Package 'socialranking'

August 23, 2023

Title Social Ranking Solutions for Power Relations on Coalitions

Version 1.0.1

Maintainer Felix Fritz <felix.fritz@dauphine.eu>

Description The notion of power index has been widely used in literature to evaluate the influence of individual players (e.g., voters, political parties, nations, stockholders, etc.) involved in a collective decision situation like an electoral system, a parliament, a council, a management board, etc., where players may form coalitions. Traditionally this ranking is determined through numerical evaluation. More often than not however only ordinal data between coalitions is known. The package 'socialranking' offers a set of solutions to rank players based on a transitive ranking between coalitions, including through CP-Majority, ordinal Banzhaf or lexicographic excellence solution summarized by Tahar Allouche, Bruno Escoffier, Stefano Moretti and Meltem Öztürk (2020, <doi:10.24963/ijcai.2020/3>).

License GPL-3

Encoding UTF-8

RoxygenNote 7.2.3

RdMacros Rdpack

Suggests clipr (>= 0.8), testthat (>= 3.1.2), xfun (>= 0.30.0), knitr (>= 1.40), rmarkdown (>= 2.17), covr (>= 3.6.1), partitions (>= 1.10.7)

Imports relations (>= 0.6.13), rlang (>= 1.0.6), Rdpack (>= 2.4)

Config/testthat/edition 3

VignetteBuilder knitr

URL https://github.com/jassler/socialranking

BugReports https://github.com/jassler/socialranking/issues

NeedsCompilation no

Author Felix Fritz [aut, cre],
Jochen Staudacher [aut, cph, ths],
Moretti Stefano [aut, cph, ths]

Repository CRAN

Date/Publication 2023-08-23 16:00:02 UTC

R topics documented:

`appendMissingCoalitions` *Append missing coalitions*

Description

Append an equivalence class to a power relation with all coalitions of elements that do not appear in the power relation.

Usage

```
appendMissingCoalitions(powerRelation, includeEmptySet = TRUE)
```

Arguments

powerRelation	A PowerRelation object created by PowerRelation() or as.PowerRelation()
includeEmptySet	If TRUE, include the empty set in the last equivalence class if it is missing from the power relation.

Details

For a given set of elements $N = \{1, ..., n\}$, a PowerRelation object describes a total preorder of its subsets, or coalitions, $\mathcal{P} \subseteq 2^N$, where 2^N is the superset of elements.

If $\mathcal{P} \neq 2^N$, that means that there are some coalitions $S \in 2^N, S \notin \mathcal{P}$, such that we cannot compare $S \succeq T$ or $T \succeq S$ for every $T \in \mathcal{P}$.

This may be caused by 2^N having too many coalitions to consider. In certain cases, it may be more interesting to only consider the top ranking coalitions and "shoving" all remaining coalitions into the back.

For this use-case, appendMissingCoalitions() takes the set $2^N \setminus \mathcal{P}$ and attaches it in form of an equivalence class to the back of the power relation.

I.e., take as an example $12 \succ 13 \succ (1 \sim 2)$. Here, we have

$$\begin{aligned} 2^N &= \{123, 12, 13, 23, 1, 2, 3, \emptyset\} \\ \mathcal{P} &= \{12, 13, 1, 2\} \\ 2^N \setminus \mathcal{P} &= \{123, 23, 3, \emptyset\}. \end{aligned}$$

Adding the missing coalitions to the power relation then gives us $12 \succ 13 \succ (1 \sim 2) \succ (123 \sim 23 \sim 3 \sim \emptyset)$.

Value

PowerRelation object containing the following values:

- $elements: vector of elements
- $eqs: equivalence classes. Nested list of lists, each containing vectors representing groups of elements in the same equivalence class
- $coalitionLookup: function(v) taking a coalition vector v and returning the equivalence class it belongs to. See coalitionLookup() for more.
- $elementLookup: function(e) taking an element e and returning a list of 2-sized tuples. See elementLookup() for more.

See Also

Other helper functions for transforming power relations: makePowerRelationMonotonic()

Examples

```
pr <- as.PowerRelation(list(c(1,2), 3))
# 12 > 3

appendMissingCoalitions(pr)
# 12 > 3 > (123 ~ 13 ~ 23 ~ 1 ~ 2 ~ {})

appendMissingCoalitions(pr, includeEmptySet = FALSE)
# 12 > 3 > (123 ~ 13 ~ 23 ~ 1 ~ 2)
```

`as.PowerRelation` *Create PowerRelation object*

Description

Alternative ways of creating `PowerRelation` objects.

Usage

```
as.PowerRelation(x, ...)

## S3 method for class 'character'
as.PowerRelation(x, ...)

## S3 method for class 'list'
as.PowerRelation(x, ..., comparators = c(">"))
```

Arguments

`x`	An object
`...`	Optional additional parameters
`comparators`	Vector of ">" or "~" characters

Using a character string

The same way a power relation $\succeq$ may be represented in literature (or printed by an `PowerRelation` object), a simple string containing letters, numbers, > or ~ can be used to input a new power relation.

Every special character is ignored, with the exception of $\succeq$ (`"\u227B"`) and $\sim$ (`"\u223C"`).

Every letter or number is assumed to be an individual element. `"abc > ac"` therefore would represent two coalitions, the first one of size 3 with the elements a, b, and c. This method does not allow for elements to be entered that are supposed to be multiple characters long.

An empty coalitions can be simply left blank (i.e., `"abc > ~ ac"`), though it is often clearer if curly braces are used to indicate such (i.e., `"abc > {} ~ ac"`).

Using a list

Create a `PowerRelation` object with an unnested list of coalition vectors.

By default, a linear order is assumed on the coalitions. I.e., if it is given `list(c(1,2),1,2)`, these three coalitions are put into their own equivalence class, producing `12 > 1 > 2`.

The comparators in-between can be adjusted to indicate whether the relation between two coalitions is that of strict preference > or indifference ~.

Examples

```
# Using character strings
as.PowerRelation("abc > ab > ({} ~ c) > (a ~ b ~ ac) > bc")
# abc > ab > ({} ~ c) > (a ~ b ~ ac) > bc

# using createPowerset(), then shifting coalitions up and down using Alt+Up and Alt+Down
if(interactive()) {
  createPowerset(1:2, result = "copy")
}
as.PowerRelation("
  12
  > 1
  ~ {}
  > 2
")

# Using lists
as.PowerRelation(list(c(1,2), 2, c(), 1))
# 12 > 2 > {} > 1

as.PowerRelation(list(c(1,2), 2, c(), 1), comparators = c("~", ">", ">"))
# (12 ~ 2) > {} > 1

# the length of comparators doesn't necessarily matter.
# If comparators are missing, the existing ones are simply repeated...
as.PowerRelation(list(c(1,2), 2, c(), 1), comparators = "~")
# (12 ~ 2 ~ {} ~ 1)

as.PowerRelation(list(c(1,2), 2, c(), 1), comparators = c("~", ">"))
# (12 ~ 2) > ({} ~ 1)

# ... or the rest is cut off
as.PowerRelation(list(c(1,2), 2, c(), 1), comparators = c("~", ">", "~", "~", ">"))
# (12 ~ 2) > ({} ~ 1)
```

`coalitionsAreIndifferent`

Are coalitions indifferent

Description

Check if coalitions are indifferent to one another, or, in other words, if they appear in the same equivalence class.

Usage

```
coalitionsAreIndifferent(powerRelation, c1, c2)
```

Arguments

powerRelation	A PowerRelation object created by PowerRelation() or as.PowerRelation()
c1	Coalition vector
c2	Coalition vector

Value

Logical value TRUE if c1 and c2 are in the same equivalence class, else FALSE.

See Also

Other lookup functions: elementLookup(), equivalenceClassIndex()

Examples

```
pr <- PowerRelation(list(list(c(1,2)), list(1, 2)))

stopifnot(coalitionsAreIndifferent(pr, c(1,2), c(1)) == FALSE)
stopifnot(coalitionsAreIndifferent(pr, 2, 1) == TRUE)

# Note that it doesn't fail with non-existing power relations
stopifnot(coalitionsAreIndifferent(pr, 1, c()) == FALSE)
stopifnot(coalitionsAreIndifferent(pr, 3, c(1,2,3)) == TRUE)
```

---

copelandScores *Copeland-like method*

Description

Based on cpMajorityComparison(), add or subtract scores based on how an element fares against the others.

copelandRanking() returns the corresponding ranking.

Usage

```
copelandScores(powerRelation, elements = powerRelation$elements)

copelandRanking(powerRelation)
```

Arguments

powerRelation	A PowerRelation object created by PowerRelation() or as.PowerRelation()
elements	Vector of elements of which to calculate their scores. By default, the scores of all elements in powerRelation$elements are considered.

Details

Strongly inspired by the Copeland score of social choice theory (Copeland 1951), the Copeland-like solution is based on the net flow of the CP-majority graph (Allouche et al. 2020).

Individuals are ordered according to the number of pairwise winning comparisons, minus the number of pairwise losing comparisons, over the set of all CP-comparisons.

More formally, in a given PowerRelation pr with element i, count the number of elements $j \in N \setminus \{i\}$ where cpMajorityComparison(pr, i, j) >= 0 and subtract those where cpMajorityComparison(pr, i, j) <= 0.

Value

Score function returns a list of type CopelandScores and length of powerRelation$elements (unless parameter elements is specified). Each element is a vector of 2 numbers, the number of pairwise winning comparisons and the number of pairwise losing comparisons. Those two numbers summed together gives us the actual ordinal Copeland score.

Ranking function returns corresponding SocialRanking object.

References

Allouche T, Escoffier B, Moretti S, Öztürk M (2020). "Social Ranking Manipulability for the CP-Majority, Banzhaf and Lexicographic Excellence Solutions." In Bessiere C (ed.), *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI-20*, 17–23. doi:10.24963/ijcai.2020/3, Main track.

Copeland AH (1951). "A reasonable social welfare function." mimeo, 1951. University of Michigan.

See Also

Other CP-majority based functions: cpMajorityComparison(), kramerSimpsonScores()

Other ranking solution functions: L1Scores(), cumulativeScores(), kramerSimpsonScores(), lexcelScores(), ordinalBanzhafScores()

Examples

```
# (123 ~ 12 ~ 3 ~ 1) > (2 ~ 23) > 13
pr <- PowerRelation(list(
  list(c(1,2,3), c(1,2), 3, 1),
  list(c(2,3), 2),
  list(c(1,3))
))

copelandScores(pr)
# `1` = c(2, -1)
```

```
# `2` = c(2, -2)
# `3` = c(1, -2)

# only calculate results for two elements
copelandScores(pr, c(1,3))
# `1` = c(2, -1)
# `3` = c(1, -2)

# or just one element
copelandScores(pr, 2)
# `2` = c(2, -2)

# 1 > 2 > 3
copelandRanking(pr)
```

`cpMajorityComparison` *CP-Majority relation*

Description

The Ceteris Paribus-majority relation compares the relative success between two players joining a coalition.

`cpMajorityComparisonScore()` only returns two numbers, a positive number of coalitions where `e1` beats `e2`, and a negative number of coalitions where `e1` is beaten by `e2`.

Usage

```
cpMajorityComparison(
  powerRelation,
  e1,
  e2,
  strictly = FALSE,
  includeEmptySet = TRUE
)

cpMajorityComparisonScore(
  powerRelation,
  e1,
  e2,
  strictly = FALSE,
  includeEmptySet = TRUE
)
```

Arguments

`powerRelation`	A `PowerRelation` object created by `PowerRelation()` or `as.PowerRelation()`
`e1, e2`	Elements in `powerRelation$elements`

`strictly` Only include $D_{ij}(\succ)$ and $D_{ji}(\succ)$, i.e., coalitions $S \in 2^{N\setminus\{i,j\}}$ where $S\cup\{i\} \succ S \cup \{j\}$ and vice versa.

`includeEmptySet`
If `TRUE`, check $\{i\} \succeq \{j\}$ even if empty set is not part of the power relation.

Details

Given two elements i and j, go through each coalition $S \in 2^{N\setminus\{i,j\}}$. $D_{ij}(\succeq)$ then contains all coalitions S where $S\cup\{i\} \succeq S\cup\{j\}$ and $D_{ji}(\succeq)$ contains all coalitions where $S\cup\{j\} \succeq S\cup\{i\}$.

The cardinalities $d_{ij}(\succeq) = |D_{ij}|$ and $d_{ji}(\succeq) = |D_{ji}|$ represent the score of the two elements, where $i \succ j$ if $d_{ij}(\succeq) > d_{ji}(\succeq)$ and $i \sim j$ if $d_{ij}(\succeq) == d_{ji}(\succeq)$.

`cpMajorityComparison()` tries to retain all that information. The list returned contains the following information. Note that in this context the two elements i and j refer to element 1 and element 2 respectively.

- `$e1`: list of information about element 1
  - `$e1$name`: name of element 1
  - `$e1$score`: score $d_{ij}(\succeq)$. $d_{ij}(\succ)$ if `strictly == TRUE`
  - `$e1$winningCoalitions`: list of coalition `vectors` $S \in D_{ij}(\succeq)$. $S \in D_{ij}(\succ)$ if `strictly == TRUE`
- `$e2`: list of information about element 2
  - `$e2$name`: name of element 2
  - `$e1$score`: score $d_{ji}(\succeq)$. $d_{ji}(\succ)$ if `strictly == TRUE`
  - `$e1$winningCoalitions`: list of coalition `vectors` $S \in D_{ji}(\succeq)$. $S \in D_{ji}(\succ)$ if `strictly == TRUE`
- `$winner`: name of higher scoring element. `NULL` if they are indifferent.
- `$loser`: name of lower scoring element. `NULL` if they are indifferent.
- `$tuples`: a list of coalitions $S \in 2^{N\setminus\{i,j\}}$ with:
  - `$tuples[[x]]$coalition`: `vector`, the coalition S
  - `$tuples[[x]]$included`: logical, `TRUE` if $S\cup\{i\}$ and $S\cup\{j\}$ are in the power relation
  - `$tuples[[x]]$winner`: name of the winning element i where $S\cup\{i\} \succ S\cup\{j\}$. It is `NULL` if $S\cup\{i\} \sim S\cup\{j\}$
  - `$tuples[[x]]$e1`: index x_1 at which $S\cup\{i\} \in \sum_{x_1}$
  - `$tuples[[x]]$e2`: index x_2 at which $S\cup\{j\} \in \sum_{x_2}$

The much more efficient `cpMajorityComparisonScore()` only calculates `$e1$score`.

Unlike Lexcel, Ordinal Banzhaf, etc., this power relation can introduce cycles. For this reason the function `cpMajorityComparison()` and `cpMajorityComparisonScore()` only offers direct comparisons between two elements and not a ranking of all players. See the other CP-majority based functions that offer a way to rank all players.

Value

`cpMajorityComparison()` returns a list with elements described in the details.

`cpMajorityComparisonScore()` returns a vector of two numbers, a positive number of coalitions where `e1` beats `e2` ($d_{ij}(\succeq)$), and a negative number of coalitions where `e1` is beaten by `e2` ($-d_{ji}(\succeq$)).

References

Haret A, Khani H, Moretti S, Öztürk M (2018). "Ceteris paribus majority for social ranking." In *27th International Joint Conference on Artificial Intelligence (IJCAI-ECAI-18)*, 303–309.

Fayard N, Escoffier MÖ (2018). "Ordinal Social ranking: simulation for CP-majority rule." In *DA2PL'2018 (From Multiple Criteria Decision Aid to Preference Learning)*.

See Also

Other CP-majority based functions: `copelandScores()`, `kramerSimpsonScores()`

Examples

```
pr <- as.PowerRelation("ac > (a ~ b) > (c ~ bc)")

scores <- cpMajorityComparison(pr, "a", "b")
scores
# a > b
# D_ab = {c, {}}
# D_ba = {{}}
# Score of a = 2
# Score of b = 1

stopifnot(scores$e1$name == "a")
stopifnot(scores$e2$name == "b")
stopifnot(scores$e1$score == 2)
stopifnot(scores$e2$score == 1)
stopifnot(scores$e1$score == length(scores$e1$winningCoalitions))
stopifnot(scores$e2$score == length(scores$e2$winningCoalitions))

# get tuples with coalitions S in 2^(N - {i,j})
emptySetTuple <- Filter(function(x) identical(x$coalition, c()), scores$tuples)[[1]]
playerCTuple  <- Filter(function(x) identical(x$coalition, "c"), scores$tuples)[[1]]

# because {}u{a} ~ {}u{b}, there is no winner
stopifnot(is.null(emptySetTuple$winner))
stopifnot(emptySetTuple$e1 == emptySetTuple$e2)

# because {c}u{a} > {c}u{b}, player "a" gets the score
stopifnot(playerCTuple$winner == "a")
stopifnot(playerCTuple$e1 < playerCTuple$e2)
stopifnot(playerCTuple$e1 == 1L)
stopifnot(playerCTuple$e2 == 3L)

cpMajorityComparisonScore(pr, "a", "b") # c(1,0)
cpMajorityComparisonScore(pr, "b", "a") # c(0,-1)
```

createPowerset *Create powerset*

Description

Given a vector of elements generate a power set.

Usage

```
createPowerset(
  elements,
  includeEmptySet = TRUE,
  result = c("return", "print", "copy", "printCompact", "copyCompact")
)
```

Arguments

`elements`	vector of elements
`includeEmptySet`	If `TRUE`, an empty vector is added at the end
`result`	What to do with the result. Can be either: • `"return"`: return list object • `"print"`: create valid string to call `PowerRelation()` or `as.PowerRelation()` and print it • `"copy"`: create valid string to call `PowerRelation()` or `as.PowerRelation()` and copy it to clipboard • `"printCompact"` and `"copyCompact"`: same as `"print"` and `"copy"` but without newlines

Value

List of power set vectors. If the parameter `result` is set to `"print"` or `"copy"`, nothing is returned. Instead, a character string is generated that can be used in R to call and create a new `PowerRelation` object. This string is either printed or copied to clipboard (see argument `result`).

Examples

```
# normal return type is a list of vectors
createPowerset(c("Alice", "Bob"), includeEmptySet = FALSE)
## [[1]]
## [1] "Alice" "Bob"
##
## [[2]]
## [1] "Alice"
##
## [[3]]
## [1] "Bob"
```

```
# instead of creating a list, print the power set such that it can be copy-pasted
# and used to create a new PowerRelation object
createPowerset(letters[1:4], result = "print")
# prints
# as.PowerRelation("
#   abcd
#   > abc
#   > abd
#   > acd
#   > bcd
#   > ab
#   ...
#   > {}
# ")

createPowerset(letters[1:3], includeEmptySet = FALSE, result = "printCompact")
# as.PowerRelation("abc > ab > ac > bc > a > b > c")

# create the same string as before, but now copy it to the clipboard instead
if(interactive()) {
  createPowerset(1:3, result = "copyCompact")
}

# Note that as.PowerRelation(character) only assumes single-char elements.
# As such, the generated function call string with multi-character names
# looks a little different.
createPowerset(c("Alice", "Bob"), result = "print")
# PowerRelation(rlang::list2(
#   list(c("Alice", "Bob")),
#   list(c("Alice")),
#   list(c("Bob")),
#   list(c()),
# ))
```

`cumulativeScores` *Cumulative scores*

Description

Calculate cumulative score vectors for each element.

Usage

```
cumulativeScores(powerRelation, elements = powerRelation$elements)

cumulativelyDominates(powerRelation, e1, e2, strictly = FALSE)
```

Arguments

powerRelation	A PowerRelation object created by PowerRelation() or as.PowerRelation()
elements	Vector of elements of which to calculate their scores. By default, the scores of all elements in powerRelation$elements are considered.
e1, e2	Elements in powerRelation$elements
strictly	If TRUE, check if p1 *strictly* dominates p2

Details

An element's cumulative score vector is calculated by cumulatively adding up the amount of times it appears in each equivalence class in the powerRelation. I.e., in a linear power relation with eight coalitions, if element 1 appears in coalitions placed at 1, 3, and 6, its score vector is [1, 1, 2, 2, 2, 3, 3, 3].

Value

Score function returns a list of type CumulativeScores and length of powerRelation$elements (unless parameter elements is specified). Each index contains a vector of length powerRelation$eqs, cumulatively counting up the number of times the given element appears in each equivalence class.

cumulativelyDominates() returns TRUE if e1 cumulatively dominates e2, else FALSE.

Dominance

i dominates j if, for each index x, $\text{Score}(i)_x \geq \text{Score}(j)_x$.

i *strictly* dominates j if there exists an x such that $\text{Score}(i)_x > \text{Score}(j)_x$.

References

Moretti S (2015). "An axiomatic approach to social ranking under coalitional power relations." *Homo Oeconomicus*, **32**(2), 183–208.

Moretti S, Öztürk M (2017). "Some axiomatic and algorithmic perspectives on the social ranking problem." In *International Conference on Algorithmic Decision Theory*, 166–181. Springer.

See Also

Other ranking solution functions: L1Scores(), copelandScores(), kramerSimpsonScores(), lexcelScores(), ordinalBanzhafScores()

Examples

```
pr <- as.PowerRelation("12 > 1 > 2")

# `1`: c(1, 2, 2)
# `2`: c(1, 1, 2)
cumulativeScores(pr)

# calculate for selected number of elements
cumulativeScores(pr, c(2))
```

```
# TRUE
d1 <- cumulativelyDominates(pr, 1, 2)

# TRUE
d2 <- cumulativelyDominates(pr, 1, 1)

# FALSE
d3 <- cumulativelyDominates(pr, 1, 1, strictly = TRUE)

stopifnot(all(d1, d2, !d3))
```

`dominates` *Dominance*

Description

Check if one element dominates the other.

Usage

```
dominates(powerRelation, e1, e2, strictly = FALSE, includeEmptySet = TRUE)
```

Arguments

`powerRelation`	A `PowerRelation` object created by `PowerRelation()` or `as.PowerRelation()`
`e1, e2`	Elements in `powerRelation$elements`
`strictly`	If `TRUE`, check if `p1` *strictly* dominates `p2`
`includeEmptySet`	If `TRUE`, check $\{i\} \succeq \{j\}$ even if empty set is not part of the power relation.

Details

i is said to dominate j if $S \cup \{i\} \succeq S \cup \{j\}$ for all $S \in 2^{N \setminus \{i,j\}}$.

i *strictly* dominates j if there also exists an $S \in 2^{N \setminus \{i,j\}}$ such that $S \cup \{i\} \succ S \cup \{j\}$.

Value

Logical value `TRUE` if `e1` dominates `e2`, else `FALSE`.

Examples

```
pr <- as.PowerRelation("12 > 1 > 2")

# TRUE
d1 <- dominates(pr, 1, 2)

# FALSE
d2 <- dominates(pr, 2, 1)

# TRUE (because it's not strict dominance)
d3 <- dominates(pr, 1, 1)

# FALSE
d4 <- dominates(pr, 1, 1, strictly = TRUE)

stopifnot(all(d1, !d2, d3, !d4))
```

---

doRanking *Create a* SocialRanking *object*

Description

Rank elements based on their scores.

Usage

```
doRanking(scores, compare = NULL, decreasing = TRUE)
```

Arguments

scores	A vector or list representing each element's score. If names(scores) is not NULL, those will be used as element names. Else a number sequence corresponding to the elements is generated.
compare	Optional comparison function taking in two elements and returning a numerical value based on the relation between these two elements. If set to NULL, the default order() function is called. See details for more information.
decreasing	If TRUE (default), elements with higher scores are ranked higher.

Details

All ranking solutions in the package are tied to the scores or score vectors of the elements. For these kinds of solutions, doRanking() offers a simple way that turns a (named) vector or list of scores for each element into a SocialRanking object. For example, doRanking(c(a=1,b=2)) produces b > a ($bP^{\succeq}a$), because b with a score of 2 should be placed higher than a with a score of 1.

Ranking solutions in the package include lexcelRanking(), ordinalBanzhafRanking() and L1Ranking(), among others. These functions take a power relation, calculate the scores of each element and returns a SocialRanking object.

R natively supports sorting for vectors, but not for lists. If the use of lists is necessary, or if the native sort method in vectors does not produce the desired results, there are two possible ways to solve this:

1. by the introduction of custom S3 classes, or
2. by setting the `compare` parameter in `doRanking()`.

For S3 classes, the class for the score object has to be set and the `==` and `>` (and `[` for lists) operators overloaded. I.e., `lexcelScores()` returns a list with the custom class `LexcelScores` that implements `==.LexcelScores`, `>.LexcelScores`, `[.LexcelScores` and `is.na.LexcelScores`.

In cases where we only want to experiment, introducing new S3 classes can be cumbersome. As an alternative, the `compare` parameter can be assigned a function. This function must take two parameters, i.e., `function(a, b)`, where `a` and `b` are the scores of two arbitrary elements. The function then must return one of the following:

- `> 0` (positive value) if score `a` is ranked higher than score `b`,
- `< 0` (negative value) if score `a` is ranked lower than score `b`, or
- `= 0` if both scores `a` and `b` are considered equal.

In `doRanking(c(a=3,b=2,c=2), compare = function(a,b) a - b)`, the compare function returns a positive value of the first parameter is larger than the second. `a` has the highest value and will there for be ranked highest, `a > b ~ c`.

Conversely, `doRanking(c(a=3,b=2,c=2), compare = function(a,b) b - a)` favors elements with lower scores, resulting in the element ranking `b ~ c > a`.

Value

A list of type `SocialRanking`. Each element of the list contains a vector of elements in `powerRelation$elements` that are indifferent to one another.

See Also

`SocialRanking()`

Examples

```
doRanking(c(a=1,b=2))
# b > a

doRanking(c(a=2,b=2))
# a ~ b

# a custom ranking function. Here, we implement the following ranking solution:
# disregard any big coalitions and only rank elements based on their individual performances
# iRj if and only if {i} >= {j}
singletonRanking <- function(pr) {
  scores <- sapply(pr$elements, equivalenceClassIndex, powerRelation = pr)
  # note that coalitions in higher indexed equivalence classes are less preferable
  # hence, scores should be sorted in an increasing order
  doRanking(scores, decreasing = FALSE)
}
```

```
pr <- as.PowerRelation("abc > ab > ac > b ~ c ~ bc > a")
singletonRanking(pr)
# b ~ c > a

# a reverse lexcel ranking, where vectors are compared right to left
# here, we introduce a compare function. It returns:
# * 0, if a and b are identical
# * a positive value, if a[i] > b[i] and every value after that is equal
# * a negative value, if a[i] < b[i] and every value after that is equal
reverseLexcelCompare <- function(a, b) {
  i <- which(a != b) |> rev()
  if(length(i) == 0) 0
  else a[i[1]] - b[i[1]]
}

scores <- unclass(cumulativeScores(pr))

# R cannot natively sort a class. Instead:
# Method 1 - utilize the compare parameter
doRanking(scores, compare = reverseLexcelCompare)


# Method 2 - introduce S3 class
`[.RevLex` <- function(x, i, ...) structure(unclass(x)[i], class = "RevLex")
`==.RevLex` <- function(a, b) reverseLexcelCompare(a[[1]],b[[1]]) == 0
`>.RevLex` <- function(a, b) reverseLexcelCompare(a[[1]],b[[1]]) > 0
is.na.RevLex <- function(x) FALSE
doRanking(structure(scores, class = "RevLex"))

stopifnot(
  doRanking(scores, compare = reverseLexcelCompare) ==
  doRanking(structure(scores, class = "RevLex"))
)
```

`elementLookup` *Element lookup*

Description

List coalitions that an element appears in.

Usage

```
elementLookup(powerRelation, element)
```

Arguments

`powerRelation`	A `PowerRelation` object created by `PowerRelation()` or `as.PowerRelation()`
`element`	an element in `powerRelation$elements`

Details

This function calls `powerRelation$elementLookup(element)`. The returned list contains tuples containing the index to find the corresponding coalitions in `powerRelation$eqs`.

If `elementLookup(powerRelation, 2)` returns `list(c(1,1), c(1,2), c(3,1))`, we can determine that the element 2 appears twice in equivalence class 1 and once in equivalence class 3. The specific coalition then can be accessed with `powerRelation$eqs[[i]][[j]]`, where `i` is the equivalence class index and `j` is the coalition in that equivalence class containing the element.

Value

List of tuples, each of size 2. First value of a tuple indicates the equivalence class index, the second value the index inside that equivalence class with the coalition containing the element. Returns `NULL` if the element does not exist.

See Also

Other lookup functions: `coalitionsAreIndifferent()`, `equivalenceClassIndex()`

Examples

```
pr <- as.PowerRelation("12 > 2 ~ 1")

l <- elementLookup(pr, 1)
l
# (1,1), (2,2)

sapply(l, function(tuple) 1 %in% pr$eqs[[tuple[1]]][[tuple[2]]]) |> all() |> stopifnot()

# if element does not exist, it returns NULL
elementLookup(pr, 3) |> is.null() |> stopifnot()
```

---

equivalenceClassIndex *Get index of equivalence class containing a coalition*

Description

Given a `coalition` vector, return the equivalence class index it appears in.

Usage

```
equivalenceClassIndex(powerRelation, coalition)

coalitionLookup(powerRelation, coalition)
```

Arguments

`powerRelation`	A `PowerRelation` object created by `PowerRelation()` or `as.PowerRelation()`
`coalition`	a coalition vector or that is part of `powerRelation`

Details

This function calls `powerRelation$coalitionLookup(coalition)`.

`equivalenceClassIndex()` serves as an alias to `coalitionLookup()`.

Value

Numeric value, equivalence class index containing `coalition`. `NULL` if the coalition does not exist. If the `powerRelation` contains cycles, it is possible that multiple values are returned.

See Also

Other lookup functions: `coalitionsAreIndifferent()`, `elementLookup()`

Examples

```
pr <- as.PowerRelation("12 > 2 ~ 1")

(e1 <- equivalenceClassIndex(pr, c(1, 2)))
# 1

(e2 <- equivalenceClassIndex(pr, c(1)))
# 2

(e3 <- equivalenceClassIndex(pr, c(2)))
# 2

(e4 <- equivalenceClassIndex(pr, c()))
# NULL <- empty set does not exist

stopifnot(all(c(e1,e2,e3,e4) == c(1,2,2)))
```

`generateNextPartition` *Next partition*

Description

Skip to the next partition of the generator.

Usage

```
generateNextPartition(gen)
```

Arguments

`gen`	A generator object.

Value

A generator function. If the generator is already down to its last partition, it will throw an error.

See Also

Other generator functions: `powerRelationGenerator()`

Examples

```
coalitions <- createPowerset(c('a','b'), includeEmptySet = FALSE)
# list(c('a','b'), 'a', 'b')

gen <- powerRelationGenerator(coalitions)
gen()
# (ab ~ a ~ b)

gen()
# (ab ~ b) > a

# skipping partition of size two, where the first partition has
# 2 coalitions and the second partition has 1 coalition
gen <- generateNextPartition(gen)
gen()
# ab > (a ~ b)

# only remaining partition is one of size 3, wherein each
# equivalence class is of size 1
gen <- generateNextPartition(gen)
gen()
# ab > a > b

# went through all partitions, it will only generate NULL now
gen <- generateNextPartition(gen)
stopifnot(is.null(gen()))
```

---

kramerSimpsonScores *Kramer-Simpson-like method*

Description

Calculate the Kramer-Simpson-like scores. Lower scores are better.

`kramerSimpsonRanking()` returns the corresponding ranking.

Usage

```
kramerSimpsonScores(
  powerRelation,
  elements = powerRelation$elements,
```

```
  compIvsI = FALSE
)

kramerSimpsonRanking(powerRelation, compIvsI = FALSE)
```

Arguments

powerRelation	A PowerRelation object created by PowerRelation() or as.PowerRelation()
elements	Vector of elements of which to calculate their scores. By default, the scores of all elements in powerRelation$elements are considered.
compIvsI	If TRUE, include CP-Majority comparison $d_{ii}(\succeq)$, or, the CP-Majority comparison score of an element against itself, which is always 0.

Details

Inspired by the Kramer-Simpson method of social choice theory (Simpson 1969) (Kramer 1975), the *Kramer-Simpson-like* method compares each element against all other elements using the CP-Majority rule.

For a given element i calculate the cpMajorityComparisonScore() against all elements j, $d_{ji}(\succeq)$ (notice that i and j are in reverse order). $\max_{j \in N \setminus \{i\}}(d_{ji}(\succeq))$ then determines the final score, where lower scoring elements are ranked higher.

Value

Score function returns a list of type KramerSimpsonScores and length of powerRelation$elements (unless parameter elements is specified). Lower scoring elements are ranked higher.

Ranking function returns corresponding SocialRanking object.

Note

By default this function does not compare $d_{ii}(\succeq)$. In other terms, the score of every element is the maximum CP-Majority comparison score against all other elements.

This is slightly different from definitions found in (Allouche et al. 2020). Since by definition $d_{ii}(\succeq) = 0$ always holds, the Kramer-Simpson scores in those cases will never be negative, possibly discarding valuable information.

For this reason kramerSimpsonScores() and kramerSimpsonRanking() includes a compIvsI parameter that can be set to TRUE if one wishes for $d_{ii}(\succeq) = 0$ to be included in the comparisons. Put into mathematical terms, if:

compIvsI	Score definition
FALSE	$\max_{j \in N \setminus \{i\}}(d_{ji}(\succeq))$
TRUE	$\max_{j \in N}(d_{ji}(\succeq))$

References

Allouche T, Escoffier B, Moretti S, Öztürk M (2020). "Social Ranking Manipulability for the CP-Majority, Banzhaf and Lexicographic Excellence Solutions." In Bessiere C (ed.), *Proceed-*

ings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI-20, 17–23. doi:10.24963/ijcai.2020/3, Main track.

Simpson PB (1969). "On defining areas of voter choice: Professor Tullock on stable voting." *The Quarterly Journal of Economics*, **83**(3), 478–490.

Kramer GH (1975). "A dynamical model of political equilibrium." *Journal of Economic Theory*, **16**(2), 310–334.

See Also

Other CP-majority based functions: `copelandScores()`, `cpMajorityComparison()`

Other ranking solution functions: `L1Scores()`, `copelandScores()`, `cumulativeScores()`, `lexcelScores()`, `ordinalBanzhafScores()`

Examples

```
# 2 > (1 ~ 3) > 12 > (13 ~ 23) > {} > 123
pr <- as.PowerRelation("2 > (1~3) > 12 > (13~23) > {} > 123")

# get scores for all elements
# cpMajorityComparisonScore(pr, 2, 1) = 1
# cpMajorityComparisonScore(pr, 3, 1) = -1
# therefore the Kramer-Simpson-Score for element
# `1` = 1
#
# Score analogous for the other elements
# `2` = -1
# `3` = 2
kramerSimpsonScores(pr)

# get scores for two elements
# `1` = 1
# `3` = 2
kramerSimpsonScores(pr, c(1,3))

# or single element
# result is still a list
kramerSimpsonScores(pr, 2)

# note how the previous result of element 2 is negative.
# If we compare element 2 against itself, its max score will be 0
kramerSimpsonScores(pr, 2, compIvsI = TRUE)

# 2 > 1 > 3
kramerSimpsonRanking(pr)
```

`L1Scores` *L1 Ranking*

Description

Calculate the $L^{(1)}$ scores.

Usage

```
L1Scores(powerRelation, elements = powerRelation$elements)

L1Ranking(powerRelation)

lexcel1Scores(powerRelation, elements = powerRelation$elements)

lexcel1Ranking(powerRelation)
```

Arguments

`powerRelation`	A `PowerRelation` object created by `PowerRelation()` or `as.PowerRelation()`
`elements`	Vector of elements of which to calculate their scores. By default, the scores of all elements in `powerRelation$elements` are considered.

Details

Similar to `lexcelRanking()`, the number of times an element appears in each equivalence class is counted. In addition, we now also consider the size of the coalitions.

Let N be a set of elements, $\succeq \in \mathcal{T}(\mathcal{P})$ be a power relation, and $\Sigma_1 \succ \Sigma_2 \succ \cdots \succ \Sigma_m$ its corresponding quotient order.

For an element $i \in N$, we get a matrix $M_i^{\succeq}$ with m columns and $|N|$ rows. Whereas each column represents an equivalence class, each row corresponds to the coalition size.

$$(M_i^{\succeq})_{pq} = |\{S \in \Sigma_q : |S| = p\}|$$

Take as an example $\succeq$: $(123 \sim 13 \sim 2) \succ (12 \sim 1 \sim 3) \succ (23 \sim \{\})$. From this, we get the following three matrices:

$$M_1^{\succeq} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix} M_2^{\succeq} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 0 \end{bmatrix} M_3^{\succeq} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}$$

The $L^{(1)}$ then ranks the elements, rewarding elements that appear in higher ranking coalitions as well as smaller coalitions. When comparing two matrices for a power relation, if $M_i^{\succeq} >_{L^{(1)}} M_j^{\succeq}$, this suggests that there exists a $p^0 \in \{1, \ldots, |N|\}$ and $q^0 \in \{1, \ldots, m\}$ such that the following holds:

1. $(M_i^{\succeq})_{p^0q^0} > (M_j^{\succeq})_{p^0q^0}$
2. $(M_i^{\succeq})_{pq^0} = (M_j^{\succeq})_{pq^0}$ for all $p < p^0$
3. $(M_i^{\succeq})_{pq} = (M_j^{\succeq})_{pq}$ for all $q < q^0$ and $p \in \{1, \ldots, |N|\}$

In the aforementioned example, we have that $M_2^{\succeq} >_{L^{(1)}} M_1^{\succeq} >_{L^{(1)}} M_3^{\succeq}$, suggesting that element 2 should be ranked highest and 3 lowest.

Value

Score function returns a list of type `Lexcel1Scores` and length of `powerRelation$elements` (unless parameter `elements` is specified). Each index contains a vector of length `powerRelation$eqs`, the number of times the given element appears in each equivalence class.

Ranking function returns corresponding `SocialRanking` object.

Aliases

For better discoverability, `lexcel1Scores()` and `lexcel1Ranking()` serve as aliases for `L1Scores()` and `L1Ranking()`, respectively.

References

Algaba E, Moretti S, Rémila E, Solal P (2021). "Lexicographic solutions for coalitional rankings." *Social Choice and Welfare*, **57**(4), 1–33.

See Also

Other ranking solution functions: `copelandScores()`, `cumulativeScores()`, `kramerSimpsonScores()`, `lexcelScores()`, `ordinalBanzhafScores()`

Examples

```
pr <- as.PowerRelation("(123 ~ 13 ~ 2) > (12 ~ 1 ~ 3) > (23 ~ {})")
scores <- L1Scores(pr)
scores$`1`
#      [,1] [,2] [,3]
# [1,]    0    1    0
# [2,]    1    1    0
# [3,]    1    0    0

L1Ranking(pr)
# 2 > 1 > 3
```

`lexcelScores` *Lexicographical Excellence*

Description

Calculate the Lexicographical Excellence (or Lexcel) score.

`lexcelRanking()` returns the corresponding ranking.

`dualLexcelRanking()` uses the same score vectors but instead of rewarding participation, it punishes mediocrity.

Usage

```
lexcelScores(powerRelation, elements = powerRelation$elements)

lexcelRanking(powerRelation)

dualLexcelRanking(powerRelation)
```

Arguments

`powerRelation`	A `PowerRelation` object created by `PowerRelation()` or `as.PowerRelation()`
`elements`	Vector of elements of which to calculate their scores. By default, the scores of all elements in `powerRelation$elements` are considered.

Details

An equivalence class $\sum_i$ contains coalitions that are indifferent to one another. In a given power relation created with `PowerRelation()` or `as.PowerRelation()`, the equivalence classes are saved in `$eqs`.

As an example, consider the power relation $\succeq$: $123 \succ (12 \sim 13 \sim 1 \sim \emptyset) \succ (23 \sim 1 \sim 2)$. The corresponding equivalence classes are:

$$\sum_1 = \{123\}, \sum_2 = \{12, 13, 1, \emptyset\}, \sum_3 = \{23, 1, 2\}.$$

The lexcel score of an element is a vector wherein each index indicates the number of times that element appears in the equivalence class. From our example, we would get

$$\text{lexcel}(1) = [1, 3, 1], \text{lexcel}(2) = [1, 1, 2], \text{lexcel}(3) = [1, 1, 1].$$

Value

Score function returns a list of type `LexcelScores` and length of `powerRelation$elements` (unless parameter `elements` is specified). Each index contains a vector of length `powerRelation$eqs`, the number of times the given element appears in each equivalence class.

Ranking function returns corresponding `SocialRanking` object.

Lexcel Ranking

The most "excellent contribution" of an element determines its ranking against the other elements. Given two Lexcel score vectors $\text{Score}(i)$ and $\text{Score}(j)$, the first index x where $\text{Score}(i)_x \neq \text{Score}(j)_x$ determines which element should be ranked higher.

From the previous example this would be $1 > 2 > 3$, because:

$\text{Score}(1)_2 = 3 > \text{Score}(2)_2 = \text{Score}(3)_2 = 1$, $\text{Score}(2)_3 = 2 > \text{Score}(3)_3 = 1$.

Dual Lexcel Ranking

The dual lexcel works in reverse order and, instead of rewarding high scores, punishes mediocrity. In that case we get $3 > 1 > 2$ because:

$\text{Score}(3)_3 < \text{Score}(2)_3$ and $\text{Score}(3)_2 < \text{Score}(1)_2$, $\text{Score}(1)_3 < \text{Score}(2)_3$.

References

Bernardi G, Lucchetti R, Moretti S (2019). "Ranking objects from a preference relation over their subsets." *Social Choice and Welfare*, **52**(4), 589–606.

Algaba E, Moretti S, Rémila E, Solal P (2021). "Lexicographic solutions for coalitional rankings." *Social Choice and Welfare*, **57**(4), 1–33.

Serramia M, López-Sánchez M, Moretti S, Rodríguez-Aguilar JA (2021). "On the dominant set selection problem and its application to value alignment." *Autonomous Agents and Multi-Agent Systems*, **35**(2), 1–38.

See Also

Other ranking solution functions: `L1Scores()`, `copelandScores()`, `cumulativeScores()`, `kramerSimpsonScores()`, `ordinalBanzhafScores()`

Examples

```
# note that the coalition {1} appears twice
# 123 > 12 ~ 13 ~ 1 ~ {} > 23 ~ 1 ~ 2
# E = {123} > {12, 13, 1, {}} > {23, 1, 2}
pr <- suppressWarnings(as.PowerRelation(
  "123 > (12 ~ 13 ~ 1 ~ {}) > (23 ~ 1 ~ 2)"
))

# lexcel scores for all elements
# `1` = c(1, 3, 1)
# `2` = c(1, 1, 2)
# `3` = c(1, 1, 1)
lexcelScores(pr)

# lexcel scores for a subset of all elements
lexcelScores(pr, c(1, 3))
lexcelScores(pr, 2)

# 1 > 2 > 3
lexcelRanking(pr)
```

```
# 3 > 1 > 2
dualLexcelRanking(pr)
```

`makePowerRelationMonotonic` *Make Power Relation monotonic*

Description

Given a `powerRelation` object, make its order monotonic.

Usage

```
makePowerRelationMonotonic(powerRelation, addMissingCoalitions = TRUE)
```

Arguments

`powerRelation`	A `PowerRelation` object created by `PowerRelation()` or `as.PowerRelation()`
`addMissingCoalitions`	If `TRUE`, also include all coalitions in the power set of `powerRelation$elements` that are not present in the current power relation.

Details

A power relation is monotonic if

$$T \subset S \Leftrightarrow S \succeq T.$$

for every coalition $S \subseteq N$.

Calling `makePowerRelationMonotonic()` on some `PowerRelation` object moves or adds coalitions to certain equivalence classes so that the power relation becomes monotonic.

Value

`PowerRelation` object containing the following values:

- `$elements`: vector of elements
- `$eqs`: equivalence classes. Nested list of lists, each containing vectors representing groups of elements in the same equivalence class
- `$coalitionLookup`: `function(v)` taking a coalition vector `v` and returning the equivalence class it belongs to. See `coalitionLookup()` for more.
- `$elementLookup`: `function(e)` taking an element `e` and returning a list of 2-sized tuples. See `elementLookup()` for more.

See Also

Other helper functions for transforming power relations: `appendMissingCoalitions()`

Examples

```
pr <- as.PowerRelation("ab > ac > abc > b > a > {} > c > bc")
makePowerRelationMonotonic(pr)
# (abc ~ ab) > ac > (bc ~ b) > a > (c ~ {})

# notice that missing coalitions are automatically added,
# except for the empty set
pr <- as.PowerRelation("a > b > c")
makePowerRelationMonotonic(pr)
# (abc ~ ab ~ ac ~ a) > (bc ~ b) > c

# setting addMissingCoalitions to FALSE changes this behavior
pr <- as.PowerRelation("a > ab > c ~ {} > b")
makePowerRelationMonotonic(pr, addMissingCoalitions = FALSE)
# (ab ~ a) > (b ~ c ~ {})

# notice that an equivalence class containing an empty coalition
# automatically moves all remaining coalitions to that equivalence class.
pr <- as.PowerRelation("a > {} > b > c")
makePowerRelationMonotonic(pr)
# (abc ~ ab ~ ac ~ a) > (bc ~ b ~ c ~ {})
```

`newPowerRelation` *New Power Relation*

Description

Deprecated. Use `PowerRelation()` instead.

Usage

```
newPowerRelation(...)
```

Arguments

`...`	Any parameter.

Value

No return value.

newPowerRelationFromString *New* PowerRelation *object*

Description

Deprecated. Use as.PowerRelation() instead.

Usage

```
newPowerRelationFromString(...)
```

Arguments

...	Any parameter.

Value

No return value.

ordinalBanzhafScores *Ordinal Banzhaf ranking*

Description

Calculate the Ordinal Banzhaf scores, the number of positive and the number of negative marginal contributions.

ordinalBanzhafRanking() returns the corresponding ranking.

Usage

```
ordinalBanzhafScores(powerRelation, elements = powerRelation$elements)

ordinalBanzhafRanking(powerRelation)
```

Arguments

powerRelation	A PowerRelation object created by PowerRelation() or as.PowerRelation()
elements	Vector of elements of which to calculate their scores. By default, the scores of all elements in powerRelation$elements are considered.

Details

Inspired by the Banzhaf index (Banzhaf III 1964), the Ordinal Banzhaf determines the score of element i by adding the amount of coalitions $S \subseteq N \setminus \{i\}$ its contribution impacts positively ($S \cup \{i\} \succ S$) and subtracting the amount of coalitions where its contribution had a negative impact ($S \succ S \cup \{i\}$)(Khani et al. 2019).

The original definition only takes total power relations into account, where either $S \succeq T$ or $T \succeq S$ for every $S, T \subseteq N$. If coalitions are missing from the power relation, we may not be able to perform certain comparisons. To indicate these missing comparisons, the ordinal Banzhaf score of an element i also includes that number at index `3`. I.e., if the ordinal Banzhaf score of an element is `c(4, -2, 1)`, it means that it contributed positively to `4` coalitions and negatively to `2` others. For one coalition, no comparison could be made.

Value

Score function returns list of class type `OrdinalBanzhafScores` and length of `powerRelation$elements`. Each index contains a vector of three numbers, the number of positive marginal contributions, the number of negative marginal contributions, and the number of coalitions for which no comparison could be done. The first two numbers summed together gives us the actual ordinal Banzhaf score.

Ranking function returns corresponding `SocialRanking` object.

References

Khani H, Moretti S, Öztürk M (2019). "An ordinal banzhaf index for social ranking." In *28th International Joint Conference on Artificial Intelligence (IJCAI 2019)*, 378–384.

Banzhaf III JF (1964). "Weighted voting doesn't work: A mathematical analysis." *Rutgers L. Rev.*, **19**, 317.

See Also

Other ranking solution functions: `L1Scores()`, `copelandScores()`, `cumulativeScores()`, `kramerSimpsonScores()`, `lexcelScores()`

Examples

```
pr <- as.PowerRelation("12 > (2 ~ {}) > 1")

# Player 1 contributes positively to {2}
# Player 1 contributes negatively to {empty set}
# Therefore player 1 has a score of 1 - 1 = 0
#
# Player 2 contributes positively to {1}
# Player 2 does NOT have an impact on {empty set}
# Therefore player 2 has a score of 1 - 0 = 0
ordinalBanzhafScores(pr)
# `1` = c(1, -1, 0)
# `2` = c(1, 0, 0)

ordinalBanzhafRanking(pr)
# 1 > 2
```

PowerRelation *PowerRelation object*

Description

Create a PowerRelation object.

Usage

```
PowerRelation(
  equivalenceClasses,
  elements = NULL,
  coalitionLookup = NULL,
  elementLookup = NULL
)

is.PowerRelation(x, ...)

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

Arguments

equivalenceClasses	A nested list of lists, each containing coalitions or groups represented as vectors that are in the same equivalence class.
elements	Vector of elements in power relation. Only set this value if you know what you are doing. See Details for more.
coalitionLookup	A function taking a vector parameter and returning an index. See return value for more details. Only set this value if you know what you are doing.
elementLookup	A function taking an element and returning a list of 2-sized tuples. See return value for more details. Only set this value if you know what you are doing.
x	An R object.
...	Additional arguments to be passed to or from methods.

Details

A power relation describes the ordinal information between elements. Here specifically, we are interested in the power relation between coalitions, or groups of elements. Each coalition is assumed to be a vector containing zero (empty coalition), one (singleton) or more elements.

createPowerset() offers a convenient way of creating a power set over a set of elements that can be used to call PowerRelation() or as.PowerRelation().

Trying to figure out what equivalence class certain coalitions or elements belong to is quite common. For these sets of problems, the functions `$coalitionLookup(v)` and `$elementLookup(e)` should be utilized. We use some redundancy to speed up the lookup methods. As such, it is highly discouraged to edit a `PowerRelation` object directly, as the different power relation representations will fall out of sync. For more information, see the vignette: `vignette(package = 'socialranking')`

The `PowerRelation()` function expects a nested list of coalitions as input. For alternatives, see `as.PowerRelation()`.

Value

`PowerRelation` object containing the following values:

- `$elements`: vector of elements
- `$eqs`: equivalence classes. Nested list of lists, each containing vectors representing groups of elements in the same equivalence class
- `$coalitionLookup`: `function(v)` taking a coalition vector `v` and returning the equivalence class it belongs to. See `coalitionLookup()` for more.
- `$elementLookup`: `function(e)` taking an element `e` and returning a list of 2-sized tuples. See `elementLookup()` for more.

Mathematical background

Let $N = \{1, ..., n\}$ be a finite set of *elements* (also called players). Any subset $S \subseteq N$ is considered to be a group or coalition of elements, where $\{\}$ is referred to as the empty coalition, $\{i\}$ as a singleton (a coalition of size 1), and N as the grand coalition. The power set 2^N denotes the set of all subsets over N.

Let $\mathcal{P} \subseteq 2^N$ be a collection of coalitions. A *power relation* on $\mathcal{P}$ is a total preorder $\succeq \subseteq \mathcal{P} \times \mathcal{P}$. That is, for any two coalitions $S, T \in \mathcal{P}$, either $(S, T) \in \succeq$, or $(T, S) \in \succeq$, or both. In other words, we can compare any two groups of elements in $\mathcal{P}$ and determine, if one group is better than, worse than, or equivalent to the other.

More commonly, the relation $(S, T) \in \succeq$ is notated as $S \succeq T$.

$\mathcal{T}(\mathcal{P})$ denotes the family of all power relations on every collection $\mathcal{P} \subseteq 2^N$. Given a power relation $\succeq \in \mathcal{T}(\mathcal{P})$, $\sim$ denotes its symmetric part whereas $\succ$ its asymmetric part. Let $S, T \in \mathcal{P}$. Then,

$$S \sim T \text{ if } S \succeq T \text{ and } T \succeq S, S \succ T \text{ if } S \succeq T \text{ and not } T \succeq S.$$

Coalitions which are deemed equivalent ($S \sim T$) can be collected into an equivalence class Σ_i. The list of equivalence classes forms a linear order, $\Sigma_1 \succ \Sigma_2 \succ \cdots \succ \Sigma_m$.

Mathematical example

As an example, consider the elements $N = \{\text{apple}, \text{banana}, \text{chocolate}\}$. Each of them individually may go well with pancakes, but we are also interested in the combination of condiments. If we consider all possibilities, we will have to compare the sets

$$\mathcal{P} = 2^N = \{\{a, b, c\}, \{a, b\}, \{a, c\}, \{b, c\}, \{a\}, \{b\}, \{c\}, \{\}\}.$$

Looking for a way to rank this group of objects, one may arrive at the following total preorder $\succeq \in \mathcal{T}(\mathcal{P})$:

$$\{b,c\} \succ (\{a\} \sim \{c\}) \succ \{b\} \succ \{\} \succ (\{a,b,c\} \sim \{a,b\} \sim \{a,c\}).$$

In this particular case, we get five equivalence classes.

$$\Sigma_1 = \{\{b,c\}\}\Sigma_2 = \{\{a\},\{c\}\}\Sigma_3 = \{\{b\}\}\Sigma_4 = \{\{\}\}\Sigma_5 = \{\{a,b,c\},\{a,b\},\{a,c\}\}$$

The power relation $\succeq$ can be copy-pasted as a character string to the `as.PowerRelation()` function (it should accept the special characters $\succeq$ and $\sim$).

```
as.PowerRelation("{b,c} > ({a} ~ {c}) > {b} > {} > ({a,b,c} ~ {a,b} ~ {a,c})")
```

References

Moretti S, Öztürk M (2017). "Some axiomatic and algorithmic perspectives on the social ranking problem." In *International Conference on Algorithmic Decision Theory*, 166–181. Springer.

Bernardi G, Lucchetti R, Moretti S (2019). "Ranking objects from a preference relation over their subsets." *Social Choice and Welfare*, **52**(4), 589–606.

Algaba E, Moretti S, Rémila E, Solal P (2021). "Lexicographic solutions for coalitional rankings." *Social Choice and Welfare*, **57**(4), 1–33.

See Also

Other ways to create a `PowerRelation()` object using `as.PowerRelation()`.

Examples

```
pr <- PowerRelation(list(
  list(c(1,2,3)),
  list(c(1, 2), 2, 3),
  list(c(2, 3), c()),
  list(c(1, 3)),
  list(1)
))

pr
# 123 > (12 ~ 2 ~ 3) > (23 ~ {}) > 13 > 1

stopifnot(pr$elements == 1:3)
stopifnot(pr$coalitionLookup(1) == 5)
stopifnot(pr$coalitionLookup(c()) == 3)
stopifnot(pr$coalitionLookup(c(1,2)) == 2)

# find coalitions an element appears in
for(t in pr$elementLookup(2)) {
  stopifnot(2 %in% pr$eqs[[t[1]]][[t[2]]])
}

# use createPowerset to help generate a valid function call
```

```
if(interactive())
  createPowerset(letters[1:3], result = "copy")

# pasted, rearranged using alt+up / alt+down in RStudio

# note that the function call looks different if elements are multiple characters long
if(interactive())
  createPowerset(c("apple", "banana", "chocolate"), result = "copy")

# pasted clipboard
PowerRelation(rlang::list2(
  list(c("banana", "chocolate")),
  list(c("apple"),
       c("chocolate")),
  list(c("banana")),
  list(c()),
  list(c("apple", "banana", "chocolate"),
       c("apple", "banana"),
       c("apple", "chocolate")),
))
# {banana, chocolate} > ({apple} ~ {chocolate}) > {banana} > {} > ...
```

`powerRelationGenerator` *Generate power relations*

Description

Based on a list of coalitions, create a generator function that returns a new `PowerRelation` object with every call. `NULL` is returned once every possible power relation has been generated.

Usage

```
powerRelationGenerator(coalitions, startWithLinearOrder = FALSE)
```

Arguments

`coalitions`	List of coalition vectors. An empty coalition can be set with `c()`.
`startWithLinearOrder`	If set to `TRUE`, the first `PowerRelation` object generated will be a linear order in the order of the list of `coalitions` they are given. If set to `FALSE`, the first `PowerRelation` object generated will have a single equivalence class containing all coalitions, as in, every coalition is equally powerful.

Details

Using the `partitions` library, `partitions::compositions()` is used to create all possible partitions over the set of coalitions. For every partition, `partitions::multinomial()` is used to create all permutations over the order of the coalitions.

Note that the number of power relations (or total preorders) grows incredibly fast.

The Stirling number of second kind $S(n, k)$ gives us the number of k partitions over n elements.

$$S(n,k) = \frac{1}{k!}\sum_{j=0}^{k}(-1)^j\binom{k}{j}(k-j)^n$$

For example, with 4 coalitions (n = 4) there are 6 ways to split it into k = 3 partitions. The sum of all partitions of any size is also known as the Bell number ($B_n = \sum_{k=0}^{n} S(n,k)$, see also `numbers::bell()`).

Regarding total preorders $\mathcal{T}(X)$ over a set X, the Stirling number of second kind can be used to determine the number of all possible total preorders $|\mathcal{T}(X)|$.

$$|\mathcal{T}(X)| = \sum_{k=0}^{|X|} k! * S(|X|, k)$$

In literature, it is referred to as the ordered Bell number or Fubini number.

In the context of social rankings we may consider total preorders over the set of coalitions 2^N for a given set of elements or players N. Here, the number of coalitions doubles with every new element. The number of preorders then are:

# of elements	# of coalitions	# of total preorders	1ms / computation
0	1	1	1ms
1	2	3	3ms
2	4	75	75ms
3	7 (w/o empty set)	47,293	47 seconds
3	8	545,835	9 minutes
4	15 (w/o empty set)	230,283,190,977,853	7,302 years
4	16	5,315,654,681,981,355	168,558 years

Value

A generator function. Every time this generator function is called, a different `PowerRelation` object is returned. Once all possible power relations have been generated, the generator function returns `NULL`.

See Also

Other generator functions: `generateNextPartition()`

Examples

```
coalitions <- createPowerset(c('a','b'), includeEmptySet = FALSE)
# list(c('a','b'), 'a', 'b')

gen <- powerRelationGenerator(coalitions)

while(!is.null(pr <- gen())) {
  print(pr)
}
# (ab ~ a ~ b)
# (ab ~ a) > b
# (ab ~ b) > a
# (a ~ b) > ab
# ab > (a ~ b)
# a > (ab ~ b)
# b > (ab ~ a)
# ab > a > b
# ab > b > a
# a > ab > b
# b > ab > a
# a > b > ab
# b > a > ab

# from now on, gen() always returns NULL
gen()
# NULL

# Use generateNextPartition() to skip certain partitions
gen <- powerRelationGenerator(coalitions)

gen <- generateNextPartition(gen)
gen <- generateNextPartition(gen)
gen()
```

`powerRelationMatrix` *Create relation matrix*

Description

For a given `PowerRelation` object create a `relations::relation()` object.

Usage

```
powerRelationMatrix(
  powerRelation,
  domainNames = c("pretty", "numericPrec", "numeric")
)
```

```
## S3 method for class 'PowerRelation'
as.relation(x, ...)
```

Arguments

powerRelation	A PowerRelation object created by PowerRelation() or as.PowerRelation()
domainNames	How should the row and column names be formatted? • pretty: Coalitions such as c(1,2) are formatted as 12. To ensure that it's correctly sorted alphabetically, every name is preceded by a certain amount of the invisible Unicode character \u200b • numericPrec: Coalitions such as c(1,2) are formatted as 1{12}, the number in front of the curly brace marking its sorted spot. While less pretty, it won't use Unicode characters. • numeric: Drop coalition names, only count from 1 upwards. Each number corresponds to the index in TODO powerRelation$rankingCoalitions • function(x): A custom function that is passed a number from 1 through length(powerRelation$rankingCoalitions). Must return a character object.
x	A PowerRelation object
...	Further parameters (ignored)

Details

Turn a PowerRelation object into a relations::relation() object. The incidence matrix can be viewed with relations::relation_incidence().

The columns and rows of a PowerRelation object are ordered by TODO powerRelation$rankingCoalitions. The relations package automatically sorts the columns and rows by their domain names, which is the reason the parameter domainNames is included. This way we ensure that the columns and rows are sorted by the order of the power relation.

Value

relations::relation() object to the corresponding power relation.

Cycles

A PowerRelation object is defined as being transitive. If a power relation includes a cycle, meaning that the same coalition appears twice in the ranking, all coalitions within that cycle will be considered to be indifferent from one another.

For example, given the power relation $1 \succ 2 \succ 3 \succ 1 \succ 12$, the relation is somewhat equivalent to $1 \sim 2 \sim 3 \succ 12$. There is no way to check for cycles in the incidence matrix only.

Call transitiveClosure() to remove cycles in a PowerRelation object.

See Also

relations::as.relation()

Examples

```
pr <- as.PowerRelation("12 > 1 > 2")
relation <- powerRelationMatrix(pr)

# do relation stuff
# Incidence matrix
# 111
# 011
# 001
relations::relation_incidence(relation)

# all TRUE
stopifnot(all(
  relations::relation_is_acyclic(relation),
  relations::relation_is_antisymmetric(relation),
  relations::relation_is_linear_order(relation),
  relations::relation_is_complete(relation),
  relations::relation_is_reflexive(relation),
  relations::relation_is_transitive(relation)
))


# a power relation where coalitions {1} and {2} are indifferent
pr <- as.PowerRelation("12 > (1 ~ 2)")
relation <- powerRelationMatrix(pr)

# Incidence matrix
# 111
# 011
# 011
relations::relation_incidence(relation)

# FALSE
stopifnot(!any(
  relations::relation_is_acyclic(relation),
  relations::relation_is_antisymmetric(relation),
  relations::relation_is_linear_order(relation)
))
# TRUE
stopifnot(all(
  relations::relation_is_complete(relation),
  relations::relation_is_reflexive(relation),
  relations::relation_is_transitive(relation)
))


# a pr with cycles
pr <- suppressWarnings(as.PowerRelation("12 > 1 > 2 > 1"))
relation <- powerRelationMatrix(pr)

# Incidence matrix
# 1111
```

```
# 0111
# 0111
# 0111
relations::relation_incidence(relation)

# custom naming convention
relation <- powerRelationMatrix(
  pr,
  function(x) paste0(letters[x], ":", paste(pr$rankingCoalitions[[x]], collapse = "|"))
)

relations::relation_incidence(relation)
# Incidences:
#       a:1|2 b:1 c:2 d:1
# a:1|2     1   1   1   1
# b:1       0   1   1   1
# c:2       0   1   1   1
# d:1       0   1   1   1
```

`SocialRanking` `SocialRanking` *object*

Description

Create a `SocialRanking` object.

Usage

```
SocialRanking(l)
```

Arguments

l	A list of vectors

Details

Similar to `PowerRelation()`, `SocialRanking` expects expects a list to represent a power relation. Unlike `PowerRelation()` however, this list should not be nested and should only contain vectors, each vector containing elements that are deemed equally preferable.

Use `doRanking()` to rank elements based on arbitrary score objects.

A social ranking solution, or ranking solution, or solution, maps each power relation between coalitions to a power relation between its elements. I.e., from the power relation $\succeq$: $\{1,2\} \succ \{2\} \succ \{1\}$, we may expect the result of a ranking solution $R^{\succeq}$ to rank element 2 over 1. Therefore $2R^{\succeq}1$ will be present, but not $1R^{\succeq}2$.

Formally, a ranking solution $R : \mathcal{T}(\mathcal{P}) \to \mathcal{T}(N)$ is a function that, given a power relation $\succeq \in \mathcal{T}(\mathcal{P})$, always produces a power relation $R(\succeq)$ (or $R^{\succeq}$) over its set of elements. For two elements $i, j \in N$, $iR^{\succeq}j$ means that applying the solution R on the ranking $\succeq$ makes i at least as preferable as j. Often times $iI^{\succeq}j$ and $iP^{\succeq}j$ are used to indicate its symmetric and asymmetric part, respectively. As in, $iI^{\succeq}j$ implies that $iR^{\succeq}j$ and $jR^{\succeq}i$, whereas $iP^{\succeq}j$ implies that $iR^{\succeq}j$ but not $jR^{\succeq}i$.

Value

A list of type `SocialRanking`. Each element of the list contains a vector of elements in `powerRelation$elements` that are indifferent to one another.

See Also

Function that ranks elements based on their scores, `doRanking()`

Examples

```
SocialRanking(list(c("a", "b"), "f", c("c", "d")))
# a ~ b > f > c ~ d
```

`socialrankingpackage` *socialranking: A package for constructing ordinal power relations and evaluating social ranking solutions*

Description

The package `socialranking` offers functions to represent ordinal information of coalitions and calculate the power relation between elements or players.

Details

`PowerRelation()` creates a `PowerRelation` object. `createPowerset()` is a convenient function to generate a `PowerRelation()` or `as.PowerRelation()` function call for all possible coalitions.

The functions used to analyze power relations can be grouped into comparison functions, score functions and ranking solutions. Ranking solutions produce a `SocialRanking` object.

Comparison Functions	Score Functions	Ranking Solutions
`dominates()`		
`cumulativelyDominates()`	`cumulativeScores()`	
`cpMajorityComparison()`^1	`copelandScores()`	`copelandRanking()`
	`kramerSimpsonScores()`	`kramerSimpsonRanking()`
	`lexcelScores()`	`lexcelRanking()`
		`dualLexcelRanking()`
	`L1Scores()`	`L1Ranking()`
	`ordinalBanzhafScores()`	`ordinalBanzhafRanking()`

^1 `cpMajorityComparisonScore()` is a faster alternative to `cpMajorityComparison()`, but it produces less data.

`powerRelationMatrix()` uses `relations::relation()` to create an incidence matrix between all competing coalitions. The incidence matrix can be displayed with `relations::relation_incidence()`.

Use `browseVignettes("socialranking")` for more information.

testRelation *Test relation between two elements*

Description

On a given PowerRelation object pr, check if e1 relates to e2 based on the given social ranking solution.

Usage

```
testRelation(powerRelation, e1)

powerRelation %:% e1

pr_e1 %>=dom% e2

pr_e1 %>dom% e2

pr_e1 %>=cumuldom% e2

pr_e1 %>cumuldom% e2

pr_e1 %>=cp% e2

pr_e1 %>cp% e2

pr_e1 %>=banz% e2

pr_e1 %>banz% e2

pr_e1 %>=cop% e2

pr_e1 %>cop% e2

pr_e1 %>=ks% e2

pr_e1 %>ks% e2

pr_e1 %>=lex% e2

pr_e1 %>lex% e2

pr_e1 %>=duallex% e2

pr_e1 %>duallex% e2
```

Arguments

powerRelation	A PowerRelation object created by PowerRelation() or as.PowerRelation()
e1, e2	Elements in powerRelation$elements
pr_e1	PowerRelation and e1 element, packed into a list using pr %:% e1

Details

The function testRelation is somewhat only used to make the offered comparison operators in the package better discoverable.

testRelation(pr, e1) is equivalent to pr %:% e1 and list(pr, e1). It should be used together with one of the comparison operators listed in the usage section.

Value

testRelation() and %:% returns list(powerRelation, e1).

Followed by a %>=comparison% or %>comparison% it returns TRUE or FALSE, depending on the relation between e1 and e2.

See Also

Comparison function: dominates(), cumulativelyDominates(), cpMajorityComparison().

Score Functions: ordinalBanzhafScores(), copelandScores(), kramerSimpsonScores(), lexcelScores().

Examples

```
pr <- as.PowerRelation("123 > 12 ~ 13 > 3 > 1 ~ 2")

# Dominance
stopifnot(pr %:% 1 %>=dom% 2)

# Strict dominance
stopifnot((pr %:% 1 %>dom% 2) == FALSE)

# Cumulative dominance
stopifnot(pr %:% 1 %>=cumuldom% 2)

# Strict cumulative dominance
stopifnot(pr %:% 1 %>cumuldom% 2)

# CP-Majority relation
stopifnot(pr %:% 1 %>=cp% 2)

# Strict CP-Majority relation
stopifnot((pr %:% 1 %>cp% 2) == FALSE)

# Ordinal banzhaf relation
stopifnot(pr %:% 1 %>=banz% 2)

# Strict ordinal banzhaf relation
```

```
# (meaning 1 had a strictly higher positive contribution than 2)
stopifnot((pr %:% 1 %>banz% 2) == FALSE)

# Copeland-like method
stopifnot(pr %:% 1 %>=cop% 2)
stopifnot(pr %:% 2 %>=cop% 1)

# Strict Copeland-like method
# (meaning pairwise winning minus pairwise losing comparison of
# 1 is strictly higher than of 2)
stopifnot((pr %:% 1 %>cop% 2) == FALSE)
stopifnot((pr %:% 2 %>cop% 1) == FALSE)
stopifnot(pr %:% 3 %>cop% 1)

# Kramer-Simpson-like method
stopifnot(pr %:% 1 %>=ks% 2)
stopifnot(pr %:% 2 %>=ks% 1)

# Strict Kramer-Simpson-like method
# (meaning ks-score of 1 is actually higher than 2)
stopifnot((pr %:% 2 %>ks% 1) == FALSE)
stopifnot((pr %:% 1 %>ks% 2) == FALSE)
stopifnot(pr %:% 3 %>ks% 1)

# Lexicographical and dual lexicographical excellence
stopifnot(pr %:% 1 %>=lex% 3)
stopifnot(pr %:% 3 %>=duallex% 1)

# Strict lexicographical and dual lexicographical excellence
# (meaning their lexicographical scores don't match)
stopifnot(pr %:% 1 %>lex% 3)
stopifnot(pr %:% 3 %>duallex% 1)
```

`transitiveClosure` *Transitive Closure*

Description

Apply transitive closure over power relation that has cycles.

Usage

```
transitiveClosure(powerRelation)
```

Arguments

`powerRelation`	A `PowerRelation` object created by `PowerRelation()` or `as.PowerRelation()`

Details

A power relation is a binary relationship between coalitions that is transitive. For coalitions $a, b, c \in 2^N$, this means that if $a \succ b$ and $b \succ c$, then $a \succ c$.

A power relation with cycles is not transitive. A transitive closure over a power relation removes all cycles and turns it into a transitive relation, placing all coalitions within a cycle in the same equivalence class. If $a \succ b \succ a$, from the symmetric definition in `PowerRelation()` we therefore assume that $a \sim b$. Similarly, if $a \succ b_1 \succ b_2 \succ \cdots \succ b_n \succ a$, the transitive closure turns it into $a \sim b_1 \sim b_2 \sim \cdots \sim b_n$.

`transitiveClosure()` transforms a `PowerRelation` object with cycles into a `PowerRelation` object without cycles. As described above, all coalitions within a cycle then are put into the same equivalence class and all duplicate coalitions are removed.

Value

`PowerRelation` object with no cycles.

Examples

```
pr <- as.PowerRelation("1 > 2")

# nothing changes
transitiveClosure(pr)


pr <- suppressWarnings(as.PowerRelation("1 > 2 > 1"))

# 1 ~ 2
transitiveClosure(pr)


pr <- suppressWarnings(
  as.PowerRelation("1 > 3 > 1 > 2 > 23 > 2")
)

# 1 > 3 > 1 > 2 > 23 > 2 =>
# 1 ~ 3 > 2 ~ 23
transitiveClosure(pr)
```

Index