Package 'socialranking'

October 27, 2022

Title Social Ranking Solutions for Power Relations on Coalitions

Version 0.1.2

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

Roxygen list(markdown = TRUE)

RoxygenNote 7.2.1

RdMacros mathjaxr,
Rdpack

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

Imports relations (>= 0.6.12),
sets (>= 1.0.21),
rlang (>= 1.0.6),
mathjaxr (>= 1.6.0),
Rdpack (>= 2.4)

Config/testthat/edition 3

VignetteBuilder knitr

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

R topics documented:

`coalitionsAreIndifferent` *Are coalitions indifferent*

Description

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

Usage

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

Arguments

`powerRelation`	A `PowerRelation` object created by `newPowerRelation()`
`c1`	Coalition vector or `sets::set()`
`c2`	Coalition vector or `sets::set()`

Details

`equivalenceClassIndex()` is called to determine, which equivalence class `c1` and `c2` belong to. It returns `TRUE` if both are in the same equivalence class.

If either coalition `c1` or `c2` is not part of the power relation, an error is thrown.

Value

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

See Also

Other equivalence class lookup functions: equivalenceClassIndex()

Examples

```
pr <- newPowerRelation(c(1,2), ">", c(1), "~", c(2))

# FALSE
coalitionsAreIndifferent(pr, c(1,2), c(1))

# TRUE
coalitionsAreIndifferent(pr, 2, 1)

# Error: The coalition {} does not appear in the power relation
tryCatch(
  equivalenceClassIndex(pr, c()),
  error = function(e) { e }
)
```

---

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 = NULL)

copelandRanking(powerRelation)
```

Arguments

powerRelation	A PowerRelation object created by newPowerRelation()
elements	vector of elements of which to calculate their scores. If elements == NULL, create vectors for all elements in pr$elements

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 `SocialRankingSolution` 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 score vector functions: `cumulativeScores()`, `kramerSimpsonScores()`, `lexcelScores()`, `ordinalBanzhafScores()`

Examples

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

# `1` = 1
# `2` = 0
# `3` = -1
copelandScores(pr)

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

# or just one element
copelandScores(pr, 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 `newPowerRelation()`
`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 `sets::sets` $S \in D_{ij}(\succeq)$. $S \in D_{ij}(\succ)$ 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 `sets::sets` $S \in D_{ji}(\succeq)$. $S \in D_{ji}(\succ)$ 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`: `sets::set`, 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 <- newPowerRelationFromString("ac > (a ~ b) > (c ~ bc)")

# a > b
# D_ab = {c, {}}
# D_ba = {{}}
# Score of a = 2
# Score of b = 1
```

```
scores <- cpMajorityComparison(pr, "a", "b")
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) x$coalition == sets::set(), scores$tuples)[[1]]
playerCTuple  <- Filter(function(x) x$coalition == sets::set("c"), scores$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 `newPowerRelation()`-valid function call with all possible coalitions.

Usage

```
createPowerset(
  elements,
  copyToClipboard = FALSE,
  writeLines = FALSE,
  includeEmptySet = TRUE
)
```

Arguments

`elements`	vector of elements
`copyToClipboard`	Copy code string to clipboard
`writeLines`	Write code string to console
`includeEmptySet`	If `TRUE`, an empty vector is added at the end

Value

List of power set vectors. If `copyToClipboard = TRUE`, it returns nothing and only copies a function-call string into the clipboard. If `writeLines = TRUE`, it returns nothing and prints a function-call string that is ready to be copy-pasted.

Examples

```
if(interactive()) {
  createPowerset(1:3, copyToClipboard = TRUE)
  createPowerset(c("a", "b", "c", "d"), writeLines = TRUE, includeEmptySet = FALSE)
}

# without copyToClipboard or writeLines set to TRUE, it returns a list
createPowerset(c("Alice", "Bob"), includeEmptySet = FALSE)
## [[1]]
## [1] "Alice" "Bob"
##
## [[2]]
## [1] "Alice"
##
## [[3]]
## [1] "Bob"
```

`cumulativeScores` *Cumulative scores*

Description

Calculate cumulative score vectors for each element.

Usage

```
cumulativeScores(powerRelation, elements = NULL)

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

Arguments

`powerRelation`	A `PowerRelation` object created by `newPowerRelation()`
`elements`	vector of elements of which to calculate their scores. If `elements == NULL`, create vectors for all elements in `pr$elements`
`e1, e2`	Elements in `powerRelation$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`. E.g. 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$elements` (unless parameter `elements` is specified). Each index contains a vector of length `powerRelation$equivalenceClasses` cumulatively counting up the number of times the given element appears in each equivalence class.

`cumulativelyDominates()` returns `TRUE` if `e1` cumulatively dominates `e2`.

Dominance

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

i *strictly* dominates j, if additionally $\text{Score}(i) \neq \text{Score}(j)$.

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 score vector functions: `copelandScores()`, `kramerSimpsonScores()`, `lexcelScores()`, `ordinalBanzhafScores()`

Examples

```
pr <- newPowerRelationFromString("12 > 1 > 2", asWhat = as.numeric)

# `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 *Domination*

Description

Test if one element dominates the other.

Usage

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

Arguments

powerRelation	A PowerRelation object created by newPowerRelation()
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 exists one $S \in 2^{N \setminus \{i,j\}}$ such that $S \cup \{i\} \succ S \cup \{j\}$.

Value

Logical value TRUE if e1 dominates e2, else FALSE.

Examples

```
pr <- newPowerRelationFromString("12 > 1 > 2", asWhat = as.numeric)

# 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* SocialRankingSolution

Description

Map a power relation between coalitions to a power relation between elements, also known as a social ranking solution.

Usage

```
doRanking(
  powerRelation,
  scores,
  isIndifferent = function(a, b) a == b,
  decreasing = TRUE
)
```

Arguments

powerRelation	A PowerRelation object created by newPowerRelation()
scores	A sortable vector or list of element scores
isIndifferent	A function that returns TRUE, if given two elements from scores the order doesn't matter. In that case the two elements are indifferent from each other, symbolized with the "~" operator.
decreasing	If TRUE (default), elements with the higher scores are ranked higher.

Value

A list of type SocialRankingSolution. Each element of the list contains a sets::set() of elements in powerRelation that are indifferent to one another.

Examples

```
pr <- newPowerRelationFromString("2 > 12 > 1", asWhat = as.numeric)

# we define our own social ranking solution.
# a player's score is determined by the equivalence class index it first appears in.
# lower is better
scores <- c(`1` = 2, `2` = 1)

# 2 > 1
doRanking(
  pr,
  scores,
  isIndifferent = function(x, y) x == y,
  decreasing = FALSE
)

# Suppose for a player to be ranked higher than the other,
# their positions have to be at least 2 apart.
# This means player 1 and 2 are indifferent,
# if they are right next to each other in the power relation.
```

```
# 2 ~ 1
doRanking(
  pr,
  scores,
  isIndifferent = function(x, y) abs(x - y) < 2,
  decreasing = FALSE
)
```

`equivalenceClassIndex` *Get index of equivalence class containing a coalition*

Description

Given a `coalition` vector or `sets::set()`, return a singular index number of the equivalence class it is located in.

Usage

```
equivalenceClassIndex(powerRelation, coalition, stopIfNotExists = TRUE)
```

Arguments

`powerRelation`	A `PowerRelation` object created by `newPowerRelation()`
`coalition`	a coalition vector or `sets::set` that is part of `powerRelation`
`stopIfNotExists`	`TRUE` if an error should be thrown when the coalition given is not in the `PowerRelation` object. If `FALSE`, `-1` will be returned

Value

Numeric value, equivalence class index where `coalition` appears in.

See Also

Other equivalence class lookup functions: `coalitionsAreIndifferent()`

Examples

```
pr <- newPowerRelation(c(1,2), ">", c(1), "~", c(2))

# 1
equivalenceClassIndex(pr, c(1, 2))

# 2
equivalenceClassIndex(pr, c(1))

# 2
equivalenceClassIndex(pr, c(2))

# Error: The coalition {} does not appear in the power relation
tryCatch(
  equivalenceClassIndex(pr, c()),
```

```
  error = function(e) { e }
)

# Error: This time only return a -1
stopifnot(-1 == equivalenceClassIndex(pr, c(), stopIfNotExists = FALSE))
```

`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 = NULL, compIvsI = FALSE)

kramerSimpsonRanking(powerRelation, compIvsI = FALSE)
```

Arguments

`powerRelation`	A `PowerRelation` object created by `newPowerRelation()`
`elements`	vector of elements of which to calculate their scores. If `elements` == `NULL`, create vectors for all elements in `pr$elements`
`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 `SocialRankingSolution` 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.), *Proceedings 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 score vector functions: `copelandScores()`, `cumulativeScores()`, `lexcelScores()`, `ordinalBanzhafScores()`

Examples

```
# 2 > (1 ~ 3) > 12 > (13 ~ 23) > {} > 123
pr <- newPowerRelation(
  2,
  ">", 1,
  "~", 3,
  ">", c(1,2),
  ">", c(1,3),
  "~", c(2,3),
  ">", c(),
  ">", c(1,2,3)
)

# 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)
```

`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 = NULL)

lexcelRanking(powerRelation)

dualLexcelRanking(powerRelation)
```

Arguments

`powerRelation`	A `PowerRelation` object created by `newPowerRelation()`
`elements`	vector of elements of which to calculate their scores. If `elements == NULL`, create vectors for all elements in `pr$elements`

Details

An equivalence class $\sum_i$ holds all coalitions that are indifferent from one another. In a given power relation created with `newPowerRelation()`, the equivalence classes are saved in `$equivalenceClasses`.

E.g. for a power relation defined as $123 \succ (12 \sim 13 \sim 1 \sim \emptyset) \succ (23 \sim 1 \sim 2)$ we would get the following equivalence classes:

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

A Lexcel score for an element is a vector where each index counts the number of times it appears in the equivalence class. Here 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$elements` (unless parameter `elements` is specified). Each index contains a vector of length `powerRelation$equivalenceClasses`, the number of times the given element appears in each equivalence class.

Ranking function returns corresponding `SocialRankingSolution` 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 score vector functions: `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(newPowerRelation(
  c(1,2,3),
  ">", c(1,2), "~", c(1,3), "~", 1, "~", c(),
  ">", c(2,3), "~", 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)
```

`newPowerRelation` *New Power Relation*

Description

Create a `PowerRelation` object based on coalition parameters separated by `">"` or `"~"`.

Usage

```
newPowerRelation(
  ...,
  rankingCoalitions = list(),
  rankingComparators = c(),
  equivalenceClasses = list()
)
```

Arguments

`...`	Coalition vector, comparison character (`">"` or `"~"`), coalition vector, comparison character, coalition vector, ...
`rankingCoalitions`	List of ordered coalition vectors. If empty, it is ignored. Corresponds to `$rankingCoalitions` list from a `PowerRelation` object.
`rankingComparators`	Vector of `">"` or `"~"` characters. If `rankingCoalitions` list is empty, it is ignored. If vector is empty, it uses the `">"` relation by default.
`equivalenceClasses`	Nested list of coalition vectors that are indifferent to another. If empty, it is ignored.

Details

A power relation describes the ordinal information between coalitions. `createPowerset()` offers a convenient way of creating a powerset over a set of elements that can be used to call the `newPowerRelation()` function. Each coalition in that case is put on a separate line (see example). In RStudio this allows us to easily rearrange the coalitions using the Alt+Up or Alt+Down shortcut (Option+Up or Option+Down on MacOS).

A coalition is a vector or a `sets::set()`. Every vector is turned into a `sets::set()`.

Value

`PowerRelation` object containing vector of elements or players `$elements`, an ordered list of coalitions `$rankingCoalitions` and an ordered vector of comparators `$rankingComparators` as well as an ordered list of equivalence classes `$equivalenceClasses` for convenience

Mathematical background

Let $N = \{1, ..., n\}$ be a finite set of *elements* (sometimes also called players). 2^N describes the powerset of N, or the set of all subsets, also *coalitions*.

Let $\mathcal{P} \subseteq 2^N$ be a collection of coalitions. A *power relation* on $\mathcal{P}$ is a total preorder $\succeq \subseteq \mathcal{P} \times \mathcal{P}$.

With that, $\mathcal{T}(\mathcal{P})$ 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. For example, let $S, T \in \mathcal{P}$. Then:

$$S \sim T \text{ if } S \succeq T \text{ and } T \succeq S$$

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

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.

See Also

Other newPowerRelation functions: `newPowerRelationFromString()`

Examples

```
if(interactive())
  createPowerset(1:3, copyToClipboard = TRUE)

# pasted clipboard and rearranged lines using
# Alt + Up, and
# Alt + Down shortcut in RStudio
pr <- newPowerRelation(
  c(1,2),
  ">", c(1,2,3),
  ">", c(1,3),
  "~", c(2),
  ">", c(1),
  "~", c(2,3),
  "~", c(3),
)

# Elements: 1 2 3
# 12 > 123 > (13 ~ 2) > (1 ~ 23 ~ 3)
print(pr)

# {1, 2, 3}
```

```
pr$elements

# {1, 2}, {1, 2, 3}, {1, 3}, {2}, {1}, {2, 3}, {3}
pr$rankingCoalitions

# ">" ">" "~" ">" ">" ">"
pr$rankingComparators

# {{1, 2}}, {{1, 2, 3}}, {{1, 3}, {2}}, {{1}, {2, 3}, {3}}
pr$equivalenceClasses

# not all coalitions of a powerset have to be present
newPowerRelation(c(1,2), ">", c(1))

# cycles produce a warning (but no errors)
newPowerRelation(c(1,2), ">", c(1), ">", c(1,2))

# use createPowerset directly
# 123 > 12 > 13 > 23 > 1 > 2 > 3 > {}
newPowerRelation(rankingCoalitions = createPowerset(1:3))

# 123 > (12 ~ 13) > (23 ~ 1) > (2 ~ 3) > {}
newPowerRelation(rankingCoalitions = createPowerset(1:3), rankingComparators = c(">", "~"))

# using equivalenceClasses parameter
# (12 ~ 13 ~ 123) > (1 ~ 3 ~ {}) > (2 ~ 23)
pr <- newPowerRelation(equivalenceClasses = list(
  list(c(1,2), c(1,3), c(1,2,3)),
  list(1, 3, c()),
  list(2, c(2,3))
))
# and manipulating the order of the equivalence classes
# (1 ~ 3 ~ {}) > (2 ~ 23) > (12 ~ 13 ~ 123)
newPowerRelation(equivalenceClasses = pr$equivalenceClasses[c(2,3,1)])


# It's discouraged to directly change the ordering of a power relation inside a
# PowerRelation object. Instead extract rankingCoalitions, rearrange the list
# and pass it to newPowerRelation
newOrdering <- rev(pr$rankingCoalitions)

# 3 > 23 > (1 ~ 2) > (13 ~ 123 ~ 12)
newPowerRelation(rankingCoalitions = newOrdering, rankingComparators = pr$rankingComparators)

# 3 > 23 > 1 > 2 > 13 > 123 > 12
newPowerRelation(rankingCoalitions = newOrdering)
```

`newPowerRelationFromString` *Create* `PowerRelation` *object from string*

Description

Given a pure string representation of a power relation, create a `PowerRelation` object.

Usage

```
newPowerRelationFromString(
  string,
  elementNames = "[0-9a-zA-Z]",
  asWhat = identity
)
```

Arguments

string	String representation of a power relation. Special characters such as $\succ$ and $\sim$ are replaced with their ASCII equivalents > and ~ respectively.
elementNames	Regular expression to match single characters in string input that should be interpreted as a name of an element. If character does not match, it is simply ignored.
asWhat	Elements are interpreted as string characters by default. base::as.numeric or base::as.integer can be passed to convert those string characters into numeric values.

Details

Elements in this power relation are assumed to be one character long. E.g., the coalitions "{1,2,3}" and 123 are equivalent, given that the elementNames parameter tells the function to only interpret the characters 1, 2 and 3 as valid element names.

Value

PowerRelation object containing vector of elements or players $elements, an ordered list of coalitions $rankingCoalitions and an ordered vector of comparators $rankingComparators as well as an ordered list of equivalence classes $equivalenceClasses for convenience

See Also

Other newPowerRelation functions: newPowerRelation()

Examples

```
# Elements: 1 2 3
# 123 > 12 > 23 > 1 > (13 ~ 2)
newPowerRelationFromString("123 > 12 > 23 > 1 > 13 ~ 2", asWhat = as.numeric)

# commas, braces and spaces are ignored by default
# notice that since an empty set is not a valid name of an element,
# it is simply ignored. Since there are no valid elements at the
# end, it is interpreted as an empty set.
newPowerRelationFromString("{1,2,3} > {1,3} > {1,2 } ~ \u2205", asWhat = as.numeric)

# use unvoncentional names
pr <- newPowerRelationFromString(".,; > .;~.,~,; > .~,~;", elementNames = "[.,;]")
stopifnot(pr$elements == sort(c(".", ",", ";")))
```

`ordinalBanzhafScores` *Ordinal Banzhaf*

Description

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

`ordinalBanzhafRanking()` returns the corresponding ranking.

Usage

```
ordinalBanzhafScores(powerRelation)

ordinalBanzhafRanking(powerRelation)
```

Arguments

`powerRelation`	A `PowerRelation` object created by `newPowerRelation()`

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).

Value

Score function returns list of class type `OrdinalBanzhafScores` and length of `powerRelation$elements`. Each index contains a vector of two numbers, the number of positive and the number of negative marginal contributions. Those two numbers summed together gives us the actual ordinal Banzhaf score.

Ranking function returns corresponding `SocialRankingSolution` 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 score vector functions: `copelandScores()`, `cumulativeScores()`, `kramerSimpsonScores()`, `lexcelScores()`

Examples

```
# 12 > (2 ~ {}) > 1
pr <- newPowerRelation(c(1,2), ">", 2, "~", c(), ">", 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
# `1` = c(1, -1)
# `2` = c(1, 0)
ordinalBanzhafScores(pr)

# 1 > 2
ordinalBanzhafRanking(pr)
```

`PowerRelation` *PowerRelation object*

Description

Use `newPowerRelation()` or `newPowerRelationFromString()` to create a PowerRelation object.

Usage

```
PowerRelation(x, ...)

is.PowerRelation(x, ...)

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

Arguments

`x`	An object
`...`	Arguments passed to or from other methods

Value

No return value.

PowerRelation.default *PowerRelation object*

Description

Use `newPowerRelation()` or `newPowerRelationFromString()` to create a PowerRelation object.

Usage

```
## Default S3 method:
PowerRelation(x, ...)
```

Arguments

`x`	An object
`...`	Arguments passed to or from other methods

Value

No return value.

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 `newPowerRelation()`
`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 `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 `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 <- newPowerRelation(c(1,2), ">", 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 <- newPowerRelation(c(1,2), ">", 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 <- newPowerRelation(c(1,2), ">", 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 = "|"))
)

# 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
relations::relation_incidence(relation)
```

`socialranking` *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

`newPowerRelation()` creates a `PowerRelation` object. `createPowerset()` is a convenient function to generate a `newPowerRelation()` 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 `SocialRankingSolution` object.

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

`SocialRankingSolution` `SocialRankingSolution` *object*

Description

Use `doRanking()` to create a `SocialRankingSolution` object.

Usage

```
SocialRankingSolution(x, ...)
```

Arguments

`x`	An object
`...`	Arguments passed to or from other methods

Value

No return value.

SocialRankingSolution.default SocialRankingSolution *object*

Description

Use doRanking() to create a SocialRankingSolution object.

Usage

```
## Default S3 method:
SocialRankingSolution(x, ...)
```

Arguments

x	An object
...	Arguments passed to or from other methods

Value

No return value.

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 newPowerRelation()
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 <- newPowerRelationFromString(
  "123 > 12 ~ 13 > 3 > 1 ~ 2", asWhat = as.numeric
)

# 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 newPowerRelation()

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 newPowerRelation() we therefore assume that $a \sim b$. Similarly if $a \succ b_1 \succ b_2 \succ \ldots \succ b_n \succ a$, the transitive closure turns it into $a \sim b_1 \sim b_2 \sim \ldots \sim b_n$.

transitiveClosure() transforms a PowerRelation object with cycles into a Powerrelation object without cycles. As described in the previous paragraph, 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 <- newPowerRelation(1, ">", 2)

# nothing changes
transitiveClosure(pr)


pr <- suppressWarnings(newPowerRelation(1, ">", 2, ">", 1))

# 1 ~ 2
transitiveClosure(pr)


pr <- suppressWarnings(
  newPowerRelation(1, ">", 3, ">", 1, ">", 2, ">", c(2,3), ">", 2)
)

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

Index