source

downloaded from https://www.public.asu.edu/~ccolbou/src/tabby in November 2024; that source is no longer available, a slightly outdated version is available at https://web.archive.org/web/20231118015055/https://www.public.asu.edu/~ccolbou/src/tabby, and a web version of the data frame is at https://github.com/ugroempi/CAs/blob/main/ColbournTables.md.
The creation of the object is documented by the R file colbournBigFrame_create.R in the extdata folder for this package, which can be located in the directory inst/extdata of the GitHub repo.

References

Colbourn (without year)

Examples

```
head(colbournBigFrame)
## Kleitman and Spencer () and Katona () provide the only construction
## that is known to be globally optimal for v=2 and t=2.
## It is implemented in function \code{\link{KSK}}.

tail(colbournBigFrame)
## some entries have excessive N
## this is also an example for the difficulty of deciphering the source entries:
## the package author is not aware of the meanings of "N-CT" (what does the "N" stand for?),
## "Arc", and "S", so that none of these constructions has been implemented.

length(grep("projection", colbournBigFrame$Source))
length(setdiff(grep("projection", colbournBigFrame$Source),
               grep("postop", colbournBigFrame$Source)))
## 137 of the 144 table entries that use a projection construction
## were improved by postprocessing (which is not yet (?) available in this package)

## an example for an erroneous entry for cyclotomy
colbournBigFrame[2224,]
Ns(3, 128, 6) ## Cyclotomy needs 768 runs (construction 4a, not 3a)
```

---

compositCA *Obtain a CA for non-prime number of levels by composition*

Description

Function to construct a strength t CA in k columns at v levels, v non-prime, from two strength t CAs in fewer levels

Usage

```
compositCA(t, k, v, ...)

N_compositCA(t, k, v, ...)
```

Arguments

t	positive integer: requested strength
k	positive integer: requested number of columns
v	positive integer, non-prime: requested number of levels for each column
...	currently not used

Value

Function compositCA yields a matrix of class ca), with levels from 0 to v-1.

Details

Function compositCA factorizes v into all variants of two factors, picks the pair that produces the smallest run size, and obtains a CA in v levels by composition from two CAs with those numbers of levels, using function crossCAs.

It uses function bestCA for obtaining the ingredient CAs. The result is rarely optimal.

Examples

```
################################################
## compositCA
################################################
D <- compositCA(3, 12, 9)
dim(D)
Ns(3, 12, 9) ## not competitive

D <- compositCA(4, 12, 4)
dim(D)
Ns(4, 12, 4) ## best implemented
eCAN(4, 12, 4)  ## this has not been implemented

Ns(3, 4, 12) ## best (orthogonal array)

Ns(4, 24, 12) ## slightly worse than fuseBushtCA

Ns(4, 12, 8)  ## best implemented

Ns(4, 12, 24) ## much worse than fuseBushtCA,
              ## which almost yields eCAN
```

---

conference_paley *Paley constructions*

Description

Functions to construct Paley-based CAs as well as to construct further objects related to a Paley construction

Usage

```
paleyCA(t, k, ...)

paleyHad(q)

show_paleyHad(minN=NULL, maxN=NULL)

paley_mat(q)

conference_paley(q)
```

Arguments

q	prime or odd prime power
k	integer number: number of columns of the CA
t	integer number: strength of the CA, 3 to 6
minN, maxN	bounds for the run size N. Without specified bounds, the returned data frame has more than 1000 rows.
...	currently not used

Value

Function paleyCA returns a matrix of class ca with levels 0 and 1 in k columns and N_PAYLEYcat(t, k, 2) rows.
Function conference_paley returns a conference matrix, function paleyHad a Hadamard matrix of the Paley construction (matrix) in q or 2q+1 2-level columns (see Section Details). Function show_paleyHad returns a data frame with columns N, k, q, constr, CAN, Source (source entry in the Colbourn tables for the design with fewest possible runs), and N_NISTcat for q up to 10009. Most of the arrays listed can be created; exceptions are q that are powers of primes larger than 50.

Background on Paley-based covering arrays

Function paleyCA uses numerical experiments of Colbourn (2015) to provide the smallest possible strength t CA for 2-level columns that can be obtained using a Paley construction. Results are based on numeric searches of Paley constructions regarding coverage properties of Paley matrices with or without adding a row of ones, as well as doubled such matrices, where doubling a matrix M means vertically stacking M and 1-M, and adding an indicator column for the two halves.
Colbourn (2015) worked out the minimum numbers of replicates for each t-tuple, and where these exceed 1, proposed to remove rows without deteriorating coverage. This package contains a data.frame PALEYcat, which contains an executable code string from processing the results of Colbourn (2015). For strength 3, only constructions without doubling are included, as CAs from doubling are very uncompetitive (small corrections to Colbourn's Table 1 were needed for primes 7, 13 (strength 3 not possible) and 17 (no row can be removed)).

Details

Function paleyCA provides a covering array of strength t for 2-level columns.
Function N_PALEYcat shows the CA run sizes that can be obtained with the paley construction using

function paleyCA,
function k_PALEYcat shows the number of columns that can be accommodated in N runs using function paleyCA.

Function conference_paley yields a q x q conference matrix, while function paley yields a q x q Paley matrix that is derived from the conference matrix by replacing the "-1" values with zeroes.

Based on finite fields and conference matrices, function paleyHad yields OAs in numbers of levels a multiple of 4. Hadamard constructions depend on the value of q %% 4, which can either be 1 or 3:
For q %% 4 = 3, a (q+1) x q OA is obtained,
q %% 4 = 1 yields a (2q+2) x (2q+1) OA.

show_paleyHad shows the Hadamard matrix run sizes that can be obtained with the constructions, and the corresponding values of k and q, as well as the smallest number for a strength 3 array in the same number of 2-level columns according to the Colbourn tables (currently-known upper bound for CAN) with the corresponding source entry.

References

Colbourn (2015), Colbourn and Keri (2009)

Examples

```
## examples for paleyCA

## Paley-based CA that is optimal
D <- paleyCA(t=4, k=67)
dim(D)
caverify::ca_verify(D, 4)
eCAN(4, 67, 2)
# as good as the optimum cyclotomy construction
# (and actually the same)

## other Paley-related functions
# conference matrix (square q x q matrix with diagonal 0)
conference_paley(5)
# paley matrix (square q x q matrix with diagonal 0)
paley_mat(5)

## Hadamard matrix based OAs
# paley construction I for N=12
D12I <- paleyHad(11)
# paley construction II for N=12
D12II <- paleyHad(5)
# they are not the same
table(D12I - D12II)
# but they have the same coverage properties
coverage(D12I, 3)
coverage(D12I, 4)
coverage(D12II, 3)
coverage(D12II, 4)
# paleyHad(13) of construction II
# and paleyHad(27) of construction I for N=28
# have different coverage properties
```

```
# for t=4

show_paleyHad(maxN=36)
  # there is no paley construction for N=16
  # but there is a CA(3,14,2) in 16 runs based on a
  # Hadamard matrix in 16 runs, which can be obtained
  # from package FrF2 as pb(16, nfactors=14)
show_paleyHad(minN=100, maxN=150)
```

`coverage` *coverage and diagnostic plot functions*

Description

functions for calculating covering proportions for a given strength, and for visualizing those

Usage

```
coverage(D, t, isInteger=TRUE, verbose=0, start0=TRUE, parallel=1)

coverage_iter(D, t, isInteger=TRUE, start0=TRUE,
       progress=FALSE, abortnot1=FALSE, start.proj=1)

coverplot(coverageD, D=NULL, t=NULL, isInteger=TRUE, start0=TRUE,
    type="projections",
    main=NULL, xlab=NULL, ylab=NULL,
    col=rgb(0,0,1,0.5), las=1, plot=TRUE, ...)
```

Arguments

`D`	array
`t`	small positive integer, interaction strength to be considered
`isInteger`	logical (default `TRUE`): are the elements of D all consecutive integers starting with 0 or 1?
`verbose`	0 (default) returns only four percentages, 1 and 2 return more detail. Anything apart from 1 and 2 behaves like 0.
`start0`	logical (default `TRUE`): do the integers start with 0 ? (irrelevant for `!isInteger`)
`parallel`	number of threads to be used (default: 1); if increased, package parallel must be available
`progress`	logical (default `FALSE`): should a progress bar be printed ?
`abortnot1`	logical (default `FALSE`): should the function stop at the first projection for which strength `t` coverage is not perfect ?
`start.proj`	integer, the number of the projection to start with in lexicographical order; ignored for `abortnot1=FALSE`; for `abortnot1=TRUE`, assumes that all projections up to `start.proj - 1` were already successfully checked

coverageD	a class CAcoverage object created by function coverage with at least verbose=1. for verbose less than 2, t must be specified. If coverageD is specified, all arguments referring to the design D, except for t in case of verbosity 1 only, are ignored.
type	"projections" (default) or "tuples"
main	title of plot
xlab, ylab	axis titles
las	orientation of tickmark labels
col	color
plot	logical (default TRUE): generate the plot?
...	further arguments to polygon

Value

Functions coverage and coverage_iter return an object of S3 class CAcoverage, which is a list with elements

total	total proportion of covered t-tuples
ave	average of t-tuple proportions taken over projections
min	the minimum proportion of t-tuples covered among the projections
simple	proportion of projections which are fully covered
tots	coverage with verbose=1: the number of t-tuples for the projections
ncovered	coverage with verbose=1: the number of covered t-tuples for the projections
proportions	coverage with verbose=1: ncovered/tots-tuples for the projection
projections	coverage with verbose=2: the projections in a t-row matrix, a column for each projection
tabs	coverage with verbose=2: a list of tables (for mixed levels CAs) or a matrix (for uniform CAs) that shows the number of coverages for each tuple in each projection; the row order of tabs is the same as is used for the numbering of tuples in notcovered (see next item)
notcovereds	coverage with verbose=2: a list of vectors of position numbers of the not-covered tuples for the projection, where position numbers arise having the left-most factor changing fastest
message	coverage_iter with abortnot1=TRUE: message which projection was the first uncovered one and thus stopped the calculations

The coverplot functions return a list of coordinates x and y, strength t and proportions (from call with verbose=1).

Details

If one only needs a TRUE/FALSE check of whether or not perfect coverage is reached, accompanied by the number of uncovered t-tuples and some examples, function ca_verify of package **caverify** is much faster than function coverage.

Function `coverage_iter` calculates the same quantities as function `coverage`; instead of creating all projections at once, it creates them iteratively, which is slower but doable for large cases for which function `coverage` fails for memory reasons. This iterative version is, at least at present, not parallelized. It allows a progress bar (which slightly deteriorates run time, e.g., on my Windows 10 machine, about 7 milliseconds per 1000 projections). It also allows to request abortion on encountering the first violation of strength `t` coverage.
In case of aborting, the function returns `NA` values for all four metrics, and as a fifth list element, the message at which projection the violation occurred; `coverage_iter` does not have a `verbose` argument, but using the `verbose` argument on printing an object that resulted from an aborted run prints that message.
For `abortnot1=TRUE`, function runs that did not abort up to a particular iteration can be restarted at the subsequent iteration. The actual restart point will be the first projection starting with the same first element as the requested `start.proj` e.g., for `t=3` and a matrix `D` with 6 columns, the first ten 3-factor projections start with 1, followed by six projections that start with 2, three projections that start with 3 and a single projection that starts with 4 (10 + 6 + 3 + 1 = 20). `start.proj=12` will start with the projection 2:3:4, i.e., the 11th projection, which is the first that starts with 2. For projections that start with 1, the second factor is also relevant for the start position: the first four projections have 2 in the second position, then there are three with 3 in the second position, then two with 4, and finally one with 5. Thus, for `start.proj=7`, `start.proj` will be reduced to 5, and the projection is 1:3:4 (the first that starts with 1:3).

Per default, the print method for class `CAcoverage` prints the first four elements only.

Function `coverplot`, with `type=projections`, plots the percentage of projections on the horizontal axis and the coverage on the vertical axis (visualizing the average taken over projections); with `type=tuples`, the percentage of tuples (i.e., projections as weighted by tuples) is shown on the horizontal axis, visualizing the total percentage covered. Sorting of the combinations on the horizontal axis is from best to worst coverage.

Warning

For function `coverage_iter`, in case of starting at projection `start.now`, the user is responsible to ensure that all prior runs have also been checked; the calculated values assume that the prior runs showed perfect coverage.

See Also

`caverify::ca_verify()`

Examples

```
## fixed levels
D <- KSK(k=10)
coverage(D, 2)
coverage(D, 3)
## mixed levels, total and ave are different
plan <- cbind(rep(0:2, each=3), c(rep(0:1, 4), 1), c(rep(0:1, each=4),0))
coverage(plan, 2)
## verbose content not printed per default
coverage(plan, 2, verbose=1)
print(coverage(plan, 2, verbose=1), digits=4, verbose=TRUE)
```

```
print(coverage(plan, 2, verbose=2), digits=4, verbose=TRUE)
## two of three projections fully covered,
## one covered 5/6 only

cD2 <- coverage(plan, 2, verbose=2)
coverplot(cD2, type="projections") ## 2/3 at 1, 1/3 at 5/6=0.833333
## the not fully covered projection involves the 3-level
## column and thus has 6/(6+6+4)=37.5 pct of the tuples instead of 1/3
coverplot(cD2, type="tuples")

## it is also possible to call coverplot on the design itself
##    however, repeated calls will then require recalculation
##    of coverage, which can be demanding.
##    Where storage space is problematic for verbose=2,
##    working directly with the design may be doable
##    (but time is likely also an issue).
##
coverplot(D=plan, t=2, type="projections")
coverplot(D=plan, t=2, type="tuples")

## the Hadamard based 16 run design for 14 columns
##    that is available from FrF2::pb;
##    the design has levels -1 and 1 and columns are factors
##    therefore, isInteger=FALSE has to be used
if (require(FrF2)){
   CA3.2tothe14 <- FrF2::pb(16, 14)
   print(coverage(CA3.2tothe14, 3, isInteger=FALSE))
   }
```

`coverageSeqCA` *Checking coverage for SeqCAs*

Description

for a given SeqCA A and a strength t

Usage

```
coverageSeqCA(A, t, stop_at_first_failure=FALSE, verbose=FALSE, ...)
```

Arguments

`A`	a sequence covering array (SeqCA) with k columns and elements 1,...,k in each row
`t`	the strength for which the coverage of all permutations of length t must be guaranteed for all `combn(k,t)` t-element subsets of 1,...,k
`stop_at_first_failure`	logical; if `TRUE`, the function returns FALSE instead of a number

verbose	logical; if TRUE, the returned object has an attribute uncovered with the uncovered sequences
...	currently not used

Value

a number between 0 and 1 with attributes; the number is the proportion of covered orders (may be made more detailed in the future).
The attribute count_effective holds the counts for each row of A of the number of sequences covered here but not earlier, as proposed by Murray and Colbourn (2015).
If the proportion is smaller than one, and if verbose=TRUE, the uncovered sequences are available in the attribute uncovered of the returned value.

Usefulness

It is particularly relevant to check whether the coverage is perfect, i.e., the outcome is 1. If this is the only interest, one can stop at the first failure.

Examples

```
### create a trivial SeqCA of strength 2
A <- rbind(1:5, 5:1)
coverageSeqCA(A, 2)
coverageSeqCA(A, 3) ## covers a third of the 3-factor orders

## a naive greedy SeqCA of strength 3
A <- greedySeqCA_naive(5, 3, seed=2323)
dim(A)
coverageSeqCA(A,3)

A <- reduce_rows_iterative_complete(A, 3)
dim(A)

## also works from worse starting matrix
A <- greedySeqCA_naive(5, 3, seed=589)
dim(A)
A <- reduce_rows_iterative_complete(A, 3)
dim(A)   ## substantially reduced in very short time
```

---

cphfCA *Making CA from CPHF*

Description

according to the process outlined in Wagner, Colbourn and Simos (2022)

Usage

```
cphfCA(t, k, v, start0 = TRUE, ...)

N_cphfCA(t, k, v)

k_cphfCA(t, N, v)

WCS(cphf, q, t, type="2022", ...)
```

Arguments

`t`	integer: target strength
`k`	integer: requested number of columns
`v`	integer: number of levels, must be a prime power
`start0`	logical: should the levels of the output array start with 0 (otherwise with 1)?
`...`	currently not used
`N`	integer: affordable run size
`cphf`	matrix with integer entries that can be expanded into permutation vectors of length `t`; must be a CPHF
`q`	integer: number of levels, must be a prime power
`type`	character string providing the type of construction; for the built-in CPHFs, `"2022"` is the correct entry. It handles CPHFs for which the first entry of the permutation vector refers to the highest power, and which were constructed with Galois fields from SageMath or the Python `numbthy` module (relevant only for prime powers 8, 9, 16, 25; also implemented in this package via the internal function `mygf`) Any type different from the default requests use of the built-in Galois fields of R package **lhs**.

Value

Function `cphfCA` returns a matrix of class `ca` with attributes, which is a covering array of strength 3, 4, 5 or 6.
Function `WCS` returns a matrix with entries in 0,...,v-1, which is a covering array of strength `t`, if a valid `cphf` for strength `t` is provided; the function is not primarily meant for direct use, but is the workhorse behind `cphfCA`.

Functions `N_cphfCA` and `k_cphfCA` return the required run size or the achievable number of columns, respectively.

Details

Function `cphfCA` implements CAs from the Wagner et al. (2022) construction for length `t` h-tuples to create length `v^t` permutation vectors for `v` levels and strength `t`. The workhorse function `WCS` uses a stored covering perfect hash family (CPHF); at present, only the CPHFs from Wagner, Colbourn and Simos (2022) are implemented via the function `cphfCA` (see `WCS_CPHFs` and `CPHFcat`).
Users who have their own valid CPHF (not SCPHF) that is not part of this package can use

the function for creating a CA. Note that the creation of a CA can take a long time for large cases; for example, creation of a CA(501121, 4, 665, 17) from the relevant CPHF with 6 rows (6*(17^4-1)+1=501121) took approximately 140 minutes on the authors Windows computer with 32GB RAM (which appeared to be the limiting factor). This package is not optimized to handle cases as large as this.

The construction relies on a Galois field GF(v). The CPHFs from Wagner et al. (2022) for v=8, v=9, v=16 and v=25 require a different Galois field than otherwise used in this package; these deviating Galois fields are created with the internal function mygf; function WCS takes the argument type, with which these Galois fields are activated (per default), but can also be deactivated in favor of the Galois fields of package **lhs** by specifying any type different from "2022".
The data frame CPHFcat holds the overview information on which constructions are implemented for function cphfCA.

References

Wagner, Colbourn and Simos (2022)

Examples

```
# applying the construction for small CPHF from the Wagner et al. (2022) paper

# strength 3
head(CPHFcat[which(CPHFcat$v==4),])
## the CPHF for the first row
WCS_CPHFs[["3"]][["4"]][["71"]]
D <- cphfCA(3,71,4)
dim(D)

  coverage(D,3)

Ns(3,71,4)   ## size is current best but not eCAN

# strength 6
Ns(6, 10, 3)  ## by far best implemented,
              ## slightly better array from group who does not share their arrays
dim(D <- bestCA(6,10,3))
coverage(D, 6)

## the CPHF for the 91st row has flexible entries
CPHFcat[91,]
rowSums(is.na(WCS_CPHFs[["4"]][["3"]][["3504"]]))
## many flexible values in the last row,
## eCAN claims one row less for this construction
Ns(4,3504,3)   ## by far the best implemented

  ## creation of this CA is slow because of the many columns
  ## it can therefore be suppressed by the run.donttest argument
  ##       to function example()
  D <- cphfCA(4,3504,3)  ## about 1 minute on author's machine
  dim(D)
```

`createDHF` *Auxiliary functions for Colbourn Torres-Jimenez Power construction*

Description

These functions support the implementation of the power constructions by Colbourn and Torres-Jimenez

Usage

```
createDHF(oa, toa, tdhf, vdhf=tdhf)

Tred(DHF, rs, reduce)
```

Arguments

`oa`	an orthogonal array
`toa`	strength of the orthogonal array
`tdhf`	requested strength of the DHF
`vdhf`	number of classes of the DHF, the default yields a PHF
`DHF`	a DHF or DHHF, for which the number of levels for rows indexed by elements of `rs` is to be reduced by `reduce`
`rs`	vector of row indices whose number of levels is to be reduced
`reduce`	number of levels to reduce

Value

Function `createDHF` creates a DHF or PHF of minimum number of rows for the combination of `toa`, `tdhf` and `vdhf`.
Function `Tred` reduces the number of levels for the specified rows by the amount stated in `reduce`, aiming for retaining the maximum number of columns.

Details

Function `createDHF` creates a distributed hash family according to Lemma 1.1 or Lemma 1.2 of Colbourn and Torres-Jimenez (2010).

Function `Tred` applies Lemma 3.1 of Colbourn and Torres-Jimenez (2010).

References

Colbourn and Torres-Jimenez (2010)

Examples

```
## a DHF of strength 3 with 2 classes
## from the Bush array of strength 2
## (which coincides with the Bose array)
mydhf <- createDHF(SCA_Busht(11, 2), 2, 3, 2)
dim(mydhf)
mydhf[,1:6]
mydhf[,116:121]
## reduce the third row by 6 levels to 5 levels
mydhhf <- Tred(mydhf, 3, 6)
dim(mydhhf)
mydhhf[,50:55]

## a PHF of strength 4 from a strength 2 OA
myphf19 <- createDHF(SCA_Bose(19), 2, 4)
dim(myphf19)

## a DHF of strength 4 with only 2 classes
## from a strength 2 OA
mydhf19 <- createDHF(SCA_Bose(19), 2, 4, 2)
dim(mydhf19)
```

`crossCAs` *Cross two strength t CAs with k columns to yield strength t in v*w levels*

Description

Function to construct a strength t CA in k columns at v*w levels from a strength t CA in k columns at v levels and a strength t CA in k columns at w levels

Usage

```
crossCAs(A, B, t=NULL, check=FALSE, ...)
```

Arguments

`A`	an N x k CA of strength `t` with v levels,
`B`	an M x k CA of strength `t` with w levels,
`t`	strength of `A` and `B`, needed for `check=TRUE`
`check`	logical; if TRUE, checks required strength of ingoing CAs (may substantially increase run time for larger CAs, especially for `t>2`)
`...`	currently not used

Value

Function `crossCAs` yields a CA(N*M, t, k, v*w) (N*M x k matrix of class `ca`).
The returned matrix has its levels coded like the ingoing matrices, i.e., integer starting with 0 or 1.

Details

Function `crossCAs` yields a CA(N*M, t, k, v*w) by combining the corresponding columns of the two CAs into `v*w` levels, according to, e.g., Theorem 2.8 of Zhang et al. (2014).

References

Chateauneuf and Kreher (2002), Zhang et al. (2014)

Examples

```
################################################
## crossCAs
################################################
## two CAs with four columns each and strength 3
A <- lhs::createBush(3, 4, bRandom=FALSE)  ## strength 3, 4 columns in 3 levels
dim(A)
B <- lhs::createBush(4, 4, bRandom=FALSE)   ## strength 3, 4 columns in 4 levels
dim(B)
## product array with 3*4=12 levels and 27*64=1728 runs
E <- crossCAs(A, B, 3)
dim(E)
tail(E)
coverage(E, 3)
eCAN(3, 4, 12)      ## the CA is optimal (as of April 2025)
eCAK(3, 1728, 12)   ## the optimal strength 3 12-level CA accommodates two more columns,
                ## and even in an OA (12 is not a prime power, thus 13 columns not possible)
```

---

crossSum — *Cross-sum two codes with k columns in v levels*

Description

Function to construct cross-sum two codes in k columns at v levels, according to CKRS (2010)

Usage

```
crossSum(A, B, ...)

directSum(A, B, ...)
```

Arguments

`A`	an N x k code with (up to) v non-negative levels.
`B`	an M x l code with (up to) v non-negative levels; for function `crossSum`, `l=k` is needed.
`...`	currently not used

Value

directSum returns an N*M x (k+l) matrix.
crossSum returns an N*M x k matrix, which has its levels coded like the ingoing matrices, i.e., integer starting with 0 or 1.

Details

The convenience function directSum yields cbind(kronecker(A, matrix(1,M,1)), kronecker(matrix(1,N,1), B), i.e., it concatenates all pairs of rows of matrices A and B.

Function crossSum yields an N*M x k matrix by adding all pairs of rows from A and B (each row of A to all columns of B). According to CKRS (2010), it is most promising to choose A as a repetition code (RC) (e.g., with rows 0000 1111 2222, for with k=4 and v=3), an RC extended by a few constant columns (ERC, e.g., 000000 111100 222200), a direct sum of repetition codes (DRC, e.g., rows 0000000000000000 1111111111111111 2222222222222222 1111111111111111 2222222222222222 0000000000000000 2222222222222222 0000000000000000 1111111111111111 for the above RC with itself), or a DRC extended by a few constant columns EDRC.

References

Colbourn, Keri, Rivas Soriani and Schlage-Puchta (2010, CKRS)

Examples

```
################################################
## crossSum
################################################
## two codes with fourteen columns each
## A a repetition code
A <- matrix(0:2, 3, 14)
dim(A)
## B from Fig. 1 of CKRS
B <- rbind(
c(0,0,0,0,0,0,0,0,0,0,0,0,0,0), c(0,2,2,2,1,1,2,0,2,0,0,2,1,1),
c(2,0,2,0,1,2,0,1,0,1,1,1,0,2), c(1,0,1,2,2,2,1,1,0,0,0,2,0,0),
c(2,0,0,2,0,0,1,1,2,1,2,1,2,1), c(0,0,1,1,0,2,1,2,2,2,1,0,1,2),
c(1,2,0,0,0,2,1,2,1,0,0,1,2,2), c(0,0,1,2,1,0,0,2,1,0,1,2,2,1),
c(2,2,1,0,2,0,1,0,0,2,1,1,2,1), c(1,2,0,2,1,0,2,1,1,2,1,0,0,2),
c(0,2,2,0,1,0,1,1,2,0,0,2,1,2), c(0,0,1,2,2,0,2,0,1,1,0,1,1,2),
c(2,1,1,1,2,0,0,1,2,0,0,0,2,2), c(0,1,0,0,2,2,2,1,2,0,1,1,0,1),
c(0,2,1,0,1,2,2,1,1,1,2,0,2,0), c(0,2,0,2,2,2,0,0,2,1,1,2,2,2),
c(0,2,1,2,0,0,1,0,2,2,2,2,0,0)
)

dim(B)
## cross-sum array with 3*17=51 runs
E <- crossSum(A, B)
dim(E)
coverage(E, 3)
eCAN(3, 14, 3)      ## the CA is not optimal
eCAK(3, 51, 3)      ## the optimal strength 3 3-level CA
                    ## in 51 runs accommodates two more columns
```

CS_CK *Function to create a strength 4, 5, or 6 CA for 2-level columns based on Colbourn-Keri cover starters*

Description

A strength 4, 5, or 6 CA in k 2-level columns is created from a starter vector of length k or k-1. It has 2*k or 2*(k-1) runs. The construction uses cyclic permutations and is based on Colbourn and Keri (2009).

Usage

```
CS_CK(k, t=4, v=2, starter=NULL, method=NULL, start0=TRUE, ...)

N_CS_CK(t, k, v=2)

k_CS_CK(t, N, v=2)
```

Arguments

`k`	the number of columns of the output CA
`t`	strength of the output CA, can be 4, 5, or 6
`v`	the number of levels of the output CA (always 2)
`starter`	`NULL` for automatic choice (recommended), or a vector of length at least `k-1`; if a vector is provided, it is the users responsibility to ensure the desired properties of the result; the specification of a value for `t` has no effect whatsoever in that case
`method`	`NULL`, or the specification of how to process a custom starter vector; if non-NULL, can be either a number from 1, 2, 3 or a character string from "CS_CK1", "CS_CK2", "CS_CK3"; `method` is ignored, unless `starter` is specified, in which case it is required
`start0`	logical; if TRUE, values of the output CA start at 0, otherwise at 1; values of the starter must always start at 0
`...`	currently not used
`N`	affordable run size

Details

The function implements the designs provided in Colbourn and Keri (2009). The `method` numbers refer to the lemmata in that paper: Methods 1 and 3 yield `N=2*k`, method 2 yields `N=2*(k-1)`; the starter vector has length `k` for method 1 and length `k-1` for methods 2 and 3. Method 3 always yields designs with the last two rows constant.

Besides methods 1, 2 and 3, the function also provides constructions for `t=4,5` by derivation from strengths 5 and 6, respectively (method `derive` in the data.frame object `ColbournKeriCombis`, that provides an overview of the constructions). The cover starters for strength 4 are in the object `ColbournKeriStarters`, the cover starters for higher strengths are calculated with the internal function `cycvec` (that is also used for the cyclotomy constructions of function `cyc`).

Value

CS_CK returns a matrix of class ca with attributes. The built-in starters allow to obtain two constant rows for all three methods, whereas the third method guarantees two constant rows in the result in general (at the bottom).
The "derive" construction does not permit two constant rows.

References

Colbourn and Keri (2009)

Examples

```
eCAN(4,21,2)  ## this is the best construction, it uses method 1
Ns(4,21,2)
D <- CS_CK(21)  ## strength 4 is the default
dim(D)
# coverage(D, 4)

Ns(4, 40, 2)
dim(CS_CK(40)) ## almost as good as best
## construction was derived, stems from a strength 5 CA made with method 3

Ns(6, 430, 2)
dim(CS_CK(430, t=6)) ## best, uses method 2
## closely related to the Paley type construction by Colbourn

## a few Colbourn Keri-related entries in the Colbourn tables apparently lack
## a postop suffix (status: June 2025)
eCAN(5, 359, 2)
eCAN(6, 360, 2)
## deriving the 720 run design yields 360 runs
## the 359 runs of the Colbourn tables must be the result of a postop step
Ns(5, 35, 2)
Ns(5, 359, 2)
Ns(5, 503, 2)  ##  from deriving 504 columns in 1008 runs
```

CS_CMMSSY *Strength 2 CAs constructed by cover starters (CMMSSY)*

Description

Function to construct a strength 2 CA based on cover starters according to Colbourn et al. 2006. k and v must be specified, the number of runs is ((v-1)*k+1) (possibly for a larger value of k if the construction does not exist for the requested k); recursive constructions based on such arrays are also included

Usage

```
CS_CMMSSY(k, v, start0=TRUE, starter=NULL, ...)

N_CS_CMMSSY(t=2, k, v, theoretical=FALSE)

k_CS_CMMSSY(N, v)
```

Arguments

k	number of factors
v	number of levels, v >= 3
start0	should the output design's levels start with zero (default) or 1?
starter	an optional cover starter vector that starts with Inf. This may be useful for cases with k larger than implemented in the stored cover starters. CAUTION: Responsibility for achieving the coverage rests with the user, please check!
...	currently not used
t	requested strength (has to be 2)
theoretical	logical: default requests size available from using stored cover starters, change to TRUE for the size assuming existence of a cover starter for k and v
N	affordable number of runs

Value

CS_MMSSY returns the smallest possible strength 2 CA from the Colbourn et al. (2006) construction for k experimental factors in v levels; it is an N x k matrix with entries from 0 to v-1 or from 1 to v, depending on start0.
N_CS_MMSSY returns the minimum run size needed for the CMMSSY construction with k factors at v levels, or NA, if no such construction exists.
k_CS_MMSSY returns the maximum number of v-level factors that can be accommodated in at most N runs, or NA, if no such construction exists.

Details

Colbourn et al. (2006) denote designs with the factors as the columns and the runs as the rows, like in this package, and contrary to Meagher and Stevens (2005).

The construction uses a distinct starter vectors of k elements with first element Inf (fixed position) and subsequent elements from 1 to v-1.

The starter vector is cycled into a k x k matrix M (internal function circular), which is one key ingredient of the construction. The other key ingredient is addition modulo v-1. Finally, the infinite element is set to v-1.

Functions N_CS_CMMSSY and k_CS_MMSSY return the minimum N or the maximum k for the requested scenario. In case of N_CS_MMSSY, the default to use the available starters can be overridden by the

argument theoretical, which allows to inspect how many runs can be obtained by a suitable user-specified starter vector.

Besides the starters from Colbourn et al. (2006), the function also returns arrays that are obtained by combining pairs, triples or quadruples of such arrays, using function productPCA, once (for pairs) or repeatedly. It would be possible to drive this further, but the function stops at quadruples. Such an approach was proposed for these distinct starters in Colbourn (2004), who also provided a smaller selection of starter vectors. (The approach is analogous to recBoseCA.)

References

Colbourn (2004), Colbourn et al. (2006)

Examples

```
## mini example of Colbourn et al. (2006)
k <- 7; v <- 5
starter <- c(Inf, 0, 0, 1, 3, 2)
CS_CMMSSY(7, 5, starter=starter) ## 29 runs, 7 columns

CS_CMMSSY(9, 6)  ## 46 rows
```

CS_LCDST *Strength 2 CAs constructed by cover starters (LCDST)*

Description

Function to construct a strength 2 CA based on cover starters according to Lobb et al. 2012. k and v must be specified, the number of runs is ((v-f)*k+CAN_(2,k,f), where f is the number of fixed points (assuming that the CAN array is available, which is the case for f=1,2,3 (eCAN in case of f=3))

Usage

```
CS_LCDST(k, v, start0=TRUE, starter=NULL, ...)

SCA_LCDST(k, v, start0=TRUE, starter=NULL, ...)

N_CS_LCDST(t=2, k, v)

k_CS_LCDST(t=2, N, v)
```

Arguments

k	number of factors
v	number of levels, v >= 3
start0	should the output design's levels start with zero (default) or 1?

`starter`	an optional cover starter vector that starts with a fixed value, denoted by a number larger than 100. This may be useful for cases with k larger than implemented in the stored cover starters. CAUTION: Responsibility for achieving the coverage rests with the user, please check!
`...`	currently not used
`t`	requested strength (must be 2, added of unified interface with other N_ and k_ functions)
`N`	affordable number of runs

Value

`CS_LCDST` returns the smallest possible strength 2 CA from the Lobb et al. (2012) construction for `k` experimental factors in `v` levels; it is an `N` x `k` matrix with entries from 0 to `v-1` or from 1 to `v`, depending on `start0`.

`SCA_LCDST` returns the SCA form of that array (works for cases with f=1 and returns an SCA with k1=k-1).
`N_CS_LCDST` returns the minimum run size needed for the LCDST construction with `k` factors at `v` levels, or `NA`, if no such construction exists.
`k_CS_LCDST` returns the maximum number of `v`-level factors that can be accommodated in at most `N` runs, or `NA`, if no such construction exists.

Details

Lobb et al. (2012) denote designs with the factors as the columns and the runs as the rows, like in this package, and contrary to Meagher and Stevens (2005).

The construction uses starter vectors of `k` elements with first element a fixed position, denoted as a number >100, depending on `v`: the fixed points are taken as the last f levels, and they will be made valid levels `v-1` to `v-f` eventually by subtracting 100.

The starter vector is cycled into a `k` x `k` matrix `M` (internal function `circular`), which is one key ingredient of the construction. The other key ingredient is the additive group `Z` of `v-f` elements, which is repeatedly applied to the non-fixed elements. The resulting matrix consists in vertical concatenations of those `v-f` matrices, augmented by the smallest possible CA for `k` f-level columns in order to cover interactions among the fixed levels (a constant row with the only fixed level in case of f=1).

Functions `N_CS_LCDST` and `k_CS_LCDST` return the minimum `N` or the maximum `k` for the requested scenario. In case of `N_CS_LCDST`, the default to use the available starters can be overridden by the argument `theoretical`, which allows to inspect how many runs can be obtained by a suitable user-specified starter vector with the perfect additional ingredient.

References

Lobb et al. (2012)

See Also

LCDSTCombis

LCDSTStarters

Examples

```
## example for k=5, v=3, f=1
starter <- c(102, 0, 1, 1, 0)
coverage(CS_LCDST(5, 3, starter=starter), 2)

## example for k=20, v=8, f=3
## levels 0 to 7, with levels 5 to 7 fixed
## array used by function CS_LCSDT for interactions among fixed points
addfix <- CAEX(20)  ##  20 3-level columns at strength 2
coverage(addfix,2)
dim(addfix)         ## 15 rows
CA115.2.20.8 <- CS_LCDST(20, 8) ## 115 runs, 20 columns at 8 levels
dim(CA115.2.20.8)   ## (v-f)*k + nrow(addfix) = (8 - 3)*20 + 15 rows
coverage(CA115.2.20.8,2)

N_CS_LCDST(k=20, v=8)  ## 115 runs are needed for 20 8-level columns
k_CS_LCDST(N=120, v=8) ## with up to 120 runs, 20 8-level columns are possible
eCAN(2, 20, 8) ## post-processing with simulated annealing may loose seven rows

## f larger than 3
## v=11 levels, f=4, k=26
D <- CS_LCDST(26, 11)
dim(D)
N_CS_LCDST(2,26,11)
LCDSTCombis[which(LCDSTCombis$k==26 & LCDSTCombis$v==11),]
```

CS_MS *Strength 2 CAs constructed by cover starters (Meagher and Stevens)*

Description

Function to construct a strength 2 CA based on cover starters according to Meagher and Stevens. k and v must be specified, the number of runs is ((v-1)*k+1) (possibly for a larger value of k if the construction does not exist for the requested k)

Usage

```
CS_MS(k, v, start0=TRUE, starter=NULL, ...)

SCA_MS(k, v, start0=TRUE, starter=NULL, ...)

N_CS_MS(t=2, k, v, theoretical=FALSE)

k_CS_MS(t=2, N, v)
```

Arguments

k	number of factors
v	number of levels, v >= 3
start0	should the output design's levels start with zero (default) or 1?
starter	an optional cover starter vector. This may be useful for cases with k larger than implemented in the stored cover starters. CAUTION: Responsibility for achieving the coverage rests with the user, please check!
...	currently not used
t	requested strength (must be 2, added of unified interface with other N_ and k_ functions)
theoretical	logical: default requests size available from using stored cover starters, change to TRUE for the size assuming existence of a cover starter for k and v
N	affordable number of runs

Value

CS_MS returns the smallest possible strength 2 CA from the Meagher/Stevens construction for k experimental factors in v levels; it is an N x k matrix with entries from 0 to v-1 or from 1 to v, depending on start0.
SCA_MS does the same, but arranges the result as an SCA (see Section "Partitioned Covering Array (PCA)" in productPCA).
N_CS_MS returns the minimum run size needed for the Meagher/Stevens construction with k factors at v levels, or NA, if no such construction exists.
k_CS_MS returns the maximum number of v-level factors that can be accommodated in at most N runs, or NA, if no such construction exists.

Details

Meagher and Stevens (2005) denote designs with the factors as the rows and the runs as the columns, this package does the opposite.

The construction uses a starter vector of k elements with first element zero and subsequent elements from 1 to v-1. These were identified by searches (exhaustive for v <= 10 with v+1 <= k <= 2*(v+1), hill-climbing for 11<=k<=20). Starter vectors identified by Meagher (2005b) in her thesis, and in part only provided in the unpublished technical report Meagher (2005a), are collected in the internal list MeagherStevensStarters, whose list entry for each v is a list of starter vectors named with k; search-based vectors for small v (3, 4, 5) have been added.
Note that there can be larger, and in some cases also smaller, starter vectors than the ones listed, which is why function CS_MS
permits the specification of a starter vector.
In search of a starter vector for k+1 if a starter vector for k is available, inserting a "1" in the second position is often successful (this technique has also been used for finding suitable vectors for cases for which Meagher only asserted existence without providing a vector).

The starter vector is cycled into a k x k matrix M (internal function circular), which is one key ingredient of the construction. The other key ingredient is the subgroup G of the symmetric group

of permutations of v elements, which cyclically permutes the second to last of the v elements and leaves the first element fixed (i.e., has v-1 elements). The resulting matrix consists in horizontally concatenating a vector of zeroes with the v-1 results of applying to M the functions corresponding to a group element of G.
Group manipulations rely on R package **permutations** (which uses coding 1 to v).

If there is no starter vector for k in the list for v, the function tries to use the next largest k, if possible. If that is not possible, an error is thrown.

Functions N_CS_MS and k_CS_MS return the minimum N or the maximum k for the requested scenario. In case of N_CS_MS, the default to use the available starters can be overridden by the argument theoretical, which allows to inspect how many runs can be obtained by a suitable user-specified starter vector.

References

Meagher and Stevens (2005), Meagher (2005a), Meagher (2005b)

Examples

```
## mini example of Meagher / Stevens
k <- 5; v <- 3
# starter <- c(0,1,1,1,2)
CS_MS(5, 3) ## 11 runs

## smallest element of Meagher / Stevens table II
## k = 9, v = 6
D <- CS_MS(9, 6)
dim(D)
coverage(D,2)
## the matrix M from cyclic permutation of the starter vector
MeagherStevensStarters[["6"]][["9"]]
## is treated by cyclic permutation of all levels expect the lowest
## (v-1) k x k matrices
## and a constant column with the lowest level is prepended

## use k=8, which is not directly in the starter table
D <- CS_MS(8, 6)
dim(D)    ## works by finding the smallest k that is in the table for v=6
N_CS_MS(k=8, v=6)
N_CS_MS(k=8, v=6, theoretical=TRUE)  ## in this case unachievability is proven
N_CS_MS(k=15, v=6) ## not implemented with stored cover starter
N_CS_MS(k=15, v=6, theoretical=TRUE) ## if a cover starter is found and provided,
                                 ## 76 runs are possible with the construction

try(D <- CS_MS(14, 6))  ## not implemented
                        ## (though a starter might exist,
                        ## but likely unattractive result)
N_CS_MS(k=15, v=6, theoretical=TRUE) ## 76 runs possible, if suitable starter provided
MeagherStevensStarters[["6"]][["14"]] ## trying a guess for a starter
starter15 <- c(0,1,1,1,1,1,1,1,1,1,2,4,3,1,2)
D <- CS_MS(15, 6, starter=starter15)
coverage(D, 2)          ## successful guess
```

```
dim(D)
```

---

cyclotomyCA *functions for cyclotomy constructions*

Description

cyc provides cyclotomy constructions according to Colbourn (2010), which yield good high strength arrays for v=2,3,4,6 (see Details section); Ncyc and kcyc provide run sizes and maximum numbers of columns for the respective q, v and construction type.

Usage

```
cyclotomyCA(t, k, v, ...)

cyc(q, v, k=NULL, type=NULL, primitive=NULL)

N_cyc(q, v, type)

k_cyc(q, v, type)
```

Arguments

t	requested strength (coverage)
k	requested number of columns; for cyc, default NULL for maximum possible
v	number of levels for each column
...	currently not used
q	prime or prime power with q=1(mod v)
type	the construction type ("1","2","3","3a","3b","4a", or "4b"), default implies "1"
primitive	the primitive of the finite field to be used for the construction (specification should not be needed, except for cases that are not adequately covered by the internal data.frame primedat)

Value

cyclotomyCA returns a matrix with k columns and values 0 to v-1 in the number of rows that can be obtained by the cyclotomy construction for strength t;
the matrix has class ca.

cyc returns a matrix with k columns and values 0 to v-1 with the number of runs corresponding to the construction (see Details section). k_cyc and N_cyc return the maximum possible k and the run size N for the given parameters.

Details

Function `cyclotomyCA` allows to specify `t`, `k` and `v` and to obtain the smallest *implemented* CA from the cyclotomy construction for this specification (if one exists). This package only automatically implements those CAs that have optimum size as of the state of the Colbourn tables of best bounds of November 2024.

As various mistakes were found in tables of Colbourn (2010), and the package author does not want to take responsibility for the strength of CAs according to those tables from the paper, the tables from Colbourn (2010) were not used for implementing further cyclotomy-based CAs. It is very well possible that there is a smaller CA for the purpose from a cyclotomy construction; users who know that a certain prime power `q` yields a smaller CA for their requirements can use function `cyc` for creating that CA.

The implemented constructions include three strength 5 cases attributed to Torres-Jimenez, for which `q%%v==2` (and not 1, as required in Colbourn 2010). Limited checks by sampling suggest that these are OK, but no full checks have been conducted, and the implementation trusts that the Colbourn tables are correct (except in one case, which was omitted because the alleged `q` is not a prime or prime power (13*229), and in three further instances (q=127, 2161, 2311), for which the size found in the Colbourn tables is too small, as it is based on construction 3a, while construction 4a is needed (thanks to Christopher Smolen for highlighting the coverage violation)).

The function `cyc` is meant for experts and yields designs with columns in `v` levels based on prime or prime power `q` as follows:

for N=k=q=1(mod v) prime power (Construction 4.1)
or N=q+v-1, k=q, with q=1(mod v) prime power (Construction 4.2)
(add constant rows with elements 1, 2, ..., v-1
to construction 4.1, as these are missing)
or N=vq, k=q, with q=1(mod v) prime power (Construction 4.3)
(juxtapose vertically A, A+1, ..., A+v-1 (mod v);
this is called developing modulo v)
or 4.3a: k=q+1
(extend A by column of 0s before developing)
or 4.3b: N=vq+v^2-v, k=q+1
(add non-zero constant rows (like in 4.2),
then add column of 0s before developing)
or N=vq+v, k=q, with q=1(mod v) prime power (Construction 4.4)
add constant row of zeroes to A before developing
or 4.4a: k=q+1
also add constant column of zeroes
or 4.4b: N=vq+v^2-v, k=q+1 (same as 3.b)
add to A all nonzero constant rows
and a column of ones, then develop

The parameters have to be specified by the user. They can, e.g., be selected from cyclotomy construction entries of `CAs:::colbournBigFrame`, or from the Colbourn (2010) paper.

Warning

For valid parameters, function `cyc` always returns an array, but coverage properties are only guaranteed for well-chosen parameters!

Examples

```
## the main function for practical use
CA768.3.128.6 <- cyclotomyCA(3, 128, 6)  ## strength 3 CA with 128 columns in 2 levels each
## the best existing CA, though a bit worse than claimed in the Colbourn tables,
## because the construction claimed there does not cover all triples (one of very few mistakes)
dim(CA768.3.128.6)
eCAN(3,128,6)
Ns(3,128,6)

## function cyc for expert use
## construction 1
## 2-level
CA19.3.19.2 <- cyc(19, 2) ## q=19, k=19
coverage(CA19.3.19.2, 3)
coverage(CA19.3.19.2, 4)

# 3-level
CA93.3.32.3 <- cyc(31, 3, type="3a")
coverage(CA93.3.32.3, 3)
dim(CA93.3.32.3)
k_cyc(31,3,"3a")
N_cyc(31,3,"3a")

## reduce k versus the default 32
CA93.3.23.3 <- cyc(31, 3, k=23, type="3a")
dim(CA93.3.23.3)
```

`derive` *Function to derive a strength t+1 CA into a smaller strength t array*

Description

A CA D of strength t+1 is reduced to a single level of a particular column, which is subsequently omitted from the array. The resulting CA has strength t.

Usage

```
derive(D, dcol=1, dval=NULL, ...)

deriveto(N, k, v, ...)

derivefrom(N, k, v, ...)
```

Arguments

D	a CA with levels as integers 0, ..., v-1 or 1,...,v
dcol	the id of the column to be deleted (default: 1)

dval	the value to be used for filtering (default: the first value that has minimum frequency in column dcol)
...	currently unused
N	integer: run size; for deriveto: the size that should result from derivation; for derivefrom: the size of the array that is to be derived
k	integer: number of columns; for deriveto: the number of columns that should result from derivation; for derivefrom: the number of columns of the array that is to be derived
v	integer: the number of levels (same for ingoing and resulting array)

Details

The name derive was taken from Zhang et al. (2014). derive returns a strength t CA from a strength t+1 array by
picking one column and selecting only the runs for a single level of that column (if not balanced, use the least frequent level for obtaining the smallest run size)
and subsequently omitting that column.
deriveto predicts the dimensions of a derived array, when deriving a CA(N,t+1,k,v), assuming maximum possible balance in the column to derive on (which is worst case).
derivefrom provides the dimensions of a CA from which the specified parameters can be obtained by derivation; note that the run size is not unique, because, e.g., deriving a 20 run Ca or a 21 run CA with v=2 will both yield 10 runs. The result for N is therefore a vector of possibilities.

Value

derive returns a matrix with class ca, which has strength one less than D; its levels are coded in the same way as those of D.
deriveto returns a named vector with integer-valued elements N and k.
derievefrom returns a list with elements N (an integer-valued vector) and k (an integer number).

Examples

```
# create a CA(6,2,10,2) from a CA(12,3,11,2)

A <- paleyHad(11)  ## a CA(12,3,11,2)

sixdefault <- derive(A)  ## first column of A was 0,
                         ## first column omitted
dim(sixdefault)
coverage(sixdefault, 2)
eCAN(2,10,2)  ## the array is optimal

## default construction can be modified:
## select all the runs with 11th column at level 1
## keep first 10 columns for these
six <- derive(A, dcol=11, dval=1)
dim(six)
## strength 3 was reduced to strength 2
```

```
coverage(six, 2)

deriveto(24, 12, 2)
derivefrom(24, 12, 2)

# t=3 v=2 k=5
eCAN(3,5,2) ## N=10, Derive from strength 4
derivefrom(10, 5, 2)
# t=4, v=2, k=6, N in 20 or 21
eCAN(4,6,2) ## N=21, Derive from strength 5
derivefrom(21, 6, 2)
# t=5, v=2, k=7, N in 42 or 23
eCAN(5,7,2)
## has 42 runs, Source "Special (Yan Jun)"
# this or equivalent CA is available in TJ library
N_TJcat(5, 7, 2)
```

DHF2CA *Make a larger CA from a DHF and a smaller CA*

Description

Function to construct a larger CA by replacing levels of a DHF with columns of a smaller CA

Usage

```
DHF2CA(P, D, v = max(D) + 1, unique=TRUE, ...)
```

Arguments

P	an M x k DHF of strength t with w levels and at least v classes (a PHF of strength t with w levels is a special case)
D	an N x w CA of strength t with v levels
v	number of levels of D, automatically detected in case of coding as 0,...,v-
unique	logical; if FALSE, does not remove non-unique rows from the outcome
...	further arguments to function maxconstant

Value

a CA(chi + (N-rho) * M, t, k, v) (matrix, not of class ca, as there are no guarantees regarding the strength, because guarding this is left to the user or to calling functions).

Details

A PHF is a perfect hash family, a DHF a distributed hash family (weaker condition).
Function `DHF2CA` yields a CA(chi + (N-rho)*M, t, k, v) from replacing the levels of the DHF (or the PHF) with the columns of the CA, leaving out the constant rows of the CA and adding a few (chi) back in, if needed, where chi=max(0, rho - (M - 1)(v - rho)), according to Corollary 2.4 of Colbourn and Torres-Jimenez (2010).
The function uses function `maxconstant` to maximize the constant rows of the CA.

The function does not check for the strengths of the ingoing arrays. Hence, it is the users responsibility to ensure that these are as needed.

References

Colbourn, Dougherty and Horsley (2019) and references therein, as well as Colbourn and Torres-Jimenez 2010.

Examples

```
########################################
## DHF2CA
########################################

## example from Table 1 of Colbourn and Torres-Jimenez
P <- t(SCA_Bose(11)[,1:3])
dim(P) ## levels 0, ..., 10
D <- paleyCA(3,11) ## 2-level, one constant row
dim(D)
(chi <- max(0, 1 - 2*(2-1)))
aus <- DHF2CA(P, D)
dim(aus)  ## 0 + 3*(12-1) rows, 121 columns
eCAN(3, 121, 2)  ## optimal

# further examples are demos only, they are not for good use cases

# a DHF[4,10,4,4,2], i.e., 4 rows, 10 columns, 4 levels, strength 4, 2 partitions
# from Colbourn et al. 2021, Table 1
P <- cbind(rep(1,4), c(1, rep(2,3)), c(1, rep(3,3)),
           rbind(2, rbind(1:3, c(2:3,1), c(3, 1:2))),
           rbind(3, rbind(1:3, c(3, 1:2), c(2:3, 1))),
           rep(4,4))
## CA with v=2 levels, 4 columns, strength 4 (--> full factorial)
D <- as.matrix(expand.grid(0:1,0:1,0:1,0:1))    ## two rows can be made constant
(chi <- max(0, 2 - 3*(2-2)))
aus <- DHF2CA(P, D, unique=FALSE)
dim(aus)     ## 2 + 4*(16-2)= 58 rows, 10 columns
aus <- DHF2CA(P, D)
dim(aus)     ## 52 x 10
             ## 6 duplicate rows removed
             ## (this happens for poorly chosen ingredients)
coverage(aus, 4)  ## works

# a PHF(4,4,19,9) from the Dwyer catalogue
```

```
P <- rbind(c(4, 2, 3, 7, 4, 0, 1, 2, 5, 3, 4, 5, 7, 6, 1, 0, 1, 8, 8),
           c(6, 2, 8, 4, 2, 1, 1, 0, 7, 5, 3, 8, 7, 4, 0, 4, 5, 6, 8),
           c(7, 1, 4, 0, 2, 5, 8, 5, 1, 7, 5, 3, 6, 4, 0, 3, 2, 6, 8),
           c(4, 8, 5, 0, 0, 1, 2, 5, 1, 1, 7, 7, 2, 4, 3, 8, 6, 3, 0))
# we need a strength 4 CA with nine columns
D <- paleyCA(4, 9)   ## is current best, has two constant rows
aus <- DHF2CA(P, D, unique=FALSE)
(chi <- max(0, 2 - (4-1)*(2-2)))
dim(aus)     ## not competitive, although D is optimal 2 + (24-2)*4
aus <- DHF2CA(P, D)
dim(aus)     ## 38 duplicate rows removed, still relevantly worse than current best
Ns(4, 19, 2)

## best for 6-level is in DWYERcat, 2768 runs for 9 columns in 6 levels

  pathGH <- "https://raw.githubusercontent.com/aadwyer/CA_Database/main/Repository/CA"
  # the instruction line starts with a comment character and is ignored,
  # therefore ninstruct=0
  D <- readCA(paste0(pathGH, "/CA_2768_4_9_6.txt"), ignore.chars=c("]", "["),
              ninstruct=0, sep=",")
  ## DHF2CA needs quite a while for this case
  ## activated one_is_enough argument for maxconstant for run time reasons
  aus <- DHF2CA(P, D, one_is_enough=TRUE)  ##
  (chi <- max(0, 1 - (2-1)*(6-1)))
  dim(aus)  ## 4*2767 = 11068, much more than the 5422 in the Colbourn tables
  coverage(aus, 4)
```

DHHF2CA *Make a larger uniform CA from a DHHF and smaller CAs*

Description

Function to construct a larger CA by replacing levels of a DHHF with columns of smaller CAs

Usage

```
DHHF2CA(P, Dlist, v = max(Dlist[[1]]) + 1, ...)
```

Arguments

P	an M x k DHHF of strength t for partition size p=min(t,v) with several blocks of rows, where the ith block has w_i levels (a PHHF is the case with partition size t)
Dlist	a list of N_i x w_i CAs of strength t with v levels each
v	number of levels of all elements of Dlist, automatically detected in case of coding as 0,...,v-1
...	further arguments to function maxconstant, e.g., one_is_enough=TRUE for suppressing the search for more constant rows

Value

a CA(chi + sum_i=1^c (N_i-rho_i)*u_i, t, k, v) (matrix, not of class `ca`, as there are no guarantees regarding the strength, because guarding this is left to the user or to calling functions).

Details

A PHHF is a perfect heterogeneous hash family, a DHHF a distributed heterogeneous hash family (weaker condition); heterogeneity means that there are c blocks with different numbers of levels, w_1,...,w_c, and there are u_i rows for the ith block, with u_1 +...+ u_c = M.
Function `DHHF2CA` yields a CA(chi + sum_i=1^c (N_i-rho_i)*u_i, t, k, v) from replacing the levels of the DHHF with the columns of the corresponding CA from* `Dlist`,
leaving out the rho_i constant rows of the ith CA and adding a few (chi) constant rows back in, if needed, where chi = max(0, v - sum_i=1^c u_i(v - rho_i)). The construction is from Theorem 2.3 of Colbourn and Torres-Jimenez (2010).
The function uses function `maxconstant` to maximize the constant rows of the CAs.

There may be (likely rare) situations with many different numbers of levels in the DHHF for which the function's handling of constant rows could be slightly improved; for these cases, the result will not have the minimum number of rows possible from the construction, but one or very few more rows.

The function does not check for the properties of the ingoing arrays. Hence, it is the users responsibility to ensure that these are as needed.

References

Colbourn and Torres-Jimenez (2010), Ji and Yin (2010) for the example

Examples

```
##########################################
## DHHF2CA
##########################################

## the Ji and Yin (2010) strength 3 OA in six 12-level columns (Lemma 3.5 with Corollary 3.3)
## available in this package as oa1728.12.6
P <- t(oa1728.12.6[,1:5])
P <- Tred(P, 4:5, 1)
## two columns reduced to 11 levels each
dim(P)
eCAN(3, 12, 2) ## 15 runs, in TJ2level_CAs
eCAN(3, 11, 2) ## 12 runs, paleyCA also yields this
D <- DHHF2CA(P,
    list(bestCA(3,12,2), bestCA(3,11,2)))
dim(D)
eCAN(3, 1452, 2)  ## optimal, this construction
```

dpCA *Direct product and generalized direct product for two CAs*

Description

Functions to construct a CA from a (generalized) direct product of two ingoing CAs

Usage

```
dpCA(k, v, ...)

productCA(D1, D2, c1=NA, c2=NA, check=FALSE, dupremove=TRUE, generalized=TRUE, ...)

productCA_raw(D1, D2)
```

Arguments

k	number of columns
v	number of levels
...	further arguments to function maxconstant, e.g., one_is_enough=TRUE in case of large D1 and D2.
D1	an N x k CA of strength 2 with v levels
D2	an M x l CA of strength 2 with v levels
c1	for generalized=TRUE, number of constant rows at the top of D1; specifying this (correctly) substantially speeds up the function for large D1
c2	for generalized=TRUE, number of constant rows at the top of D2; specifying this substantially speeds up the function for large D2, if D1 has fewer than v constant rows (ignored otherwise)
check	logical; if TRUE, checks required strength of ingoing CAs (may substantially increase run time for large CAs)
dupremove	logical; if TRUE, removes duplicated rows of the result (if any)
generalized	logical; if TRUE, exploits constant rows. TRUE is the recommended choice, as it reduces the number of runs from N+M to a run size between N+M-v and N+M-2, depending on the number of constant rows.

Value

Function productCA returns a CA(N + M - min(v, s+r), 2, kl, v) (matrix of class ca).
Function productCA_raw returns the simple (N + M) x (k*l) direct product of its two arguments.

Details

Function dpCA yields a uniform strength 2 CA with k columns in v levels, based on the pre-inspected combination of ingredient CAs contained in the data frame DPcat; the calculations invoked use the function productCA that implements (a simple form of) the generalized direct product construction presented in Colbourn and Torres-Jimenez (2013).

Function productCA yields a CA(N + M - min(v, s+r), 2, kl, v) from a product construction,
for which D1 can be rearranged to have s constant rows
and D2 can be rearranged to have r constant rows
as proposed in Corollary 4.3 of Colbourn and Torres-Jimènez (2013)
(removal of duplicates may reduce run size further, for very poor ingoing arrays).
For large and efficient ingoing arrays, it may be wise to specify one_is_enough=TRUE (as part of ...); in that case, the search for the maximum number of constant rows is skipped, and the resulting run size is N + M - 2.

Function productCA also serves as work horse function for combining two CAs in some instances of the CAEX construction.

Function productCA_raw is mainly meant for calculation of interim results in other functions, e.g., productPCA. It works on any pair of numeric matrices that do not need to be integer-valued or CAs (see Examples Section).

References

Colbourn and Torres-Jimenez (2013)

Examples

```
############################
#### dpCA ##################
############################
Ns(2, 25, 6)
D <- dpCA(25, 5)
coverage(D, 2)

########################################
## productCA
########################################
###  (N+M-2 x k*l matrix)
A <- cyc(19,2)
B <- KSK(k=12)
dim(A); dim(B)
E <- productCA(A, B)
dim(E)
## check coverage with the fast function
caverify::ca_verify(E, 2)
## 19 + 7 - 2 runs, twice as large as the optimal 12
eCAN(2,228,2)

A <- lhs::createBose(4, 5, bRandom=FALSE)
dim(A)
E <- productCA(A, A)
```

```
## check coverage with the fast function
caverify::ca_verify(E, 2)
dim(E)  ## 30 runs with 25 columns, close to optimal
eCAN(2, 25, 4) ## 29 runs
eCAK(2, 30, 4) ## 30 columns

## from two CAs with many constant rows
## (not useful for this particular A, for demo purposes only)
A <- lhs::createBush(4, 5, bRandom=FALSE)
dim(productCA(A,A)) ## using default behavior
                     ## that exploits maximum constant rows
                     ## reduces N+M by up to v runs
dim(productCA(A,A, one_is_enough=TRUE))
                     ## preventing function maxconstant from conducting a
                     ## clique search, reduces N+M by 2 runs

## the function also works for designs with flexible values (which are denoted by NA)
## productCA
D <- productCA(CAEX(N=16), CAEX(N=16))
dim(D)
eCAN(2,441,3) ## slightly worse than the best CA, which is obtained as CAEX(N=28)
eCAK(2,30,3) ## a lot more columns are possible in 30 runs

### productCA_raw works on arbitrary numeric matrices
### and applies the simple direct product
A <- matrix(1:12, nrow=4)
B <- matrix(10*(1:6), nrow=3)
productCA_raw(A, B)
```

DWYERcat *Data for function dwyerCA*

Description

The data.frame DWYERcat supports the identification and download of CAs from the DWYER GitHub repository via function dwyerCA.

Usage

```
DWYERcat
```

Format

DWYERcat is a data.frame with columns t (strength), v (number of levels), k (number of CA columns), N (number of CA rows) and Source (a brief descriptive source entry by Aaron Dwyer).

Details

The repository is large and has not been included into the package. Function dwyerCA calculates the requested array (in case of an OA) or downloads it on the fly.

Source

The source is Dwyer (2024), who cites personal communications with Colbourn as the source of many of the arrays; there is some overlap between this repository and CKRScat.
The creation of the object is documented by an R file in the extdata folder for this package, which can be located in the directory inst/extdata of the GitHub repo; the many raw arrays are not provided there, as they are available on the web.

References

Dwyer (2024)

Examples

```
head(DWYERcat) ## will be constructed with function SCA_Busht
tail(DWYERcat) ## will be downloaded on the fly, except the last one
```

---

fuseBoseCA *Function to create a strength 2 CA by fusion applied to a Bose OA*

Description

A strength 2 CA in q+1 (q-1)-level columns and q^2-3 runs is created from a q-level Bose array arranged as SCA

Usage

```
fuseBoseCA(k=NULL, v=NULL, fixNA=TRUE, ...)

fuseBose(q, fixNA=TRUE, ...)

fuseBushtCA(t, k, v, fixNA=TRUE, ...)

N_fuseBose(k=NULL, v=NULL, ...)

k_fuseBose(N, v=NULL, ...)

N_fuseBusht(t, k, v, ...)

k_fuseBusht(t, N, v, ...)
```

Arguments

k	NULL or requested number of columns (q+1); if NULL, inferred from v as k=v+2; if specified, k-1 must be a prime power, or k will be increased to the nearest such number.

v	NULL or number of levels; if NULL, inferred from k as v=k+2 or from N as sqrt(N+3)-1; if k is also specified, it takes precedence.
fixNA	logical; if FALSE, keeps flexible values; if TRUE, the first q-3 rows are made constant.
...	currently not used
q	a prime power
t	the requested strength
N	maximum acceptable number of runs

Details

Colbourn (2008) proposed the particular variant of fusion for a Bose CA, which allows to reduce the number of runs by 3 for strength 2 and at most q+1 columns. It creates a CA(q^2-3, 2, q+1, q-1) from an OA(q^2, 2, q+1, q). The reduction in rows is achieved by using a graph coloring approach (Lemma 3.2 of Colbourn 2008).

The fusion of Bush strength t CAs is included for cases for which no other better CA is implemented. It creates a CA(q^t-2*c, t, q+1, q-c) (with c the distance of v to the next largest prime power q)

Value

fuseBose returns a (q^2-3 x q+1 matrix with columns in q-1 levels labeled from 0 to q-2 that is a covering array of strength 2.
fuseBoseCA returns a matrix of class ca from inputs k and v.
fuseBushtCA returns a matrix of class ca which is obtained from the smallest possible strength t Bush construction SCA created with SCA_Busht by the necessary number of fuse steps.
N_fuseBose and k_fuseBose, N_fuseBusht and k_fuseBusht return a named vector with elements N, k, v and q.

References

Colbourn (2008, Lemma 3.2) for fuseBose

Examples

```
# create a CA(22,2,6,4)
fuseBose(5)  ## two constant rows
fuseBose(5, fixNA=FALSE) ## two flexible values
head(fuseBoseCA(6, 4)) ## two constant rows
attributes(fuseBoseCA(6, 4, fixNA=FALSE)) ## flexible values

# create a CA(46,2,8,6)
dim(D <- fuseBose(7))
coverage(D, 2)
eCAN(2,8,6) ## four runs difference

# create a CA(61,2,9,7)
dim(fuseBose(8))
eCAN(2, 9, 7)  ## two runs difference
```

```
# create a CA(166,2,14,12)
dim(fuseBose(13))
eCAN(2, 14, 12)  ## one run difference

# create a CA(508,3,9,6)
dim(fuseBushtCA(3, 9, 6))
eCAN(2, 14, 12)  ## one run difference

# querying N
N_fuseBose(k=21)
eCAN(2, 24, 22) ## one run difference
eCAK(2, 526, 22) ## as good as it gets

N_fuseBose(v=6)

## run size from fuseBusht for t=3,k=9,v=7
N_fuseBusht(3,9,7)

## fuseBusht is very close to eCAN
## postopNCK removes a few rows
## (not in its implementation in this package)
Ns(3, 18, 15)

## fuseBusht is the best implemented solution
Ns(3,14,12)
## it is behind the eCAN entry
eCAN(3, 14, 12)
## apparently, postopNCK can remove 25 runs
## with the implementation in the package the effort
## for this appears somewhat unreasonable
```

---

gen.next.cbn *auxiliary functions*

Description

non-visible functions to be used in other functions, and currently also previous drafts that may be deleted

Details

gen.next.cbn matcheck: checks a matrix for suitability cycvec: creates a cyclic start vector circular: creates matrix from cyclic start vector levels.no.NA: extracts the levels including NA iacheck: check a two-column matrix for correct interaction structure subia: checks whether ia1 is subset of ia2 levels.no from DoE.base ord from DoE.base nchoosek: matrix of all subsets rho: check for rows with a given interaction in a design D allints: function that extracts all interactions for (up to) t factors from a run failcands: fail candidates from a test suite with a single failed run failcands2: (failed) attempt to speed up failcands rhos_for_ints is.CA: checks for covering array properties,

can return distribution of covering frequencies or actual row sets is.LA: checks for locating array properties is.DA: checks for detecting array properties (not ready) metrics: a function for calculation metrics, likely obsolete proportions_slower: earlier version of coverage, likely obsolete fasttab: table replacement without using factors LS_to_D3cols: function for making a latin square into a three column design D3cols_to_LS: function for bringing a three column design to latin square format rho: extracts run numbers of D that hold the interaction ia Double: doubles a 0/1 matrix M by stacking M with 1-M and adding first column of 0/1 AGL2: from Chateauneuf and Kreher 2002

`greedySeqCA_Kuhn` *Greedy random creation of an SeqCA*

Description

for k columns and strength t according to an algorithm by Kuhn et al. (2012)

Usage

```
greedySeqCA_Kuhn(k, t, nsamp=10, postopt=TRUE, seed=NULL, ...)

iter_greedySeqCA_Kuhn(niter, k, t, nsamp=10, postopt=FALSE,
    seed_init=NULL, verbose=TRUE, ...)
```

Arguments

`k`	positive integer (at least t); the number of columns
`t`	the strength for which the coverage of all permutations of length t must be guaranteed for all `combn(k,t)` t-element subsets of 1,...,k
`nsamp`	the number of runs sampled in each step of the algorithm (the N of the paper); the default 10 will likely be too small for realistic applications
`postopt`	logical; default depends on function; if TRUE, the function `reduce_rows_iterative_complete` tries to make rows redundant and remove them. If this takes too long, `postopt` should be FALSE.
`seed`	an optional positive integer seed for random sampling; specifying a seed makes the array generation reproducible
`...`	currently not used
`niter`	the number of independent attempts to make
`seed_init`	an optional positive integer seed for the first iteration; specifying a seed makes the array generation reproducible
`verbose`	logical (default TRUE), controlling how much printed output is created

Value

a strength t SeqCA with k columns; all rows contain the integers 1 to k.

Details

The function `greedySeqCA_Kuhn` implements the construction of Kuhn et al. (2012). It starts from t! random permutations of the the integers 1,...,k and adds a further random permutation, until all orders of all t-tuples are covered.
For `postopt=TRUE`, it then *identifies the redundant elements and removes all rows that contain redundant elements only (would be easiest, but at present not done)* applies iterative run size reduction using function `reduce_rows_iterative_complete`.

The function `iter_greedySeqCA_Kuhn` iteratively repeats the process and keeps the best outcome.

Use of AI

Claude 4 was involved in the development of these functions (mainly for debugging).

Author(s)

Ulrike Groemping

References

Kuhn et al. (2012)

See Also

`greedySeqCA_naive()` for a naive random assembly of an SeqCA and `greedySeqCA_TJ()` for the graph-based greedy stage method of Torres-Jimenez et al. (2022)

Examples

```
## without post-optimization
nrow(greedySeqCA_Kuhn(5, 3, seed=2323, postopt=FALSE))
## relatively unlucky:
nrow(greedySeqCA_Kuhn(5, 3, seed=2415, postopt=FALSE))
## relatively lucky:
nrow(A <- greedySeqCA_Kuhn(5, 3, seed=16, postopt=FALSE))
coverageSeqCA(A, 3)
coverageSeqCA(A, 4)

## with post-optimization
## differences become smaller (still relevant esp. for larger cases)
nrow(greedySeqCA_Kuhn(5, 3, seed=2323))
## relatively unlucky:
nrow(greedySeqCA_Kuhn(5, 3, seed=2415))
## relatively lucky:
nrow(A <- greedySeqCA_Kuhn(5, 3, seed=16)) ## unchanged

## iteration
nrow(iter_greedySeqCA_Kuhn(10, 5, 3, seed_init=2415)) ## start unlucky
  ## larger niter brings the run size down to 8
```

greedySeqCA_naive *Greedy random creation of an SeqCA*

Description

for k columns and strength t

Usage

```
greedySeqCA_naive(k, t, seed=NULL, postopt=TRUE, ...)

iter_greedySeqCA_naive(niter, k, t, seed_init=NULL, verbose=TRUE, ...)
```

Arguments

k	positive integer (at least t); the number of columns
t	the strength for which the coverage of all permutations of length t must be guaranteed for all `combn(k,t)` t-element subsets of 1,...,k
seed	an optional positive integer seed for random sampling; specifying a seed makes the array generation reproducible
postopt	logical; default TRUE, i.e., rows with all elements redundant are identified and removed. If this takes too long, change to FALSE
...	currently not used
niter	number of independent attempts
seed_init	initial seed for iterative application
verbose	logical (default TRUE), controlling how much printed output is created

Value

a strength t SeqCA with k columns; all rows contain the integers 1 to k.

Details

Function `greedySeqCA_naive` starts from t! random permutations of the the integers 1,...,k and adds a further random permutation, until all orders of all t-tuples are covered. For `postopt=TRUE`, it then identifies the redundant elements and removes all rows that contain redundant elements only.

Function `iter_greedySeqCA_naive` makes `niter` independent attempts and improves the best of those by removing rows that contain redundant elements only.

Use of AI

Claude 4 was involved in the development of these functions.

Author(s)

Ulrike Groemping

See Also

greedySeqCA_Kuhn() for the greedy method by Kuhn et al. (2012) and greedySeqCA_TJ() for the greedy stage method of Torres-Jimenez et al. (2022)

Examples

```
## without post-optimization
nrow(greedySeqCA_naive(5, 3, seed=2323, postopt=FALSE))
## relatively unlucky:
nrow(greedySeqCA_naive(5, 3, seed=2415, postopt=FALSE))
## relatively lucky:
nrow(A <- greedySeqCA_naive(5, 3, seed=9456, postopt=FALSE))
coverageSeqCA(A, 3)
coverageSeqCA(A, 4)

## with post-optimization
## differences become less dramatic (but still relevant)
nrow(greedySeqCA_naive(5, 3, seed=2323))
## relatively unlucky:
nrow(greedySeqCA_naive(5, 3, seed=2415))
## relatively lucky:
nrow(A <- greedySeqCA_naive(5, 3, seed=9456))
coverageSeqCA(A, 3)
coverageSeqCA(A, 4) ## worse coverage of 4-sequences than for larger SeqCA

## iterative with unlucky start
Aiter <- iter_greedySeqCA_naive(10, 5, 3, seed_init=2415)
nrow(Aiter)
## niter=20 better than niter=10
Aiter <- iter_greedySeqCA_naive(20, 5, 3, seed_init=2415)
nrow(Aiter)

## further improvement is possible by more substantial attempt
nrow(reduce_rows_iterative_complete(Aiter, 3))
```

---

greedySeqCA_TJ *Greedy random creation of an SeqCA*

Description

for k columns and strength t according to an algorithm by Torres-Jimenez et al. (2022)

Usage

```
greedySeqCA_TJ(k, t, postopt=TRUE, seed=NULL, ...)

iter_greedySeqCA_TJ(niter, k, t, seed_init=NULL, verbose=TRUE, ...)
```

Arguments

k	positive integer (at least t); the number of columns
t	the strength for which the coverage of all permutations of length t must be guaranteed for all `combn(k,t)` t-element subsets of 1,...,k
postopt	logical; default depends on function; if TRUE, the function `reduce_rows_iterative_complete` tries to make rows redundant and remove them. If this takes too long, `postopt` should be FALSE.
seed	an optional positive integer seed for random sampling; specifying a seed makes the array generation reproducible
...	currently not used
niter	the number of independent attempts to make
seed_init	an optional positive integer seed for the first iteration; specifying a seed makes the array generation reproducible
verbose	logical (default TRUE), controlling how much printed output is created

Value

a strength t SeqCA with k columns; all rows contain the integers 1 to k.

Details

The function `greedySeqCA_TJ` implements the construction of Torres-Jimenez et al. (2022). It uses an approach based on directed acyclic graphs and relies on the R package igraph for that purpose. For `postopt=TRUE`, it then *identifies the redundant elements and removes all rows that contain redundant elements only (would be easiest, but at present not done)* applies iterative run size reduction using function `reduce_rows_iterative_complete`.

The function `iter_greedySeqCA_TJ` iteratively repeats the process and keeps the best outcome.

Torres-Jimenez et al. (2022) describe a 3-stage approach, the third stage of which is simulated annealing. Simulated annealing is invoked if the iterative reduction process is unable to find a row that can be removed without deteriorating coverage. While this stage is needed for achieving the very good performance that the paper reports on, it is often excessively slow at least in the implementation of this package.

Use of AI

Claude 4 was involved in the development of these functions (mainly for debugging).

Author(s)

Ulrike Groemping

References

Torres-Jimenez et al. (2022)

See Also

greedySeqCA_naive() for a naive greedy random assembly of an SeqCA and greedySeqCA_Kuhn() for the greedy method by Kuhn et al. (2012)

Examples

```
## without post-optimization
nrow(greedySeqCA_TJ(5, 3, seed=2323, postopt=FALSE))
## relatively unlucky:
nrow(greedySeqCA_TJ(5, 3, seed=222, postopt=FALSE))
## relatively lucky:
nrow(A <- greedySeqCA_TJ(5, 3, seed=8881, postopt=FALSE))
coverageSeqCA(A, 3)

## with post-optimization
## differences become smaller (still relevant esp. for larger cases)
## formerly relatively unlucky
nrow(A <- greedySeqCA_TJ(5, 3, seed=222))
coverageSeqCA(A, 3)

## iteration, with good seed
nrow(Aiter <- iter_greedySeqCA_TJ(8, 5, 3, seed_init=9055))
  ## brings the run size down to 7
coverageSeqCA(Aiter, 3)
```

---

identify_needed *Identifying needed and redundant elements of SeqCAs*

Description

from a given SeqCA A and its strength t

Usage

```
identify_needed(A, t, ...)
```

Arguments

A	a sequence covering array (SeqCA) with k columns and elements 1,...,k in each row
t	the strength for which the coverage of all permutations of length t must be guaranteed for all combn(k,t) t-element subsets of 1,...,k
...	currently not used

Value

a logical matrix of the same dimensions as A with TRUE for needed values and FALSE for values that can be moved.

The meanings of TRUE and FALSE in the resulting matrix

In an SeqCA of strength t, it is guaranteed that all orders of all t-tuples of elements of 1,...,k are covered; the ordered tuples are also counted as covered if they are interleaved by other values. For example, the sequence 1->4->3->2->5 covers the 3-sequence 1->3->5. The run 14325 covers `choose(5,3)=10` 3-sequences, and each of its elements is part of six of these. If all six such sequences are already covered in earlier runs, the element is marked as redundant (`FALSE`), otherwise it is needed (`TRUE`); for example, if the sequences 1->4->3, 1->4->2, 1->4->5, 4->3->2, 4->3->5, 4->2->5 were already covered by earlier runs, the position of the "4" becomes `FALSE`. Thus, if the position is `TRUE`, these sequences might still be covered by later runs, but are not guaranteed to.

Use of AI

Claude 4 was involved in the development of this function.

Examples

```
### create a trivial SeqCA of strength 2
A <- rbind(1:5, 5:1)
identify_needed(A,2)

## a naive greedy SeqCA
A <- greedySeqCA_naive(5, 3, seed=2323, postop=FALSE)
dim(A)
## identify the necessary elements
(neededs <- identify_needed(A,3))
## there are various FALSE-only rows
## remove those
A <- A[!rowSums(neededs)==0,]
neededs <- neededs[!rowSums(neededs)==0,]
dim(A)
## all sequences are still covered
(cove <- coverageSeqCA(A, 3))
## the count_effective attribute of cove shows
## that the last three rows are only needed for 1 sequence each
attr(cove, "count_effective")
## correspondingly, two elements of each of those rows are redundant
neededs
A[11:13,]
## row 11 is needed for the sequence 3->1->4,
## row 12 for the sequence 5->4->3,
## and row 13 for the sequence 5->1->4
```

`int2poly` *auxiliary functions for Galois field arithmetics*

Description

non-visible functions to be used in other functions, including the construction for OA(q^3, 3, q+2, q) for q=2^s

Details

Most functions are taken from SOAs. Function createBush3PowerOf2 implements the construction via permutation vectors as stated in Sherwood (2006)

`juxtapose` *Function to juxtapose two strength t CAs*

Description

Two CAs D1 and D2 with the same strength t, the same number of columns k, and the same numbers of levels in all but at most one column are juxtaposed, such that the levels of one corresponding column in each array are concatenated, resulting in a new strength t CA with the sum of the numbers of levels for that column.

Usage

```
juxtapose(D1, D2, jcol=NULL, start0=TRUE, ...)
```

Arguments

`D1`	a CA with levels as integers 0, ..., v-1 or 1,...,v (may have different number of levels for each column, but same coding)
`D2`	a CA like `D1`, same number of columns, same numbers of levels, except perhaps for the `jcol`th column; must have the same coding as `D1`
`jcol`	the id of the column to be concatenated; if `NULL`, it is either the only column id for which the numbers of levels differ, or 1.
`start0`	logical: Do integer valued levels start with 0 ? (Otherwise, they must start with 1.)
`...`	further arguments to function `coverage`

Details

Juxtaposing two CAs of a particular strength yields a new CA of the same strength. If `D1` and `D2` do not have the same strength, the weaker strength prevails.

Value

If `D1` is a `N1 x k` CA of strength `t` and `D2` is a `N2 x k` CA of strength `t`, the function returns an `(N1 + N2) x k` matrix of class `ca`, which is a CA of strength `t` and has the same numbers of levels as the ingoing arrays, except for the juxtaposed column `jcol`, for which the number of levels is the sum of the numbers of levels of the ingoing CA.

Examples

```
# create a CA(24,3,2^10 4^1) from two CA(12,3,11,2)

## do it by hand
A <- paleyHad(11)
A2 <- A
A2[,11] <- A[,11] + 2
coverage(rbind(A,A2),3)

## use the function
juxtapose(A, A, jcol=11)

## create a CA(13, 2^10 3^1, 2) by juxtaposing two
##    strength 2 CAs (one with a column that has only 1 level):
juxtapose(KSK(11), cbind(KSK(10), 0))
   ## note: it is not necessary to specify jcol
   ##       because there is only one column with
   ##       different number of levels
```

`lower_bound_CA` *functions for bounds for CA sizes*

Description

The function(s) provide bounds for CA sizes.

Usage

```
lower_bound_CA(t, k, v)
```

Arguments

`t`	requested strength of CA (positive integer number >= 2)
`k`	number of columns, must be at least `t`
`v`	number of levels for each column: a single integer number or a vector of integer numbers of length `k`

Value

`lower_bound_CA` returns a positive number.

Details

Function `lower_bound_CA` provides the trivial lower bound which is the largest product of `t` numbers of levels.

Examples

```
lower_bound_CA(3, 75, 2)  ## 2^3
lower_bound_CA(3, 75, c(rep(2,72), 3, 5, 8)) ## 3*5*8
```

`markflex` *Redundancy checks and profile*

Description

Functions for manipulating flexible values: determination of flexible positions (flexpos), determination of a flexible set (markflex), and calculation of the profile (flexprofile)

Usage

```
flexpos(D, t, ...)

uniquecount(D, t, ...)

markflex(D, t, fixrows=0, verbose=0, ...)

flexprofile(D, ...)

postopNCK(D, t, fixrows=0, verbose=0, outerRetry = 50, outerMaxnochange = 10,
       innerRetry=10, innerMaxnochange=25, seed=NULL, ...)
```

Arguments

`D`	an N x k CA of strength t with v levels, coded from 0 to v-1 or from 1 to v; flexible values, if any, must be denoted as `NA`
`t`	integer-valued (>=2), the strength of `D`
`fixrows`	integer from 0 to `N` for preventing the top `fixrows` rows from being moved
`verbose`	integer-valued degree of verbosity; 1 requests that the latest stages of unsuccessful attempts are added as an attribute to the returned unchanged matrix in case of a failure
`...`	currently not used
`outerRetry`	integer-valued number of retries for escaping local optimum, if `outerMaxnochange` iterations were not successful
`outerMaxnochange`	integer-valued number of iterations to try and find optimum positions within a candidate row or its automatically-determined replacements
`innerRetry`	integer-valued number of reshuffles for escaping local optimum, if `innerMaxnochange` iterations were not successful
`innerMaxnochange`	integer-valued number of iterations to try and find optimum positions within a candidate row or its automatically-determined replacements

seed	NULL, or an integer-valued seed for the random process; a non-NULL seed is stored in the seed attribute of the output object

Value

Function flexpos returns a logical matrix with TRUE values indicating flexible positions of D. Any position that is already NA in D is set to TRUE.
Function markflex returns the ingoing CA, with values denoted as before, with rows potentially reordered (for facilitating all-flexible rows), and with flexible positions changed to NA. **this may change**
Function postopNCK returns an object of class CA with prior attributes preserved and attributes Call and seed augmented by the postopNCK call details.

Details

Function flexpos marks flexible *positions*, i.e. positions all of whose t-way interactions are also covered in at least one tuple without them. It will not work to treat all these positions as flexible *at the same time*, because this would eliminate *all* instances of the affected interactions.

Function markflex uses the greedy algorithm of @nayeriRandomizedPostoptimizationCovering2013 for marking elements as flexible. It makes use of function flexpos for swapping rows so that it becomes more likely that entire rows consist of wildcards (flexible set points). It works reasonably fast for strength 2 or small k, but becomes slow for larger strength and/or many columns. Nayeri et al. (2013) ascertain that the determination of the maximum set of flexible points is an NP-hard problem, and this algorithm is not guaranteed to find the maximum flexible set. The returned flexible positions tend to be at the bottom of the array.

Function flexprofile obtains the profile of column-wise flexible values for D.

Function postopNCK likely needs some tuning for speed and efficiency, but is nevertheless often successful.
It attempts to remove rows by trying to make entire rows contain flexible values; in fact, it removes rows that have fewer than t non-flexible values, as those cannot contribute to t-coverage. It implements a variant of the proposed algorithm by Nayeri et al. (2013). If successful, it returns D after removal of as many as possible flexible rows, as created by markflex. If that immediate and cheap approach does not work, the function tries to iteratively improve the number of flexible values in the last row, until the row has flexible entries only. If that is not successful in a given number of iterations (innerMaxnochange), an inner retry step switches to a different last row to make flexible (at most innerRetry attempts).
An outer loop makes outerRetry such attempts and keeps the user informed via messages about the current status. If the user interrupts via the keyboard (<Esc> or <Ctrl>-<c>), the result of the latest successful *outer* retry is returned (make sure to not abort iterations before the next outer retry was started).
Without such an interrupt, if no attempt is successful, the function returns the unchanged input matrix, adding as an attribute a list with the last states of each retry, if requested by the verbose argument.

References

Nayeri, Colbourn and Konjevod (2013)

Examples

```
# identify flexible values
A <- cyc(19,2)
coverage(A, 3)
(AwithFlex <- markflex(A, 3))
coverage(AwithFlex, 3)
caverify::ca_verify(AwithFlex, 3)

## also works for mixed levels
(L18withFlex <- markflex(DoE.base::L18, 2))
flexprofile(L18withFlex)
## last two rows removable
## row 16 only needed for pair 23 in first two columns
## change element [8,1] to 2 --> row 16 becomes removable
L18withFlex[8,1] <- 2
flexprofile(L18withFlex)  ## frequency reduced by 1 for column 1
(L18withFlexmod <- markflex(L18withFlex, 2))
## now, three rows can be removed
coverage(L18withFlexmod[1:15,], 2, start0=FALSE)

## do the replacement on the raw design
L18mod <- DoE.base::L18; L18mod[8,1] <- 2
(L18modwithFlex <- markflex(L18mod, 2, flexible="NA"))

## example of Colbourn and Torres-Jimenez (2013)
## Figure 2 of the paper
hilf <- strsplit(c("2222001", "0022222", "1121201", "0120110",
                       "2210202", "2102122", "1200020", "0211121",
                       "1012100", "2001210", "0000001", "1111012",
                       "2222222", "2222011"), "")
plan <- do.call(rbind, lapply(hilf, as.numeric))
dim(plan)

## flexpos marks all positions that are not involved in any unique pairs
flexpos(plan, 2)
## these cannot be simultaneously made flexible

## markflex creates actual flexible values (=don't care values)
## and moves flexible rows to the end
(planMod <- markflex(plan, 2))
## there are two rows with all values flexible (can be omitted)

coverage(planMod[1:12,], 2)     ## now optimal
eCAN(2,7,3)

## fixing the first two rows from being moved
## allows only one row to be removed
## fixing rows comes at a price
(planMod2 <- markflex(plan, 2, fixrows=2))

## if removal of rows is the goal,
## postopNCK does the entire job automatically
```

```
## and very fast for this trivial case
plan_postopNCK <- postopNCK(plan, 2, outerRetry=1)
dim(plan_postopNCK)
coverage(plan_postopNCK, 2)

## postopNCK is fast on trivial problems like the above,
## but may take its time otherwise
## such examples can be found in the help for MCA2
```

MCAt *Making mixed-level CAs*

Description

either from a given CA D, by makes unused levels of some columns flexible (arbitrary strength), or for strength 2 only, from required numbers of levels, by a construction generalized from Sherwood (2008). In either case, the result is treated by function postopNCK for reducing its run size.

Usage

```
MCA2(nlevels, D=NULL, outerRetry=10, innerRetry=1, seed=NULL, ...)

MCAt(nlevels, t, outerRetry=10, innerRetry=1, seed=NULL, ...)

CA_to_MCA(D, cs, tolevs, t=attr(D, "t"), outerRetry=10, innerRetry=1, seed=NULL, ...)

N_upper_MCA(nlevels, t=2, internet = TRUE, ...)

projBoseMCA(nlevels, t=2, ...)
```

Arguments

nlevels	a vector of numbers of levels, or a data.frame that is the table of that vector, with levels in decreasing order and column names `level` and `frequency`; function `MCA2` always sorts the numbers of levels into decreeasing order
t	integer, the strength of `D` (smaller strength would be permitted, but would not make sense) or the requested strength
outerRetry	passed to `postopNCK`; if `0`, run size reduction will not be attempted.
innerRetry	positive integer value, passed to `postopNCK`; it is sometimes but not always beneficial to increase it versus the default 1
seed	the seed for making `postopNCK` reproducible
...	currently not used
D	a strength `t` CA (it may have mixed levels)
cs	integer vector of column numbers
tolevs	integer vector of target numbers of levels for the columns indicated by `cs`; must not be larger than the number of levels of the related columns in `cs`, and must not be smaller than 2
internet	logical: is an internet connection available (for DWYER or NIST download)

Value

Functions MCA2, MCAt and CA_to_MCA return a matrix of class ca with attributes of the ingoing matrix (dimensions modified), and attributes Call and seed modified, and often with flexible values in most columns. For function MCA2, the number of levels is in decreasing order (regardless of the order specified in nlevels).
Function N_upper_MCA returns an upper bound for the run size of an MCA obtainable via the constructions (named integer, obtained by bestN, where the size can be obtained by using the CA of the construction indicated by the name as the D in function CA_to_MCA).
Function projBoseMCA returns a matrix of class ca with attributes; for numeric nlevels, the columns are in the order of nlevels, otherwise the columns with more levels come before the columns with fewer levels.

Details

Function MCA2 implements Groemping's (2025) generalization of the first construction of Sherwood (2008) for creating mixed covering arrays (MCAs). The algorithm is described in Groemping (2025). Its key idea is to start from a uniform CA, make superfluous levels flexible, and expand columns with enough flexible entries into more columns of the same number of levels, using a recursive substitution of flexible values with ordered designs. The result is then post-processed by function postopNCK, unless this is suppressed by the argument outerRetry=0. Post-processing can also be done separately by directly using function postopNCK on the result of function MCA2, regardless whether it was created with or without post-processing.

Function MCAt implements an automated version of function CA_to_MCA, in which the start array D for that function is automatically determined, so that the user can simply specify the required numbers of levels with the required strength.

Function CA_to_MCA sets unused levels of the columns in cs to NA, i.e., makes them flexible, as, e.g., described in Moura et al. (2003). Subsequently, the function postopNCK applies the method of Nayery et al. (2013) for making entire rows flexible and removing them. Repeated runs can remove more rows. The outerRetry argument gives the number of calls to postopNCK. if the bottom rows of the outcome still look promising, the function can be called again for removing even more rows.

Without specifying a seed, the results will differ from call to call. If you want it reproducible, make sure to set an integer seed. Run sizes and run times can be very seed-dependent in some situations, and it can happen that encountering a small design is a rare lucky case. Therefore, one may sometimes want to walk over a vector of random seeds and pick the best resulting design (reproducible by keeping track of the seeds).

The run size optimization can take a long time. It can be suppressed by setting outerRetry to zero, and this should be done for large settings. Per default, the optimization is switched on. Its progress is reported by interim messages. The functions (try to) react beneficial to user interrupts: They return the result from the previous successful *outer* retry (make sure the outer retry has finished before escaping calculations!).

Function N_upper_MCA reports an upper bound on the run number for an MCA of the setting. That bound is obtained by using function bestN for the maximum number of levels with the total number of columns. Depending on the situation, the bound may be sharp (e.g., for eight five-level columns with two 3-level columns) or dramatically too large (e.g., for one eight-level column with ten 2-level columns).

Function projBoseMCA implements Theorem 2.3 of Colbourn (2008) as well as Corollary 2.2 of the same paper, which also references Stevens, Ling and Mendelsohn (2002, their Theorem 2.6). It

works for two different numbers of levels only. It is most powerful for q+1 columns at q-1 levels with a single column at q+1 levels, where q is a prime power (the Corollary 2.2 situation); in these cases the run size is optimal ((q-1)*(q+1)=q^2-1).

Warning

There is not much experience yet with the post-optimization performance. The defaults for the related parameters may change in the future.

Author(s)

Ulrike Groemping

References

Colbourn (2008), Groemping (2025), Moura et al. (2003), Nayeri et al. (2013), Sherwood (2008), Stevens, Ling and Mendelsohn (2002)

Examples

```
##################################################
### MCA2
##################################################
## small example with fast optimization to global optimum
## which is 2*8
## 16 runs
D <- MCA2(c(rep(2,10), 8))
dim(D)
coverage(D, 2)
## N_upper_MCA is very pessimistic
N_upper_MCA(c(rep(2,10), 8))

## another case for which N_upper_MCA seems to be tight
## (postopNCK cannot achieve an improvement in reasonable time)
N_upper_MCA(c(rep(5,8), 3,3))
D <- CA_to_MCA(CS_LCDST(10,5), 9:10, c(3,3), outerRetry=0)
dim(D)
head(D); tail(D)

## create a large design without trying a run size optimization
D <- MCA2(c(rep(2,70), rep(3,80), rep(4,8), rep(6,2)),
         outerRetry=0)
dim(D)
coverage(D, 2)
## alternative call for the same setting
D <- MCA2(data.frame(levels=c(6,4,3,2), frequency=c(2,16,80,70)),
          outerRetry=0)
## Not run:
  ## make a lucky run reproducible by using a random seed
  myseed <- sample(32000, 1)
  Doptimized <- postopNCK(D, 2, seed=myseed)
  ## This runs for an excessive time without user interruption.
```

```
  ## There have occasionally been cases (e.g. seed 75 on windows),
  ## for which the run size was reduced to 45 runs early.
  ## It is not known whether it eventually improves more
  ##    (lower bound 24 runs is likely not achievable).
  ## The run can be stopped by <ESC>, once patience runs out
  ##    or enough improvement was achieved.

## End(Not run)

## use the argument D for starting with a naive
##      CA of 45 runs for 30 5-level columns
## (default would start with CA(29, 2, 7, 5), which is the best for
##      seven 5-level columns and permits expansion as required);
D <- MCA2(c(5, rep(4,2), rep(3,7), rep(2,20)), D=bestCA(2,30,5),
         outerRetry=0)
## Not run:
   ## make a lucky run reproducible by using a random seed
   myseed <- sample(32000, 1)
   ## The code below uses the default number of steps
   ## (which would run for excessive time),
   ## but can be interrupted with <ESC> if run size improvement
   ## has been sufficient or seems to stall
   ## (make sure to wait until the outer retry has safely finished);
   ## run size optimization quickly reduces the 45 (or 29) runs
   ##      to 22 or 23 runs (20 is the lower bound);
   ##      21 runs have also been observed.
   ## Without user interrupt, the code will run
   ##      for excessively long time.
   Doptimized <- postopNCK(D, 2, seed=myseed)

## End(Not run)

#######################################################
### MCAt
#######################################################
## create a strength 4 CA, initially unoptimized
D <- MCAt(c(4,4,3,2,2,2), 4, outerRetry=0)
dim(D) ## much larger than the optimum run size

#######################################################
### CA_to_MCA
#######################################################
# a small example
## six 3-level columns in 33 runs
D <- bestCA(3,6,3)
## to three 3-level and three 2-level columns in 27 runs
Dmixed <- CA_to_MCA(D, cs=4:6, tolev=rep(2, 3), t=3, outerRetry=0)
dim(Dmixed)
## (optimal)
coverage(Dmixed, 3)
## works

## five 3-level columns in 33 runs
```

```
D <- bestCA(3,5,3)  ## CA(33,3,5,3)
## to four 3-level columns and one 2-level column in 33 runs,

Dmixed <- CA_to_MCA(D, cs=5, tolev=2, t=3) ## last column reduced to 2 levels
## a lengthy search cannot reduce the number of runs

## D can be easily reduced to the optimal 27 runs,
## if two columns are reduced to 2 levels each:
Dmixed <- CA_to_MCA(D, cs=4:5, tolev=c(2,2), t=3)

#######################################################
####           projBoseMCA               #############
#######################################################

## corollary 2.2 cases (a single number of levels is different,
## and two larger than the others)
D <- projBoseMCA(c(rep(4,6),6))
head(D)
dim(D)

D <- projBoseMCA(c(rep(6,8),8))
head(D)
dim(D)


## NOTE: example() skips the examples below (for MCA2 and MCAt),
## unless the user specifies
## options(run_heavy_examples = TRUE)
## This is because they could be very slow depending on sample.kind.

## different setting,
## optimize run size within the call (using a good seed)
D <- MCA2(c(rep(2,30), rep(3,2)), seed=333, outerRetry=3)
coverage(D, 2)
attributes(D)
## reasonably quickly reduces to 11 runs on my Windows machine
## lower bound is 9 runs (3*3)
## CAgen yields 14 runs with IPOG-F

D <- MCAt(c(4,4,3,2,2,2), 4, outerRetry=0)
## with seed 888, the optimum is quickly reached on Windows
Doptimized <- postopNCK(D, 4, innerRetry = 3, seed=888)
dim(Doptimized)
  ## optimum 4*4*3*2=96
  ## many other seeds stop at 97 or 98 runs after longer iteration
  ## could also use
     D <- MCAt(c(4,4,3,2,2,2), 4, innerRetry=3, seed=888)
```

`MeagherStevensCombis` *Data for cover starter constructions*

Description

Lists ...Combis and ...Starters support cover starter constructions.

Usage

```
MeagherStevensCombis

MeagherStevensStarters

LCDSTCombis

LCDSTStarters

CMMSSYCombis

CMMSSYStarters

ColbournKeriCombis

ColbournKeriStarters
```

Format

`LCDSTStarters`, `CMMSSYStarters`, `MeagherStevensStarters` and `ColbournKeriStarters` are lists of lists:

The first hierarchy level is either the number of levels per column for, i.e., `v`, or the strength `t`:
`CMMSSYStarters` holds 12 lists for 3 to 10, 12, 13, 15, and 17 levels,
`MeagherStevensStarters` holds 16 lists for 3 to 18 levels,
`LCDSTStarters` 21 lists for 3 to 23 levels; these three constructions always yield strength 2.
`ColbournKeriStarters` holds a single list for strength 4, as the starter vectors for the other strengths can be calculated and do not need to be stored; this construction works for 2 levels only.

The second hierarchy level relates to the number of columns (`k`):
Each list element at the first level is a list whose entries are the starter vectors and whose names are the length of the starter; except for `CMMSSYStarters`, the length of the starter equals the permissible numbers of array columns, while for `CMMSSYStarters`, the array has one more column than the length of the starter.

`MeagherStevensCombis` is a matrix with columns `v`, `k` and `N` that indicates combinations for which starter vectors for the construction function `CS_MS` are available from `MeagherStevensCombis`.

`CMMSSYCombis` is a data.frame that indicates combinations for which starter vectors for function `CS_CMMSSY` are available; it has columns `v`, `k`, `N`, `k1`, `code`, and `stage`.
`k1` indicates the number of columns in the left-hand side partition of a PCA structure.

`LCDSTCombis` is a data.frame that indicates combinations for which starter vectors for function `CS_LCDST` are available; it has columns `v`, `k`, `f`, `NAs`, `N`, `ingredient` and `N_eCANbased`.
`f` indicates how many values are fixed;
`N` is the number of runs achieved in the package based on installed constructions for the ingredient CA with `f` levels,
`ingredient` is the ingredient used in the construction (interesting for f>=4),

and N_eCANbased is the number of runs achieved when using the best CA referenced in the Colbourn tables (which may not be implemented in the package).

ColbournKeriCombis is a data.frame with columns t, k, v, N, method, starter, code and nconst that indicates combinations for which starter vectors for function CS_CK are available; method refers to four different methods that are introduced by Colbourn and Keri (2009) (one of which is deriving from the next higher strength), code (built on starter) is used by function CS_CK for constructing the actual arrays.

An object of class list of length 16.

An object of class data.frame with 134 rows and 8 columns.

An object of class list of length 21.

An object of class data.frame with 32 rows and 4 columns.

An object of class list of length 12.

An object of class data.frame with 217 rows and 8 columns.

An object of class list of length 1.

Details

For more details, see also the documentation of functions CS_MS, CS_LCDST and CS_CK.
The creation of the objects is documented by R files in the extdata folder for this package, which can be located in the directory inst/extdata of the GitHub repo.

Source

References are Meagher and Stevens (2005), Lobb, Colbourn, Danziger, Stevens and Torres-Jimenez (2012, LCDST), and Colbourn and Keri (2009).

References

Meagher and Stevens (2005), Colbourn and Keri (2009), Lobb, Colbourn, Danziger, Stevens and Torres-Jimenez (2012, LCDST)

Examples

```
## an impression of the content
head(ColbournKeriCombis)
tail(ColbournKeriCombis)
length(ColbournKeriStarters[[1]])

head(MeagherStevensCombis)
tail(MeagherStevensCombis)
length(MeagherStevensStarters)

head(CMMSSYCombis)
tail(CMMSSYCombis)
length(CMMSSYStarters)

head(LCDSTCombis)
tail(LCDSTCombis)
```

```
length(LCDSTStarters)
```

`miscCA` *Incorporate miscellaneous arrays*

Description

incorporates miscellaneous arrays that are created in the package or obtained from package DoE.base

Usage

```
miscCA(t, k, v, fixNA=TRUE, seed=NULL, maxconstant=FALSE, makePCA=FALSE, ...)
```

Arguments

`t`	integer: the strength
`k`	integer: the number of columns
`v`	integer: the number of levels for each column
`fixNA`	logical: should flexible values be fixed? If the array has flexible values, `fixNA=TRUE` randomly assigns a fixed value to each flexible value. Thus, a seed is needed for reproducibility.
`seed`	`NULL` or an integer seed for making the fixing of NA values reproducible; a non-NULL seed is stored in the attribute `fixNA_seed` of the returned object.
`maxconstant`	logical: should constant rows be maximized ? ignored for internal arrays, applied for arrays loaded from package **DoE.base** only; per default, the array is not modified and may not even have a single constant row
`makePCA`	logical: should a PCA structure be enforced ? ignored for internal arrays, applied for arrays loaded from package **DoE.base** only If both `makePCA` and `maxconstant` are TRUE, `maxconstant` takes precedence.
`...`	currently not used

Value

an array of class `ca` with attributes

Details

The function relies on the data frame `miscCAcat` for a description of the available CAs.

References

Ji and Yin (2010), Colbourn and Torres-Jimenez (2013), Kuhfeld (w/o year), **DoE.base**

Examples

```
# a CA (without flexible values) from DoE.base
miscCA(3, 5, 4)

## without making rows constant and without enforcing PCA
attributes(miscCA(3, 5, 4, maxconstant=FALSE))

# a CA that is internally stored
# it has a single flexible value, which is per default fixed
# the seed makes the fixing reproducible
attributes(D <- miscCA(2, 14, 4, seed=374))
Ns(2,14,4)  ## best possible
```

MTVTRouxtypeCA *Functions to construct a strength 4 or strength 5 CA from several input CAs*

Description

based on Section 4 of Martirosyan and van Trung (2004). The functions obtain a strength 4 or strength 5 CA in 2k columns at v levels each from several smaller arrays.

Usage

```
MTVTRouxtypeCA(t, k, v, method=NULL, ...)

N_upper_MTVTRouxtypeCA(t, k, v, method=NULL, theoretical=FALSE, ...)
```

Arguments

t	4 or 5: strength of the CA
k	positive integer: number of columns in the result
v	positive integer: number of levels
method	NULL, or "large"
...	currently not used
theoretical	logical: if TRUE, N is calculated from the best known CAs, not from the best implemented ones

Value

Function MTVTRouxtypeCA returns a strength t CA with k columns at v levels each, as requested. N_upper_MTVTRouxtypeCA returns the size of the constructed CA before removing duplicates.

Details

The constructions are based on a strength `t` CA A, a strength `t-1` CA B, a strength `t-2` CA C and, for one of the strength 4 constructions (with Theorem 4.3) as well as for the strength 5 construction, a strength 2 CA D. These ingredient CAs have c columns at `v` levels, where `c=ceiling(k/2)`, except for Theorem 4.3, where D has `v` columns at `v` levels.

The construction for strength 4 is based on Theorem 4.13 (default) or on request on Theorem 4.3 (may be better for large `k`).

The construction for strength 5 is based on Theorem 4.13 (default) or on request on Theorem 4.11 (may be better for large `k`).

Warning

Function `MTVTRouxtypeCA` implements the constructions of Section 4 from Martirosyan and van Trung (2004). It uses internal workhorse functions that correspond to the different theorems of the section.
Note that the constructions are of the so-called Roux type, i.e., recursive with several CA ingredients. Such constructions are competitive for very large scenarii only, and when they get good ingredients.
WARNING: Many ingredients for large situations in this package are (at present) quite poor,
and it is easy to bring the computer to the brink of crash or beyond by naively requesting a CA of strength 5 for 2000 columns. If in doubt, check the expected size with `N_upper_MTVTRouxtypeCA`; on the author's Windows machine, the creation of a CA(172792, 5, 1000, 4) by the command `MTVTRouxtypeCA(5, 1000, 4)` was successful after 10 minutes (about half of which were used for the creation of the largest ingredient with the command `powerCA(5, 500, 4)`) and produced an object of size 1.4GB, with large RAM usage during the slow array generation.
Furthermore, note that the creation of many or even most large arrays requires an internet connection, as arrays from the NIST (2008) library are involved as ingredients (in the above case, as ingredients to the `powerCA` construction; without an internet connection, the ingredient size would more than double).
Large arrays are not a priority of this package, so that this situation may remain unchanged for a long time.

References

Martirosyan and van Trung (2004), NIST (2008)

Examples

```
dim(D <- MTVTRouxtypeCA(4, 12, 3))
N_upper_MTVTRouxtypeCA(4,12,3) ## two duplicates were removed
N_upper_MTVTRouxtypeCA(4,12,3, theoretical=TRUE) ## ingredients are optimal
coverage(D, 4)
Ns(t=4, k=12, v=3)  ## D is far from optimal
                   ## the method is not part of Ns

## the other method is worse for small k
dim(Dworse <- MTVTRouxtypeCA(4,12,3, method="large"))
N_upper_MTVTRouxtypeCA(4,12,3, method="large") ## 32 duplicates were removed
```

```
coverage(Dworse, 4)

dim(D5 <- MTVTRouxtypeCA(5, 10, 4))
N_upper_MTVTRouxtypeCA(5, 10, 4) ## 48 duplicates were removed
caverify::ca_verify(D5, 5)
Ns(t=5, k=10, v=4)  ## run size of D5 is far from optimal
dim(D6 <- MTVTRouxtypeCA(5, 10, 4, method="large"))
```

N_Hartman74 *Construction in Hartman Theorem 7.4*

Description

The number of columns of an N x k design A can be squared; the design size is usually not optimal.

Usage

```
Hartman74(A, v, t, B=NULL, ...)

N_Hartman74(N_A, k_A, v, t, ...)

k_Hartman74(N_A, k_A, v, t, ...)

Turan(t, v)
```

Arguments

N_A	run size of the matrix A (called N above)
k_A	number of columns of the matrix A (called k above)
v	the number of levels (or the number of parts for Turan)
t	the strength (or the number of vertices for Turan)
...	currently not used
A	an N x k CA of strength t with columns in v levels each, denoted as 0 to v-1 or 1 to v
B	NULL, or a strength 2 CA in at least Turan(t,v) columns with levels from 1 to k (levels from 0 to (k-1) are acceptable too). If not specified by the user, the Bose construction from createBose is used, which yields at most k+1 columns, for a prime power k. If k is not a prime power or k+1 columns are not enough, the user must provide a suitable B or an A with more columns (may not be easy, at least not for achieving acceptable run sizes).

Value

function Hartman74 returns a strength t CA with k^2 columns (matrix of class ca),
the identical functions N_Hartman74 and k_Hartman74 return a named vector with the run size N, the number of columns k, the strength t and the number of levels v
from using an N_A x k_A matrix A of strength t in v levels,
assuming this matrix and a suitable B exists (warning, if this is not the case for the Bose construction for prime power k),
and function Turan returns the maximum number of edges in a v-partite graph with t vertices, which is used in the construction.

Details

The construction vertically stacks Turan(t,v) copies of column selections from A,
where the ith copy is obtained by picking the A columns according to the elements of the ith column of B.
Eventually, duplicate rows are removed.

The construction requires that the number of columns of B is at least Turan(t,v), which may be impossible if A has too few columns. Sometimes, it can be possible to increase the number of columns for A (see examples).

References

Hartman (2005)

Examples

```
# Turan numbers (minimum number of columns needed for B)
Turanvec <- Vectorize(Turan)
TuranTable <- outer(1:13, 2:7, function(X, Y) Turanvec(X, Y))
dimnames(TuranTable) <- list(t=1:13, v=2:7)
TuranTable
## the first seven rows remain constant for larger values of v

A <- lhs::createBusht(4,5,3,bRandom=FALSE) ## 64 x 5
D <- Hartman74(A, v=4, t=3)
coverage(D, 3)
dim(D)
eCAN(3,25,4) ## D is far from optimal (roughly factor 1.6)

A <- lhs::createBusht(3,4,4,bRandom=FALSE) ## 81 x 4
A5 <- derive(DoE.base::L243.3.6-1)         ## 81 x 5
     ## optimal CA for strength 4 in 5 3-level columns according to Colbourn tables
# D <- Hartman74(A, v=3, t=4) ## would yield an error, too few columns in Bose design
D <- Hartman74(A5, v=3, t=4)
# coverage(D, 4)  ## commented out because of check time
dim(D)
eCAN(4,25,3) ## D is far from optimal (roughly factor 1.44)
```

N_KSK *function for optimal 2-level CAs of strength 2*

Description

implements the construction by Kleitman and Spencer (1973) and Katona (1973), which have optimal size

Usage

```
KSK(k=NULL, N=NULL)

k_KSK(N)

N_KSK(k)
```

Arguments

k	number of columns
N	number of runs

Value

KSK returns a matrix of class ca with k columns and N runs and levels 0 and 1 in each column, with coverage strength 2.

N_KSK and k_KSK return the minimum N for a given k or the maximum k for a given N, respectively.

Examples

```
## smallest design for 10 columns
KSK(k=10)
## largest design for 6 rows
ca6.2.10.2 <- KSK(N=6)
coverage(ca6.2.10.2, 2)
coverage(ca6.2.10.2, 3)
## largest k for 10 rows
k_KSK(10)
## smallest N for 2800 columns
N_KSK(2800)
## design for 8 columns in 6 rows (not necessarily best subset of columns)
KSK(8,6)
## design for 2 columns (full factorial needed)
KSK(2)
```

NISTcat *Data for function nistCA*

Description

The data.frame NISTcat supports the identification and download of CAs from the NIST library of covering arrays.

Usage

```
NISTcat
```

Format

NISTcat is a matrix with columns t (strength), v (number of levels), k (number of CA columns) and N (number of CA rows). Its rownames are the file names in the NIST library of covering arrays.

Details

The library is very large and has not been included into the package. Function nistCA downloads the requested array as a zip file into the temp directory (using function tempdir to locate that directory), unzips it there and subsequently reads the CA into R.

The CAs in the NIST library of covering arrays were obtained via computer search. For many of them, there are meanwhile better alternatives.

Source

The source is NIST (2008).
The creation of the object is documented by the R file NISTcat_create.R in the extdata folder for this package, which can be located in the directory inst/extdata of the GitHub repo; the many raw arrays are not provided there, as they are available on the web.

References

NIST (2008)

Examples

```
## an impression of the content
head(NISTcat)
tail(NISTcat)
xtabs(~t+v, as.data.frame(NISTcat))

## the following is not tested because of run times
## in CRAN checks

fun <- function(t,k,v) eCAN(t,k,v)$CAN
fun <- Vectorize(fun)
```

```
diff <- NISTcat[,"N"]-fun(NISTcat[,"t"], NISTcat[,"k"], NISTcat[,"v"])
fivenum(diff)
## 0 13 37 161 50698
ratio <- NISTcat[,"N"]/fun(NISTcat[,"t"], NISTcat[,"k"], NISTcat[,"v"])
fivenum(ratio)
## 1 1.286 1.356 1.402 2.315
```

Ns *Functions for N and k from Colbourn tables, other libraries, and implemented constructions*

Description

The functions extract the number of runs needed for a specified setting according to the Colbourn tables, according to the NIST library of CAs and the Torres-Jimenez library of CAs, as well as according to implemented constructions and available catalogues. Or they extract the number of columns achievable for a specified number of runs.

Usage

```
eCAN(t, k, v)

eCAK(t, N, v)

Ns(t, k, v, exclude=NULL)

Ns_derive(t, k, v, exclude=NULL)

Ns_fuse(t, k, v, maxfuse = 1)

N_fuseBoseCA(t,k,v)

N_fuseBushtCA(t,k,v)

Ns_CK_doubling(t=3, k, v)

Ns_productCA(t=2, k, v)

N_NISTcat(t, k, v)

N_TJcat(t, k, v)

N_WKScat(t=6, k, v=2)

N_CKRScat(t, k, v)
```

```
N_miscCAcat(t, k, v)

N_DWYERcat(t, k, v)

N_PALEYcat(t, k, v=2)

N_CYCLOTOMYcat(t, k, v)

N_CAEX(t=2, k, v=3)

N_recBoseCA(t=2, k, v, type="PCA")

N_projBoseCA(t=2, k, v, cmax=3)

N_SCA_Busht(t, k, v)

N_dpCA(t=2, k, v)

N_pcaCA(t=2, k, v)

ks(t, N, v)

ks_productCA(t=2, N, v)

k_NISTcat(t, N, v)

k_TJcat(t, N, v)

k_WKScat(t=6, N, v=2)

k_CKRScat(t, N, v)

k_miscCAcat(t, N, v)

k_DWYERcat(t, N, v)

k_PALEYcat(t, N, v=2)

k_CYCLOTOMYcat(t, N, v)

k_CAEX(t=2, N, v=3)

k_fuseBoseCA(t=2, N, v)

k_fuseBushtCA(t,N,v)

k_recBoseCA(t=2, N, v, type="PCA")
```

```
k_SCA_Busht(t, N, v)

k_dpCA(t=2, N, v)

k_pcaCA(t=2, N, v)
```

Arguments

t	coverage strength
k	number of columns
v	number of levels for each column
exclude	NULL or a character vector of method(s) to exclude, in support of size calculations for recursive constructions which need to calculate sizes of best ingredients
maxfuse	integer number of levels to fuse (i.e., difference between number of levels before and after fusing)
N	number of runs
type	character string: "PCA" or "CA"
cmax	integer number: maximum c to consider for projection (N_projBoseCA returns NA, if a design cannot be found within this limit; increase to Inf, if arbitrarily large N are of interest)

Details

Functions eCAN and eCAK provide the current best N for a given k and vice versa, according to the Colbourn tables, including source information as stated in those tables.
Functions Ns and ks provide the respective run sizes for a requested number of columns k (Ns) or the affordable numbers of columns from an affordable run size N (ks). They take into account all available catalogues and constructions, for which sizes can be easily provided; their scope will grow with time, and Ns is prioritized over ks, as it is considered more important for applications.
Functions Ns_derive, Ns_fuse, Ns_CK_doubling, Ns_productCA provide run sizes from deriving, fusing, doubling or taking the product with the smallest possible strength 2 CA for the requested v in comparison to run sizes without applying such techniques. In case of doubling, the function is restricted to strength $t=3$ (the only strength for which Chateauneuf-Kreher doubling works) with $v=2,3$, for which a natural choice of a the strength 2 design D2 for CK_doubling is known (implemented in functions KSK and CAEX, respectively).
Functions ks_derive and ks_fuse provide affordable numbers of columns from an affordable run size N; for doubling, this is more complicated and has not yet been implemented.

Specific functions for run sizes N and numbers of columns k can also be inspected separately.
The functions typically look for a value of k for *at most* N runs, and a value of N for at least k columns.
The functions look for the exact strength t; this implies that a CA for a higher strength may exist, where none is implemented for a lower strength.

At present, the individual functions, except for eCAN and eCAK, return a number only. This may change in the future.

For the TJcat-related functions, a message provides the command for creating the CA, if a creation from within the package is possible (with one of functions CAEX or SCA_Bose).

Value

eCAN returns a data frame with the smallest known run size (empirical CAN, based on the Colbourn table) and the corresponding source entry (Source),
eCAK does the same with the largest possible number of columns k,
N_NISTcat returns the smallest run size for the requested k of a catalogued array from the publicly available NIST covering array library,
k_NISTcat returns the maximum column size for the requested N of a catalogued array from the publicly available NIST covering array library,
and N_TJcat returns the smallest run size for the requested k of a catalogued array from the Jose Torres-Jimenez library (as of Feb 6 2025, small selection, which is expected to grow; partly available in this package),
and k_TJcat returns the largest number of columns k for the requested N of a catalogued array from the Jose Torres-Jimenez library (as of Feb 6 2025, small selection, which is expected to grow; partly available in this package)
The analogous functionality holds for N_ and k_ functions for DWYERcat, CKRScat, miscCAcat, WKScat and CYCLOTOMYcat, also for CS_MS, CS_LCDST, CS_CK and various other constructions. . In cases for which there is no entry in the respective table or library, the returned results are missing values.

Examples

```
eCAN(3, 199, 2)
N_NISTcat(3, 199, 2)
N_TJcat(3, 199, 2) ## equals the best-known array

eCAN(4, 199, 2)
N_TJcat(4, 199, 2) ## Colbourn table outdated

eCAK(2,11,3)  ## also obtained by CAEX
k_NISTcat(2,11,3)
k_TJcat(2,11,3) ## equals the best-known array

N_CYCLOTOMYcat(3, 45, 3) ## t=3 and v=3 not implemented
N_CYCLOTOMYcat(4, 45, 3) ## quite large!
k_CYCLOTOMYcat(4, 669, 3) ## can accommodate 224 columns
N_CYCLOTOMYcat(4, 5, 3) ## is the smallest *implemented*
                        ## cyclotomy strength 4 CA for 3 levels
N_CYCLOTOMYcat(4, 670, 3) ## the next smallest is for 1051 columns
eCAN(4, 670, 3)           ## which is a current optimum

# overview of available constructions
Ns(2, 45, 2)  ## function KSK is known to yield the overall optimum
              ## projBoseCA is very unsuitable, but Ns uses N_projBoseCA with cmax=Inf
N_projBoseCA(2, 45, 2, cmax=25)
N_projectionBose(45, 2)
Ns(2, 16, 11) ## CS_LCDST is the best,
              ## projBoseCA and CS_MS are competitive
Ns(2, 40, 12) ## pcaCA and dpCA are the best,
Ns(3, 45, 2)  ## the best array is in the TJ or the DWYER-catalogue
              ## CK_doublingCA is close
Ns(3, 400, 2) ## powerCT construction is best
```

```
Ns(2, 800, 3) ## function CAEX produces the best available array
Ns(3, 200, 3) ## cphfCA is best
Ns(4, 150, 3) ## cyclotomy is very competitive
Ns(4, 150, 2) ## TJ is best (function tjCA)
Ns(4, 50, 2) ## Paley and Cyclotomy are the same, and both optimal
Ns(4, 50, 3) ## the best easily available CA is
   # https://github.com/aadwyer/CA_Database/blob/main/Repository/CA/CA_507_4_53_3.txt
Ns(5, 50, 2) ## CS_CK and Paley are optimal
Ns(5, 500, 2) ## CS_CK is near-optimal
Ns(5, 50, 3) ## the best easily available CA is
   # https://github.com/aadwyer/CA_Database/blob/main/Repository/CA/CA_2067_5_50_3.txt
Ns(5, 500, 3) ## Cyclotomy is optimal
Ns(6, 50, 2) ## WKS is optimal
   # WKS_CAs['50']; bestCA(6, 50, 2)
Ns(6, 200, 2) ## CS_CK and Paley are optimal
Ns(6, 2000, 2) ## Cyclotomy is optimal
Ns(6, 200, 3) ## powerCT is best implemented,
              ## but far worse than optimum

Ns_CK_doubling(3, 22, 2)
## with doubling a Paley design for ceiling(22/2) columns,
## the optimum run size can be achieved
## (obtained also directly with CK_doublingCA)

Ns_CK_doubling(3, 12, 3)
## the direct CKRS construction is best (ckrsCA)
```

OD2 *auxiliary ingredients*

Description

non-visible functions for creating ingredients

Details

OD2: ordered designs of strength 2, for the Sherwood mixed level construction OD3: ordered designs of strength 3, for ordered design (CCL) DCA: DCA according to Shokri and Moura (2025)

ODbasedCA *Ordered design based constructions by Cohen Colbourn and Ling (CCL)*

Description

Functions to construct a strength 3 CA with k columns at v levels based on the "ordered design (CCL)" construction by Cohen, Colbourn and Ling (2003)

Usage

```
ODbasedCA(t, k, v, ...)

N_ODbasedCA(t, k, v, nconstant=NULL, ...)

k_ODbasedCA(t, N, v, ...)
```

Arguments

`t`	integer, strength of the CA
`k`	integer, number of columns of the CA; `k <= v`
`v`	integer, number of levels of the CA; `v >= k`, and `v-1` must be a prime power
`...`	currently not used
`N`	integer, affordable run size
`nconstant`	`NULL` for the default; otherwise 1 or 2, for the number of constant rows in the ingredient CA (default matches the available ingredients)

Value

Function `powerCA` returns a strength `t` CA in `N_powerCA(t, k, v)` runs and `k` columns. `k_powerCA(t, N, v)` returns the maximum number of columns that can be achieved with `N` runs when requesting strength `t` at `v` levels.

Details

The function `ODbasedCA` implements constructions according to Cohen, Colbourn and Ling (2003); the run size is more explicitly stated in Cohen, Colbourn and Ling (2008).

The construction works in three steps: It creates an ordered design of strength 3 for all triples with three distinct levels. It then uses `choose(v,2)` copies of the best strength 3 CA in `k` 2-level columns (called "ingredient CA" below) for covering all triples with at most two distinct levels at a time.

The ingredient CA always has at least one constant row, and in a few cases two (at present for `k<=8` and `k=12`).
Each of its `choose(v,2)` copies is used for one of the `choose(v,2)` pairs of distinct levels. If the ingredient CA has two constant rows, the run size can be optimized by omitting both from all copies of the ingredient CA (i.e., by initially not covering (most) triples with all elements the same) and by adding all `v` constant rows in the end (the function actually leaves them all in and removes duplicates in the end).

If the ingredient CA has only a single constant row, this row can be omitted entirely without adding any constant rows in the end, by making sure that each of the `v` levels occurs at least once in the role of 0 and in the role of 1 in the ingredient CA (this is achieved by swapping 0 and `v-1` in that respective pair).

This package knows how to obtain the best available ingredient CAs, so the run size of the output object matches the Colbourn table entries (which have at least `k=10`). For `k=12`, the arrays are even smaller than the Colbourn table entries (which do not reflect the benefit from having two constant elements in the ingredient CA).

Function N_ODbasedCA per default calculates the run size for an ingredient with two constant rows for k<=6 and k=12, and with one constant row for all other cases. The argument nconstant can be set to 1 or 2 in order to inspect the respective non-default case.

Function k_ODbasedCA calculates the maximum k for a given run size. The upper limit for k is always v, so that this is less interesting than for other constructions.

References

Cohen, Colbourn and Ling (2003, 2008)

Examples

```
D <- ODbasedCA(3, 8, 8)
coverage(D, 3)
dim(D)
N_ODbasedCA(3, 8, 8)
## not using a 12 run design with 2 constant row
## would imply 20 more rows
N_ODbasedCA(3, 8, 8, nconstant=1)
k_ODbasedCA(3, 630, 8) ## pessimistic, assumes nconstant=1
k_ODbasedCA(3, 650, 8) ## assumes nconstant=1

D <- ODbasedCA(3, 12, 14)
coverage(D, 3)
dim(D)
N_ODbasedCA(3, 12, 14)
N_ODbasedCA(3, 12, 14, nconstant=1)
eCAN(3, 12, 14) ## based on one constant row,
                ## although two are in the best-known CA(15,3,12,2)
```

---

one_to_howmany *auxiliary functions for mixed level CAs*

Description

non-visible functions to be used in functions for MCA creation, and currently also previous drafts that may be deleted key resource: Sherwood (2008) and Groemping (2025)

Details

OD: ingredient creation for array expansion one_to_howmany: determines the number of factors a column can be expanded to colexpandOD: expands a column makeNA: makes appropriate levels into flexible (NA) values colexpandODr: repeat colexpandOD find_mix_construction_separate: finds construction details for a requested setting of levels, for use in MCA2 find_mix_construction: unfinished, attempts to be more efficient but is way more complicated outerseparate: workhorse function of function MCA2

PALEYcat *Data for function mathematical constructions*

Description

The data.frames CYCLOTOMYcat and PALEYcat support the identification of valid scenarii for use of the respective constructions, and the actual implementation.

Usage

```
PALEYcat

CYCLOTOMYcat
```

Format

CYCLOTOMYcat is a data.frame in 213 rows with columns t (strength), k (number of CA columns), v (number of levels), N (number of CA rows), q (prime or prime power behind the construcion), type (the construction type), and the code for the workhorse function that the interface function cyclotomyCA will use.

PALEYcat is a data.frame in 178 rows with columns t (strength), k (number of CA columns), v (number of levels), N (number of CA rows), q (prime or prime power behind the construcion), and nconst (number of constant rows that can be achieved by function maxconstant). Here, v is always 2. The code is for the workhorse function paley_mat, if necessary with rows added or removed, will be used by the interface function paleyCA.

An object of class data.frame with 213 rows and 7 columns.

Details

For more details, see also the documentation of functions paleyCA and cyclotomyCA.

Source

The reference for the cyclotomy construction is Colbourn (2010). Some parameters in the paper were found to be wrong. Thus, the parameters in CYCLOTOMYcat are from the Colbourn tables, where hope is that they are correct.
One instance from the Colbourn tables was removed, as its apparent underlying q is not a prime, but 13*229 so that either a different construction was used or the entry carries wrong information. For three further instances, the run size was corrected to reflect the correct construction 4a instead of the erroneous construction 3a underlying the run size in the Colbourn tables (thanks to Christopher Smolen for highlighting the coverage violation).
The cases that were checked were found correct. It was not possible with the available computing resources to check the actual coverage for large cases, and sources apart from the Colbourn tables were neither found for many of the arrays.

The reference for Paley type constructions is Colbourn (2015), and the constructions were taken from there. For strength 3, only constructions without doubling are included, as CAs from doubling

are not competitive. Small corrections to Colbourn's Table 1 were needed for primes 7, 13 (strength 3 not possible) and 17 (no row can be removed).

The creation of the objects is documented by R files in the `extdata` folder for this package, which can be located in the directory `inst/extdata` of the GitHub repo.

References

Colbourn (2010), Colbourn (2015)

Examples

```
## an impression of the content
head(PALEYcat)
tail(PALEYcat)
boxplot(N ~ t, PALEYcat, las=1)

head(CYCLOTOMYcat)
tail(CYCLOTOMYcat)
xtabs(~ t + v, CYCLOTOMYcat)

## the following is not tested because of run times
## in CRAN checks

fun <- function(t,k,v) eCAN(t,k,v)$CAN
fun <- Vectorize(fun)
diff <- PALEYcat[,"N"]-fun(PALEYcat[,"t"], PALEYcat[,"k"], PALEYcat[,"v"])
fivenum(diff)
## -102.0    2.0   35.5  168.0  438.0

## one entry is much better than the current Colbourn table entry
## coverage of this entry (t=6, k=380) has been proven
## by Hriday Rungta (see https://github.com/ugroempi/CAs/issues/62).
##
```

`pcaCA` *Functions for partitioned CAs and their products*

Description

Functions for partitioned CAs (PCAs) of coverage strength 2, and for a product of two PCAs. The number of rows of a product of a PCA(N, 2, k, v) and a PCA(M, 2, l, v) is N + M - v, the number of columns depends on the splits k = k1+k2 and l = l1+l2.

Usage

```
pcaCA(k, v, ...)

productPCA(D1, D2=NULL, k1=NULL, l1=NULL, ...)

CA_to_PCA(D, tryhard=FALSE, ignorePCAstatus=TRUE, ...)

is.PCA(D, start0=NULL, ...)
```

Arguments

k	integer: the number of requested columns
v	integer: the number of levels for each column
...	currently not used
D1	an N x k CA of strength 2 with v levels, split into k1 + k2 columns for productPCA
D2	an M x l CA of strength 2 with v levels, split into l1 + l2 columns for productPCA
D	an N x k CA of strength 2 with v levels
k1	width of P block in D1 (see Details section)
l1	width of P block in D2 (see Details section)
tryhard	logical: if TRUE, CA_to_PCA tries to achieve v constant rows for k-1 or k-2 columns; this may take a while in case of many columns.
ignorePCAstatus	logical: should CA_to_PCA ignore the PCAstatus attribute of D (if any)?
start0	NULL, or logical: do the values start with 0 (otherwise with 1)?

Value

Functions pcaCA and productPCA return a matrix of class ca that is a PCA with attribute PCAstatus. For pcaCA, the number of columns is as requested, and the number of rows is the smallest possible according to the pre-calculated information data frame PCAcat; it can be explored using function N_pcaCA or function Ns (where the construction is applicable). The returned matrix has its levels coded from 0 to v-1.
For productPCA, the number of columns is k1*l1 + k1*l2 + k2*l1 (of which k1*l1 make up the left part), and the number of rows is N + M - v.
The returned matrix has its levels coded like the ingoing matrices, i.e., integers starting with 0 or 1.

Function CA_to_PCA returns an equivalent CA of S3 class ca that has PCA (or SCA) structure (see Section "Partitioned Covering Array (PCA)"), if possible.

The resulting CAs have an attribute PCAstatus, which is a list with elements type (PCA or SCA), k1 and k2.

Function is.PCA returns a logical with the analogous attribute; it can be (mis)used for checking consistency between the codings of designs by specifying start0.

Details

Function pcaCA returns a PCA from - possibly recursive - application to of function productPCA according to rules provided in the data frame PCAcat.

Function productPCA exploits a partitioned structure of the two CAs it combines (see Section "Partitioned Covering Array (PCA)"). The construction can be viewed as a kind of product, but is sometimes also called "cut-and-paste". It was proposed in Colbourn et al. (2006) and was slightly generalized later, see, e.g., in Colbourn and Torres-Jimenez (2013). Function productPCA also serves as work horse function for combining two CAs in many instances of the CAEX and recBoseCA constructions.

If two matrices D1 and D2 are specified, the function productPCA does a conventional PCAxPCA construction according to Theorem 3.2 of Colbourn et al. (2006).
If D2=NULL, the function uses the SCA constructed by function link{SCA_Bose} and exploits the fact that the sub matrix B1 of that SCA is itself a PCA (i.e., the SCA is a 2PCA in the notation of the Colbourn tables); for such cases, Theorem 3.3 of Colbourn et al. (2006) states that the number of columns of the first part in the result can be increased (which becomes relevant whenever using the result as the in-going matrix in another productPCA call). This is implemented by moving the bottom-most block of v rows from the resulting matrix to the top and applying function CA_to_PCA to the result; for avoiding this from happening, explicitly specify SCA_Bose(v) as the second matrix. Likewise, for any other 2PCA (i.e., PCA whose bottom left block is a PCA) in the role of D2, the user can postprocess the result of function productPCA in order to benefit from the properties of that matrix.

CA_to_PCA rearranges D into a PCA structure (see Section "Partitioned Covering Array (PCA)") by rearranging columns, swapping symbols, and in case of tryhard=TRUE also rearranging rows.
In case of tryhard=TRUE, it is tried to achieve PCA status k2=1 or k2=2; in that case, if it is possible to achieve v constant rows (i.e., k2=0), the function finds these.
Without tryhard=TRUE, the resulting PCA will have k1 as the number of columns with v different levels in the first v rows.
tryhard=TRUE may be useful for small designs, but may substantially increase run times for D with many rows or many columns.

Function CA_to_PCA brings its argument into the best possible PCA shape by swapping columns only, working on the first q rows. If the argument tryhard is invoked, the function tries to achieve k1=k-1 or k1=k-2 by trying maxconstant on subsets of columns. Function is.PCA checks the PCA status of its argument without modifying it in any way.

Partitioned Covering Array (PCA)

For an N x k CA(N, t, k, v) to be a PCA, the number of columns is split into k = k1 + k2, with k1>0 (otherwise no PCA), the number of rows is split into N = v + (N - v); such a PCA is denoted as PCA(N, t, (k1,k2), v).
The PCA structure, in this package, is arranged with top and bottom parts swapped versus partitioning of Colbourn et al. (2006). Parts are arranged here such that
the v x k1 matrix P (with each column consisting of the identity permutation of all levels) makes up the top left block,
the top right v x k2 block (which may be empty) is constant with all elements equal to the smallest level (for an SCA, which is a special case of a PCA) or has arbitrary levels (for a PCA), however, ensuring that the level in row i must not be larger than the ith level (important for the product to work on PCAs and not on SCAs only, and always possible to achieve by level permutation),

and the remaining two blocks (bottom left and bottom right) may contain arbitrary elements.
It is beneficial for the number of columns in a product, if the left-hand side block is as wide as possible.
It is beneficial for recursively taking products, if the bottom left block is itself a PCA, i.e., contains q rows for which a large number of columns consists of permutations of the values 0 to q-1 (also called a 2PCA).

References

Colbourn et al. (2006) and Torres-Jimenez et al. (2019) for the product with an SCA, Colbourn and Torres-Jimenez (2013) for the more general version with two PCAs

Examples

```
##################################################
### pcaCA
##################################################
## 44 6-level columns
dim(D <- pcaCA(44, 6))
coverage(D, 2)
## best implemented construction
Ns(2,44,6)
## as Torres-Jimenez does not share his arrays
eCAN(2, 44, 6)

### larger number of columns
N_pcaCA(2, 500, 8)
Ns(2, 500, 8) ## not best implemented (prime power v)
Ns(2, 500, 6) ## not eCAN, but best implemented (non-prime power v)


##################################################
### productPCA
##################################################
A <- SCA_Bose(4)  ## Bose array, arranged as SCA
EPCA <- productPCA(A, A)
coverage(EPCA, 2)
dim(EPCA)  ## 24 columns only, 28 runs only
eCAN(2, 24, 4) ## optimal, this construction!
eCAK(2, 28, 4) ## optimal, this construction
attr(EPCA, "PCAstatus")
## better, if D2 is left unspecified, because
## the structure of SCA_Bose(4) in the role of D2
## is exploited
EPCA2 <- productPCA(A)
attr(EPCA2, "PCAstatus")


########################################
## utilities CA_to_PCA and is.PCA
########################################
SCA61.2..9.1..7 <- CA_to_PCA(CS_MS(10, 7), tryhard=TRUE)
```

```
attributes(SCA61.2..9.1..7)

is.PCA(DoE.base::L36.2.8.6.3[,-(1:8)])
CA_to_PCA(DoE.base::L36.2.8.6.3[,-(1:8)])

## utility SCA_Bose
SCA_Bose(3)

## apply PCA construction
E <- productPCA(SCA_Bose(3))
dim(E)
coverage(E,2)
eCAN(2,15,3)  ## optimal for 15 columns
eCAK(2,15,3)  ## but does not cover 20 columns

## for non-prime power case
## use optimum CA(46,2,9,6)
MS6 <- CS_MS(k=9, v=6)
MS6 <- CA_to_PCA(MS6)
MS6 <- CA_to_PCA(MS6, tryhard=TRUE)
head(MS6)  ## it is an SCA with k1=8 and k2=9
dim(CA.86.6.80 <- productPCA(MS6, MS6))
## four too many runs
eCAN(2,80,6)

## another case
head(DoE.base::L36.2.8.6.3[,-(1:8)]) ## it is an SCA with k1=3 and k2=0,
                                     ## as all columns have six distinct
                                     ## values in first six rows
dim(productPCA(MS6, CA_to_PCA(DoE.base::L36.2.8.6.3[,-(1:8)] - 1)))
eCAN(2,27,6)  ## 8 too many runs

## the function also works for designs with flexible values (which are denoted by NA)
## productPCA
D <- productPCA(CAEX(N=16), CAEX(N=16))
dim(D)
eCAN(2,405,3) ## slightly worse than the best CA, which is obtained as CAEX(N=27)
is.PCA(D) ## is already good and cannot be further improved by the simple method used in CA_to_PCA
coverage(D, 2) ## slightly longer run time
## faster with ca_verify
caverify::ca_verify(D, 2)
```

PCAcat *Data for productPCA and productCA constructions*

Description

The data frames PCAcat and DPcat hold the information for strength 2 productPCA or productCA constructions based on cover starter ingredients, Bose SCAs and additional good small CAs.

Usage

```
PCAcat

DPcat
```

Format

PCAcat is a data frame with columns v, k and N, type, k1, code and stage. type indicates which types of ingredients are used, k1 has the number of columns of the left-hand side parts in the resulting PCA, code gives the code that is run for constructing the resulting CA, and stage gives the number of ingredients multiplied for obtaining the result.

DPcat is a data frame with columns v, k and N, type, nconst, code and stage. type indicates which types of ingredients are used, nconst has the number of constant rows of the resulting PCA (guaranteed, in a few cases slightly more would be possible), code gives the code that is run for constructing the resulting CA, and stage gives the number of ingredients involved in obtaining the result.

An object of class data.frame with 484 rows and 7 columns.

Details

The functions productPCA and productCA do the calculations, based on ingredients created with the functions SCA_Bose, miscCA, CS_MS, CS_LCDST (f=1 only) and CS_CMMSSY.
The creation of the objects is documented by the R files create_PCAcat.R and create_DPcat.R in the extdata folder for this package, which can be located in the directory inst/extdata of the GitHub repo. That folder also holds the candidate ingredients used in the constructions in the file PCA_cands.rda and DP_cands.rda.

Source

Colbourn and Torres-Jimenez (2013) and the references given below for the ingredients.

References

Colbourn et al. (2006), Colbourn and Torres-Jimenez (2013), Bose (1938), Meagher and Stevens (2005), Colbourn and Keri (2009), Lobb, Colbourn, Danziger, Stevens and Torres-Jimenez (2012, LCDST)

Examples

```
## an impression of the content
head(PCAcat)
tail(PCAcat)
boxplot(N ~ v, PCAcat, las=1, horizontal=TRUE)

head(DPcat)
tail(DPcat)
boxplot(N ~ v, DPcat, las=1, horizontal=TRUE)

fun <- function(k,v) eCAN(2,k,v)$CAN
```

```
fun <- Vectorize(fun)
diff <- PCAcat[,"N"]-fun(PCAcat[,"k"], PCAcat[,"v"])
fivenum(diff)
## 0    8   23  75  376
quot <- PCAcat[,"N"]/fun(PCAcat[,"k"], PCAcat[,"v"])
fivenum(quot)

diff <- DPcat[,"N"]-fun(DPcat[,"k"], DPcat[,"v"])
fivenum(diff)
## 0    8   23  75  376
quot <- DPcat[,"N"]/fun(DPcat[,"k"], DPcat[,"v"])
fivenum(quot)
```

---

permvals *Function to make the maximum possible number of rows constant*

Description

applies maximum clique search for making maximally many rows constant, which may be useful when using an array in a product or power based construction; the constant rows are moved to the front or removed (if requested)

Usage

```
maxconstant(D, verbose=0, remove=FALSE, one_is_enough=FALSE, dupcheck=FALSE, ...)
```

Arguments

D	a uniform CA without duplicated rows, unless dupcheck is set to TRUE
...	currently not used
verbose	0 for neither comments nor attributes added to output objects, 1 for printing messages with the row numbers used for the constant rows, 2 for not printing anything but adding the list of possible row sets (that could have been made constant) as an attribute to the returned object, 12 for combining both
remove	logical, should constant rows be removed? If TRUE, the function returns a matrix with fewer rows than D.
one_is_enough	logical; should the algorithm limit itself to making one row constant?
dupcheck	logical, should duplicate rows be checked and removed? If FALSE, the function assumes that there are no duplicate rows.

Value

an equivalent version of D that has as many constant rows as possible, located at the start of the array, or removed in case of remove=TRUE.
For verbose=2, the attribute constant_rows is attached to the returned object; it is a list with elements design_name and row_set, where the latter contains the row numbers in the original design of the rows that were moved to the front.

Details

The function uses a method presented by Torres-Jimenez at NIST that identifies the maximum number of constant rows via maximum clique size in a graph, whose vertices are the rows of D and where there are edges between rows that do not coincide in any column. For a clique, all rows can be made constant by the equivalent transform of swapping levels within columns. Of course, the maximum clique size is at most the number of levels, because any further row will have to coincide with an existing one.

If the array already has the maximum possible number of constant rows, these are moved to the first runs (or removed), and a search is avoided.
If there are as many constant rows in the array as is maximally possible as asserted in a clique search, these are again moved to the first runs (or removed).
If there are constant rows that are all together in one of the largest cliques, the first such clique is used.
If there are no constant rows and only one constant row is possible, the first row is made constant.
For all other cases, the last largest clique is used.
The constant rows have ascending levels, starting with the lowest level.

Warning

For arrays with many rows, the search for the largest cliques can take a long time.

References

Torres-Jimenez, Acevedo-Juarez and Avila-George (2021) for the mention of the largest clique idea

Examples

```
set.seed(1212) ## for array randomization
D <- lhs::createBush(4,5)
table(nvals <- lengths(lapply(1:64, function(obj) unique(D[obj,]))))
## one row is already constant:
which(nvals==1)
## 4 rows can be made constant (that is the maximum for 4 level CAs)
Dmoreconst <- maxconstant(D, verbose=1)
head(Dmoreconst)
## previous constant row 22 is now the 2nd row,
## the other clique rows were made constant and moved to positions 1, 3 and 4

## only a single row can be made constant (that is always possible)
D <- cyc(19,2) ## k=19, q=19, type 1, t=3
table(nvals <- lengths(lapply(1:19, function(obj) unique(D[obj,]))))
## no row is already constant (all have 2 distinct values)
## only a single constant row is achievable
head(maxconstant(D, verbose=1))

## two rows can be made constant
CA.22.3.12.2 <- cyc(11,2, type="3a") ## N=22, k=12, q=11, type 3a, t=3
table(nvals <- lengths(lapply(1:12, function(obj) unique(CA.22.3.12.2[obj,]))))
## no row is already constant (all have 2 distinct values)
## two constant rows are achievable
```

```
head(Dc <- maxconstant(CA.22.3.12.2, verbose=12))
## which rows of design CA.22.3.12.2 were used ?
attributes(Dc)

## example of Colbourn and Torres-Jimenez (2013)
## Figure 2 of the paper
hilf <- strsplit(c("2222001", "0022222", "1121201", "0120110",
                       "2210202", "2102122", "1200020", "0211121",
                       "1012100", "2001210", "0000001", "1111012",
                       "2222222", "2222011"), "")
plan <- do.call(rbind, lapply(hilf, as.numeric))
table(nvals <- lengths(lapply(1:14, function(obj) unique(plan[obj,]))))
## one row is already constant, two rows can be made constant
maxconstant(plan)
```

---

powerCA *Power constructions*

Description

Functions to construct a strength t CA in with k columns at v levels based on the power construction by Colbourn and Torres-Jimenez (2010)

Usage

```
powerCA(t, k, v, type="CT", ...)

N_powerCT(t, k, v, type="CT", ...)

k_powerCT(t, N, v, type="CT", ...)
```

Arguments

t	integer, strength of the CA
k	integer, number of columns of the CA
v	integer, number of levels of the CA
type	character string, currently only CT is implemented
...	currently not used
N	integer, number of runs of the CA

Value

Function powerCA returns a strength t CA in N_powerCA(t, k, v) runs and k columns. k_powerCA(t, N, v) returns the maximum number of columns that can be achieved with N runs when requesting strength t at v levels.

Details

The function powerCA implements constructions according to Colbourn and Torres-Jimenez (2010); it is based on the data.frame object powerCTcat that contains scenarii for which the power construction with ideal ingredients yields the smallest possible CAs according to colbournBigFrame (and for which the construction string in colbournBigFrame is understood). Many rows of the data.frame object powerCTcat refer to instances for which the resulting design is so large that its construction requires a computer with massive resources (and the naive call of powerCA will crash R or even the computer); it is therefore recommended to inspect the expected run size with function N_powerCT before calling the construction function.

The actual arrays that can be generated with this package may be larger than the best possible ones if the perfect ingredients are not (yet) available; they may also be somewhat smaller, because this package makes use of the maximum number of constant rows of the ingredient CAs where possible, whereas the numbers in colbournBigFrame do not always do that (it appears that they do it where the Source entry contains "c" next to the respective ingredient).

The construction uses functions Tred, createDHF, and DHHF2CA, as well as CA construction functions as indicated in powerCTcat.

References

Colbourn and Torres-Jimenez (2010)

Examples

```
Ns(3, 25, 2)  ## not a good case, but small enough for checking coverage
D <- powerCA(3, 25, 2)
dim(D)
coverage(D, 3)

## a larger example
Ns(6, 25, 2) ## way too large for so few columns
ks(6, 5321, 2) ## would cover many more columns
eCAN(6, 5321, 2) ## eCAN is PowerCZ
```

---

powerCTcat *Data for function powerCA*

Description

The data.frame powerCTcat holds the details for power constructions according to Colbourn and Torres-Jimenez 2010

Usage

```
powerCTcat
```

Format

powerCTcat is a data.frame with columns t (strength), v (number of levels), k (number of CA columns), N (number of CA rows), Source (the Source entry in colbournBigFrame), Pbase, expon (the base with its exponent and the resulting OA for the construction of the DH(H)F), M (the number of rows of the DHHF), w1 (the number of levels of the first group of u1 rows of the DHHF, corresponding to the number of columns of the first CA to be constructed with construction constr1, whose number of rows is N1 and whose number of constant rows is (at least) c1, ... (same for 2, 3, 4, where applicable), chi the number of rows to re-add into the final CA for covering constant t-tuples), claimedN and claimedk (the corresponding N and k entries in colbournBigFrame).

Details

The data.frame powerCTcat holds information on ingredients of the power construction according to Colbourn and Torres-Jimenez (2010). It is used by the functions powerCA, N_powerCT and k_powerCT.

The columns constr1 to constr4 hold labels of the constructions used for creating the ingredients; these were determined using function bestN. In some cases, they may change in the future due to availability of improved constructions, which will then also impact the achievable overall run size. Users who spot such opportunities are invited to give a shout.

The columns claimedN and claimedk contain the eCAN and eCAK values from the Colbourn tables in colbournBigFrame;
the columns N and k contain the run size and number of columns of actual constructions based on the CAs that are implemented in this package.
Implemented N are equal to claimedN for strengths 3 and 4, within about 10% of the optimum for strength 5 and a bit more spread-out for strength 6 (note that claimedN can only be reasonably compared to N, where claimedk equals k).

Constructions with claimedN>1000000 have been omitted with version 0.19; this greatly improves maintenance and is not a practical limitation, as the large CAs are likely not of value and cannot be created with this package by a typical computer.

Source

The source are the Colbourn tables, combined with Colbourn and Torres-Jimenez (2010).
The data have been compiled based on the data.frame colbournBigFrame that holds the information from the Colbourn tables, and based on the paper by Colbourn and Torres-Jimenez (2010) for the construction. The code can be found the extdata folder for this package, which can be located in the directory inst/extdata of the GitHub repo.

References

Colbourn and Torres-Jimenez (2010), Colbourn (wo year)

Examples

```
## an impression of the content
xtabs( ~ t + v, data=powerCTcat)
xtabs( ~ t, data=powerCTcat)
```

```
boxplot(N / claimedN ~ t, data=powerCTcat,
         horizontal = TRUE, las=1)
xtabs( ~ t, data=powerCTcat, subset=claimedN<69000)
boxplot(N / claimedN ~ t, data=powerCTcat, subset=claimedN<69000,
         horizontal = TRUE, las=1)
xtabs( ~ t, data=powerCTcat, subset=claimedN<73000)
boxplot(N / claimedN ~ t, data=powerCTcat, subset=claimedN<73000,
         horizontal = TRUE, las=1)

with(powerCTcat, data.frame(t, v, k, N, claimedN, diff=(N-claimedN), Source)[claimedN>N,])
```

`print.ca` *Class ca and its methods*

Description

This documentation file documents the class ca (for CAs = Covering Arrays) and its method.

Usage

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

Arguments

`x`	the object to be printed
`...`	further arguments to `print.default`

Value

The print method invisibly returns the input objext `x`

Details

The print method for class `ca` suppresses printing of most attributes, but indicates which attributes are available.

The S3 class `ca` is for a matrix with attributes. Common attributes are `origin`, `t`, `Call`, `eCAN`, `date`, `CAs-version`, `flexible` or `PCAstatus`. Not all attributes are present for all `ca` objects.

The matrix elements are consecutive non-negative integers starting with 0 or 1, and possibly missing values that stand for flexible values, i.e., values that can be arbitrarily chosen without deteriorating the strength of the CA.

print.CAcoverage — *Methods for class CAcoverage*

Description

print method for class CAcoverage

Usage

```
## S3 method for class 'CAcoverage'
print(x, ..., verbose = FALSE)
```

Arguments

x	an object of class CAcoverage, created by function coverage or coverage_iter
...	further arguments to print
verbose	logical, if TRUE, prints the verbose content as well

Value

invisibly returns x

projBoseCA — *Function to create a strength 2 CA by projection of a Bose OA*

Description

A strength 2 CA with q^2-c runs with q+1 (q-c)-level columns and c s-level columns is created from a q-level Bose array arranged as SCA

Usage

```
projBoseCA(k, v, fixNA=TRUE, ...)

projectionBose(q, c=1, s=q-c, fixNA=FALSE, ...)

N_projectionBose(k, v, cmax=Inf, ...)

k_projectionBose(N, v, cmax=Inf, ...)
```

Arguments

k	requested number of columns
v	number of levels
fixNA	logical; if FALSE, keeps flexible values; if TRUE, flexible values are arbitrarily fixed.
...	currently not used
q	a prime power
c	an integer number, 0 <= c <= q-2; for c=0, the unchanged array created by SCA_Bose is returned.

s	an integer number; for c > 1, 2 <= s <= q-c;
	for c=1, 2 <= s <= q+1;
	for c=0, s is irrelevant.

N	maximum acceptable number of runs
cmax	the maximum number permitted for c; setting cmax prevents ridiculous values of N for cases for which k and v are too far apart

Value

projBoseCA returns a covering array in k columns at v levels each with v constant rows, if this is reasonably possible by projection of a strength 2 Bose OA, which should be the case whenever v<k and k-v not larger than seven and v+(k-v-1)/2 a prime power.

projectionBose returns a (q^2-c x q+1+c matrix with the first q+1 columns in q-c levels (labeled from 0 to q-c-1) and the last c columns in s levels (labeled from 0 to s-1) that is a covering array of strength 2.
N_projectionBose and k_projectionBose return a named vector with elements N, k, kmax, v, q and c.

References

Colbourn (2008, Theorem 2.3 and Corollary 2.2), Torres-Jimenez et al. (2019, Fig. 25)

Examples

```
# create a CA(24,2,7,4)
projectionBose(5)  ## four constant rows, some flexible values
projectionBose(5, fixNA=TRUE) ## flexible values arbitrarily filled
projectionBose(5, c=2, s=2) ## 6 3-level columns (3=5-2), two 2-level columns

# create a CA(48,2,9,6)
dim(D <- projectionBose(7))
coverage(D, 2)
eCAN(2,9,6) ## two runs difference

# create a CA(63,2,9,7)
dim(projBoseCA(9,7))
eCAN(2, 9, 7)  ## four runs difference
```

```
# create a CA(166,2,15,12)
dim(projBoseCA(15,12))
eCAN(2, 15, 12)  ## optimal

# querying k or N
N_projectionBose(k=21, v=15)
k_projectionBose(N=357, v=15)
eCAN(2, 21, 15) ## the above design is the basis
eCAK(2, 357, 15) ## four more columns

# the mixed level construction of Stevens, Ling and Mendelssohn (2002)
# Thm 2.6, yields a CA(q^2-1, 2, q+2, (q-1)^(q+1) (q+1)^1),
# e.g., a CA(24, 2, 7, 4^6 6^1)
# as quoted also in Colbourn 2008
D <- projectionBose(5, 1, 6)  ## q=5, c=1, s=
D
coverage(D, 2)
```

---

readCA *Function for reading a covering array from a text file*

Description

reads a text file into a matrix of class ca (for covering array)

Usage

```
readCA(path, flexible.symbols=c("*","-","."), comment.symbol="C",
ninstruct=1, skiplines=0, ignore.chars=NULL, nosep=FALSE, origin=NULL, ...)

funmakefromstrings(stringvec)
```

Arguments

`path`	a character string that specifies the path, including the file name; see Section Details for requirements the file must satisfy.
`flexible.symbols`	characters to be treated as flexible values (will be set to NA)
`comment.symbol`	character that starts a comment line
`ninstruct`	integer, number of lines with instructions (1 or 0)
`skiplines`	integer, number of lines with no readable instructions and no defined comment character that should nevertheless be skipped; those are assumed to be above the instructions line
`ignore.chars`	`NULL`, or characters to be removed from data lines (e.g., [and])

nosep	logical; set to TRUE, if the data lines do not contain separators and each character is a data value
origin	character string to be attached to the output array as origin-attribute; if NULL, the path and filename information is used
...	further arguments for function read.table, which is used separately for each line (after reading them with readLines)
stringvec	vector of n character strings of equal lengths (k, say). The character strings should consist of digits and small letters. Digits will remain unchanged, and the letters a to z will be replaced by numbers 10 to 35.

Value

The output object of readCA is a matrix of class ca. The output object of funmakefromstrings is an n x k matrix of numeric values. funmakefromstrings is used by readCA, but can also be used on its own.

Details

The file fn may start with comment lines that begin with a comment symbol, and an instruction line.
The first line after removing the comments lines contains instructions about the file content, in the form N k v1^k1 v2^k2 ... It is permissible for the instructions line to have a single string without blanks or ^ (e.g., for the strength) after the exponential notation for the columns; this will be ignored in the reading process. ninstruct=0 indicates that there is no instruction line.
All columns of the array must have the same starting value, i.e., start all with 0 or all with 1.

Examples

```
## values start with 0
funmakefromstrings(c("0123456789abc", "1032547698cba"))
## values start with 1
funmakefromstrings(c("123456789abc", "132547698cba"))
## values start with 4 (would usually not make sense)
funmakefromstrings(c("456789abc", "547698cba"))
```

---

recBoseCA *function to create a strength 2 CA based on recursive multiplication of Bose CAs*

Description

recBoseCA applies productCA or productPCA recursively to Bose CAs. The number of levels must be a prime or prime power.

Usage

```
recBoseCA(t=2, k=NULL, v=4, type="PCA", ...)

recursiveBose(q, d=NULL, k=NULL, type="PCA", ...)

N_k_recursiveBose(q, d=2, type="PCA", ...)

N_d_recursiveBose(q, k, type="PCA", ...)

Dd(d, q)

kd(d, q)
```

Arguments

t	the requested strength; must be 2
k	NULL, or the number of columns; if NULL, the maximum possible number for d is returned (with d defaulting to 2, if not specified).
v	the same as q
type	"CA" or "PCA", indicating whether productCA or productPCA should be used for the product
...	currently not used
q	a prime or prime power number of levels
d	NULL, or a small positive integer; d=1 returns SCA_Bose(q) (q^2 x (q+1)), d > 1 the product (productCA or productPCA, depending on type) of d SCA_Bose(q); For d=NULL, the minimum d for accommodating k is used. If k is also NULL, the default is d=2.

Value

recBoseCA returns a uniform CA strength 2 CA with k v level columns (matrix of class ca); its rowsize can be determined a-priori with function N_recBoseCA.
recursiveBose returns a uniform strength 2 CA with q level columns.
For type="PCA" (default), the result has d*q^2 - (d-1)*q rows and q^d + d*q^(d-1) columns, levels are coded from 0 to q - 1.
For type="CA", the result has d*q^2 - (d-1)*2 rows (upper bound, may be slightly smaller depending on constant rows) and (q+1)^d columns, levels are coded from 0 to q - 1.
N_k_recursiveBose returns a named vector with the number of rows and the number of columns k for given q and d.
N_d_recursiveBose returns a named vector with the run size N, the d for its creation for the requested number of columns k, and the maximum possible number of columns kmax for that construction.

Details

The function recBoseCA uses function recursiveBose, which in turn uses SCA_Bose for creating the q^2 x (q+1) ingredient matrix, and either function productPCA or productCA for obtaining the recursive product (depending on the choice for argument type).

type=PCA: For a given v, the function can construct up to kd(d,v) columns in d*v^2-(d-1)*v runs, where d is the number of Bose CAs that are combined, and k(d,v) has been determined by experimentation; for d=2,3, k(d,v) coincides with the value stated in Colbourn et al. (2006) Lemma 3.5, which uses a recursively defined Dd(0,v)=0, Dd(1,v)=1, Dd(d+1,v)=v*(Dd(d,v) + Dd(d-1,v)) to yield k=(v+1)*Dd(d,v) + v*Dd(d-1,v). Different from the construction underlying Lemma 3.5 of Colbourn et al. (2006), this function improves the size of the left-hand side partition in each interim step; this is achieved by applying function CA_to_PCA after moving the first block of v rows Experimentation showed that this choice yielded the overall optimum result among blocks of v rows in all cases that were investigated.

For type=="CA": For a given v, the function can construct up to (v+1)^d columns in up to d*v^2-(d-1)*2 runs (type=="CA"), where d is the number of Bose CAs that are combined.

References

Colbourn et al. (2006)

Examples

```
D <- recBoseCA(k=12)   ## v=4 is the default, and type="PCA"
dim(D)
coverage(D,2)
eCAN(2, 12, 4)
Ns(2, 12, 4) ## the optimum CA is available

D <- recBoseCA(k=25)   ## v=4 is the default, and type="PCA"
dim(D)
coverage(D,2)
eCAN(2, 25, 4)
Ns(2, 25, 4) ## the optimum CA is available
## type="CA" gets close to optimal here
## because k=25 makes it still work with d=2 SCA_Bose
D <- recBoseCA(k=25, type="CA")
dim(D)

dim(D <- recursiveBose(3))
eCAN(2,15,3)  ## optimal
dim(recursiveBose(3, d=3))
eCAN(2,57,3)  ## reasonably close to optimal
dim(recursiveBose(3, d=3, type="CA"))
eCAN(2,64,3)  ## reasonably close to optimal

## query functions
N_k_recursiveBose(4)      ## d=2
N_k_recursiveBose(4, 3)   ## d=3
kd(3,4)   ## work horse function for k for default type PCA
N_k_recursiveBose(4, 3, type="CA")
```

```
N_d_recursiveBose(4, 12)  ## k=12
```

`recursiveBoseHartman` *function to increase numbers of columns for strength 2 CA in q levels (q prime power) from k to kq+1*

Description

applies construction principle of Hartman (2005) Thm 7.1 and Corollary 7.2

Usage

```
recursiveBoseHartman(q, A=NULL, d=NULL, ...)

N_k_recursiveBoseHartman_A(q, A, ...)

N_k_recursiveBoseHartman_d(q, d=2, ...)

N_d_recursiveBoseHartman(q, k, ...)
```

Arguments

`q`	a prime or prime power number of levels
`A`	`NULL`, or a uniform CA of strength 2 in N rows and k columns with integer-valued levels (0 to q-1 or 1 to q). `A` takes precedence over `d`.
`d`	`NULL`, or a small positive integer; 1 returns the Bose OA (q^2 x (q+1)), d>1 the processed Bose OA in d*(q^2) - (d-1)*q runs and q^d+q^(d-1)+...+q+1 columns. For `recursiveBose`, if `A` is specified, `d` is ignored. For `k_recursiveBose`, the default is `d=2` for `N=2q^2 - q`; it can be increased, if k is too small; each increase by 1 will increase N by `q^2 - q`
`...`	currently not used
`k`	the number of columns requested

Value

`recursiveBoseHartman` returns a uniform strength 2 CA with `q` level columns.
If neither `A` nor `d` are specified, the result has `2q^2 - q` rows and `q^2 + q + 1` columns, levels are coded from 0 to `q - 1`.
If `A` is specified, the result has `N + q^2 - q` rows and `q*k + q + 1` columns, in the same coding as `A`.
If `d` is specified and `A=NULL`, the result has `d*q^2 - (d-1)*q` rows and `sum_i=0^d q^i` columns, coded from 0 to q-1.
`N_k_recursiveBoseHartman_A` returns a named vector with the run size `N` and the number of columns `k` for given `q` and `A`.
`k_recursiveBoseHartman` returns the number of columns k for given `q` and `d`.
`N_d_recursiveBoseHartman` returns a named vector with the run size `N`, the `d` for its creation for

the requested number of columns k, and the maximum possible number of columns kmax for that construction.

Details

The function uses a method presented by Hartman (2005) (Theorem 7.1 and Corollary 7.2). For A=NULL, the function uses lhs::createBose(q, q+1, bRandom=FALSE) for creating A The user can specify any uniform strength 2 CA for A; the extension to more columns and runs will always use the Bose OA without the first q rows, and thus always add q^2-q rows.

Examples

```
dim(D <- recursiveBoseHartman(3))
eCAN(2,13,3)  ## optimal
dim(recursiveBoseHartman(3, d=2))
dim(D2 <- recursiveBoseHartman(3, D))
eCAN(2,40,3)  ## reasonably close to optimal
dim(recursiveBoseHartman(3, d=3))
dim(D3 <- recursiveBoseHartman(3, D2))
eCAN(2,121,3)  ## still reasonably close to optimal
dim(recursiveBoseHartman(3,d=4))

## constructions with A=NULL are optimal
for (q in c(4,5,7,9,11,13,16)){
  D <- recursiveBoseHartman(q)
  cat(paste0("q=", q, "\n"))
  cat(paste0("constructed: ",
     paste(paste0(c("N=","ncolumns="),dim(D)), collapse=", "), "\n"))
  Nopt <- eCAN(2,ncol(D),q)[1]   ## optimal
  cat(paste0("optimal for ", ncol(D), " columns: ", Nopt, " runs\n"))
}

#' ## constructions of first recursion are optimal or reasonably close
for (q in c(4,5,7,9,11,13,16)){
  D <- recursiveBoseHartman(q)
  D <- recursiveBoseHartman(q, D)
  cat(paste0("q=", q, "\n"))
  cat(paste0("constructed: ",
     paste(paste0(c("N=","ncolumns="),dim(D)), collapse=", "), "\n"))
  Nopt <- eCAN(2,ncol(D),q)[1]   ## optimal
  cat(paste0("optimal for ", ncol(D), " columns: ", Nopt, " runs\n"))
}

## a general A, an OA in 11 5-level columns at strength 2
## coded starting with 1
A <- matrix(as.numeric(as.matrix(
     DoE.base::oa.design(nlevels=rep(5,11), randomize=FALSE))),
   nrow=50)
head(A)
## number of columns increased from 11 to 56
## number of rows increased by 5^2 - 5 = 20
```

```
## still strength 2
dim(D <- recursiveBoseHartman(5, A))
coverage(D, 2, start0=FALSE)
## 25% more runs than the best CA
eCAN(2,56,5)

## recursive with d=1
recursiveBose(3,d=1)

## query functions
N_k_recursiveBoseHartman_A(4, lhs::createBose(4, 5))
N_k_recursiveBoseHartman_d(4,4)
N_d_recursiveBoseHartman(4, 12)
```

`reduce_rows_complete` *Reducing the number of rows for an SeqCA*

Description

by covering all t-orders that are missing after removing a certain row through rearrangements of redundant elements in other rows

Usage

```
reduce_rows_complete(A, t, verbose=FALSE, ...)

reduce_rows_iterative_complete(A, t, verbose=FALSE, ...)
```

Arguments

`A`	a sequence covering array (SeqCA) with k columns and elements 1,...,k in each row
`t`	the strength for which the coverage of all permutations of length t must be guaranteed for all `combn(k,t)` t-element subsets of 1,...,k
`verbose`	logical; if `TRUE`, printed output is more detailed
`...`	currently not used

Value

an SeqCA with attributes

Details

This can likely be made faster by first ordering the rows in decreasing order of the count_effective attribute of coverage.

Use of AI

Claude 4 was heavily involved in the development of these functions.

Examples

```
# a naive greedy SeqCA
A <- greedySeqCA_naive(6, 4, seed=17)
dim(A)  ## 67 rows
coverageSeqCA(A, 4) ## perfect coverage

  ## somewhat slow (about 60s on my machine)
  Areduced <- reduce_rows_iterative_complete(A, 4)
  dim(Areduced)  ## 42 runs, i.e. reduced by about 1/3
  coverageSeqCA(Areduced,4)   ## still perfect coverage
```

SCA_Bose — *Functions for regular orthogonal arrays with prime power numbers of levels*

Description

Functions for regular orthogonal arrays with prime power numbers of levels q. SCA_Bose creates strength 2 OAs in q^2 runs with q+1 columns at q levels, SCA_Busht creates strength t OAs (including t=2) in q^t runs with q+1 columns at q levels.

Usage

```
SCA_Bose(q)

SCA_Busht(q, t)
```

Arguments

q	integer: a prime power
t	integer: the requested strength

Value

Both functions yield a matrix of class `ca` which is an orthogonal array in `q+1` columns with `q` levels (exception: `q+2` columns for `t=3` and `q` a power of 2); the number of rows is `q^2` for `SCA_Bose` and `q^t` for `SCA_Busht`. The arrays are rearranged such that their first `q` columns have their first `q` rows constant, which is beneficial for some CA constructions, e.g., for `powerCA`.

Details

Function SCA_Bose obtains the strength 2 Bose constructed CA (q+1 columns in q levels, q^2 runs, q a prime power) and rearranges it into SCA shape. The array actually is a 2PCA, in the sense that its bottom left (q^2-q)xq block is itself a PCA with width of the left parts equal to q (and thus width of the right-hand side equal to zero). Every consecutive qxq block of the matrix consists of columns whose elements are a permutation of the number 0 to q-1. This structure of the matrix is exploited by function productPCA.

Function SCA_Busht obtains the strength t Bush constructed CA (q+1 columns in q levels, q^t runs, q a prime power; exception: for t=3 and q a power of 2, $q+2$ columns are achieved.
In all cases, the array is arranged into SCA shape with k1=q. For t=2, it equals the array created by SCA_Bose. This array is also a 2PCA; however, productPCA is a strength 2 construction only.

Function SCA_Bose uses function createBose, function SCA_Busht uses function createBusht, except for t=3 with q a power of 2, where it uses the internal function createBush3PowerOf2 of this package, which is based on Sherwood et al. (2006).

References

Bose (1938), Bush (1952), Sherwood, Martirosyan and Colbourn (2006)

Examples

```
A <- SCA_Bose(4)  ## Bose array, arranged as SCA
B <- SCA_Busht(4,2)
all.equal(A, B)
## arrays the same, difference only in attributes

dim(SCA_Busht(4, 3))  ## q+2 columns for t=3 and q=2^s
dim(SCA_Busht(4, 4))  ## q+1 columns otherwise
dim(SCA_Busht(5, 3))  ## q+1 columns otherwise
```

---

scphfCA *Making CA from SCPHF*

Description

according to the process outlined in Sherwood, Martirosyan and Colbourn (2006) (non-reduced permutation vectors).

Usage

```
scphfCA(t, k, v, start0 = TRUE, ...)

SMC(cphf, q, t, type="2006", ...)

N_scphfCA(t, k, v)

k_scphfCA(t, N, v)
```

Arguments

`t`	integer: target strength
`k`	integer: requested number of columns
`v`	integer: number of levels, must be a prime power
`start0`	logical: should the levels of the output array start with 0 (otherwise with 1)?
`...`	currently not used
`cphf`	matrix, which is a covering perfect hash family (CPHF)
`q`	integer: number of levels, prime power
`type`	string: "2006" refers to the original implementation by Sherwood, Martirosyan and Colbourn (2006), which uses non-default Galois fields for q=8,9; string "2009" refers to SCPHFs from Walker and Colbourn (2009) which also uses non-default Galois fields for q=8,9, and uses a different one for `type="2006"` for q=9; anything else uses the built-in Galois fields of package **lhs**
`N`	integer: affordable run size

Value

Function `scphfCA` returns a matrix of class `ca` with attributes, which is a covering array of strength 3 or 4.
The function `SMC` returns a matrix with entries in 0,...,q-1, which is a covering array of strength `t`, if a valid `cphf` for strength `t` is provided; the function is not primarily meant for direct use, but is the workhorse behind `scphfCA`. It can be used with user-provided CPHFs.

Functions `N_scphfCA` and `k_scphfCA` return the required run size or the achievable number of columns, respectively.

Details

Function `scphfCA` implements CAs from the Sherwood et al. (2006) construction for length `t-1` h-tuples to create length `v^t` permutation vectors for `v` levels and strength `t`. The workhorse function `SMC` uses a stored covering perfect hash family of a specific form (SCPHF). The built-in SCPHFs are from Sherwood et al. (2006) and from Lanus as provided in the Dwyer (2024) GitHub repository (see `SMC_SCPHFs` and `CL_SCPHFs`). Users who want to use their own SCPHF can implement it with the function `SMC`; large cases can take a long time to construct.
The construction relies on a Galois field GF(`v`). The SCPHFs from Sherwood et al. (2006) for `v=8` or `v=9` require a different Galois field than otherwise used in this package (created with internal function `mygf`); the `type` argument of function `SMC` can be used for requesting those (type="2006") or the default ones (type="2018").
The data frame `SCPHFcat` holds the overview information on which constructions are implemented.

References

Sherwood, Martirosyan and Colbourn (2006), Walker II and Colbourn (2009), Colbourn and Lanus (2018), Colbourn, Lanus and Sarkar (2018), Dwyer (2024)

Examples

```
# applying the construction for small SCPHF from the Sherwood et al. (2006) paper

# strength 3
head(SCPHFcat[which(SCPHFcat$v==4),])
## the SCPHF for the first row (type 2006)
SMC_SCPHFs[["3"]][["4"]][["16"]]
D <- scphfCA(3,16,4)
dim(D)
coverage(D,3)
eCAN(3,16,4)   ## size matches eCAN

## the SCPHF for the third row (type 2018)
CL_SCPHFs[["3"]][["4"]][["55"]]
D <- scphfCA(3,55,4)
dim(D)
coverage(D,3, parallel=2) ## about 8 seconds
caverify::ca_verify(D, 3)
Ns(3,55,4)   ## best implemented

# strength 4
Ns(4, 10, 3)  ## best possible
```

`SeqCA_Levenshtein` *Levenshtein creation of an optimal strength t SCA for k = t + 1 columns*

Description

based on the very clear presentation in the master thesis by Na (2021)

Usage

```
SeqCA_Levenshtein(k, t = k - 1, a=0, ...)
```

Arguments

`k`	positive integer, at least 3: the number of columns
`t`	the strength for which the coverage of all permutations of length t must be guaranteed for all `combn(k,t)` t-element subsets of 1,...,k; for this construction, `t = k - 1` is required.
`a`	the partition class
`...`	currently not used

Value

a strength t SeqCA with k columns; all rows contain the integers 1 to k. The SeqCA has t! runs, i.e., optimal run size.

Details

The construction is based on partitioning the k! permutations of the values in `1:k` according to the weights of their derivatives, where the derivative of a permutation p is obtainable as `d <- (p[1:(k-1)] - p[2:k] > 0) + 0L`, and its weight is `w <- sum(d*1:t)`; partitioning is w.r.t. to classes generated by the modulo k result of the weights (`w %% k`). Each of the t + 1 parts is an SeqCA of strength t (Levenshtein 1991 as presented in Na 2021).

The function works fast for small k; as k! increases very fast (much faster than 2^k), k larger than 10 need a lot of storage space and are also slow to process (k=10 on my machine takes about 26 seconds).

Author(s)

Ulrike Groemping

References

Levenshtein (1991) and Na (2021)

Examples

```
# example code

## t=3, k=4
A0 <- SeqCA_Levenshtein(4, a=0)
A1 <- SeqCA_Levenshtein(4, a=1)
dim(A0); dim(A1)
coverageSeqCA(A0, 3)
coverageSeqCA(A1, 3)
## together, these cover half of the 4-sequences
coverageSeqCA(rbind(A0, A1), 4)

## t=6, k=7
dim(SeqCA_Levenshtein(7))

## t=15, k=16
try(SeqCA_Levenshtein(16))

## t=20, k=21
try(dim(SeqCA_Levenshtein(21)))


# NOTE: example() skips the following code,
# unless the user specifies
# options(run_heavy_examples = TRUE)
# This is because it could be very slow depending on sample.kind.
## runs about 16 seconds in R 4.6.1 on Windows 11
## with 32 GB RAM and Intel(R) Core(TM) ultra 7 286 V CPU @ (2.2GHz)

## t=9, k=10
system.time(print(dim(SeqCA_Levenshtein(10))))
```

```
## t=10, k=11 will take more than 11 times as long
```

---

simulated_annealing_sca

Simulated Annealing for improving an SeqCA

Description

for k columns and strength t for the third stage of an algorithm by Torres-Jimenez et al. (2022)

Usage

```
optimize_SeqCA(A, t, sa_params = list(), target=factorial(t),
           skipsa=FALSE, skipreduce=FALSE, verbose = FALSE, ...)

simulated_annealing_sca(A, t, maxAccepted = 100, maxAttempts = 500,
           initialTemperature = 100, finalTemperature = 0.1,
           temperatureReduction = 0.95, probreshuffle=0, verbose = FALSE, ...)
```

Arguments

`A`	an SeqCA of strength `t`, or for `simulated_annealing_sca`, a partial SeqCA for which coverage is to be completed
`t`	the strength for which the coverage of all permutations of length t must be guaranteed for all `combn(k,t)` t-element subsets of 1,...,k (k the number of columns of A)
`maxAccepted`	positive integer, maximum accepted perturbations for each temperature; default 100
`maxAttempts`	positive integer, maximum attempts for each temperature; default 500
`initialTemperature`	positive number, default 100
`finalTemperature`	positive number, less than initialTemperature, default 0.1
`temperatureReduction`	positive number, less than 1; the closer to 1, the slower the temperature reduction, and the more temperatures are run; default 0.95, corresponding to 135 temperatures with the other defaults
`probreshuffle`	probability of a proper reshuffle in case that is possible; value between 0 and 1; positive values do not work yet.
`verbose`	logical; if TRUE, printing progress information
`...`	currently not used
`skipsa`	logical, default FALSE; if TRUE, simulated annealing is suppressed

skipreduce	logical, default FALSE; if TRUE, the initial iterative reduction step is suppressed (useful if the SeqCA is the result of such a reduction)
target	number of rows at which no further improvement is attempted; defaults to the number of permutations needed to cover all orders for a single t-tuple, which is likely unachievable for most relevant problem sizes
sa_params	list of parameters whose default values are to be changed; all parameters not in the list keep their default states; the possible paramters are maxAccepted, maxAttempts, initialTemperature, finalTemperature, temperatureReduction, probreshuffle

Value

in case of success, a strength t SeqCA with k columns is returned; all rows contain the integers 1 to k.

Details

The function simulated_annealing_sca implements the simulated annealing approach of Torres-Jimenez et al. (2022). It can be applied directly to a partial SeqCA, but its main purpose is being called by function optimize_SeqCA for reducing the number of rows of an existing strength t SeqCA (see below).#' The perturbation function was not described in detail by Torres-Jimenez et al. 2022; they only stated that the correct order for a currently non-covered randomly selected sequence is forced for the row for which this action creates the fewest missings.
This is implemented in the internal perturbation function within function simulated_annealing_sca by simply reordering the sequence elements (which might of course break existing coverages of the row, but might also bring useful new coverages).
optimize_SeqCA implements a workflow for applying reduce_rows_iterative_complete and subsequently applying simulated annealing repeatedly.

Torres-Jimenez et al. (2022) describe a 3-stage approach, the third stage of which is simulated annealing. Simulated annealing is invoked if postprocessing of an initial greedy SeqCA is unable to find further rows that can be removed without deteriorating coverage. The post-processing and simulated annealing are combined in function optimze_SeqCA. While the simulated annealing is needed for achieving the very good performance that the paper reports on, it can be very slow. Therefore, it can be suppressed by setting skipsa=TRUE, in which case the function optimize_SeqCA is basically a wrapper for reduce_rows_iterative_complete.
The paper claims that the method is most useful for larger problems; but for these, the run time will often be prohibitive.

If optimize_SeqCA is stuck in simulated annealing iterations, having achieved some progress but neither improving nor finishing, the process can be interrupted by pressing the <ESC>-key, in which case it will return the last successful outcome.

Use of AI

Claude 4 was heavily involved in the development of these functions, by initially implementing the simulated annealing pseudo code from the paper; it worked reasonably well for everything except implementing the perturbation.

Warning

The implementation of simulated annealing is very preliminary, and the `probreshuffle` argument does not work yet for positive probabilities.

Author(s)

Ulrike Groemping

References

Torres-Jimenez et al. (2022)

See Also

`greedySeqCA_TJ()` for the greedy method by Torres-Jimenez et al. (2022) and `reduce_rows_iterative_complete()` for the postoptimization stage method of that paper that is used as the first stage of function `optimize_SeqCA`

Examples

```
## a small SeqCA with originally 13 rows
A_TJ <- greedySeqCA_TJ(6,3, postopt = FALSE, seed=2323)
dim(A_TJ)


## on Windows, the above A_TJ is quickly postoptimized to 9 runs
A_TJ_reduced <- optimize_SeqCA(A_TJ, 3, target=9, verbose=TRUE)
coverageSeqCA(A_TJ_reduced, 3)

## one less row is possible, but requires a somewhat slow simulated
## annealing optimization
A_TJ_reduced8 <- optimize_SeqCA(A_TJ, 3, target=8, verbose=TRUE)



# NOTE: example() skips the following code,
# unless the user specifies
# options(run_heavy_examples = TRUE)

## without setting the optimization target, one would have to wait until
## the simulated annealing for reducing to 7 runs failed
optimize_SeqCA(A_TJ_reduced, 3, verbose=TRUE, skipreduce=TRUE)

## Here: Kuhn much slower than TJ
nrow(A <- greedySeqCA_Kuhn(6, 4, seed=75))  ## 52 before, 41 after reduce
## Not run:
## this should only be run interactively, with a user prepared
## to interrupt it if it seems stuck
aus <- optimize_SeqCA(A, 4, skipreduce=TRUE, verbose=TRUE)

## End(Not run)
```

SMC_SCPHFs *Data for the SCPHF-based (permutation vector) constructions*

Description

Lists SMC_SCPHFs, CL_SCPHFs and WC_SCPHFs and data frame SCPHFcat that support the SCPHF-based constructions

Usage

```
SMC_SCPHFs

SCPHFcat

CL_SCPHFs

WC_SCPHFs
```

Format

SMC_SCPHFs, CL_SCPHFs and WC_SCPHFs are lists of lists:

The first hierarchy level is the strength t:
All lists hold at least elements for strengths 3 and 4, WC_SCPHFs also a few elements for strengths 5 and 6.
Each of these holds lists for the implemented numbers of levels v, which in turn hold entries for the implemented numbers of columns k.

SCPHFcat is a data frame with columns t, v, k, N and type. The latter indicates the source: for "2006", the source is the Sherwood et al. (2006) paper and SCPHFs are in SMC_SCPHFs,
for "2009" the source is Walker and Colbourn (2009) with arrays from Walker's GitHub repository and arrays are in WC_SCPHFs,
for "2018" the source is work by Erin Lanus and co-authors, and SCPHFs are in CL_SCPHFs (arrays found in the Dwyer (2024) GitHub repository).

An object of class data.frame with 79 rows and 5 columns.

An object of class list of length 2.

An object of class list of length 4.

Details

For more details, see also the documentation of function scphfCA.
The creation of the objects is documented in the R files SherwoodMartirosyanColbournCPHF.R, Lanus_SCPHFs_from_DwyerRepo.R and SCPHF_dimchecks.R in the extdata folder for this package, which can be located in the directory inst/extdata of the GitHub repo; the implemented raw SCPHFs from the DWYER repo are in the sub folder SCPHFs of that folder.

Source

Reference Sherwood, Martirosyan, Colbourn (2006), CPHFs and constructions in the paper; Colbourn and Lanus (2018) and/or Colbourn, Lanus and Sarkar (2018), SCPHFs from the repository of Dwyer (2024)

References

Colbourn and Lanus (2018), Colbourn, Lanus and Sarkar (2018), Dwyer (2024), Sherwood, Martirosyan, Colbourn (2006), Walker and Colbourn (2009)

Examples

```
## an impression of the content
SCPHFcat
SMC_SCPHFs[["4"]][["3"]]
```

---

swapvals *function to reduce the number of levels by 1*

Description

reduces the number of levels by 1, achieving a reduction of the number of rows by 2. fuse can also reduce the number of levels by more than 1, reducing the number of rows by 2 for each reduction of 1 level.

Usage

```
fuse(D, vprior, vpost, ...)
```

Arguments

D	a CA with vprior levels in each column (integers, starting a 0 or 1)
vprior	the number of levels prior to applying fuse
vpost	the number of levels after applying fuse, must be smaller than vprior
...	currently not used

Value

an array of the same coverage properties and the same number of columns as D and at most nrow(D) - 2 * (vprior - vpost) rows with vpost levels per column

Details

Note that there is a variant of fusion, implemented in function fuseBose, for which the number of runs can be reduced by three; this is applicable for strength 2 with vprior a prime or prime power and up to vprior + 1 columns in D.

References

Colbourn et al. (2010, CKRS; Lemma 3.1)

See Also

fuseBose() for the case mentioned in the Details section

Examples

```
set.seed(1212) ## for array randomization
D <- lhs::createBush(11, 12)
table(D)
dim(D)
D10 <- fuse(D, 11, 10) ## reduce by two rows and one level
table(D10)
dim(D10)
coverage(D, 3)
coverage(D10, 3)
```

TJcat *Data for function tjCA*

Description

The data.frame TJcat supports the identification of CAs from the list TJ2level_CAs

Usage

```
TJcat

TJ2level_CAs
```

Format

TJcat is a data.frame with columns t (strength), k (number of CA columns), v (number of levels), N (number of CA rows), filesize (the file size in bytes), nameInTJ2level_CAs, replaceable and code. Row names hold the names of the downloaded files.
CAs corresponding to rows with entries in code are created by executing the code of column code (either with function CAEX for 3-level or with one of several constructions for 2-level),
CAs corresponding to rows with entries in nameInTJ2level_CAs by picking the respective entry from TJ2levelCAs.
CAs corresponding to rows with neither entry cannot be created at present (many might be constructable by the power construction which will be implemented shortly and will then be incorporated in the code column).

TJ2level_CAs is a list of arrays (class ca) corresponding to the rows of TJcat with entries in nameInTJ2level_CAs (access via names).

An object of class list of length 141.

Details

The data.frame `TJcat` holds information on 2-level CAs of strength 3 from Torres-Jimenez (2021), covered by function `CAEX` of this package,
and on 2-level CAs of strengths 3 to 6. The list `TJ2level_CAs` holds actual arrays for the latter case. These have been selected by size, i.e., too large ones have been omitted. The unavailable 2-level arrays (empty string in column `TJ2level_CAs`) can *in principle* be constructed by the construction of Colbourn and Torres-Jimenez (2010). In various cases, post-processing steps by Torres-Jimenez, not implemented in the package, reduced the run size of the pure construction or added a column to the pure construction. Furthermore, as not all ingredients are available in this package (yet), some sizes can not be achieved by the package. The size of the arrays has nevertheless precluded the inclusion of the larger arrays into the package, in the interest interest of limiting package size.

Source

The source is the no longer functional website `https://www.tamps.cinvestav.mx/~oc/` by Torres-Jimenez (without year). Web archive versions of that site are useless, because they do not include the arrays.
The creation of the objects is documented by R files in the `extdata` folder for this package, which can be located in the directory `inst/extdata` of the GitHub repo; the actual raw arrays can be found there as well (omitting large ones that can be constructed at least approximately by a recursive construction).

References

Colbourn and Torres-Jimenez (2010), Torres-Jimenez (without year), Torres-Jimenez (2021)

Examples

```
## an impression of the content
head(TJcat)
fivenum(TJcat$N)
xtabs(~ v + t, TJcat)

fun <- function(t,k,v) eCAN(t,k,v)$CAN
fun <- Vectorize(fun)
diff <- TJcat[,"N"]-fun(TJcat[,"t"], TJcat[,"k"], TJcat[,"v"])
fivenum(diff)
## -11 0 0 0 23
ratio <- TJcat[,"N"]/fun(TJcat[,"t"], TJcat[,"k"], TJcat[,"v"])
fivenum(ratio)
## 0.9286 1.0000 1.0000 1.0000 1.0446
## the arrays are pretty close to the best-known,
##    occasionally even better
##    (i.e., colbournBigFrame would have to be updated)
fun <- function(t,k,v) eCAN(t,k,v)$Source
fun <- Vectorize(fun)
```

WCS_CPHFs *Data for the CPHF-based (permutation vector) constructions*

Description

List WCS_CPHFs and data frame CPHFcat that support the CPHF-based constructions

Usage

```
WCS_CPHFs

CPHFcat
```

Format

WCS_CPHFs is a list of lists:

The first hierarchy level is the strength t:
There are four lists for strengths 3 to 6,
each of which holds lists for 13 implemented numbers of levels v, which in turn hold lists of the implemented numbers of columns k.

CPHFcat is a data frame with columns t, v, k, nCPHF, hasNA and N. The construction is from Wagner, Colbourn and Simos (2022), with arrays provided by Michael Wagner.

An object of class data.frame with 323 rows and 6 columns.

Details

For more details, see also the documentation of function cphfCA.
The creation of the objects is documented in the R file WCS_CPHFs_from_read.R in the extdata folder for this package, which can be located in the directory inst/extdata of the GitHub repo; the implemented raw CPHFs are not available in the package.

Source

Wagner, Colbourn and Simos (2022)

References

Wagner, Colbourn and Simos (2022) and https://srd.sba-research.org/data/cphfipo/cphfs.zip (partly improved versus the paper, which was published based on an interim optimization state)

Examples

```
## an impression of the content
head(CPHFcat)
tail(CPHFcat)
names(WCS_CPHFs[["6"]][["3"]])
WCS_CPHFs[["6"]][["3"]][["7"]]  ## only one row --> CA with 3^6 runs
```

```
dim(WCS_CPHFs[["6"]][["3"]][["36"]])  ## 10 rows --> CA with 10*(3^6-1) + 1 runs
```

WKScat *Data for function wksCA*

Description

The data.frame WKScat supports the identification of CAs from the list WKS_CAs

Usage

```
WKScat

WKS_CAs
```

Format

`WKScat` is a data.frame in 43 rows with columns `t` (strength), `k` (number of CA columns), `v` (number of levels) and `N` (number of CA rows).

`WKS_CAs` is a list of stored strength 6 CAs for 2-level factors, constructed by Wagner, Kampel and Simos (2021). Its element names are the run numbers.

An object of class `list` of length 43.

Details

All arrays have strength 6 and 2-level columns. The data.frame `WKScat` holds information on available settings, the list `WKS_CAs` holds the actual arrays.

Source

The reference is Wagner, Kampel and Simos (2021), and the arrays were downloaded at https://srd.sba-research.org/data/sipo/.
The creation of the objects is documented by R files in the `extdata` folder for this package, which can be located in the directory `inst/extdata` of the GitHub repo; the relatively large raw arrays are not provided there, as they are available on the web.

References

Wagner, Kampel and Simos (2021)

Examples

```
## an impression of the content
head(WKScat)
tail(WKScat)
fivenum(WKScat$N)  ## 440.0 530.5 605.0 669.0 718.0
fivenum(WKScat$k)  ## 30.0 40.5 51.0 61.5 72.0

fun <- function(t,k,v) eCAN(t,k,v)$CAN
fun <- Vectorize(fun)
diff <- WKScat[,"N"]-fun(WKScat[,"t"], WKScat[,"k"], WKScat[,"v"])
fivenum(diff)
## 0 0 0 0 0
## at present, they are all optimal
```

Index

∗ **datasets**
ca12.2.8, 14
CAEX_lineages, 20
CKRScat, 28
colbournBigFrame, 29
DWYERcat, 63
MeagherStevensCombis, 83
NISTcat, 92
PALEYcat, 100
PCAcat, 105
powerCTcat, 110
SMC_SCPHFs, 130
TJcat, 132
WCS_CPHFs, 134
WKScat, 135
'CAs-package' *(CAs-package)*, 3

add1CA, 10
add1Column *(add1CA)*, 10

bestCA, *4*, 11, *22*, *24*, *31*
bestN, *10*, *23*, *24*, *27*, *80*, *111*
bestN *(bestCA)*, 11

ca12.2.8, 14
ca23.4.8 *(ca12.2.8)*, 14
ca26.4.15 *(ca12.2.8)*, 14
ca30.5.7 *(ca12.2.8)*, 14
ca41.6.6 *(ca12.2.8)*, 14
CA_to_MCA, *6*
CA_to_MCA *(MCAt)*, 79
CA_to_PCA, *17*, *118*
CA_to_PCA *(pcaCA)*, 101
ca_verify, *35*
CAEX, *5*, 16, *20*, *62*, *95*, *103*, *132*, *133*
CAEX_CAs, *6*
CAEX_CAs *(CAEX_lineages)*, 20
CAEX_lineages, *6*, 19
CAs *(CAs-package)*, 3
CAs-package, 3
caverify::ca_verify(), *36*
checkcond1, 21
checkcond2 *(checkcond1)*, 21
checkcond3 *(checkcond1)*, 21
checkcond3a *(checkcond1)*, 21
checkcond3b *(checkcond1)*, 21
checkcond4 *(checkcond1)*, 21
checkcond4a *(checkcond1)*, 21
checkcond4b *(checkcond1)*, 21
CK_constrD, 22
CK_constrDCA *(CK_constrD)*, 22
CK_doubling, *4*, 23, *95*
CK_doublingCA, *4*
CK_doublingCA *(CK_doubling)*, 23
CK_NRB, 25
CKRS_augment, 26
CKRS_augmentCA *(CKRS_augment)*, 26
CKRS_CAs, *6*
CKRS_CAs *(CKRScat)*, 28
ckrsCA *(bestCA)*, 11
CKRScat, *6*, 28, *64*
CL_SCPHFs, *6*, *124*
CL_SCPHFs *(SMC_SCPHFs)*, 130
CMMSSYCombis, *6*
CMMSSYCombis *(MeagherStevensCombis)*, 83
CMMSSYStarters, *6*
CMMSSYStarters *(MeagherStevensCombis)*, 83
CohenSA *(ca12.2.8)*, 14
colbournBigFrame, *6*, 29, *110*, *111*
ColbournKeriCombis, *6*
ColbournKeriCombis *(MeagherStevensCombis)*, 83
ColbournKeriStarters, *6*
ColbournKeriStarters *(MeagherStevensCombis)*, 83
compositCA, 30
conference_paley, 31
coverage, *4*, *23*, *24*, 34, *74*, *113*

coverage_iter (coverage), 34
coverageSeqCA, *6*, 37
coverplot, *4*
coverplot (coverage), 34
cphfCA, *5*, 38, *134*
CPHFcat, *6*, *39*, *40*
CPHFcat (WCS_CPHFs), 134
createBose, *89*, *123*
createBusht, *123*
createDHF, 41, *110*
crossCAs, *5*, *31*, 42
crossSum, 43
CS_CK, 45, *85*
CS_CMMSSY, *5*, 46, *84*, *106*
CS_LCDST, *5*, 48, *84*, *85*, *106*
CS_MS, *5*, 50, *84*, *85*, *106*
cyc, *4*, *45*
cyc (cyclotomyCA), 53
cyclotomyCA, *4*, 53, *100*
CYCLOTOMYcat, *6*
CYCLOTOMYcat (PALEYcat), 100

Dd (recBoseCA), 116
derive, 55
derivefrom (derive), 55
deriveto (derive), 55
DHF2CA, 57
DHHF2CA, 59, *110*
directSum (crossSum), 43
DoE.base, *15*, *86*
dpCA, *5*, 61
DPcat, *6*, *62*
DPcat (PCAcat), 105
dwyerCA, *63*
dwyerCA (bestCA), 11
DWYERcat, *6*, *7*, 63

eCAK, *111*
eCAK (Ns), 93
eCAN, *17*, *111*
eCAN (Ns), 93

flexpos (markflex), 76
flexprofile (markflex), 76
funmakefromstrings (readCA), 115
fuse (swapvals), 131
fuseBose, *131*
fuseBose (fuseBoseCA), 64
fuseBose(), *132*
fuseBoseCA, *5*, 64
fuseBushtCA, *5*
fuseBushtCA (fuseBoseCA), 64

gen.next.cbn, 66
greedySeqCA_Kuhn, *6*, 67
greedySeqCA_Kuhn(), *70*, *72*
greedySeqCA_naive, *6*, 69
greedySeqCA_naive(), *68*, *72*
greedySeqCA_TJ, *6*, 70
greedySeqCA_TJ(), *68*, *70*, *129*

Hartman74 (N_Hartman74), 89

identify_needed, *6*, 72
int2poly, 73
is.PCA (pcaCA), 101
iter_greedySeqCA_Kuhn
(greedySeqCA_Kuhn), 67
iter_greedySeqCA_naive
(greedySeqCA_naive), 69
iter_greedySeqCA_TJ (greedySeqCA_TJ), 70

juxtapose, 74

k_CAEX (Ns), 93
k_CK_NRB (CK_NRB), 25
k_CKRScat (Ns), 93
k_cphfCA (cphfCA), 38
k_CS_CK (CS_CK), 45
k_CS_CMMSSY (CS_CMMSSY), 46
k_CS_LCDST (CS_LCDST), 48
k_CS_MS (CS_MS), 50
k_cyc (cyclotomyCA), 53
k_CYCLOTOMYcat (Ns), 93
k_detail_CAEX (CAEX), 16
k_dpCA (Ns), 93
k_DWYERcat (Ns), 93
k_fuseBose (fuseBoseCA), 64
k_fuseBoseCA (Ns), 93
k_fuseBusht (fuseBoseCA), 64
k_fuseBushtCA (Ns), 93
k_Hartman74 (N_Hartman74), 89
k_KSK (N_KSK), 91
k_miscCAcat (Ns), 93
k_NISTcat (Ns), 93
k_ODbasedCA (ODbasedCA), 97
k_PALEYcat, *33*
k_PALEYcat (Ns), 93

k_pcaCA (Ns), 93
k_powerCT, *111*
k_powerCT (powerCA), 109
k_projectionBose (projBoseCA), 113
k_recBoseCA (Ns), 93
k_SCA_Busht (Ns), 93
k_scphfCA (scphfCA), 123
k_TJcat (Ns), 93
k_WKScat (Ns), 93
kd (recBoseCA), 116
ks (Ns), 93
ks_productCA (Ns), 93
KSK, *4*, *22*, *24*, *95*
KSK (N_KSK), 91

LCDSTCombis, *6*, *50*
LCDSTCombis (MeagherStevensCombis), 83
LCDSTStarters, *6*, *50*
LCDSTStarters (MeagherStevensCombis), 83
lhs, *39, 40*, *124*
lower_bound_CA, 75

markflex, 76
maxconstant, *57*, *59*, *61*, *100*, *103*
maxconstant (permvals), 107
MCA2 (MCAt), 79
MCAt, 79
MeagherStevensCombis, *6*, 83
MeagherStevensStarters, *6*
MeagherStevensStarters (MeagherStevensCombis), 83
miscCA, *4*, 86, *106*
miscCAcat, *6*, *86*
miscCAcat (ca12.2.8), 14
MTVTRouxtypeCA, 87

N_add1CA (add1CA), 10
N_CAEX (Ns), 93
N_CK_constrDCA (CK_constrD), 22
N_CK_doublingCA (CK_doubling), 23
N_CK_NRB (CK_NRB), 25
N_CKRS_augmentCA (CKRS_augment), 26
N_CKRScat (Ns), 93
N_compositCA (compositCA), 30
N_cphfCA (cphfCA), 38
N_CS_CK (CS_CK), 45
N_CS_CMMSSY (CS_CMMSSY), 46
N_CS_LCDST (CS_LCDST), 48
N_CS_MS (CS_MS), 50
N_cyc (cyclotomyCA), 53
N_CYCLOTOMYcat (Ns), 93
N_d_recursiveBose (recBoseCA), 116
N_d_recursiveBoseHartman (recursiveBoseHartman), 119
N_dpCA (Ns), 93
N_DWYERcat (Ns), 93
N_fuseBose (fuseBoseCA), 64
N_fuseBoseCA (Ns), 93
N_fuseBusht (fuseBoseCA), 64
N_fuseBushtCA (Ns), 93
N_Hartman74, 89
N_k_recursiveBose (recBoseCA), 116
N_k_recursiveBoseHartman_A (recursiveBoseHartman), 119
N_k_recursiveBoseHartman_d (recursiveBoseHartman), 119
N_KSK, 91
N_miscCAcat (Ns), 93
N_NISTcat (Ns), 93
N_ODbasedCA (ODbasedCA), 97
N_PALEYcat, *32*
N_PALEYcat (Ns), 93
N_pcaCA, *102*
N_pcaCA (Ns), 93
N_powerCT, *111*
N_powerCT (powerCA), 109
N_projBoseCA (Ns), 93
N_projectionBose (projBoseCA), 113
N_recBoseCA, *117*
N_recBoseCA (Ns), 93
N_SCA_Busht (Ns), 93
N_scphfCA (scphfCA), 123
N_TJcat, *17*
N_TJcat (Ns), 93
N_upper_MCA (MCAt), 79
N_upper_MTVTRouxtypeCA (MTVTRouxtypeCA), 87
N_WKScat (Ns), 93
nistCA, *92*
nistCA (bestCA), 11
NISTcat, *6*, *7*, 92
Ns, *4*, *12*, *29*, 93, *102*
Ns(), *12*
Ns_CK_doubling (Ns), 93
Ns_derive (Ns), 93
Ns_fuse (Ns), 93
Ns_productCA (Ns), 93

oa1728.12.6 (ca12.2.8), 14
oa3375.15.6 (ca12.2.8), 14
oa9261.21.6 (ca12.2.8), 14
OD2, 97
ODbasedCA, 97
one_to_howmany, 99
optimize_SeqCA, *6*
optimize_SeqCA (simulated_annealing_sca), 127

paley_mat (conference_paley), 31
paleyCA, *4*, *100*
paleyCA (conference_paley), 31
PALEYcat, *6*, 100
paleyHad (conference_paley), 31
pcaCA, *5*, 101
PCAcat, *6*, *102*, *103*, 105
permvals, 107
postopNCK, *5*, *79*, *80*
postopNCK (markflex), 76
powerCA, *5*, 109, *111*, *122*
powerCTcat, *6*, 110, *110*
prepcoverplot (coverage), 34
print.ca, 112
print.CAcoverage, 113
productCA, *5*, *20*, *106*, *117*, *118*
productCA (dpCA), 61
productCA_raw (dpCA), 61
productPCA, *5*, *15*, *48*, *51*, *62*, *106*, *117*, *118*, *123*
productPCA (pcaCA), 101
projBoseCA, *5*, 113
projBoseMCA, *6*
projBoseMCA (MCAt), 79
projectionBose, *5*
projectionBose (projBoseCA), 113

read.table, *116*
readCA, 115
readLines, *116*
recBoseCA, *5*, *48*, *103*, 116
recursiveBose, *5*
recursiveBose (recBoseCA), 116
recursiveBoseHartman, 119
reduce_rows_complete, 121
reduce_rows_iterative_complete, *6*, *67*, *71*, *128*
reduce_rows_iterative_complete (reduce_rows_complete), 121
reduce_rows_iterative_complete(), *129*

SCA_Bose, *4*, *106*, *114*, *118*, 122
SCA_Busht, *4*, *65*
SCA_Busht (SCA_Bose), 122
SCA_LCDST (CS_LCDST), 48
SCA_MS (CS_MS), 50
scphfCA, *5*, 123, *130*
SCPHFcat, *6*, *124*
SCPHFcat (SMC_SCPHFs), 130
SeqCA_Levenshtein, *6*, 125
show_paleyHad (conference_paley), 31
simulated_annealing_sca, 127
SMC (scphfCA), 123
SMC_SCPHFs, *6*, *124*, 130
swapvals, 131

tempdir, *92*
TJ2level_CAs, *6*
TJ2level_CAs (TJcat), 132
tjCA (bestCA), 11
TJcat, *6*, *20*, 132
Tred, *110*
Tred (createDHF), 41
Turan (N_Hartman74), 89

uniquecount (markflex), 76

WC_SCPHFs, *6*
WC_SCPHFs (SMC_SCPHFs), 130
WCS (cphfCA), 38
WCS_CPHFs, *6*, *39*, 134
WKS_CAs, *6*
WKS_CAs (WKScat), 135
WKScat, *6*, 135