

Package: cafast (via r-universe)

October 29, 2026

Type Package

Title Fast Companions to the 'CAs' Package for Covering Arrays

Version 0.1.0

Description Compiled implementations and generalizations of selected functions of the 'CAs' package by Ulrike Groemping for covering arrays. The package provides a C version of the search for the largest set of rows that can be made constant by permuting symbols, the power construction of Colbourn and Torres-Jimenez (2010) <[doi:10.1090/conm/523/10309](https://doi.org/10.1090/conm/523/10309)> for any strength, number of columns and number of levels, a counting kernel that tabulates value tuples for every set of t columns, and fast versions of the flexible-value tools and of the randomized post-optimization of Nayeri, Colbourn and Konjevod (2013) <[doi:10.1016/j.ejc.2012.07.017](https://doi.org/10.1016/j.ejc.2012.07.017)>. Each function was tested against its 'CAs' counterpart, and the remaining differences are documented.

License GPL (>= 3)

URL <https://github.com/chrispbsmolen/ca-tools>

BugReports <https://github.com/chrispbsmolen/ca-tools/issues>

Encoding UTF-8

Language en-US

Depends R (>= 4.0)

Imports CAs

Suggests caverify, lhs

NeedsCompilation yes

Author Christopher Smolen [aut, cre]

Maintainer Christopher Smolen <chrispbsmolen@gmail.com>

Config/pak/sysreqs libgmp-dev libxml2-dev libssl-dev

Repository <https://chrispbsmolen.r-universe.dev>

Date/Publication 2026-10-28 12:30:10 UTC

RemoteUrl <https://github.com/cran/cafast>
RemoteRef HEAD
RemoteSha 8ad434cddda03abcd1e44edcde91d7025da2048

Contents

cafast-package	2
count_core	3
flexpos_c	4
maxconstant_c	6
postopNCK_c	8
powerCT_any	10
Index	12

cafast-package	<i>Fast Companions to the CAs Package for Covering Arrays</i>
----------------	---

Description

Compiled implementations and generalizations of selected functions of the package **CAs** by Ulrike Groemping, for working with covering arrays.

- **maxconstant_c**: makes the largest possible number of rows constant, as `CAs::maxconstant` does, with the clique search in C.
- **powerCT_any**, **power_plan**, **power_build**: the power construction of Colbourn and Torres-Jimenez for any strength, number of columns and number of levels, where `CAs::powerCA` covers its catalogued settings.
- **count_core**: tuple counts for every set of `t` columns, the counting behind coverage checks.
- **flexpos_c**, **markflex_c**: flexible positions and greedy marking of flexible values, as `CAs::flexpos` and `CAs::markflex`.
- **postopNCK_c**: the post-optimization of Nayeri, Colbourn and Konjevod, as `CAs::postopNCK`.

Each function was tested against its counterpart in **CAs**. The help pages list the differences that remain, which concern tie-breaks, missing values and input checks. The test batteries, which download arrays from the NIST covering array tables, are installed in the ‘tests’ folder of the installed package.

Author(s)

Christopher Smolen <chrispbsmolen@gmail.com>

See Also

The package **CAs**, <https://CRAN.R-project.org/package=CAs>.

count_core	<i>Tuple counts for every set of t columns</i>
------------	--

Description

The counting kernel behind coverage checks of covering arrays. For every set of t columns, in lexicographic order, the function counts how often each value tuple occurs among the rows. A row with an NA in a projection contributes nothing to that projection.

Usage

```
count_core(x, t, vs = apply(x, 2, function(col) max(col, na.rm = TRUE) + 1L),
  rowmult = TRUE)
```

Arguments

<code>x</code>	an integer matrix (or a matrix or data frame that can be converted to one) with rows as runs and columns as factors, coded from 0. NA cells are allowed.
<code>t</code>	the number of columns per projection, an integer from 1 to $\text{ncol}(x)$.
<code>vs</code>	the number of levels of each column, a vector of length $\text{ncol}(x)$. The default is the largest symbol of each column plus one. Every symbol of column j must lie in $\emptyset: (\text{vs}[j]-1)$, otherwise an error is given.
<code>rowmult</code>	logical. If TRUE (the default), the matrix of row multiplicities is computed as well.

Details

The order of the column sets matches `combn` and the column sets reported by `CAs::coverage`. Tuples are linearized with the leftmost column varying fastest. The tuple space of one column set is limited to 2^{31} values, and the `rowmult` matrix to 2×10^9 cells. The computation is single threaded and can be interrupted.

Value

A list with elements

<code>tots</code>	for every set of t columns, the number of possible value tuples, the product of their <code>vs</code> .
<code>ncovered</code>	for every set of t columns, the number of tuples that occur in at least one row.
<code>missing</code>	<code>tots - ncovered</code> .
<code>rowmult</code>	if <code>rowmult = TRUE</code> , an $\text{nrow}(x)$ by <code>choose(ncol(x), t)</code> matrix whose entry for a row and a column set is the number of rows with the same tuple on that column set, or 0 if the row has an NA there. NULL otherwise.

The column sets are taken in lexicographic order, the order of `combn(ncol(x), t)`. The first three elements are numeric vectors of length `choose(ncol(x), t)`.

See Also

coverage in package **CAs**.

Examples

```
## the full factorial 2^3 covers every pair, each pair tuple twice
x <- as.matrix(expand.grid(0:1, 0:1, 0:1))
r <- count_core(x, 2)
r$ncovered
r$missing
r$rowmult[1:3, ]

## four of the runs form an orthogonal array of strength 2
oa <- x[c(1, 4, 6, 7), ]
count_core(oa, 2)$missing
## without its first run, one pair is missing in every two columns
count_core(oa[-1, ], 2, vs = c(2, 2, 2))$missing
```

flexpos_c

Flexible positions and greedy marking of flexible values

Description

C implementations of flexpos and markflex from the package **CAs**. A position of a covering array is flexible when every t-tuple through it is also covered by another row, so the position can be changed on its own without loss of coverage. markflex_c applies the greedy marking of Nayeri, Colbourn and Konjevod (2013), which turns a set of positions into NA (“don’t care”) values that are flexible together.

Usage

```
flexpos_c(D, t, conv = c("flexpos", "coverage"))
```

```
markflex_c(D, t, fixrows = 0)
```

Arguments

D	a matrix or data frame with rows as runs and columns as factors, coded from 0 or from 1 with consecutive symbols in each column. NA marks a flexible value. The number of levels of each column is its largest symbol plus one (after a shift of codes from 1). Codes that are not consecutive from 0 or from 1 give an error.
t	the strength, an integer from 2 to ncol(D).
conv	the reading of NA cells in flexpos_c. "flexpos" (the default) is the convention of CAs::flexpos: NA is an ordinary symbol that matches only NA. With "coverage" a row whose projection contains an NA contributes nothing to that projection and imposes no constraint on its cells. The two agree on arrays without NA, and "flexpos" never flags a cell that "coverage" would not flag.
fixrows	an integer from 0 to nrow(D). The first fixrows rows are not moved.

Details

A position counts as flexible in `flexpos_c` when, for every set of `t` columns through its column, the row's tuple on that set occurs in at least two rows.

`markflex_c` sorts the rows by their number of flexible positions in ascending order (a stable sort, keeping the first `fixrows` rows in place, and no sort at all if `fixrows` $\geq$ `nrow(D) - 1`), as `CAs::markflex` does. It then goes through the column sets in lexicographic order and the rows from the top. The first row that covers a tuple keeps its `t` cells of that set, and a row with an NA in the projection covers nothing. Cells that are kept by no column set become NA.

Both functions reproduce `CAs::flexpos` and `CAs::markflex` cell for cell on the test batteries of the package. The differences are in the input handling. Codes from 1 are shifted to 0 internally and back on output, codes that are not consecutive give an informative error, and `markflex_c` returns the attributes `rowOrder` and `flexcount`, where `CAs::markflex` returns `rowOrder` and `flexible`.

Value

`flexpos_c` returns a logical matrix of the dimensions of `D`, TRUE at flexible positions. Cells that are NA in `D` are TRUE.

`markflex_c` returns a matrix in the coding of `D`, with the rows in their sorted order and the flexible values set to NA. It has the attributes `rowOrder` (the indices of the rows of `D` in the order returned) and `flexcount` (the number of NA cells in each returned row).

References

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

See Also

`flexpos` and `markflex` in package `CAs`, [postopNCK_c](#).

Examples

```
plan <- rbind(c(2,2,2,2,0,0,1), c(0,0,2,2,2,2,2), c(1,1,2,1,2,0,1),
             c(0,1,2,0,1,1,0), c(2,2,1,0,2,0,2), c(2,1,0,2,1,2,2),
             c(1,2,0,0,0,2,0), c(0,2,1,1,1,2,1), c(1,0,1,2,1,0,0),
             c(2,0,0,1,2,1,0), c(0,0,0,0,0,0,1), c(1,1,1,1,0,1,2),
             c(2,2,2,2,2,2,2), c(2,2,2,2,0,1,1))
## positions not involved in any pair that only one row covers
flexpos_c(plan, 2)
## a set of positions that can be flexible together
## the last two rows are entirely flexible and can be dropped
(m <- markflex_c(plan, 2))
attr(m, "flexcount")
```

maxconstant_c

Make the largest possible number of rows constant

Description

A C implementation of maxconstant from the package **CAs** by Ulrike Groemping. A set of rows can be made constant by permuting symbols within columns exactly when every two of its rows differ in every column. The function finds a largest such set (a largest clique in the graph whose vertices are the rows and whose edges join rows that differ in every column), moves it to the front and relabels the symbols of each column so that these rows become constant.

Usage

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

Arguments

D	a numeric matrix, a uniform array with rows as runs and columns as factors. It should have no duplicated rows unless dupcheck = TRUE. NA cells are accepted (see Details), but a row consisting only of NA cells is an error.
verbose	one of 0, 1, 2 or 12, as in CAs::maxconstant. With 1 the rows used for the constant rows are reported as messages. With 2 the attribute constant_rows is added to the result. With 12 both happen. With 0 (the default) neither.
remove	logical. If TRUE, the constant rows are removed and the result has fewer rows than D. If FALSE (the default), they are moved to the front.
one_is_enough	logical. If TRUE, no clique search is made. Existing constant rows are moved to the front, and if there are none, the first row is made constant by swapping symbols within the columns.
dupcheck	logical. If TRUE, duplicated rows are removed before anything else is done.
...	currently not used.

Details

The graph is built and searched in C. The clique search is a branch and bound that stops as soon as it holds v rows, the number of levels, since no more rows can be constant. Existing constant rows are tried first as a forced start of the clique. If an unconstrained clique is strictly larger it is used, otherwise the clique containing the existing constant rows is kept, which is the preference of CAs::maxconstant. If the array already has v constant rows, or no larger clique exists, the existing constant rows are moved to the front and nothing else changes.

Differences from CAs::maxconstant (version 0.24-4). When several largest cliques exist, CAs::maxconstant takes the last one returned by igraph::largest_cliques and this function takes the first one found in row order. The number of constant rows is the same. The arrays then differ by a row order and a symbol permutation, and on arrays with duplicated rows the number of rows outside the clique that happen to be constant as well can differ. The levels are taken from the whole array, whereas

CAs::maxconstant takes them from the first column and stops unless every column has the same number of levels.

Missing values are treated as in CAs::maxconstant. A row with an NA cell is not an existing constant row. Two rows are adjacent when they differ in every column in which both are non-missing, so an NA cell matches no symbol. After the clique rows are moved to the front, the NA cells among them are filled column by column, from the top, with the levels that do not occur among the clique rows in that column, in ascending order. The symbols are then permuted, and NA cells outside the clique rows stay NA. With one_is_enough = TRUE and no existing constant row, NA cells in the first row are left as they are, as in CAs::maxconstant.

Two cases with NA cells give different results. First, the clique search here stops at v rows, while igrph::largest_cliques can return a clique of more than v rows when NA cells make extra rows compatible. No filling makes more than v rows constant, and CAs::maxconstant then returns fewer constant rows (often none), while this function returns v . Second, when only one row can be made constant, none is constant yet and the first row has an NA cell, CAs::maxconstant stops with an error and this function fills the cell. The fill uses the array's own levels, where CAs::maxconstant uses $0:(v-1)$, which is the same for arrays coded from 0.

Value

A matrix equivalent to D (rows reordered, symbols permuted within columns) whose first rows are constant, with the levels of D in ascending order: the first constant row holds the smallest level, the second the next one, and so on. With remove = TRUE these rows are dropped instead. For verbose 2 or 12 the result carries the attribute constant_rows, a list with elements design_name (the deparsed argument) and row_set_list (the indices, in D after any dupcheck, of the rows that became the constant rows).

References

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

See Also

maxconstant in package CAs, powerCT_any, which uses this function on its ingredient.

Examples

```
## a 3-level array: one row is constant and one more can be made constant
plan <- rbind(c(2,2,2,2,0,0,1), c(0,0,2,2,2,2,2), c(1,1,2,1,2,0,1),
             c(0,1,2,0,1,1,0), c(2,2,1,0,2,0,2), c(2,1,0,2,1,2,2),
             c(1,2,0,0,0,2,0), c(0,2,1,1,1,2,1), c(1,0,1,2,1,0,0),
             c(2,0,0,1,2,1,0), c(0,0,0,0,0,0,1), c(1,1,1,1,0,1,2),
             c(2,2,2,2,2,2,2), c(2,2,2,2,0,1,1))
maxconstant_c(plan, verbose = 1)

## an orthogonal array OA(64, 4, 4, 3) with three NA cells, one of
## them in row 22, which becomes the second constant row
D <- as.matrix(expand.grid(0:3, 0:3, 0:3))
D <- unname(cbind(D, (D[, 1] + D[, 2] + D[, 3]) %% 4))
D[c(2, 22, 30), 4] <- NA
```

```
head(maxconstant_c(D, verbose = 1))
```

postopNCK_c

Randomized post-optimization of a covering array

Description

The post-optimization of Nayeri, Colbourn and Konjevod (2013) as implemented in postopNCK of the package **CAs**, with the marking steps in C (`flexpos_c`, `markflex_c`). Rows whose cells can all be made flexible are removed. The other flexible values are filled at random and the array is marked again, repeatedly, so that more rows become removable.

Usage

```
postopNCK_c(D, t, fixrows = 0, outerRetry = 50, outerMaxnochange = 10,
  innerRetry = 10, innerMaxnochange = 25, seed = NULL,
  timeBudget = Inf, restarts = 0, verbose = 0)
```

Arguments

D	a covering array of strength t, a matrix or data frame with rows as runs, coded from 0 or from 1 with consecutive symbols in each column. Every column must contain all of its levels. NA marks a flexible value.
t	the strength, an integer from 2 to ncol(D).
fixrows	an integer from 0 to nrow(D). The first fixrows rows are never moved or removed.
outerRetry	the largest number of outer rounds.
outerMaxnochange	the number of consecutive outer rounds that remove no row after which the search stops (or restarts, see restarts).
innerRetry	the number of inner retries in one outer round. Each retry works on a different last row.
innerMaxnochange	the number of inner iterations without an increase in the number of flexible values of the last row after which the next inner retry starts.
seed	NULL or an integer seed for the random process. If NULL, a seed is drawn with <code>sample(32000, 1)</code> . The seed is passed to <code>set.seed</code> and stored in the attribute <code>seed</code> of the result.
timeBudget	a time limit in seconds (default Inf). When it expires, the best array found so far is returned with the attribute <code>timeout = TRUE</code> .
restarts	the number of restarts allowed after the stopping rule of outerMaxnochange is reached. A restart fills every NA of the best array at random, marks it again and continues. The default 0 keeps the stopping behaviour of <code>CAs::postopNCK</code> .
verbose	0 (the default) prints nothing. With 1 or more, the number of rows is printed in every outer round, and with 2 or more every inner retry is printed as well.

Details

A row with fewer than t non-flexible cells covers nothing and is removed. The inner loop moves a row with the most flexible values to the bottom, fills the flexible values (in a column where the bottom row has a symbol, with that symbol, elsewhere at random), marks the array again and repeats. Inner retries and outer rounds follow `CAs::postopNCK`.

The function was calibrated against `CAs::postopNCK` by the distribution of the achieved number of rows over many seeds, not by trajectory. The same seed gives different runs in the two functions. The deliberate differences, agreed with the author of **CAs**, are these. Random fills use the array's own coding. Exhausted rows are kept in an explicit vector at the top of the array, and exactly these rows are frozen between inner retries (`CAs::postopNCK` freezes up to the last exhausted index, which in practice freezes the rows with flexible values as well). An inner pass that removes nothing hands its perturbed array to the next pass instead of returning its input. The function is silent by default. The time budget and the restarts are additions taken from the paper. An interrupt returns the best array found so far.

Value

The array with the fewest rows found, of class "ca" added to the matrix classes, in the coding of D, with its remaining flexible values as NA. It has the attribute `seed`, the attribute `flexible` (`list(value = NA)`) if it contains NA cells, and `timeout = TRUE` or `interrupt = TRUE` if the search was stopped by `timeBudget` or by a user interrupt. If `nrow(D)` already equals the product of the t largest numbers of levels, D is returned unchanged.

References

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

See Also

`postopNCK` in package **CAs**, [markflex_c](#).

Examples

```
plan <- rbind(c(2,2,2,2,0,0,1), c(0,0,2,2,2,2,2), c(1,1,2,1,2,0,1),
             c(0,1,2,0,1,1,0), c(2,2,1,0,2,0,2), c(2,1,0,2,1,2,2),
             c(1,2,0,0,0,2,0), c(0,2,1,1,1,2,1), c(1,0,1,2,1,0,0),
             c(2,0,0,1,2,1,0), c(0,0,0,0,0,0,1), c(1,1,1,1,0,1,2),
             c(2,2,2,2,2,2,2), c(2,2,2,2,0,1,1))
A <- postopNCK_c(plan, 2, seed = 1)
dim(A)
## still covering: no pair is missing in any two columns
sum(count_core(A, 2, vs = rep(3, 7), rowmult = FALSE)$missing)
```

powerCT_any

*Power construction of covering arrays for any parameters***Description**

The power construction of Colbourn and Torres-Jimenez (2010) for any strength t , number of columns k and number of levels v , in the homogeneous case with one ingredient. The function powerCA of the package **CAs** builds the 132 settings of its catalogue, and these functions extend the same construction to other settings.

Usage

```
powerCT_any(t, k, v, emax = 4, qextra = 1, ingredient = NULL,
  verbose = TRUE, ...)

power_plan(t, k, v, emax = 4, qextra = 1, ingredient = NULL, maxq = 400,
  verbose = TRUE, ...)

power_build(t, k, v, q, e, ingredient = NULL, ...)
```

Arguments

t	the strength.
k	the number of columns.
v	the number of levels.
emax	the largest exponent e tried. Exponents 2:emax are enumerated.
qextra	for each exponent, the number of admissible prime powers q tried beyond the smallest one.
ingredient	NULL, a matrix, or a function. NULL (the default) uses <code>CAs::bestCA(t, q, v, ...)</code> , which may download arrays from the NIST covering array tables. A function is called as <code>ingredient(t, q, v, ...)</code> and must return a $CA(t, q, v)$ with at least q columns. A matrix is used as it is (only its first q columns), which is meant for <code>power_build</code> . NA cells of the ingredient are set to 0, and the ingredient is passed through <code>maxconstant_c</code> .
maxq	the largest prime power q considered.
verbose	logical. If TRUE, every admissible (q, e) and the resulting number of rows is printed.
q	a prime power, the number of levels of the orthogonal array and the number of columns of the ingredient.
e	the strength of the orthogonal array, at least 2.
...	passed to the ingredient function or to <code>CAs::bestCA</code> .

Details

The columns of a Bush orthogonal array $OA(q^e, q + 1, q, e)$ (from `CAs::SCA_Busht`), transposed, give a distributing hash family with q^e columns. Its first $M = (e - 1)T(t, v) + 1$ rows are used, where $T(t, v)$ is the Turan number, so $q \geq M - 1$ is needed, and $q^e \geq k$. An ingredient $CA(t, q, v)$ with N_1 rows, ρ of them constant after `maxconstant_c`, is placed in every row of the hash family, shifted by $(r - 1)\rho$ modulo v in row r . Then $\chi = \max(0, v - M(v - \rho))$ constant rows are added back, which gives $N = M(N_1 - \rho) + \chi$ rows. The assembly follows `CAs::DHHF2CA`.

`power_plan` enumerates the admissible (q, e) , builds each ingredient once and reports N . `power_build` builds the array for one (q, e) . `powerCT_any` plans and builds the smallest. The constructions with several ingredients or with T-reductions in the catalogue of `CAs::powerCA` are not covered, and the result is often larger than the best known array.

Value

`power_plan` returns a data frame with one row per admissible (q, e) , sorted by the number of rows N , with columns `t`, `k`, `v`, `e`, `q`, `M` (rows of the hash family), `N1` (rows of the ingredient), `rho` (its constant rows), `chi` (constant rows added back), `N` and `kmax` (q^e , the largest number of columns of this (q, e)). The attribute `ingredients` is a list of the ingredient arrays by q . If no (q, e) is admissible, the result is `NULL`.

`power_build` returns an integer matrix with `k` columns, levels $0:(v-1)$ and no duplicated rows, with the attributes `t`, `q`, `e`, `M`, `N1` and `origin`.

`powerCT_any` returns the array of `power_build` for the first row of the plan, with the plan as the further attribute `plan`. It gives a warning if the number of rows differs from the plan.

References

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

See Also

`powerCA`, `bestCA` and `DHHF2CA` in package **CAs**.

Examples

```
## a strength 2 binary ingredient that needs no download
ing <- function(t, q, v) CAs::KSK(k = q)
power_plan(2, 100, 2, ingredient = ing)
A <- powerCT_any(2, 100, 2, ingredient = ing, verbose = FALSE)
dim(A)
sum(count_core(A, 2, vs = rep(2, 100), rowmult = FALSE)$missing)

B <- power_build(2, 100, 2, q = 11, e = 2, ingredient = CAs::KSK(k = 11))
dim(B)
```

Index

*** package**

cafast-package, [2](#)

cafast (cafast-package), [2](#)

cafast-package, [2](#)

count_core, [2](#), [3](#)

flexpos_c, [2](#), [4](#), [8](#)

markflex_c, [2](#), [8](#), [9](#)

markflex_c (flexpos_c), [4](#)

maxconstant_c, [2](#), [6](#), [10](#), [11](#)

postopNCK_c, [2](#), [5](#), [8](#)

power_build, [2](#)

power_build (powerCT_any), [10](#)

power_plan, [2](#)

power_plan (powerCT_any), [10](#)

powerCT_any, [2](#), [7](#), [10](#)