Package: RelationalContracts (via r-universe)

September 22, 2024

Type Package

Title Characterize relational contracts in repated or stochastic games

Version 0.2.0

Author Sebastian Kranz

Maintainer Sebastian Kranz <sebastian.kranz@uni-ulm.de>

Description Characterize relational contracts in repated or stochastic games. Can also analyse repeated negotiation equilibria.

License GPL >= 2.0

Encoding UTF-8

LazyData true

Depends restorepoint, dplyr, matrixStats,RelationalContractsCpp, tidyr

Suggests ggplot2, plotly,DiagrammeR

RoxygenNote 7.1.1

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

RemoteUrl https://github.com/skranz/RelationalContracts

RemoteRef master

RemoteSha e29d080e8bfff5d9033f9b0f8b86ff5e002c6369

Contents

`animate_capped_rne_history` *Use ggplotly to show an animation of the payoff sets of a capped RNE going from t=T to t=1*

Description

Use ggplotly to show an animation of the payoff sets of a capped RNE going from t=T to t=1

Usage

```
animate_capped_rne_history(
  g,
  x = g$sdf$x[1],
  hist = g$eq.history,
  colors = c("#377EB8", "#E41A1C", "#4DAF4A", "#984EA3", "#FF7F00", "#FFFF33",
    "#A65628", "#F781BF"),
  alpha = 0.4,
  add.state.label = TRUE,
```

```
    add.grid = FALSE,
    add.diag = FALSE,
    add.plot = NULL,
    eq.li = NULL
  )
```

`animate_eq_li` *Use ggplotly to show an animation of the payoff sets of a list of equilibria*

Description

Use ggplotly to show an animation of the payoff sets of a list of equilibria

Usage

```
animate_eq_li(g, eq.li, x = g$sdf$x[1], ...)
```

`compare_eq` *Helper function to find differences between two equilibria*

Description

Helper function to find differences between two equilibria

Usage

```
compare_eq(eq1, eq2 = g[["eq"]], g, verbose = TRUE)
```

`diagnose_transitions` *Take a look at the computed transitions for each state using separate data frames*

Description

Take a look at the computed transitions for each state using separate data frames

Usage

```
diagnose_transitions(g)
```

eq_combine_xgroup *Aggregate equilibrium behavior in games with random active player*

Description

Often it is useful to specify games such that players don't move simultaneously but a random player ap is chosen to be active in a given state.

Usage

```
eq_combine_xgroup(
  g,
  eq = g[["eq"]],
  ap.col = ifelse(has.col(eq, "ap"), "ap", NA)
)
```

Arguments

g	the game object
eq	the equilibrium, by default the last solved eq of g.
ap.col	the column as a character in x.df that is the index of the active player. By default "ap".

Details

The active player in a state x is defined by the variable ap in x.df and the original state by xgroup.

This function aggregates equilibrium outcomes from x to xgroup. For payoffs r1,r2,v1,v2 and U we take the mean over the payoffs given the two possible actvive players.

The columns move.adv1 and move.adv2 describe the difference in negotiation payoffs of a player when is the active player who can make a move compared to the other player being active.

Finally we create action labels by combining the actions chosen when a player is active.

eq_diagram *Draws a diagram of equilibrium state transition*

Description

Draws an arrow from state x to state y if and only if on the equilibrium path there is a positive probability to directly transist from x to y.

Usage

```
eq_diagram(
  g,
  show.own.loop = FALSE,
  show.terminal.loop = FALSE,
  use.x = NULL,
  just.eq.chain = FALSE,
  x0 = g$sdf$x[1],
  hide.passive.edge = TRUE,
  label.fun = NULL,
  tooltip.fun = NULL,
  active.edge.color = "#000077",
  passive.edge.color = "#dddddd",
  add.passive.edge = TRUE,
  passive.edge.width = 1,
  return.dfs = FALSE,
  eq = g[["eq"]],
  font.size = 24,
  font = paste0(font.size, "px Arial black")
)
```

Arguments

`g`	The solved game object
`show.own.loop`	Shall a loop from a state to itself be drawn if there is a positive probability to stay in the state? (Default=FALSE)
`show.terminal.loop`	Only relevant if `show.own.loop = TRUE`. If still `show.terminal.loop=FALSE` omit loops in terminal state that don't transist to any other state.
`use.x`	optionally a vector of state ids that shall only be shown.
`just.eq.chain`	If TRUE only show states that can be reached with positive probability on the equilibrium path when starting from state x0.
`x0`	only relevant if `just.eq.chain=TRUE`. The ID of the x0 state. By default the first defined state.
`label.fun`	An optional function that takes the equilibrium object and game and returns a character vector that contains a label for each state.
`tooltip.fun`	Similar to `label.fun` but for the tooltip shown on a state.
`return.dfs`	if TRUE don't show diagram but only return the relevant edge and node data frames that can be used to call `DiagrammeR::create_graph`. Useful if you want to manually customize graphs further.
`font.size`	The font size

eq_diagram_xgroup *Draws a diagram of equilibrium state transition*

Description

Draws an arrow from state x to state y if and only if on the equilibrium path there is a positive probability to directly transist from x to y.

Usage

```
eq_diagram_xgroup(
  g,
  show.own.loop = FALSE,
  show.terminal.loop = FALSE,
  use.x = NULL,
  just.eq.chain = FALSE,
  x0 = g$sdf$x[1],
  hide.passive.edge = TRUE,
  add.passive.edge = TRUE,
  label.fun = NULL,
  tooltip.fun = NULL,
  active.edge.color = "#000077",
  passive.edge.color = "#dddddd",
  passive.edge.width = 1,
  return.dfs = FALSE,
  eq = g[["eq"]],
  ap.col = if (has.col(eq, "ap")) "ap" else NA,
  font.size = 24,
  font = paste0(font.size, "px Arial black")
)
```

Arguments

- `g` The solved game object
- `show.own.loop` Shall a loop from a state to itself be drawn if there is a positive probability to stay in the state? (Default=FALSE)
- `show.terminal.loop` Only relevant if `show.own.loop = TRUE`. If still `show.terminal.loop=FALSE` omit loops in terminal state that don't transist to any other state.
- `use.x` optionally a vector of state ids that shall only be shown.
- `just.eq.chain` If TRUE only show states that can be reached with positive probability on the equilibrium path when starting from state x0.
- `x0` only relevant if `just.eq.chain=TRUE`. The ID of the x0 state. By default the first defined state.
- `label.fun` An optional function that takes the equilibrium object and game and returns a character vector that contains a label for each state.

tooltip.fun	Similar to `label.fun` but for the tooltip shown on a state.
return.dfs	if TRUE don't show diagram but only return the relevant edge and node data frames that can be used to call `DiagrammeR::create_graph`. Useful if you want to manually customize graphs further.

get_eq — *Get the last computed equilibrium of game g*

Description

Get the last computed equilibrium of game g

Usage

```
get_eq(g, extra.cols = "ae", eq = g[["eq"]], add.vr = FALSE)
```

get_repgames_results — *Get the results of all solved repeated games assuming the state is fixed*

Description

Returns for all discount factors the optimal simple strategy profiles maximum joint payoffs and punishment profiles

Usage

```
get_repgames_results(
  g,
  action.details = TRUE,
  delta = g$param$delta,
  rho = g$param$rho
)
```

get_rne — *Get the last computed RNE of game g*

Description

Get the last computed RNE of game g

Usage

```
get_rne(g, extra.cols = "ae", eq = g[["rne"]])
```

`get_rne_details` *Retrieve more details about the last computed RNE*

Description

Retrieve more details about the last computed RNE

Usage

```
get_rne_details(g, x = NULL)
```

`get_spe` *Get the last computed SPE of game g*

Description

Get the last computed SPE of game g

Usage

```
get_spe(g, extra.cols = "ae", eq = g[["spe"]])
```

`get_T_rne_history` *Get the intermediate steps in from t = T to t = 1 for a T-RNE or capped RNE that has been solved with `save.history = TRUE`*

Description

Get the intermediate steps in from t = T to t = 1 for a T-RNE or capped RNE that has been solved with `save.history = TRUE`

Usage

```
get_T_rne_history(g)
```

irv — *Helper functions to specify state transitions*

Description

To be used as argument of irv_joint_dist

Usage

```
irv(var, ..., default = NULL, lower = NULL, upper = NULL, vals.unique = TRUE)
```

Details

See vignette for examples

irv_joint_dist — *Helper function to specify state transitions*

Description

See vignette for examples

Usage

```
irv_joint_dist(
  df,
  ...,
  enclos = parent.frame(),
  remove.zero.prob = TRUE,
  prob.var = "prob"
)
```

irv_val — *Helper functions to specify state transitions*

Description

To be used as argument of irv

Usage

```
irv_val(val, prob)
```

Details

See vignette for examples

plot_eq_payoff_set *Show a base R plot of equilibrium payoff set*

Description

Show a base R plot of equilibrium payoff set

Usage

```
plot_eq_payoff_set(
  g,
  x = eq$x[1],
  t = 1,
  eq = if (use.vr) get_eq(g, add.vr = TRUE) else g[["eq"]],
  xlim = NULL,
  ylim = NULL,
  add = FALSE,
  plot.r = TRUE,
  alpha = 0.8,
  black.border = TRUE,
  add.state.label = is.null(labels),
  labels = NULL,
  colors = c("#377EB8", "#E41A1C", "#4DAF4A", "#984EA3", "#FF7F00", "#FFFF33",
    "#A65628", "#F781BF"),
  add.xlim = NULL,
  add.ylim = NULL,
  extend.lim.perc = 0.05,
  use.vr = FALSE,
  ...
)
```

Arguments

g	The game object for which an equilibrium has been solved
x	A character vector of the state(s) for which the (continuation) equilibrium payoff set shall be shown. By default only the first stage.
eq	An equilibrium object. By default the last solved equilibrium.
xlim	as in `plot.default`
ylim	as in `plot.default`
add	as in `plot.default` Setting `add=FALSE` can be useful to compare payoff sets of different games.
plot.r	Shall negotiation payoffs be shown as a point on the Pareto-frontier (default = TRUE)
alpha	opacity of the fill color

`rel_after_cap_actions` *Fix action profiles for the equilibrium path (ae) and during punishment (a1.hat and a2.hat) that are assumed to be played after the cap in period T onwards. The punishment profile a1.hat is the profile in which player 1 already plays a best-reply (in a1 he might play a non-best reply). From the specified action profiles in all states, we can compute the relevant after-cap payoffs U(x), v1(x) and v2(x) assuming that state transitions would continue.*

Description

Fix action profiles for the equilibrium path (ae) and during punishment (a1.hat and a2.hat) that are assumed to be played after the cap in period T onwards. The punishment profile a1.hat is the profile in which player 1 already plays a best-reply (in a1 he might play a non-best reply). From the specified action profiles in all states, we can compute the relevant after-cap payoffs U(x), v1(x) and v2(x) assuming that state transitions would continue.

Usage

```
rel_after_cap_actions(g, x = NA, ae, a1.hat, a2.hat, x.T = NA)
```

Arguments

`g`	a relational contracting game created with rel_game
`x`	The state(s) for which this after-cap payoff set is applied. If NA (default) and also x.T is NA, it applies to all states.
`ae`	A named list that specifies the equilibrum action profiles.
`a1.hat`	A named list that specifies the action profile when player 1 is punished.
`a2.hat`	A named list that specifies the action profile when player 2 is punished.
`x.T`	Instead of specifiying the argument x, we can specify as x.T a name of the after-cap state. This can be refereed to as the argument x.T in rel_state and rel_states

Value

Returns the updated game

`rel_after_cap_payoffs` *Specify the SPE payoff set(s) of the truncated game(s) after a cap in period T. While we could specify a complete repeated game that is played after the cap, it also suffices to specify just an SPE payoff set of the truncated game of the after-cap state.*

Description

Specify the SPE payoff set(s) of the truncated game(s) after a cap in period T. While we could specify a complete repeated game that is played after the cap, it also suffices to specify just an SPE payoff set of the truncated game of the after-cap state.

Usage

```
rel_after_cap_payoffs(
  g,
  x = NA,
  U,
  v1 = NA,
  v2 = NA,
  v1.rep = NA,
  v2.rep = NA,
  x.T = NA
)
```

Arguments

`g`	a relational contracting game created with rel_game
`x`	The state(s) for which this after-cap payoff set is applied. If NA (default) and also x.T is NA, it applies to all states.
`U`	The highest joint payoff in the truncated repeated game starting from period T.
`v1`	The lowest SPE payoff of player 1 in the truncated game. These are average discounted payoffs using delta as discount factor.
`v2`	Like v1, but for player 2.
`v1.rep`	Alternative to v1. Player 1 lowest SPE payoff in the repeated game with adjusted discount factor delta*(1-rho). Will be automatically converted into v1_trunc based on rho, delta, and bargaining weight. Are often easier to specify.
`v2.rep`	Like v1.rep, but for player 2.
`x.T`	Instead of specifiying the argument x, we can specify as x.T a name of the after-cap state. This can be refereed to as the argument x.T in rel_state and rel_states

Value

Returns the updated game

`rel_capped_rne` *Solve an RNE for a capped version of a game*

Description

In a capped version of the game we assume that after period T the state cannot change anymore and always stays the same. I.e. after T periods players play a repeated game. For a given T a capped game has a unique RNE payoff. Also see `rel_T_rne`.

Usage

```
rel_capped_rne(
  g,
  T,
  delta = g$param$delta,
  rho = g$param$rho,
  adjusted.delta = NULL,
  beta1 = g$param$beta1,
  tie.breaking = c("equal_r", "slack", "random", "first", "last", "max_r1", "max_r2",
    "unequal_r")[1],
  tol = 1e-12,
  add.iterations = FALSE,
  save.details = FALSE,
  save.history = FALSE,
  use.cpp = TRUE,
  T.rne = FALSE,
  spe = NULL,
  res.field = "eq"
)
```

Arguments

g	The game
T	The number of periods in which new negotiations can take place.
delta	the discount factor
rho	the negotiation probability
adjusted.delta	the adjusted discount factor (1-rho)*delta. Can be specified instead of delta.
beta1	the bargaining weight of player 1. By default equal to 0.5. Can also be initially specified with `rel_param`.
tie.breaking	A tie breaking rule when multiple action profiles could be implemented on the equilibrium path with same joint payoff U. Can take the following values: • "equal_r" (DEFAULT) prefer actions that in expectation move to states with more equal negotiation payoffs. • "slack" prefer the action profile with the highest slack in the incentive constraints • "random" pick randomly from all eligible action profiles • "max_r1" pick action profiles that in moves to states with highest negotiation payoff for player 1. • "max_r2" pick action profiles that in moves to states with highest negotiation payoff for player 2.
tol	Due to numerical inaccuracies the calculated incentive constraints for some action profiles may be vialoated even though with exact computation they should hold, yielding unexpected results. We therefore also allow action profiles whose numeric incentive constraints is violated by not more than tol. By default we have `tol=1e-10`.

`add.iterations`	if TRUE just add T iterations to the previously computed capped RNE or T-RNE.
`save.details`	if set TRUE details of the equilibrium are saved that can be analysed later by calling `get_rne_details`. For an example, see the vignette for the Arms Race game.
`save.history`	saves the values for intermediate T.

`rel_change_param` *Add parameters to a relational contracting game*

Description

Add parameters to a relational contracting game

Usage

```
rel_change_param(g, ...)
```

Arguments

`g`	a relational contracting game created with rel_game
`...`	other parameters that can e.g. be used in payoff functions
`delta`	The discount factor
`rho`	The negotiation probability

Value

Returns the updated game

`rel_compile` *Compiles a relational contracting game*

Description

Compiles a relational contracting game

Usage

```
rel_compile(g, ..., compute.just.static = FALSE)
```

rel_eq_as_discounted_sums — *Translate equilibrium payoffs as discounted sum of payoffs*

Description

By default equilibrium payoffs are given as average discounted payoffs. This is the discounted sum of payoffs multiplied by (1-delta).

Usage

```
rel_eq_as_discounted_sums(g)
```

Details

Call this function after you have solved an equilibrium if you want to present the equilibrium

@param g the game for which an equilibrium was computedayoffs as the discounted sum of payoffs instead.

rel_first_best — *Compute first-best.*

Description

We compute the "equilibrium" play that would maximize joint payoffs if incentive constraints could be completely ignored.

Usage

```
rel_first_best(g, delta = g$param$delta, ...)
```

Arguments

`g`	the game object
`delta`	The discount factor
`...`	additional parameters of `rel_spe`

Details

Note that we create the same columns as for a spe, e.g. punishment payoffs v1 and v2 that would arise if every action profile could be implemented as punishment. This allows to use the same functions, like `eq_diagram` as for equilibria.

rel_game — *Creates a new relational contracting game*

Description

Creates a new relational contracting game

Usage

```
rel_game(name = "Game", ..., enclos = parent.frame())
```

rel_is_eq_rne — *Checks if an equilibrium eq with negotiation payoffs is an RNE*

Description

We simply solve the truncated game with r1 and r2 and check whether the resultig r1 and r2 are the same

Usage

```
rel_is_eq_rne(
  g,
  eq = g[["eq"]],
  r1 = eq$r1,
  r2 = eq$r2,
  r.tol = 1e-10,
  verbose = TRUE
)
```

rel_mpe — *Tries to find a MPE by computing iteratively best replies*

Description

Returns a game object that contains the mpe. Use the function get_mpe to retrieve a data frame that describes the MPE.

Usage

```
rel_mpe(
  g,
  delta = g$param$delta,
  static.eq = NULL,
  max.iter = 100,
  tol = 1e-08,
  a.init.guess = NULL
)
```

Arguments

`g`	the game
`delta`	the discount factor
`max.iter`	maximum number of iterations
`tol`	we finish if payoffs in a subsequent iteration don't change by more than tol
`a.init.guess`	optionaly an initially guess of the action profiles. A vector of size nx (number of states) that describes for each state the integer index of the action profile. For a game g look at 'g$ax.grid' to find the indeces of the desired action profiles.

`rel_options` *Set some game options*

Description

Set some game options

Usage

```
rel_options(g, lab.action.sep = " ", lab.player.sep = " | ")
```

`rel_param` *Add parameters to a relational contracting game*

Description

Add parameters to a relational contracting game

Usage

```
rel_param(
  g,
  ...,
  delta = non.null(param[["delta"]], 0.9),
  rho = non.null(param[["rho"]], 0),
  beta1 = non.null(param[["beta1"]], 1/2),
  param = g[["param"]]
)
```

Arguments

g	a relational contracting game created with rel_game
...	other parameters that can e.g. be used in payoff functions
delta	The discount factor
rho	The negotiation probability

Value

Returns the updated game

---

rel_rne   *Find an RNE for a (weakly) directional game*

Description

If the game is strongly directional, i.e. non-terminal states will be reached at most once, there exists a unique RNE payoff.

Usage

```
rel_rne(
  g,
  delta = g$param$delta,
  rho = g$param$rho,
  adjusted.delta = NULL,
  beta1 = g$param$beta1,
  verbose = TRUE,
  ...
)
```

Arguments

`g`	The game object
`delta`	the discount factor
`rho`	the negotiation probability
`adjusted.delta`	the adjusted discount factor (1-rho)*delta. Can be specified instead of delta.
`beta1`	the bargaining weight of player 1. By default equal to 0.5. Can also be initially specified with `rel_param`.
`verbose`	if `TRUE` give more detailed information over the solution process.

Details

For weakly directional games no RNE or multiple RNE payoffs may exist.

You can call rel_capped_rne to solve a capped version of the game that allows state changes only up to some period T. Such a capped version always has a unique RNE payoff.

`rel_scale_eq_payoffs` *Scale equilibrium payoffs*

Description

Scale equilibrium payoffs

Usage

```
rel_scale_eq_payoffs(g, factor)
```

Arguments

`g`	the game for which an equilibrium was computed
`factor`	the factor by which the payoffs U,v1,v2,r1 and r2 are multiplied

---

`rel_solve_repgames` *Solves for all specified states the repeated game assuming the state is fixed*

Description

Solves for all specified states the repeated game assuming the state is fixed

Usage

```
rel_solve_repgames(
  g,
  x = g$sdf$x,
  overwrite = FALSE,
  rows = match(x, g$sdf$x),
  use.repgame.package = FALSE
)
```

Value

Returns a game object that contains a field 'rep.games.df'. This data frame contains the relevant information to compute equilibrium payoffs and equilibria for all discount factors for all states.

---

rel_spe — *Finds an optimal simple subgame perfect equilibrium of g. From this the whole SPE payoff set can be deduced.*

Description

Finds an optimal simple subgame perfect equilibrium of g. From this the whole SPE payoff set can be deduced.

Usage

```
rel_spe(
  g,
  delta = g$param$delta,
  tol.feasible = 1e-10,
  no.exist.action = c("warn", "stop", "nothing"),
  verbose = FALSE,
  r1 = NULL,
  r2 = NULL,
  rho = g$param$rho,
  add.action.labels = TRUE,
  max.iter = 10000,
  first.best = FALSE
)
```

Arguments

`g`	the game object
`delta`	The discount factor. By default the discount factor specified in g.
`tol.feasible`	Due to numerical inaccuracies, sometimes incentive constraints which theoretically should exactly hold, seem to be violated. To avoid this problem, we will consider all action profiles feasible whose incentive constraint is not violated by

more then `tol.feasible`. This means we compute epsilon equilibria in which `tol.feasible` is the epsilon.

`no.exist.action`
: What shall be done if no pure SPE exists? Default is `no.exist.action = "warning"`, alternatives are `no.exist.action = "error"` or `no.exist.action = "nothing"`.

`verbose`
: if `TRUE` give more detailed information over the solution process.

`r1`
: (or `r2`) if not NULL we want to find a SPE in a truncated game. Then r1 and r2 need to specify for each state the exogenously fixed negotiation payoffs.

`rho`
: Only relevant if r1 and r2 are not null. In that case the negotiation probability.

`rel_states` *Add one or multiple states. Allows to specify action spaces, payoffs and state transitions via functions*

Description

Add one or multiple states. Allows to specify action spaces, payoffs and state transitions via functions

Usage

```
rel_states(
  g,
  x,
  A1 = NULL,
  A2 = NULL,
  pi1,
  pi2,
  A.fun = NULL,
  pi.fun = NULL,
  trans.fun = NULL,
  static.A1 = NULL,
  static.A2 = NULL,
  static.A.fun = NULL,
  static.pi1,
  static.pi2,
  static.pi.fun = NULL,
  x.T = NULL,
  pi1.form,
  pi2.form,
  ...
)

rel_state(
  g,
```

```
  x,
  A1 = NULL,
  A2 = NULL,
  pi1,
  pi2,
  A.fun = NULL,
  pi.fun = NULL,
  trans.fun = NULL,
  static.A1 = NULL,
  static.A2 = NULL,
  static.A.fun = NULL,
  static.pi1,
  static.pi2,
  static.pi.fun = NULL,
  x.T = NULL,
  pi1.form,
  pi2.form,
  ...
)
```

Arguments

`g`	a relational contracting game created with rel_game
`x`	The names of the states
`A1`	The action set of player 1. A named list, like `A1=list(e1=1:10)`, where each element is a numeric or character vector.
`A2`	The action set of player 2. See A1.
`pi1`	Player 1's payoff. (Non standard evaluation)
`pi2`	Player 2's payoff. (Non standard evaluation)
`A.fun`	Alternative to specify A1 and A2, a function that returns action sets.
`pi.fun`	Alternative to specify pi1 and pi2 as formula. A vectorized function that returns payoffs directly for all combinations of states and action profiles.
`trans.fun`	A function that specifies state transitions
`x.T`	Relevant when solving a capped game. Which terminal state shall be set in period T onwards. By default, we stay in state x.
`pi1.form`	Player 1's payoff as formula with standard evaluation
`pi2.form`	Player 2's payoff as formula with standard evaluation

Value

Returns the updated game

Functions

- `rel_state`: rel_state is just a synonym for the rel_states. You may want to use it if you specify just a single state.

`rel_state_probs` *Compute the long run probability distribution over states if an equilibrium is played for many periods.*

Description

Adds a column `state.prob` to the computed equilibrium data frame, which you can retrieve by calling `get_eq`.

Usage

```
rel_state_probs(
  g,
  x0 = c("equal", "first", "first.group")[1],
  start.prob = NULL,
  n.burn.in = 100,
  n.averaging = 100,
  tol = 1e-13,
  eq.field = "eq"
)
```

Arguments

`g`	the game object for which an equilibrium has been solved
`x0`	the initial state, by default the first state. If `initial` is not specified we assume the game starts with probability 1 in the initial state. We have 3 reserved keywords: `x0="equal"` means all states are equally likely, `x0="first"` means the game starts in the first state, `x0="first.group"` means all states of the first `xgroup` are equally likely.
`start.prob`	an optional vector of probabilities that specifies for each state the probability that the game starts in that state. Overwrites "x0" unless kept NULL.
`n.burn.in`	Number of rounds before probabilities are averaged.
`n.averaging`	Number of rounds for which probabilities are averaged.
`tol`	Tolerance such that computation stops already in burn-in phase if transition probabilities change not by more than tol.

Details

If the equilibrium strategy induces a unique stationary distribution over the states, this distribution should typically be found (or at least approximated). Otherwise the result can depend on the parameters.

The initial distribution of states is determined by the parameters `x0` or `start.prob`. We then multiply the current probabilities susequently `n.burn.in` times with the transitition matrix on the equilibrium path. This yields the probability distribution over states assuming the game is played for `n.burn.in` periods.

We then continue the process for `n.averaging` rounds, and return the mean of the state probability vectors over these number of rounds.

If between two rounds in the burn-in phase the transitition probabilities of no state pair change by more than the parameter `tol`, we immediately stop and use the resulting probabilit vector.

Note that for T-RNE or capped RNE we always take the transition probabilities of the first period, i.e. we don't increase the t in the actual state definition.

`rel_transition` *Add a state transition from one state to one or several states. For more complex games, it may be preferable to use the arguments* `trans.fun` *of* `link{rel_states}` *instead.*

Description

Add a state transition from one state to one or several states. For more complex games, it may be preferable to use the arguments `trans.fun` of `link{rel_states}` instead.

Usage

```
rel_transition(g, xs, xd, ..., prob = 1)
```

Arguments

`g`	a relational contracting game created with rel_game
`xs`	Name(s) of source states
`xd`	Name(s) of destination states
`...`	named action and their values
`prob`	transition probability

Value

Returns the updated game

`rel_T_rne` *Compute a T-RNE*

Description

The idea of a T-RNE is that only for a finite number of T periods relational contracts will be newly negoatiated. After T periods no new negotiations take place, i.e. every SPE continuation payoff can be implemented. For fixed T there is a unique RNE payoff.

Usage

```
rel_T_rne(
  g,
  T,
  delta = g$param$delta,
  rho = g$param$rho,
  adjusted.delta = NULL,
  beta1 = g$param$beta1,
  tie.breaking = c("equal_r", "slack", "random", "first", "last", "max_r1", "max_r2",
    "unequal_r")[1],
  tol = 1e-12,
  save.details = FALSE,
  add.iterations = FALSE,
  save.history = FALSE,
  use.cpp = TRUE,
  spe = g[["spe"]],
  res.field = "eq"
)
```

Arguments

`g`	The game
`T`	The number of periods in which new negotiations can take place.
`delta`	the discount factor
`rho`	the negotiation probability
`adjusted.delta`	the adjusted discount factor (1-rho)*delta. Can be specified instead of delta.
`beta1`	the bargaining weight of player 1. By default equal to 0.5. Can also be initially specified with `rel_param`.
`tie.breaking`	A tie breaking rule when multiple action profiles could be implemented on the equilibrium path with same joint payoff U. Can take the following values: • "equal_r" (DEFAULT) prefer actions that in expectation move to states with more equal negotiation payoffs. • "slack" prefer the action profile with the highest slack in the incentive constraints • "random" pick randomly from all eligible action profiles • "max_r1" pick action profiles that in moves to states with highest negotiation payoff for player 1. • "max_r2" pick action profiles that in moves to states with highest negotiation payoff for player 2.
`tol`	Due to numerical inaccuracies the calculated incentive constraints for some action profiles may be vialoated even though with exact computation they should hold, yielding unexpected results. We therefore also allow action profiles whose numeric incentive constraints is violated by not more than tol. By default we have `tol=1e-10`.

save.details	if set TRUE details of the equilibrium are saved that can be analysed later by calling get_rne_details. For an example, see the vignette for the Arms Race game.
add.iterations	if TRUE just add T iterations to the previously computed capped RNE or T-RNE.
save.history	saves the values for intermediate T.

Index