Package 'rdecision'

January 10, 2024

Title Decision Analytic Modelling in Health Economics

Version 1.2.0

Description Classes and functions for modelling health care interventions using decision trees and semi-Markov models. Mechanisms are provided for associating an uncertainty distribution with each source variable and for ensuring transparency of the mathematical relationships between variables. The package terminology follows Briggs ``Decision Modelling for Health Economic Evaluation" (2006, ISBN:978-0-19-852662-9).

Depends R (>= 3.1.0)

Imports grid, R6, rlang (>= 0.4.2), stats, withr

Suggests covr, DiagrammeR, grDevices, knitr, pander, rmarkdown, testthat (>= 3.0.0), utf8

License GPL-3

Language en-GB

Encoding UTF-8

RoxygenNote 7.2.3

VignetteBuilder knitr

Config/testthat/edition 3

NeedsCompilation no

Author Andrew Sims [aut, cre] (<https://orcid.org/0000-0002-9553-7278>), Kim Fairbairn [aut] (<https://orcid.org/0000-0001-5108-6279>), Paola Cognigni [aut] (<https://orcid.org/0000-0001-5418-3103>)

Maintainer Andrew Sims <andrew.sims@newcastle.ac.uk>

Repository CRAN

Date/Publication 2024-01-10 10:10:03 UTC

R topics documented:

`Action` *An action in a decision tree*

Description

R6 class representing an action (choice) edge.

Details

A specialism of class `Arrow` which is used in a decision tree to represent an edge whose source node is a `DecisionNode`.

Super classes

`rdecision::Edge` -> `rdecision::Arrow` -> `Action`

Methods

Public methods:

- Action$new()
- Action$modvars()
- Action$p()
- Action$set_cost()
- Action$cost()
- Action$set_benefit()
- Action$benefit()
- Action$clone()

Method `new()`**:** Create an object of type `Action`. Optionally, a cost and a benefit may be associated with traversing the edge. A *pay-off* (benefit minus cost) is sometimes used in edges of decision trees; the parametrization used here is more general.

Usage:

```
Action$new(source_node, target_node, label, cost = 0, benefit = 0)
```

Arguments:

`source_node` Decision node from which the arrow leaves.

`target_node` Node to which the arrow points.

`label` Character string containing the arrow label. This must be defined for an action because the label is used in tabulation of strategies. It is recommended to choose labels that are brief and not punctuated with spaces, dots or underscores.

`cost` Cost associated with traversal of this edge (numeric or `ModVar`).

`benefit` Benefit associated with traversal of the edge.

Returns: A new `Action` object.

Method `modvars()`**:** Find all the model variables of type `ModVar` that have been specified as values associated with this `Action`. Includes operands of these `ModVars`, if they are expressions.

Usage:

```
Action$modvars()
```

Returns: A list of `ModVars`.

Method `p()`**:** Return the current value of the edge probability, i.e. the conditional probability of traversing the edge.

Usage:

```
Action$p()
```

Returns: Numeric value equal to 1.

Method `set_cost()`**:** Set the cost associated with the action edge.

Usage:

```
Action$set_cost(c = 0)
```

Arguments:

`c` Cost associated with traversing the action edge. Of type numeric or `ModVar`.

Returns: Updated `Action` object.

Method `cost()`**:** Return the cost associated with traversing the edge.

Usage:

```
Action$cost()
```

Returns: Cost.

Method `set_benefit()`**:** Set the benefit associated with the action edge.

Usage:

```
Action$set_benefit(b = 0)
```

Arguments:

`b` Benefit associated with traversing the action edge. Of type numeric or `ModVar`.

Returns: Updated `Action` object.

Method `benefit()`**:** Return the benefit associated with traversing the edge.

Usage:

```
Action$benefit()
```

Returns: Benefit.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
Action$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Author(s)

Andrew J. Sims <andrew.sims@newcastle.ac.uk>

`Arborescence` *A rooted directed tree*

Description

An R6 class representing an *arborescence* (a rooted directed tree).

Details

Class to encapsulate a directed rooted tree specialization of a digraph. An arborescence is a directed tree with exactly one root and unique directed paths from the root. Inherits from class `Digraph`.

Super classes

`rdecision::Graph` -> `rdecision::Digraph` -> `Arborescence`

Methods

Public methods:

Method `new()`**:** Create a new `Arborescence` object from sets of nodes and edges.

Usage:

```
Arborescence$new(V, A)
```

Arguments:

`V` A list of Nodes.

`A` A list of Arrows.

Returns: An `Arborescence` object.

Method `parent()`**:** Find the parent of a Node.

Usage:

```
Arborescence$parent(v)
```

Arguments:

`v` Index node, or a list of index Nodes.

Returns: A list of Nodes of the same length as v, if v is a list, or a scalar Node if v is a single node. NA if v (or an element of v) is the root node.

Method `is_parent()`**:** Test whether the given node(s) is (are) parent(s).

Usage:

```
Arborescence$is_parent(v)
```

Arguments:

`v` Node to test, or a list of Nodes.

Details: In an arborescence, `is_parent()` and `is_leaf()` are mutually exclusive.

Returns: A logical vector of the same length as v, if v is a list, or a logical scalar if v is a single node.

Method `is_leaf()`**:** Test whether the given node is a leaf.

Usage:

```
Arborescence$is_leaf(v)
```

Arguments:

v Node to test, or a list of Nodes.

Details: In an arborescence, `is_parent()` and `is_leaf()` are mutually exclusive.

Returns: A logical vector of the same length as v, if v is a list, or a logical scalar is v is a single node.

Method `root()`**:** Find the root vertex of the arborescence.

Usage:

```
Arborescence$root()
```

Returns: The root vertex.

Method `is_root()`**:** Is the specified node the root?

Usage:

```
Arborescence$is_root(v)
```

Arguments:

v Vertex to test, or list of vertexes

Returns: A logical vector if v is a list, or a logical scalar if v is a single node.

Method `siblings()`**:** Find the siblings of a vertex in the arborescence.

Usage:

```
Arborescence$siblings(v)
```

Arguments:

v Vertex to test (only accepts a scalar Node).

Returns: A (possibly empty) list of siblings.

Method `root_to_leaf_paths()`**:** Find all directed paths from the root of the tree to the leaves.

Usage:

```
Arborescence$root_to_leaf_paths()
```

Returns: A list of ordered node lists.

Method `postree()`**:** Implements function `POSITIONTREE` (Walker, 1989) to determine the coordinates for each node in an arborescence.

Usage:

```
Arborescence$postree(
  SiblingSeparation = 4,
  SubtreeSeparation = 4,
  LevelSeparation = 1,
  RootOrientation = "SOUTH",
  MaxDepth = Inf
)
```

Arguments:

`SiblingSeparation` Distance in arbitrary units for the distance between siblings.

`SubtreeSeparation` Distance in arbitrary units for the distance between neighbouring subtrees.

`LevelSeparation` Distance in arbitrary units for the separation between adjacent levels.

`RootOrientation` Must be one of "NORTH", "SOUTH", "EAST", "WEST". Defined as per Walker (1989), but noting that Walker assumed that y increased down the page. Thus the meaning of NORTH and SOUTH are opposite to his, with the default (SOUTH) having the child nodes at positive y value and root at zero, as per his example (figure 12).

`MaxDepth` The maximum depth (number of levels) to be drawn; if the tree exceeds this, an error will be raised.

Details: In the `rdecision` implementation, the sibling order is taken to be the lexicographic order of the node labels, if they are unique among siblings, or the node indexes otherwise.

Returns: A data frame with one row per node and three columns (n, x and y) where `n` gives the node index given by the `Graph::vertex_index()` function.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
Arborescence$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Author(s)

Andrew Sims <andrew.sims@newcastle.ac.uk>

References

Walker, John Q II. A A node-positioning algorithm for general trees. University of North Carolina Technical Report TR 89-034, 1989.

Arrow *A directed edge in a digraph*

Description

An R6 class representing an directed edge in a digraph.

Details

An arrow is the formal term for an edge between pairs of nodes in a directed graph. Inherits from class `Edge`.

Super class

`rdecision::Edge` -> `Arrow`

Methods

Public methods:

- `Arrow$new()`
- `Arrow$source()`
- `Arrow$target()`
- `Arrow$clone()`

Method `new()`**:** Create an object of type `Arrow`.

Usage:

```
Arrow$new(source_node, target_node, label = "")
```

Arguments:

`source_node` Node from which the arrow leaves.

`target_node` Node to which the arrow points.

`label` Character string containing the arrow label.

Returns: A new `Arrow` object.

Method `source()`**:** Access source node.

Usage:

```
Arrow$source()
```

Returns: Node from which the arrow leads.

Method `target()`**:** Access target node.

Usage:

```
Arrow$target()
```

Returns: Node to which the arrow points.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
Arrow$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Author(s)

Andrew J. Sims <andrew.sims@newcastle.ac.uk>

`BetaDistribution` *A parametrized Beta Distribution*

Description

An R6 class representing a Beta distribution with parameters.

Details

A Beta distribution with hyperparameters for shape (`alpha` and `beta`). Inherits from class `Distribution`.

Super class

`rdecision::Distribution` -> `BetaDistribution`

Methods

Public methods:

- `BetaDistribution$new()`
- `BetaDistribution$distribution()`
- `BetaDistribution$mean()`
- `BetaDistribution$mode()`
- `BetaDistribution$SD()`
- `BetaDistribution$sample()`
- `BetaDistribution$quantile()`
- `BetaDistribution$clone()`

Method `new()`**:** Create an object of class `BetaDistribution`.

Usage:

```
BetaDistribution$new(alpha, beta)
```

Arguments:

`alpha` parameter of the Beta distribution.

`beta` parameter of the Beta distribution.

Returns: An object of class `BetaDistribution`.

Method `distribution()`**:** Accessor function for the name of the uncertainty distribution.

Usage:

```
BetaDistribution$distribution()
```

Returns: Distribution name as character string.

Method `mean()`**:** The expected value of the distribution.

Usage:

```
BetaDistribution$mean()
```

Returns: Expected value as a numeric value.

Method `mode()`**:** The mode of the distribution (if `alpha`, `beta` > 1)

Usage:

```
BetaDistribution$mode()
```

Returns: mode as a numeric value.

Method `SD()`**:** The standard deviation of the distribution.

Usage:

```
BetaDistribution$SD()
```

Returns: Standard deviation as a numeric value

Method `sample()`**:** Draw and hold a random sample from the model variable.

Usage:

```
BetaDistribution$sample(expected = FALSE)
```

Arguments:

`expected` If TRUE, sets the next value retrieved by a call to `r()` to be the mean of the distribution.

Returns: Updated distribution.

Method `quantile()`**:** The quantiles of the Beta distribution.

Usage:

```
BetaDistribution$quantile(probs)
```

Arguments:

`probs` Vector of probabilities, in range [0,1].

Returns: Vector of quantiles.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
BetaDistribution$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Author(s)

Andrew J. Sims <andrew.sims@newcastle.ac.uk>

BetaModVar *A model variable whose uncertainty follows a Beta distribution*

Description

An R6 class representing a model variable whose uncertainty is described by a Beta distribution.

Details

A model variable for which the uncertainty in the point estimate can be modelled with a Beta distribution. The hyperparameters of the distribution are the shape parameters (`alpha` and `beta`) of the uncertainty distribution. Inherits from class `ModVar`.

Super class

`rdecision::ModVar` -> `BetaModVar`

Methods

Public methods:

- `BetaModVar$new()`
- `BetaModVar$is_probabilistic()`
- `BetaModVar$clone()`

Method `new()`**:** Create an object of class `BetaModVar`.

Usage:

```
BetaModVar$new(description, units, alpha, beta)
```

Arguments:

`description` A character string describing the variable.

`units` Units of the variable, as character string.

`alpha` parameter of the Beta distribution.

`beta` parameter of the Beta distribution.

Returns: An object of class `BetaModVar`.

Method `is_probabilistic()`**:** Tests whether the model variable is probabilistic, i.e. a random variable that follows a distribution, or an expression involving random variables, some of which follow distributions.

Usage:

```
BetaModVar$is_probabilistic()
```

Returns: TRUE if probabilistic

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
BetaModVar$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Author(s)

Andrew J. Sims <andrew.sims@newcastle.ac.uk>

`BriggsEx47` *Probabilistic results of HIV model*

Description

A dataset containing PSA results of Briggs example 2.5.

Usage

```
data(BriggsEx47)
```

Format

A data frame with 1000 rows and 7 columns:

Mono.LYs Life years gained with monotherapy

Mono.Cost Incremental cost with monotherapy, in GBP

Comb.LYs Life years gained with combination therapy

Comb.Cost Incremental cost with combination therapy, in GBP

Diff.LYG Difference in life years gained

Diff.incCost Difference in incremental cost, GBP

ICER Incremental cost effectiveness ratio, GBP/QALY

Details

A dataset containing the results of probabilistic sensitivity analysis of Briggs (2006) example 2.5 (HIV model), provided as Example 4.7 in the book. These data were generated from the solution spreadsheet provided as a companion to the book (Exercise 4.7 solution) via an Excel macro written to record 1000 runs of the model.

Source

https://www.herc.ox.ac.uk/downloads/decision-modelling-for-health-economic-evaluation/

References

Briggs A, Claxton K, Sculpher M. Decision modelling for health economic evaluation. Oxford, UK: Oxford University Press; 2006.

ChanceNode *A chance node in a decision tree*

Description

An R6 class representing a chance node in a decision tree.

Details

A chance node is associated with at least two branches to other nodes, each of which has a conditional probability (the probability of following that branch given that the node has been reached). Inherits from class Node.

Super class

rdecision::Node -> ChanceNode

Methods

Public methods:

- ChanceNode$new()
- ChanceNode$clone()

Method new(): Create a new ChanceNode object

Usage:

```
ChanceNode$new(label = "")
```

Arguments:

label An optional label for the chance node.

Returns: A new ChanceNode object

Method clone(): The objects of this class are cloneable with this method.

Usage:

```
ChanceNode$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

Author(s)

Andrew Sims <andrew.sims@newcastle.ac.uk>

ConstModVar *A constant model variable*

Description

An R6 class representing a constant in a model.

Details

A ModVar with no uncertainty in its value. Its distribution is treated as a Dirac delta function $\delta(x - c)$ where c is the hyperparameter (value of the constant). The benefit over using a regular numeric variable in a model is that it will appear in tabulations of the model variables associated with a model and therefore be explicitly documented as a model input. Inherits from class ModVar.

Super class

rdecision::ModVar -> ConstModVar

Methods

Public methods:

- ConstModVar$new()
- ConstModVar$is_probabilistic()
- ConstModVar$clone()

Method new(): Create a new constant model variable.

Usage:

```
ConstModVar$new(description, units, const)
```

Arguments:

description A character string description of the variable and its role in the model. This description will be used in a tabulation of the variables linked to a model.

units A character string description of the units, e.g. "GBP", "per year".

const The constant numerical value of the object.

Returns: A new ConstModVar object.

Method is_probabilistic(): Tests whether the model variable is probabilistic.

Usage:

```
ConstModVar$is_probabilistic()
```

Details: Does the random variable follow a distribution, or is it an expression involving' random variables, some of which follow distributions?

Returns: TRUE if probabilistic

Method clone(): The objects of this class are cloneable with this method.

Usage:

```
ConstModVar$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

Author(s)

Andrew Sims <andrew.sims@newcastle.ac.uk>

`DecisionNode` *A decision node in a decision tree*

Description

An R6 class representing a decision node in a decision tree.

Details

A class to represent a decision node in a decision tree. The node is associated with one or more branches to child nodes. Inherits from class `Node`.

Super class

`rdecision::Node` -> `DecisionNode`

Methods

Public methods:

- `DecisionNode$new()`
- `DecisionNode$clone()`

Method `new()`**:** Create a new decision node.

Usage:

```
DecisionNode$new(label)
```

Arguments:

`label` A label for the node. Must be defined because the label is used in tabulation of strategies. The label is automatically converted to a syntactically valid (in R) name to ensure it can be used as a column name in a data frame.

Returns: A new `DecisionNode` object.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
DecisionNode$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Author(s)

Andrew J. Sims <andrew.sims@newcastle.ac.uk>

DecisionTree *A decision tree*

Description

An R6 class to represent a decision tree model.

Details

A class to represent a decision tree. An object contains a tree of decision nodes, chance nodes and leaf nodes, connected by edges (either actions or reactions). It inherits from class `Arborescence` and satisfies the following conditions:

1. Nodes and edges must form a tree with a single root and there must be a unique path from the root to each node. In graph theory terminology, the directed graph formed by the nodes and edges must be an *arborescence*.
2. Each node must inherit from one of `DecisionNode`, `ChanceNode` or `LeafNode`. Formally the set of vertices must be a disjoint union of sets of decision nodes, chance nodes and leaf nodes.
3. All and only leaf nodes must have no children.
4. Each edge must inherit from either `Action` or `Reaction`.
5. All and only edges that have source endpoints joined to decision nodes must inherit from `Action`.
6. All and only edges that have source endpoints joined to chance nodes must inherit from `Reaction`.
7. The sum of probabilities of each set of reaction edges with a common source endpoint must be 1.
8. Each `DecisionNode` must have a label, and the labels of all `DecisionNodes` must be unique within the model.
9. Each `Action` must have a label, and the labels of `Actions` that share a common source endpoint must be unique.

Super classes

`rdecision::Graph` -> `rdecision::Digraph` -> `rdecision::Arborescence` -> `DecisionTree`

Methods

Public methods:

- `DecisionTree$new()`
- `DecisionTree$decision_nodes()`
- `DecisionTree$chance_nodes()`
- `DecisionTree$leaf_nodes()`
- `DecisionTree$actions()`
- `DecisionTree$modvars()`

- DecisionTree$modvar_table()
- DecisionTree$draw()
- DecisionTree$is_strategy()
- DecisionTree$strategy_table()
- DecisionTree$strategy_paths()
- DecisionTree$edge_properties()
- DecisionTree$evaluate_walks()
- DecisionTree$evaluate()
- DecisionTree$tornado()
- DecisionTree$threshold()
- DecisionTree$clone()

Method `new()`**:** Create a new decision tree.

Usage:

```
DecisionTree$new(V, E)
```

Arguments:

`V` A list of nodes.

`E` A list of edges.

Details: The tree must consist of a set of nodes and a set of edges which satisfy the conditions given in the details section of this class.

Returns: A `DecisionTree` object

Method `decision_nodes()`**:** Find the decision nodes in the tree.

Usage:

```
DecisionTree$decision_nodes(what = "node")
```

Arguments:

`what` A character string defining what to return. Must be one of "node", "label" or "index".

Returns: A list of `DecisionNode` objects (for what = "node"); a list of character strings (for what = "label"), or an integer vector with indexes of the decision nodes (for what = "index").

Method `chance_nodes()`**:** Find the chance nodes in the tree.

Usage:

```
DecisionTree$chance_nodes(what = "node")
```

Arguments:

`what` A character string defining what to return. Must be one of "node", "label" or "index".

Returns: A list of `ChanceNode` objects (for what = "node"); a list of character strings (for what = "label"), or an integer vector with indexes of the decision nodes (for what = "index").

Method `leaf_nodes()`**:** Find the leaf nodes in the tree.

Usage:

```
DecisionTree$leaf_nodes(what = "node")
```

Arguments:

what One of "node" (returns Node objects), "label" (returns the leaf node labels) or "index" (returns the vertex indexes of the leaf nodes).

Returns: A list of LeafNode objects (for what = "node"); a list of character strings (for what = "label"); or an integer vector of leaf node indexes (for what = "index").

Method actions(): Find the edges that have the specified decision node as their source.

Usage:

```
DecisionTree$actions(d)
```

Arguments:

d A decision node.

Returns: A list of Action edges.

Method modvars(): Find all the model variables of type ModVar.

Usage:

```
DecisionTree$modvars()
```

Details: Find ModVars that have been specified as values associated with the nodes and edges of the tree.

Returns: A list of ModVars.

Method modvar_table(): Tabulate the model variables.

Usage:

```
DecisionTree$modvar_table(expressions = TRUE)
```

Arguments:

expressions A logical that defines whether expression model variables should be included in the tabulation.

Returns: Data frame with one row per model variable, as follows:

Description As given at initialization.

Units Units of the variable.

Distribution Either the uncertainty distribution, if it is a regular model variable, or the expression used to create it, if it is an ExprModVar.

Mean Mean; calculated from means of operands if an expression.

E Expectation; estimated from random sample if expression, mean otherwise.

SD Standard deviation; estimated from random sample if expression, exact value otherwise.

Q2.5 p=0.025 quantile; estimated from random sample if expression, exact value otherwise.

Q97.5 p=0.975 quantile; estimated from random sample if expression, exact value otherwise.

Est TRUE if the quantiles and SD have been estimated by random sampling.

Method draw(): Draw the decision tree to the current graphics output.

Usage:

```
DecisionTree$draw(border = FALSE)
```

Arguments:

border If TRUE draw a light grey border around the plot area.

Details: Uses the algorithm of Walker (1989) to distribute the nodes compactly (see the Arborescence class help for details).

Returns: No return value.

Method `is_strategy()`**:** Tests whether an object is a valid strategy.

Usage:

```
DecisionTree$is_strategy(strategy)
```

Arguments:

`strategy` A list of Action edges.

Details: A strategy is a unanimous prescription of an action taken at each decision node, coded as a list of action edges. This checks whether the strategy is valid for this decision tree.

Returns: TRUE if the strategy is valid for this tree. Returns FALSE if the list of Action edges are not a valid strategy.

Method `strategy_table()`**:** Find all potential strategies for the decision tree.

Usage:

```
DecisionTree$strategy_table(what = "index", select = NULL)
```

Arguments:

`what` A character string defining what to return. Must be one of "label" or "index".

`select` A single strategy (given as a list of action edges, with one action edge per decision node). If provided, only that strategy is selected from the returned table. Intended for tabulating a single strategy into a readable form.

Details: A strategy is a unanimous prescription of the actions at each decision node. If there are decision nodes that are descendants of other nodes in the tree, the strategies returned will not necessarily be unique.

Returns: A data frame where each row is a potential strategy and each column is a decision node, ordered lexicographically. Values are either the index of each action edge, or their label. The row names are the edge labels of each strategy, concatenated with underscores.

Method `strategy_paths()`**:** Find all paths walked in each possible strategy.

Usage:

```
DecisionTree$strategy_paths()
```

Details: A strategy is a unanimous prescription of an action in each decision node. Some paths can be walked in more than one strategy, if there exist paths that do not pass a decision node.

Returns: A data frame, where each row is a path walked in a strategy. The structure is similar to that returned by `strategy_table` but includes an extra column, `Leaf` which gives the leaf node index of each path, and there is one row for each path in each strategy.

Method `edge_properties()`**:** Properties of all actions and reactions as a matrix.

Usage:

```
DecisionTree$edge_properties()
```

Details: Gets the properties (probability, cost, benefit) of each action and reaction in the decision tree in matrix form.

Returns: A numeric matrix with one row per edge, and with four columns: the index of the edge, the conditional probability of traversing the edge, the cost of traversing the edge and the benefit associated with traversing the edge. The column names are `index`, `probability`, `cost`, `benefit` and the row names are the labels of the edges.

Method `evaluate_walks()`**:** Evaluate the components of pay-off associated with a set of walks in the decision tree.

Usage:

```
DecisionTree$evaluate_walks(W = NULL, Wi = NULL)
```

Arguments:

`W` A list of root-to-leaf walks. A walk is a sequence of edges (actions and reactions), stored as a list. Each walk must start with an edge whose source is the root node and end with an edge whose target is a leaf node. The list of walks is normally the walks associated with all the root to leaf paths in a tree.

`Wi` As *W* but with edge indices instead of Edge objects. One of *W* and *Wi* must be NULL. It is more efficient to provide *Wi* during PSA, where the paths do not change between cycles, to avoid repeated conversion of edges to indices.

Details: For each walk, probability, cost, benefit and utility are calculated. There is minimal checking of the argument because this function is intended to be called repeatedly during tree evaluation, including PSA.

Returns: A pay-off table, represented as a matrix of numeric values with response columns as follows:

`Probability` The probability of traversing the pathway.

`Path.Cost` The cost of traversing the pathway.

`Path.Benefit` The benefit derived from traversing the pathway.

`Path.Utility` The utility associated with the outcome (leaf node).

`Path.QALY` The QALYs associated with the outcome (leaf node).

`Cost` `Path.Cost` $*$ probability of traversing the pathway.

`Benefit` `Path.Benefit` $*$ probability of traversing the pathway.

`Utility` `Path.Utility` $*$ probability of traversing the pathway.

`QALY` `Path.QALY` $*$ probability of traversing the pathway.

The matrix has one row per path, with the row label equal to the character representation of the index of the leaf node at the end of the path.

Method `evaluate()`**:** Evaluate each strategy.

Usage:

```
DecisionTree$evaluate(setvars = "expected", N = 1L, by = "strategy")
```

Arguments:

`setvars` One of "expected" (evaluate with each model variable at its mean value), "random" (sample each variable from its uncertainty distribution and evaluate the model), "q2.5", "q50", "q97.5" (set each model variable to its 2.5%, 50% or 97.5% quantile, respectively, and evaluate the model) or "current" (leave each model variable at its current value prior to calling the function and evaluate the model).

`N` Number of replicates. Intended for use with PSA (`modvars` = `"random"`); use with `modvars` = "expected" will be repetitive and uninformative.

`by` One of {"path", "strategy", "run"}. If "path", the table has one row per path walked per strategy, per run, and includes the label of the terminating leaf node to identify each path. If "strategy" (the default), the table is aggregated by strategy, i.e., there is one row per strategy per run. If "run", the table has one row per run and uses concatenated strategy names (as above) and one (cost, benefit, utility, QALY) as row names.

Details: Starting with the root, the function works though all possible paths to leaf nodes and computes the probability, cost, benefit and utility of each, optionally aggregated by strategy or run. The columns of the returned data frame are:

`by = "path"` `Run` Run number

`<label of first decision node>` label of action leaving the node

`<label of second decision node (etc.)>` label of action leaving the node

`Leaf` The label of terminating leaf node

`Probability` Probability of traversing the path

`Cost` Cost of traversing the path

`Benefit` Benefit of traversing the path

`Utility` Utility of traversing the path

`QALY` QALY of traversing the path

`by = "strategy"` `Run` Run number

`<label of first decision node>` label of action leaving the node

`<label of second decision node (etc)` label of action

`Probability` Σp_i for the run (1)

`Cost` Aggregate cost of the strategy

`Benefit` Aggregate benefit of the strategy

`Utility` Aggregate utility of the strategy

`QALY` Aggregate QALY of the strategy

`by = "run"` `Run` Run number

`Probability.<S>` Probability for strategy S

`Cost.<S>` Cost for strategy S

`Benefit.<S>` Benefit for strategy S

`Utility.<S>` Benefit for strategy S

`QALY.<S>` QALY for strategy S

where <S> is a label associated with strategy `S`. Each strategy label is a list of the labels of the action edges that are traversed in the strategy, concatenated with underscores. The ordering of each label part follows the lexicographical order of the decision node labels concatenated with underscores. For example, if there are three decision nodes labelled d1, d2 and d3, each strategy label will be of the form a1i_a2i_a3i where a1i is the label of one action edge emanating from decision node d1, etc. There will be one probability, cost, benefit, utility and QALY column for each strategy.

Returns: A data frame whose columns depend on `by`; see "Details".

Method `tornado()`**:** Create a "tornado" diagram.

Usage:

```
DecisionTree$tornado(
  index,
  ref,
  outcome = "saving",
  exclude = NULL,
  draw = TRUE
)
```

Arguments:

`index` The index strategy (option) to be evaluated.

`ref` The reference strategy (option) with which the index strategy will be compared.

`outcome` One of `"saving"` or `"ICER"`. For `"saving"` (e.g. in cost consequence analysis), the x axis is cost saved (cost of reference minus cost of index), on the presumption that the new technology will be cost saving at the point estimate. For `"ICER"` the x axis is $\Delta C/\Delta E$ and is expected to be positive at the point estimate (i.e. in the NE or SW quadrants of the cost-effectiveness plane), where ΔC is cost of index minus cost of reference, and ΔE is utility of index minus utility of reference.

`exclude` A list of descriptions of model variables to be excluded from the tornado.

`draw` TRUE if the graph is to be drawn; otherwise return the data frame silently.

Details: Used to compare two strategies for traversing the decision tree. A strategy is a unanimous prescription of the actions at each decision node. The extreme values of each input variable are the upper and lower 95% confidence limits of the uncertainty distributions of each variable. This ensures that the range of each input is defensible (Briggs 2012).

Returns: A data frame with one row per input model variable and columns for: minimum value of the variable, maximum value of the variable, minimum value of the outcome and maximum value of the outcome. NULL if there are no `ModVars`.

Method `threshold()`**:** Find the threshold value of a model variable at which the cost difference is zero or the ICER is equal to a threshold, for an index strategy compared with a reference strategy.

Usage:

```
DecisionTree$threshold(
  index,
  ref,
  outcome,
  mvd,
  a,
  b,
  tol,
  lambda = NULL,
  nmax = 1000L
)
```

Arguments:

`index` The index strategy (option) to be evaluated.

`ref` The reference strategy (option) with which the index strategy will be compared.

`outcome` One of `"saving"` or `"ICER"`. For `"saving"` (e.g., in cost consequence analysis), the value of `mvd` is found at which cost saved is zero (cost saved is cost of reference minus cost of index, on the presumption that the new technology will be cost saving at the point estimate). For `"ICER"` the value of `mvd` is found for which the incremental cost effectiveness ratio (ICER) is equal to the threshold `lambda`. ICER is calculated as $\Delta C/\Delta E$, which will normally be positive at the point estimate (i.e. in the NE or SW quadrants of the cost-effectiveness plane), where ΔC is cost of index minus cost of reference and ΔE is utility of index minus utility of reference.

`mvd` The description of the model variable for which the threshold is to be found.

`a` The lower bound of the range of values of `mvd` to search for the root (numeric).

`b` The upper bound of the range of values of `mvd` to search for the root (numeric).

`tol` The tolerance to which the threshold should be calculated (numeric).

`lambda` The ICER threshold (threshold ratio) for outcome="ICER".

`nmax` Maximum number if iterations allowed to reach convergence.

Details: Uses a rudimentary bisection method method to find the root. In PSA terms, the algorithm finds the value of the specified model variable for which 50% of runs are cost saving (or above the ICER threshold) and 50% are cost incurring (below the ICER threshold).

Returns: Value of the model variable of interest at the threshold.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
DecisionTree$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Author(s)

Andrew J. Sims <andrew.sims@newcastle.ac.uk>

References

Briggs A, Claxton K, Sculpher M. Decision modelling for health economic evaluation. Oxford, UK: Oxford University Press; 2006.

Briggs AH, Weinstein MC, Fenwick EAL, Karnon J, Sculpher MJ, Paltiel AD. Model Parameter Estimation and Uncertainty: A Report of the ISPOR-SMDM Modeling Good Research Practices Task Force-6. *Value in Health* 2012;**15**:835–42, doi:10.1016/j.jval.2012.04.014.

Kaminski B, Jakubczyk M, Szufel P. A framework for sensitivity analysis of decision trees. *Central European Journal of Operational Research* 2018;**26**:135–59, doi:10.1007/s1010001704796.

Digraph *A directed graph*

Description

An R6 class representing a digraph (a directed graph).

Details

Encapsulates and provides methods for computation and checking of directed graphs (digraphs). Inherits from class Graph.

Super class

rdecision::Graph -> Digraph

Methods

Public methods:

- Digraph$new()
- Digraph$digraph_adjacency_matrix()
- Digraph$digraph_incidence_matrix()
- Digraph$topological_sort()
- Digraph$is_connected()
- Digraph$is_weakly_connected()
- Digraph$is_acyclic()
- Digraph$is_tree()
- Digraph$is_polytree()
- Digraph$is_arborescence()
- Digraph$direct_successors()
- Digraph$direct_predecessors()
- Digraph$arrow_source()
- Digraph$arrow_target()
- Digraph$paths()
- Digraph$walk()
- Digraph$as_DOT()
- Digraph$clone()

Method new(): Create a new Digraph object from sets of nodes and edges.

Usage:

```
Digraph$new(V, A)
```

Arguments:

V A list of Nodes.

A A list of Arrows.

Returns: A Digraph object.

Method `digraph_adjacency_matrix()`**:** Compute the adjacency matrix for the digraph.

Usage:

```
Digraph$digraph_adjacency_matrix(boolean = FALSE)
```

Arguments:

`boolean` If `TRUE`, the adjacency matrix is logical, each cell is {`FALSE`,`TRUE`}.

Details: Each cell contains the number of edges from the row vertex to the column vertex, with the convention of self loops being counted once, unless `boolean` is `TRUE` when cells are either `FALSE` (not adjacent) or `TRUE` (adjacent).

Returns: A square integer matrix with the number of rows and columns equal to the order of the graph. The rows and columns are in the same order as `V`. If the nodes have defined and unique labels the dimnames of the matrix are the labels of the nodes.

Method `digraph_incidence_matrix()`**:** Compute the incidence matrix for the digraph.

Usage:

```
Digraph$digraph_incidence_matrix()
```

Details: Each row is a vertex and each column is an edge. Edges leaving a vertex have value -1 and edges entering have value +1. By convention self loops have value 0 (1-1). If all vertexes have defined and unique labels and all edges have defined and unique labels, the dimnames of the matrix are the labels of the vertexes and edges.

Returns: The incidence matrix of integers.

Method `topological_sort()`**:** Topologically sort the vertexes in the digraph.

Usage:

```
Digraph$topological_sort()
```

Details: Uses Kahn's algorithm (Kahn, 1962).

Returns: A list of vertexes, topologically sorted. If the digraph has cycles, the returned ordered list will not contain all the vertexes in the graph, but no error will be raised.

Method `is_connected()`**:** Test whether the graph is connected.

Usage:

```
Digraph$is_connected()
```

Details: For digraphs this will always return `FALSE` because *connected* is not defined. Function `weakly_connected` calculates whether the underlying graph is connected.

Returns: `TRUE` if connected, `FALSE` if not.

Method `is_weakly_connected()`**:** Test whether the digraph is weakly connected, i.e. if the underlying graph is connected.

Usage:

```
Digraph$is_weakly_connected()
```

Returns: `TRUE` if connected, `FALSE` if not.

Method `is_acyclic()`**:** Checks for the presence of a cycle in the graph.

Usage:

```
Digraph$is_acyclic()
```

Details: Attempts to do a topological sort. If the sort does not contain all vertexes, the digraph contains at least one cycle. This method overrides `is_acyclic` in `Graph`.

Returns: `TRUE` if no cycles detected.

Method `is_tree()`**:** Is the digraph's underlying graph a tree?

Usage:

```
Digraph$is_tree()
```

Details: It is a tree if it is connected and acyclic.

Returns: `TRUE` if the underlying graph is a tree; `FALSE` if not.

Method `is_polytree()`**:** Is the digraph's underlying graph a polytree?

Usage:

```
Digraph$is_polytree()
```

Details: It is a polytree if it is directed, connected and acyclic. Because the object is a digraph (directed), this is synonymous with `tree`.

Returns: `TRUE` if the underlying graph is a tree; `FALSE` if not.

Method `is_arborescence()`**:** Is the digraph an arborescence?

Usage:

```
Digraph$is_arborescence()
```

Details: An *arborescence* is a tree with a single root and unique paths from the root.

Returns: `TRUE` if the digraph is an arborescence; `FALSE` if not.

Method `direct_successors()`**:** Find the direct successors of a node.

Usage:

```
Digraph$direct_successors(v)
```

Arguments:

`v` The index vertex (a scalar; does not accept a vector of nodes).

Returns: A list of Nodes or an empty list if the specified node has no successors.

Method `direct_predecessors()`**:** Find the direct predecessors of a node.

Usage:

```
Digraph$direct_predecessors(v)
```

Arguments:

`v` The index vertex (a scalar; does not accept an index of nodes).

Returns: A list of Nodes or an empty list if the specified node has no predecessors.

Method `arrow_source()`**:** Find the node that is the source of the given arrow.

Usage:

```
Digraph$arrow_source(a)
```

Arguments:

`a` An arrow (directed edge), which must be in the digraph.

Details: The source node is a property of the arrow, not the digraph of which it is part, hence the canonical method for establishing the source node of an arrow is via method `$source` of an `Arrow` object. This function is provided for convenience when iterating the arrows of a digraph. It raises an error if the arrow is not in the graph. It returns the index of the source node, which is a property of the graph; the node object itself may be retrieved using the `$vertex_at` method of the graph.

Returns: Index of the source node of the specified edge.

Method `arrow_target()`**:** Find the node that is the target of the given arrow.

Usage:

```
Digraph$arrow_target(a)
```

Arguments:

`a` An arrow (directed edge), which must be in the digraph.

Details: The target node is a property of the arrow, not the digraph of which it is part, hence the canonical method for establishing the target node of an arrow is via method `$target` of an `$Arrow` object. This function is provided for convenience when iterating the arrows of a digraph. It raises an error if the arrow is not in the graph. It returns the index of the target node, which is a property of the graph; the node itself may be retrieved using the `$vertex_at` method of the graph.

Returns: Index of the target node of the specified edge.

Method `paths()`**:** Find all directed simple paths from source to target.

Usage:

```
Digraph$paths(s, t)
```

Arguments:

`s` Source node.

`t` Target node.

Details: In simple paths all vertexes are unique. Uses a recursive depth-first search algorithm.

Returns: A list of ordered node lists.

Method `walk()`**:** Sequence of edges which join the specified path.

Usage:

```
Digraph$walk(P, what = "edge")
```

Arguments:

`P` A list of Nodes

`what` One of "edge" or "index".

Returns: A list of Edges for `what = "edge"` or a list of Edge indices for `what = "index"`.

Method `as_DOT()`**:** Exports the digraph in DOT notation.

Usage:

```
Digraph$as_DOT(rankdir = "LR", width = 7, height = 7)
```

Arguments:

`rankdir` One of "LR" (default), "TB", "RL" or "BT".

`width` of the drawing, in inches

`height` of the drawing, in inches

Details: Writes a representation of the digraph in the `graphviz` DOT language (https://graphviz.org/doc/info/lang.html) for drawing with one of the `graphviz` tools, including `dot` (Gansner, 1993). If all nodes have labels, these are used in the graph, otherwise the labels are the node indices.

Returns: A character vector. Intended for passing to `writeLines` for saving as a text file.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
Digraph$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Author(s)

Andrew Sims <andrew.sims@newcastle.ac.uk>

References

Gansner ER, Koutsofios E, North SC, Vo K-P. A technique for drawing directed graphs. *IEEE Transactions on Software Engineering*, 1993;**19**:214–30, doi:10.1109/32.221135.

Gross JL, Yellen J, Zhang P. Handbook of Graph Theory. Second edition, Chapman and Hall/CRC.; 2013, doi:10.1201/b16132.

Kahn AB, Topological Sorting of Large Networks, *Communications of the* ACM, 1962;**5**:558-562, doi:10.1145/368996.369025.

`DiracDistribution` *A Dirac delta function*

Description

An R6 class representing a Dirac Delta function.

Details

A distribution modelled by a Dirac delta function $\delta(x - c)$ where c is the hyperparameter (value of the constant). It has probability 1 that the value will be equal to c and zero otherwise. The mode, mean, quantiles and random samples are all equal to c. It is acknowledged that there is debate over whether Dirac delta functions are true distributions, but the assumption makes little practical difference in this case. Inherits from class `Distribution`.

Super class

`rdecision::Distribution` -> `DiracDistribution`

Methods

Public methods:

- `DiracDistribution$new()`
- `DiracDistribution$distribution()`
- `DiracDistribution$mode()`
- `DiracDistribution$mean()`
- `DiracDistribution$SD()`
- `DiracDistribution$quantile()`
- `DiracDistribution$sample()`
- `DiracDistribution$clone()`

Method `new()`**:** Create a new Dirac Delta function distribution.

Usage:

```
DiracDistribution$new(const)
```

Arguments:

`const` The value at which the distribution is centred.

Returns: A new `DiracDistribution` object.

Method `distribution()`**:** Accessor function for the name of the distribution.

Usage:

```
DiracDistribution$distribution()
```

Returns: Distribution name as character string.

Method `mode()`**:** Return the mode of the distribution.

Usage:

```
DiracDistribution$mode()
```

Returns: Numeric Value where the distribution is centered.

Method `mean()`**:** Return the expected value of the distribution.

Usage:

```
DiracDistribution$mean()
```

Returns: Expected value as a numeric value.

Method `SD()`**:** Return the standard deviation of the distribution.

Usage:

```
DiracDistribution$SD()
```

Returns: Standard deviation as a numeric value

Method `quantile()`**:** Quantiles of the distribution.

Usage:

```
DiracDistribution$quantile(probs)
```

Arguments:

`probs` Numeric vector of probabilities, each in range [0,1].

Details: For a Dirac Delta Function all quantiles are returned as the value at which the distribution is centred.

Returns: Vector of numeric values of the same length as `probs`.

Method `sample()`**:** Draw and hold a random sample from the model variable.

Usage:

```
DiracDistribution$sample(expected = FALSE)
```

Arguments:

`expected` If TRUE, sets the next value retrieved by a call to `r()` to be the mean of the distribution.

Returns: Updated distribution.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
DiracDistribution$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Author(s)

Andrew Sims <andrew.sims@newcastle.ac.uk>

`DirichletDistribution` *A parametrized Dirichlet distribution*

Description

An R6 class representing a multivariate Dirichlet distribution.

Details

A multivariate Dirichlet distribution. See https://en.wikipedia.org/wiki/Dirichlet_distribution for details. Inherits from class `Distribution`.

Super class

`rdecision::Distribution` -> `DirichletDistribution`

Methods

Public methods:

Method `new()`**:** Create an object of class `DirichletDistribution`.

Usage:

```
DirichletDistribution$new(alpha)
```

Arguments:

`alpha` Parameters of the distribution; a vector of `K` numeric values each > 0, with $K > 1$.

Returns: An object of class `DirichletDistribution`.

Method `distribution()`**:** Accessor function for the name of the distribution.

Usage:

```
DirichletDistribution$distribution()
```

Returns: Distribution name as character string.

Method `mean()`**:** Mean value of each dimension of the distribution.

Usage:

```
DirichletDistribution$mean()
```

Returns: A numerical vector of length K.

Method `mode()`**:** Return the mode of the distribution.

Usage:

```
DirichletDistribution$mode()
```

Details: Undefined if any alpha is ≤ 1.

Returns: Mode as a vector of length `K`.

Method `quantile()`**:** Quantiles of the univariate marginal distributions.

Usage:

```
DirichletDistribution$quantile(probs)
```

Arguments:

`probs` Numeric vector of probabilities, each in range [0,1].

Details: The univariate marginal distributions of a Dirichlet distribution are Beta distributions. This function returns the quantiles of each marginal. Note that these are not the true quantiles of the multivariate Dirichlet.

Returns: A matrix of numeric values with the number of rows equal to the length of `probs`, the number of columns equal to the order; rows are labelled with quantiles and columns with the dimension (1, 2, etc).

Method `varcov()`**:** Variance-covariance matrix.

Usage:

```
DirichletDistribution$varcov()
```

Returns: A positive definite symmetric matrix of size `K` by `K`.

Method `sample()`**:** Draw and hold a random sample from the distribution.

Usage:

```
DirichletDistribution$sample(expected = FALSE)
```

Arguments:

`expected` If TRUE, sets the next value retrieved by a call to `r()` to be the mean of the distribution.

Returns: Void; sample is retrieved with call to `r()`.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
DirichletDistribution$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Author(s)

Andrew J. Sims <andrew.sims@newcastle.ac.uk>

`Distribution` *A probability distribution*

Description

An R6 class representing a (possibly multivariate) distribution.

Details

The base class for particular univariate or multivariate distributions.

Methods

Public methods:

- Distribution$new()
- Distribution$order()
- Distribution$distribution()
- Distribution$mean()
- Distribution$mode()
- Distribution$SD()
- Distribution$varcov()
- Distribution$quantile()
- Distribution$sample()
- Distribution$r()
- Distribution$clone()

Method `new()`**:** Create an object of class `Distribution`.

Usage:

```
Distribution$new(name, K = 1L)
```

Arguments:

`name` Name of the distribution ("Beta" etc.)

`K` Order of the distribution (1 = univariate, 2 = bivariate etc.). Must be an integer; use 1L, 3L etc. to avoid an error.

Returns: An object of class `Distribution`.

Method `order()`**:** Order of the distribution

Usage:

```
Distribution$order()
```

Returns: Order (`K`).

Method `distribution()`**:** Description of the uncertainty distribution.

Usage:

```
Distribution$distribution()
```

Details: Includes the distribution name and its parameters.

Returns: Distribution name and parameters as character string.

Method `mean()`**:** Mean value of the distribution.

Usage:

```
Distribution$mean()
```

Returns: Mean value as a numeric scalar (`K = 1L`) or vector of length `K`.

Method `mode()`**:** Return the mode of the distribution.

Usage:

```
Distribution$mode()
```

Details: By default returns NA, which will be the case for most because an arbitrary distribution is not guaranteed to be unimodal.

Returns: Mode as a numeric scalar (K = 1L) or vector of length K.

Method SD(): Return the standard deviation of a univariate distribution.

Usage:

```
Distribution$SD()
```

Details: Only defined for univariate (K = 1L) distributions; for multivariate distributions, function varcov returns the variance-covariance matrix.

Returns: Standard deviation as a numeric value.

Method varcov(): Variance-covariance matrix.

Usage:

```
Distribution$varcov()
```

Returns: A positive definite symmetric matrix of size K by K, or a scalar for K = 1L, equal to the variance.

Method quantile(): Marginal quantiles of the distribution.

Usage:

```
Distribution$quantile(probs)
```

Arguments:

probs Numeric vector of probabilities, each in range [0,1].

Details: If they are defined, this function returns the marginal quantiles of the multivariate distribution; i.e. the quantiles of each univariate marginal distribution of the multivariate distribution. For example, the univariate marginal distributions of a multivariate normal are univariate normals, and the univariate marginal distributions of a Dirichlet distribution are Beta distributions. Note that these are not the true quantiles of a multivariate distribution, which are contours for K = 2L, surfaces for K = 3L, etc. For example, the 2.5% and 97.5% marginal quantiles of a bivariate normal distribution define a rectangle in x_1, x_2 space that will include more than 95% of the distribution, whereas the contour containing 95% of the distribution is an ellipse.

Returns: For K = 1L a numeric vector of length equal to the length of probs, with each entry labelled with the quantile. For K > 1L a matrix of numeric values with the number of rows equal to the length of probs, the number of columns equal to the order; rows are labelled with probabilities and columns with the dimension (1, 2, etc).

Method sample(): Draw and hold a random sample from the distribution.

Usage:

```
Distribution$sample(expected = FALSE)
```

Arguments:

expected If TRUE, sets the next value retrieved by a call to r() to be the mean of the distribution.

Returns: Void

Method r(): Return a random sample drawn from the distribution.

Usage:

```
Distribution$r()
```

Details: Returns the sample generated at the last call to `sample`.

Returns: A vector of length `K` representing one sample.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
Distribution$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Author(s)

Andrew J. Sims <andrew.sims@newcastle.ac.uk>

`Edge` *An edge in a graph*

Description

An R6 class representing an edge in a graph.

Details

Edges are the formal term for links between pairs of nodes in a graph. A base class.

Methods

Public methods:

- `Edge$new()`
- `Edge$is_same_edge()`
- `Edge$endpoints()`
- `Edge$label()`
- `Edge$clone()`

Method `new()`**:** Create an object of type `Edge`.

Usage:

```
Edge$new(v1, v2, label = "")
```

Arguments:

`v1` Node at one endpoint of the edge.

`v2` Node at the other endpoint of the edge.

`label` Character string containing the edge label.

Returns: A new `Edge` object.

Method `is_same_edge()`**:** Is this edge the same as the argument?

Usage:

```
Edge$is_same_edge(e)
```

Arguments:

`e` edge to compare with this one

Returns: `TRUE` if `e` is also this one.

Method `endpoints()`**:** Retrieve the endpoints of the edge.

Usage:

```
Edge$endpoints()
```

Returns: List of two nodes to which the edge is connected.

Method `label()`**:** Access label.

Usage:

```
Edge$label()
```

Returns: Label of the edge; character string.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
Edge$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Author(s)

Andrew J. Sims <andrew.sims@newcastle.ac.uk>

`EmpiricalDistribution` *An empirical distribution*

Description

An R6 class representing an empirical (1D) distribution.

Details

An object representing an empirical distribution. It inherits from class `Distribution`.

Super class

`rdecision::Distribution` -> `EmpiricalDistribution`

Methods

Public methods:

- EmpiricalDistribution$new()
- EmpiricalDistribution$distribution()
- EmpiricalDistribution$mean()
- EmpiricalDistribution$mode()
- EmpiricalDistribution$SD()
- EmpiricalDistribution$sample()
- EmpiricalDistribution$quantile()
- EmpiricalDistribution$clone()

Method `new()`**:** Create an object of class `EmpiricalDistribution`.

Usage:

```
EmpiricalDistribution$new(x, interpolate.sample = TRUE)
```

Arguments:

`x` a sample of at least 1 numerical value from the population of interest.

`interpolate.sample` Logical; if true, each call to `sample()` make a random draw from $U_{0,1}$ to find a p value, then finds that quantile of the sample, using the `quantile` function in R, via interpolation from the eCDF. If false, the `sample()` function makes a random draw from `x`.

Details: Empirical distributions based on very small sample sizes are supported, but not recommended.

Returns: An object of class `EmpiricalDistribution`.

Method `distribution()`**:** Accessor function for the name of the distribution.

Usage:

```
EmpiricalDistribution$distribution()
```

Returns: Distribution name as character string.

Method `mean()`**:** Return the expected value of the distribution.

Usage:

```
EmpiricalDistribution$mean()
```

Returns: Expected value as a numeric value.

Method `mode()`**:** Return the mode of the distribution,

Usage:

```
EmpiricalDistribution$mode()
```

Returns: NA because an empirical distribution is not guaranteed to be unimodal.

Method `SD()`**:** Return the standard deviation of the distribution.

Usage:

```
EmpiricalDistribution$SD()
```

Returns: Standard deviation as a numeric value

Method `sample()`**:** Draw and hold a random sample from the distribution.

Usage:

```
EmpiricalDistribution$sample(expected = FALSE)
```

Arguments:

`expected` If TRUE, sets the next value retrieved by a call to `r()` to be the mean of the distribution.

Details: Samples with interpolation or by random draw from the supplied distribution (see parameter `interpolate.sample` in `new()`).

Returns: Updated distribution.

Method `quantile()`**:** Return the quantiles of the empirical uncertainty distribution.

Usage:

```
EmpiricalDistribution$quantile(probs)
```

Arguments:

`probs` Vector of probabilities, in range [0,1].

Returns: Vector of quantiles.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
EmpiricalDistribution$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Author(s)

Andrew J. Sims <andrew.sims@newcastle.ac.uk>

`ExprModVar` *A model variable constructed from an expression of other variables*

Description

An R6 class representing a model variable constructed from an expression involving other variables.

Details

A class to support expressions involving objects of base class `ModVar`, which itself behaves like a model variable. For example, if `A` and `B` are variables with base class `ModVar` and `c` is a variable of type `numeric`, then it is not possible to write, for example, `x <- 42*A/B + c`, because R cannot manipulate class variables using the same operators as regular variables. But such forms of expression may be desirable in constructing a model and this class provides a mechanism for doing so. Inherits from class `ModVar`.

Super class

rdecision::ModVar -> ExprModVar

Methods

Public methods:

- ExprModVar$new()
- ExprModVar$add_method()
- ExprModVar$is_probabilistic()
- ExprModVar$operands()
- ExprModVar$distribution()
- ExprModVar$mean()
- ExprModVar$mode()
- ExprModVar$SD()
- ExprModVar$quantile()
- ExprModVar$mu_hat()
- ExprModVar$sigma_hat()
- ExprModVar$q_hat()
- ExprModVar$set()
- ExprModVar$get()
- ExprModVar$clone()

Method `new()`**:** Create a `ModVar` formed from an expression involving other model variables.

Usage:

```
ExprModVar$new(description, units, quo, nemp = 1000L)
```

Arguments:

`description` Name for the model variable expression. In a complex model it may help to tabulate how model variables are combined into costs, probabilities and rates.

`units` Units in which the variable is expressed.

`quo` A `quosure` (see package **rlang**), which contains an expression and its environment. The usage is `quo(x+y)` or `rlang::quo(x+y)`.

`nemp` sample size of the empirical distribution which will be associated with the expression, and used to estimate values for `mu_hat`, `sigma_hat` and `q_hat`.

Returns: An object of type `ExprModVar`

Method `add_method()`**:** Create a new `quosure` from that supplied in `new()` but with each `ModVar` operand appended with `$x` where `x` is the argument to this function.

Usage:

```
ExprModVar$add_method(method = "mean()")
```

Arguments:

`method` A character string with the method, e.g. `"mean()"`.

Details: This method is mostly intended for internal use within the class and will not generally be needed for normal use of `ExprModVar` objects. The returned expression is *not* syntactically checked or evaluated before it is returned.

Returns: A *quosure* whose expression is each `ModVar` `v` in the expression replaced with `v$method` and the same environment as specified in the quosure supplied in new().

Method `is_probabilistic()`**:** Tests whether the model variable is probabilistic, i.e. a random variable that follows a distribution, or an expression involving random variables, at least one of which follows a distribution.

Usage:

```
ExprModVar$is_probabilistic()
```

Returns: `TRUE` if probabilistic

Method `operands()`**:** Return a list of operands.

Usage:

```
ExprModVar$operands(recursive = TRUE)
```

Arguments:

`recursive` Whether to include nested variables in the list.

Details: Finds operands that are themselves `ModVars` in the expression. if `recursive=TRUE`, the list includes all `ModVars` that are operands of expression operands, recursively.

Returns: A list of model variables.

Method `distribution()`**:** Accessor function for the name of the expression model variable.

Usage:

```
ExprModVar$distribution()
```

Returns: Expression as a character string with all control characters having been removed.

Method `mean()`**:** Return the value of the expression when its operands take their mean value (i.e. value returned by call to `mean` or their value, if numeric). See notes on this class for further explanation.

Usage:

```
ExprModVar$mean()
```

Returns: Mean value as a numeric value.

Method `mode()`**:** Return the mode of the variable. By default returns `NA`, which will be the case for most `ExprModVar` variables, because an arbitrary expression is not guaranteed to be unimodal.

Usage:

```
ExprModVar$mode()
```

Returns: Mode as a numeric value.

Method `SD()`**:** Return the standard deviation of the distribution as `NA` because the variance is not available as a closed form for all functions of distributions.

Usage:

```
ExprModVar$SD()
```

Returns: Standard deviation as a numeric value

Method `quantile()`**:** Find quantiles of the uncertainty distribution. Not available as a closed form, and returned as `NA`.

Usage:

```
ExprModVar$quantile(probs)
```

Arguments:

`probs` Numeric vector of probabilities, each in range [0,1].

Returns: Vector of numeric values of the same length as `probs`.

Method `mu_hat()`**:** Return the estimated expected value of the variable.

Usage:

```
ExprModVar$mu_hat()
```

Details: This is computed by numerical simulation because there is, in general, no closed form expressions for the mean of a function of distributions. It is derived from the empirical distribution associated with the object.

Returns: Expected value as a numeric value.

Method `sigma_hat()`**:** Return the estimated standard deviation of the distribution.

Usage:

```
ExprModVar$sigma_hat()
```

Details: This is computed by numerical simulation because there is, in general, no closed form expressions for the SD of a function of distributions. It is derived from the empirical distribution associated with the object.

Returns: Standard deviation as a numeric value.

Method `q_hat()`**:** Return the estimated quantiles by sampling the variable.

Usage:

```
ExprModVar$q_hat(probs)
```

Arguments:

`probs` Vector of probabilities, in range [0,1].

Details: This is computed by numerical simulation because there is, in general, no closed form expressions for the quantiles of a function of distributions. The quantiles are derived from the empirical distribution associated with the object.

Returns: Vector of quantiles.

Method `set()`**:** Sets the value of the `ExprModVar`.

Usage:

```
ExprModVar$set(what = "random", val = NULL)
```

Arguments:

`what` Until `set` is called again, subsequent calls to `get` will return a value determined by the `what` parameter. as follows:

`"random"` a random sample is derived by taking a random sample from each of the operands and evaluating the expression. It does not draw from the empirical distribution because of the possibility of nested autocorrelation. For example, if $z = xy$, where x is a model variable and y is an expression which involves x, then y and x are correlated and will produce a different distribution for z than if x and y were independent. However, if z was sampled from the empirical distribution of y and the uncertainty distribution of x independently, the effect of correlation would be lost;

`"expected"` the value of the expression when each of its operands takes its expected value. This will not - in general - be the mean of the uncertainty distribution for the expression which can be estimated by calling `mu_hat`;

`"q2.5"` the value of the expression when each of its operands is equal to the 2.5th centile of their own uncertainty distribution. In general, this will be a more extreme value than the 2.5th centile of the uncertainty distribution of the expression, which can be found by using `q_hat(p=0.025)`;

`"q50"` as per `"q2.5"` but for the 50th centile (median);

`"q97.5"` as per `"q2.5"` but for the 97.5th centile;

`"current"` leaves the `what` parameter of method `set` unchanged *for each operand* and causes the expression to be re-evaluated at subsequent calls to `get`. Thus, after calling `set(what="current")` for the expression, subsequent calls to `get` for the expression may not return the same value, if method `set` has been called for one or more operands in the meantime;

`"value"` sets the value of the expression to be equal to parameter `val`. This is not recommended for normal usage because it allows the model variable to be set to an implausible value, based on its defined uncertainty. An example of where this may be needed is in threshold finding.

`val` A numeric value, only used with `what="value"`, ignored otherwise.

Details: The available options for parameter `what` are identical to those available for the `set` method of `ModVar`. However, because an `ExprModVar` represents the left hand side of an expression involving operands, the effect of some options is different from its effect on a non-expression `ModVar`.

Returns: Updated `ExprModVar`.

Method `get()`**:** Gets the value of the `ExprModVar` that was set by the most recent call to `set()`.

Usage:

```
ExprModVar$get()
```

Returns: Value determined by last `set()`.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
ExprModVar$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Note

For many expressions involving model variables there will be no closed form expressions for the mean, standard deviation and the quantiles. When an `ExprModVar` is created, an empirical distribution is generated by repeatedly drawing a random sample from each operand and evaluating the expression. The empirical distribution, which becomes associated with the object, is used to provide estimates of the mean, standard deviation and the quantiles via functions `mu_hat`, `sigma_hat` and `q_hat`.

For consistency with `ModVars` which are not expressions, the function `mean` returns the value of the expression when all its operands take their mean values. This will, in general, not be the mean of the expression distribution (which can be obtained via `mu_hat`), but is the value normally used in the base case of a model as the point estimate. As Briggs *et al* note (section 4.1.1) "in all but the most non-linear models, the difference between the expectation over the output of a probabilistic model and that model evaluated at the mean values of the input parameters, is likely to be modest."

Functions `SD`, `mode` and `quantile` return `NA` because they do not necessarily have a closed form. The standard deviation can be estimated by calling `sigma_hat` and the quantiles by `q_hat`. Because a unimodal distribution is not guaranteed, there is no estimator provided for the mode.

Method `distribution` returns the string representation of the expression used to create the model variable.

Author(s)

Andrew J. Sims <andrew.sims@newcastle.ac.uk>

References

Briggs A, Claxton K, Sculpher M. Decision modelling for health economic evaluation. Oxford, UK: Oxford University Press; 2006.

GammaDistribution *A parametrized Gamma distribution*

Description

An R6 class representing a Gamma distribution.

Details

An object representing a Gamma distribution with hyperparameters shape (`k`) and scale (`theta`). In econometrics this parametrization is more common but in Bayesian statistics the shape (`alpha`) and rate (`beta`) parametrization is more usual. Note, however, that although Briggs *et al* (2006) use the shape, scale formulation, they use `alpha`, `beta` as parameter names. Inherits from class `Distribution`.

Super class

`rdecision::Distribution` -> `GammaDistribution`

Methods

Public methods:

- GammaDistribution$new()
- GammaDistribution$distribution()
- GammaDistribution$mean()
- GammaDistribution$mode()
- GammaDistribution$SD()
- GammaDistribution$sample()
- GammaDistribution$quantile()
- GammaDistribution$clone()

Method `new()`**:** Create an object of class `GammaDistribution`.

Usage:

```
GammaDistribution$new(shape, scale)
```

Arguments:

`shape` shape parameter of the Gamma distribution.

`scale` scale parameter of the Gamma distribution.

Returns: An object of class `GammaDistribution`.

Method `distribution()`**:** Accessor function for the name of the distribution.

Usage:

```
GammaDistribution$distribution()
```

Returns: Distribution name as character string.

Method `mean()`**:** Return the expected value of the distribution.

Usage:

```
GammaDistribution$mean()
```

Returns: Expected value as a numeric value.

Method `mode()`**:** Return the mode of the distribution (if `shape` >= 1)

Usage:

```
GammaDistribution$mode()
```

Returns: mode as a numeric value.

Method `SD()`**:** Return the standard deviation of the distribution.

Usage:

```
GammaDistribution$SD()
```

Returns: Standard deviation as a numeric value

Method `sample()`**:** Draw and hold a random sample from the distribution.

Usage:

```
GammaDistribution$sample(expected = FALSE)
```

Arguments:

`expected` If TRUE, sets the next value retrieved by a call to `r()` to be the mean of the distribution.

Returns: Updated distribution.

Method `quantile()`**:** Return the quantiles of the Gamma uncertainty distribution.

Usage:

```
GammaDistribution$quantile(probs)
```

Arguments:

`probs` Vector of probabilities, in range [0,1].

Returns: Vector of quantiles.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
GammaDistribution$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Author(s)

Andrew J. Sims <andrew.sims@newcastle.ac.uk>

References

Briggs A, Claxton K, Sculpher M. Decision modelling for health economic evaluation. Oxford, UK: Oxford University Press; 2006.

`GammaModVar` *A model variable whose uncertainty follows a Gamma distribution*

Description

An R6 class for a model variable with Gamma uncertainty.

Details

A model variable for which the uncertainty in the point estimate can be modelled with a Gamma distribution. The hyperparameters of the distribution are the shape (`k`) and the scale (`theta`). Note that although Briggs *et al* (2006) use the shape, scale formulation, they use `alpha`, `beta` as parameter names. Inherits from class `ModVar`.

Super class

`rdecision::ModVar` -> `GammaModVar`

Methods

Public methods:

- GammaModVar$new()
- GammaModVar$is_probabilistic()
- GammaModVar$clone()

Method `new()`**:** Create an object of class `GammaModVar`.

Usage:

```
GammaModVar$new(description, units, shape, scale)
```

Arguments:

`description` A character string describing the variable.

`units` Units of the variable, as character string.

`shape` shape parameter of the Gamma distribution.

`scale` scale parameter of the Gamma distribution.

Returns: An object of class `GammaModVar`.

Method `is_probabilistic()`**:** Tests whether the model variable is probabilistic, i.e., a random variable that follows a distribution, or an expression involving random variables, some of which follow distributions.

Usage:

```
GammaModVar$is_probabilistic()
```

Returns: `TRUE` if probabilistic

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
GammaModVar$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Note

The Gamma model variable class can be used to model the uncertainty of the mean of a count quantity which follows a Poisson distribution. The Gamma distribution is the conjugate prior of a Poisson distribution, and the shape and scale relate directly to the number of intervals from which the mean count has been estimated. Specifically, the shape (k) is equal to the total count of events in $1/\theta$ intervals, where θ is the scale. For example, if 200 counts were observed in a sample of 100 intervals, setting `shape=200` and `scale=1/100` gives a Gamma distribution with a mean of 2 and a 95% confidence interval from 1.73 to 2.29.

Author(s)

Andrew J. Sims <andrew.sims@newcastle.ac.uk>

References

Briggs A, Claxton K, Sculpher M. Decision modelling for health economic evaluation. Oxford, UK: Oxford University Press; 2006.

Graph *An undirected graph*

Description

An R6 class to represent a graph (from discrete mathematics).

Details

Encapsulates and provides methods for computation and checking of undirected graphs. Graphs are systems of vertices connected in pairs by edges. A base class.

Methods

Public methods:

- Graph$new()
- Graph$order()
- Graph$size()
- Graph$vertexes()
- Graph$vertex_along()
- Graph$vertex_index()
- Graph$vertex_at()
- Graph$has_vertex()
- Graph$vertex_label()
- Graph$edges()
- Graph$edge_along()
- Graph$edge_index()
- Graph$edge_at()
- Graph$has_edge()
- Graph$edge_label()
- Graph$graph_adjacency_matrix()
- Graph$is_simple()
- Graph$is_connected()
- Graph$is_acyclic()
- Graph$is_tree()
- Graph$degree()
- Graph$neighbours()
- Graph$as_DOT()
- Graph$clone()

Method `new()`**:** Create a new `Graph` object from sets of nodes and edges.

Usage:

```
Graph$new(V, E)
```

Arguments:

V An unordered set of Nodes, as a list.

E An unordered set of Edges, as a list.

Returns: A Graph object.

Method order(): Return the order of the graph (number of vertices).

Usage:

```
Graph$order()
```

Returns: Order of the graph (integer).

Method size(): Return the size of the graph (number of edges).

Usage:

```
Graph$size()
```

Returns: Size of the graph (integer).

Method vertexes(): A list of all the Node objects in the graph.

Usage:

```
Graph$vertexes()
```

Details: The list of Node objects is returned in the same order as their indexes understood by vertex_index, vertex_at and vertex_along, which is not necessarily the same order in which they were supplied in the V argument to new.

Method vertex_along(): Sequence of vertex indices.

Usage:

```
Graph$vertex_along()
```

Details: Similar to base::seq_along, this function provides the indices of the vertices in the graph. It is intended for use by graph algorithms which iterate vertices.

Returns: A numeric vector of indices from 1 to the order of the graph. The vertex at index i is not guaranteed to be the same vertex at V[[i]] of the argument V to new (i.e., the order in which the vertices are stored internally within the class may differ from the order in which they were supplied).

Method vertex_index(): Find the index of a vertex in the graph.

Usage:

```
Graph$vertex_index(v)
```

Arguments:

v A vertex, or list of vertexes.

Returns: Index of *v*. The index of vertex v is the one used internally to the class object, which is not necessarily the same as the order of vertices in the V argument of new. NA if *v* is not a vertex, or is a vertex that is not in the graph.

Method vertex_at(): Find the vertex at a given index.

Usage:

```
Graph$vertex_at(index, as_list = FALSE)
```

Arguments:

`index` Index of vertex in the graph, as an integer, or vector of integers.

`as_list` Boolean. If TRUE the method returns list of Nodes, even if the length of `index` is 1.

Details: The inverse of function `vertex_index`. The function will raise an abort signal if all the supplied indexes are not vertexes. The function is vectorized, but for historical compatibility the return object is a single `Node` if `index` is a scalar. The return object can be guaranteed to be a list if `as_list` is set.

Returns: Node at `index` if `index` is a scalar, a list of Nodes at the values of `index` if `index` is a vector, or an empty list if index is an empty array.

Method `has_vertex()`**:** Test whether a vertex is an element of the graph.

Usage:

```
Graph$has_vertex(v)
```

Arguments:

`v` Subject vertex.

Returns: TRUE if v is an element of V(G).

Method `vertex_label()`**:** Find label of vertexes at index i.

Usage:

```
Graph$vertex_label(iv)
```

Arguments:

`iv` Index of vertex, or vector of indexes.

Returns: Label(s) of vertex at index i

Method `edges()`**:** A list of all the Edge objects in the graph.

Usage:

```
Graph$edges()
```

Details: The list of Edge objects is returned in the same order as their indexes understood by `edge_index`, `edge_at` and `edge_along`, which is not necessarily the same order in which they were supplied in the `E` argument to `new`.

Method `edge_along()`**:** Sequence of edge indices.

Usage:

```
Graph$edge_along()
```

Details: Similar to `base::seq_along`, this function provides the indices of the edges in the graph. It is intended for use by graph algorithms which iterate edges. It is equivalent to `seq_along(g$edges())`, where `g` is a graph.

Returns: A numeric vector of indices from 1 to the size of the graph. The edge at index i is not guaranteed to be the same edge at `E[[i]]` of the argument `E` to `new` (i.e., the order in which the edges are stored internally within the class may differ from the order in which they were supplied).

Method `edge_index()`**:** Find the index of an edge in a graph.

Usage:

```
Graph$edge_index(e)
```

Arguments:

`e` An edge object, or list of edge objects.

Details: The index of edge `e` is the one used internally to the class object, which is not necessarily the same as the order of edges in the `E` argument of `new`.

Returns: Index of `e`. `NA` if *e* is not an edge, or is an edge that is not in the graph.

Method `edge_at()`**:** Find the edge at a given index.

Usage:

```
Graph$edge_at(index, as_list = FALSE)
```

Arguments:

`index` Index of edge in the graph, as an integer, vector of integers, or list of integers.

`as_list` Boolean. If TRUE the method returns list of Edges, even if the length of `index` is 1.

Details: The inverse of function `edge_index`. The function will raise an abort signal if the supplied index is not an edge. The function is vectorized, but for historical compatibility the return object is a single `Edge` if `index` is a scalar. The return object can be guaranteed to be a list if `as_list` is set.

Returns: The edge, or list of edges, with the specified index.

Method `has_edge()`**:** Test whether an edge is an element of the graph.

Usage:

```
Graph$has_edge(e)
```

Arguments:

`e` Edge or list of edges.

Returns: Logical vector with each element `TRUE` if the corresponding element of `e` is an element of $E(G)$.

Method `edge_label()`**:** Find label of edge at index i

Usage:

```
Graph$edge_label(ie)
```

Arguments:

`ie` Index of edge, or vector of indexes.

Returns: Label of edge at index i, or character vector with the labels at indexes `ie`.

Method `graph_adjacency_matrix()`**:** Compute the adjacency matrix for the graph.

Usage:

```
Graph$graph_adjacency_matrix(boolean = FALSE)
```

Arguments:

`boolean` If `TRUE`, the adjacency matrix is logical, each cell is {`FALSE`, `TRUE`}.

Details: Each cell contains the number of edges joining the two vertexes, with the convention of self loops being counted twice, unless `binary` is `TRUE` when cells are either 0 (not adjacent) or 1 (adjacent).

Returns: A square integer matrix with the number of rows and columns equal to the order of the graph. The rows and columns are labelled with the node labels, if all the nodes in the graph have unique labels, or the node indices if not.

Method `is_simple()`**:** Is this a simple graph?

Usage:

```
Graph$is_simple()
```

Details: A simple graph has no self loops or multi-edges.

Returns: `TRUE` if simple, `FALSE` if not.

Method `is_connected()`**:** Test whether the graph is connected.

Usage:

```
Graph$is_connected()
```

Details: Graphs with no vertices are considered unconnected; graphs with 1 vertex are considered connected. Otherwise a graph is connected if all nodes can be reached from an arbitrary starting point. Uses a depth first search.

Returns: `TRUE` if connected, `FALSE` if not.

Method `is_acyclic()`**:** Checks for the presence of a cycle in the graph.

Usage:

```
Graph$is_acyclic()
```

Details: Uses a depth-first search from each node to detect the presence of back edges. A back edge is an edge from the current node joining a previously detected (visited) node, that is not the parent node of the current one.

Returns: `TRUE` if no cycles detected.

Method `is_tree()`**:** Compute whether the graph is connected and acyclic.

Usage:

```
Graph$is_tree()
```

Returns: `TRUE` if the graph is a tree; `FALSE` if not.

Method `degree()`**:** The degree of a vertex in the graph.

Usage:

```
Graph$degree(v)
```

Arguments:

`v` The subject node.

Details: The number of incident edges.

Returns: Degree of the vertex, integer.

Method `neighbours()`**:** Find the neighbours of a node.

Usage:

```
Graph$neighbours(v)
```

Arguments:

`v` The subject node (scalar, not a list).

Details: A property of the graph, not the node. Does not include self, even in the case of a loop to self.

Returns: A list of nodes which are joined to the subject.

Method `as_DOT()`**:** Export a representation of the graph in DOT format.

Usage:

```
Graph$as_DOT()
```

Details: Writes the representation in the `graphviz` DOT language (https://graphviz.org/doc/info/lang.html) for drawing with one of the `graphviz` tools including `dot` (Gansner, 1993).

Returns: A character vector. Intended for passing to `writeLines` for saving as a text file.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
Graph$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Author(s)

Andrew Sims <andrew.sims@newcastle.ac.uk>

References

Gansner ER, Koutsofios E, North SC, Vo K-P. A technique for drawing directed graphs. *IEEE Transactions on Software Engineering*, 1993;**19**:214–30, doi:10.1109/32.221135.

Gross JL, Yellen J, Zhang P. Handbook of Graph Theory. Second edition, Chapman and Hall/CRC.; 2013, doi:10.1201/b16132

LeafNode *A leaf node in a decision tree*

Description

An R6 class representing a leaf (terminal) node in a decision tree.

Details

Represents a terminal state in a tree, and is associated with an incremental utility. Inherits from class `Node`.

Super class

rdecision::Node -> LeafNode

Methods

Public methods:

- LeafNode$new()
- LeafNode$modvars()
- LeafNode$set_utility()
- LeafNode$utility()
- LeafNode$interval()
- LeafNode$QALY()
- LeafNode$clone()

Method new(): Create a new LeafNode object; synonymous with a clinical outcome.

Usage:

```
LeafNode$new(
  label,
  utility = 1,
  interval = as.difftime(365.25, units = "days")
)
```

Arguments:

label Character string; a label for the state; must be defined because it is used in tabulations. The label is automatically converted to a syntactically valid (in R) name to ensure it can be used as a column name in a data frame.

utility The incremental utility that a user associates with being in the health state for the interval. Intended for use with cost benefit analysis. Can be numeric or a type of ModVar. If the type is numeric, the allowed range is -Inf to 1; if it is of type ModVar, it is unchecked.

interval The time interval over which the utility parameter applies, expressed as an R difftime object; default 1 year.

Returns: A new LeafNode object

Method modvars(): Find all the model variables of type ModVar that have been specified as values associated with this LeafNode. Includes operands of these ModVars, if they are expressions.

Usage:

```
LeafNode$modvars()
```

Returns: A list of ModVars.

Method set_utility(): Set the incremental utility associated with the node.

Usage:

```
LeafNode$set_utility(utility)
```

Arguments:

utility The incremental utility that a user associates with being in the health state for the interval. Intended for use with cost benefit analysis. Can be numeric or a type of ModVar. If the type is numeric, the allowed range is -Inf to 1; if it is of type ModVar, it is unchecked.

Returns: Updated `Leaf` object.

Method `utility()`**:** Return the incremental utility associated with being in the state for the interval.

Usage:

```
LeafNode$utility()
```

Returns: Incremental utility (numeric value).

Method `interval()`**:** Return the interval associated with being in the state.

Usage:

```
LeafNode$interval()
```

Returns: Interval (as a `difftime`).

Method `QALY()`**:** Return the quality adjusted life years associated with being in the state.

Usage:

```
LeafNode$QALY()
```

Returns: QALY.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
LeafNode$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Author(s)

Andrew J. Sims <andrew.sims@newcastle.ac.uk>

`LogNormDistribution` *A parametrized log Normal probability distribution*

Description

An R6 class representing a log Normal distribution.

Details

A parametrized Log Normal distribution inheriting from class `Distribution`. Swat (2017) defined seven parametrizations of the log normal distribution. These are linked, allowing the parameters of any one to be derived from any other. All 7 parametrizations require two parameters as follows:

LN1 $p_1 = \mu, p_2 = \sigma$, where μ and σ are the mean and standard deviation, both on the log scale.

LN2 $p_1 = \mu, p_2 = v$, where μ and v are the mean and variance, both on the log scale.

LN3 $p_1 = m$, $p_2 = \sigma$, where m is the median on the natural scale and σ is the standard deviation on the log scale.

LN4 $p_1 = m$, $p_2 = c_v$, where m is the median on the natural scale and c_v is the coefficient of variation on the natural scale.

LN5 $p_1 = \mu$, $p_2 = \tau$, where μ is the mean on the log scale and τ is the precision on the log scale.

LN6 $p_1 = m$, $p_2 = \sigma_g$, where m is the median on the natural scale and σ_g is the geometric standard deviation on the natural scale.

LN7 $p_1 = \mu_N$, $p_2 = \sigma_N$, where μ_N is the mean on the natural scale and σ_N is the standard deviation on the natural scale.

Super class

`rdecision::Distribution` -> `LogNormDistribution`

Methods

Public methods:

- `LogNormDistribution$new()`
- `LogNormDistribution$distribution()`
- `LogNormDistribution$sample()`
- `LogNormDistribution$mean()`
- `LogNormDistribution$mode()`
- `LogNormDistribution$SD()`
- `LogNormDistribution$quantile()`
- `LogNormDistribution$clone()`

Method `new()`**:** Create a log normal distribution.

Usage:

```
LogNormDistribution$new(p1, p2, parametrization = "LN1")
```

Arguments:

`p1` First hyperparameter, a measure of location. See *Details*.

`p2` Second hyperparameter, a measure of spread. See *Details*.

`parametrization` A character string taking one of the values `"LN1"` (default) through `"LN7"` (see *Details*).

Returns: A `LogNormDistribution` object.

Method `distribution()`**:** Accessor function for the name of the distribution.

Usage:

```
LogNormDistribution$distribution()
```

Returns: Distribution name as character string (`"LN1"`, `"LN2"` etc.).

Method `sample()`**:** Draw a random sample from the model variable.

Usage:

```
LogNormDistribution$sample(expected = FALSE)
```

Arguments:

`expected` If TRUE, sets the next value retrieved by a call to `r()` to be the mean of the distribution.

Returns: Updated `LogNormDistribution` object.

Method `mean()`**:** Return the expected value of the distribution.

Usage:

```
LogNormDistribution$mean()
```

Returns: Expected value as a numeric value.

Method `mode()`**:** Return the point estimate of the variable.

Usage:

```
LogNormDistribution$mode()
```

Returns: Point estimate (mode) of the log normal distribution.

Method `SD()`**:** Return the standard deviation of the distribution.

Usage:

```
LogNormDistribution$SD()
```

Returns: Standard deviation as a numeric value

Method `quantile()`**:** Return the quantiles of the log normal distribution.

Usage:

```
LogNormDistribution$quantile(probs)
```

Arguments:

`probs` Vector of probabilities, in range [0,1].

Returns: Vector of quantiles.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
LogNormDistribution$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Note

The log normal distribution may be used to model the uncertainty in an estimate of relative risk (Briggs 2006, p90). If a relative risk estimate is available with a 95% confidence interval, the `"LN7"` parametrization allows the uncertainty distribution to be specified directly. For example, if RR = 0.67 with 95% confidence interval 0.53 to 0.84 (Leaper, 2016), it can be modelled with `LogNormModVar$new("rr", "RR", p1=0.67,p2=(0.84-0.53)/(2*1.96)), "LN7")`.

Author(s)

Andrew J. Sims <andrew.sims@newcastle.ac.uk>

References

Briggs A, Claxton K and Sculpher M. Decision Modelling for Health Economic Evaluation. Oxford 2006, ISBN 978-0-19-852662-9.

Leaper DJ, Edmiston CE and Holy CE. Meta-analysis of the potential economic impact following introduction of absorbable antimicrobial sutures. *British Journal of Surgery* 2017;**104**:e134-e144.

Swat MJ, Grenon P and Wimalaratne S. Ontology and Knowledge Base of Probability Distributions. *Bioinformatics* 2016;**32**:2719-2721, doi:10.1093/bioinformatics/btw170.

`LogNormModVar` *A model variable whose uncertainty follows a log Normal distribution*

Description

An R6 class representing a model variable with log Normal uncertainty.

Details

A model variable for which the uncertainty in the point estimate can be modelled with a log Normal distribution. One of seven parametrizations defined by Swat *et al* can be used. Inherits from `ModVar`.

Super class

`rdecision::ModVar` -> `LogNormModVar`

Methods

Public methods:

- `LogNormModVar$new()`
- `LogNormModVar$is_probabilistic()`
- `LogNormModVar$clone()`

Method `new()`**:** Create a model variable with log normal uncertainty.

Usage:

```
LogNormModVar$new(description, units, p1, p2, parametrization = "LN1")
```

Arguments:

`description` A character string describing the variable.

`units` Units of the quantity; character string.

`p1` First hyperparameter, a measure of location. See *Details*.

`p2` Second hyperparameter, a measure of spread. See *Details*.

`parametrization` A character string taking one of the values `"LN1"` (default) through `"LN7"` (see *Details*).

Returns: A `LogNormModVar` object.

Method `is_probabilistic()`**:** Tests whether the model variable is probabilistic, i.e., a random variable that follows a distribution, or an expression involving random variables, some of which follow distributions.

Usage:

```
LogNormModVar$is_probabilistic()
```

Returns: `TRUE` if probabilistic

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
LogNormModVar$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Author(s)

Andrew J. Sims <andrew.sims@newcastle.ac.uk>

References

Briggs A, Claxton K and Sculpher M. Decision Modelling for Health Economic Evaluation. Oxford 2006, ISBN 978-0-19-852662-9.

Leaper DJ, Edmiston CE and Holy CE. Meta-analysis of the potential economic impact following introduction of absorbable antimicrobial sutures. *British Journal of Surgery* 2017;**104**:e134-e144.

Swat MJ, Grenon P and Wimalaratne S. Ontology and Knowledge Base of Probability Distributions. *Bioinformatics* 2016;**32**:2719-2721, doi:10.1093/bioinformatics/btw170.

See Also

`LogNormDistribution`.

`MarkovState` *A state in a Markov model*

Description

An R6 class representing a state in a Markov model.

Details

Represents a single state in a Markov model. A Markov model is a digraph in which states are nodes and transitions are arrows. Inherits from class `Node`.

Value

Updated `MarkovState` object

Super class

rdecision::Node -> MarkovState

Methods

Public methods:

- MarkovState$new()
- MarkovState$name()
- MarkovState$set_cost()
- MarkovState$cost()
- MarkovState$utility()
- MarkovState$modvars()
- MarkovState$clone()

Method `new()`**:** Create an object of type `MarkovState`.

Usage:

```
MarkovState$new(name, cost = 0, utility = 1)
```

Arguments:

`name` The name of the state (character string).

`cost` The annual cost of state occupancy (numeric or `ModVar`). Default 0.0.

`utility` The utility associated with being in the state (numeric or `ModVar`).

Details: Utility must be in the range `[-Inf,1]`. If it is of type numeric, the range is checked on object creation.

Returns: An object of type `MarkovState`.

Method `name()`**:** Accessor function to retrieve the state name.

Usage:

```
MarkovState$name()
```

Returns: State name.

Method `set_cost()`**:** Set the annual occupancy cost

Usage:

```
MarkovState$set_cost(cost)
```

Arguments:

`cost` The annual cost of state occupancy

Method `cost()`**:** Gets the annual cost of state occupancy.

Usage:

```
MarkovState$cost()
```

Returns: Annual cost; numeric.

Method `utility()`**:** Gets the utility associated with the state.

Usage:

`MarkovState$utility()`

Returns: Utility; numeric.

Method `modvars()`**:** Find all the model variables.

Usage:

`MarkovState$modvars()`

Details: Find variables of type `ModVar` that have been specified as values associated with this `MarkovState`. Includes operands of these `ModVars`, if they are expressions.

Returns: A list of `ModVars`.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
MarkovState$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Author(s)

Andrew J. Sims <andrew.sims@newcastle.ac.uk>

`ModVar` *A model variable incorporating uncertainty*

Description

An R6 class for a variable in a health economic model.

Details

Base class for a variable used in a health economic model. The base class wraps a numerical value which is used in calculations. It provides a framework for creating classes of model variables whose uncertainties are described by statistical distributions parametrized with hyperparameters.

Methods

Public methods:

- `ModVar$new()`
- `ModVar$is_expression()`
- `ModVar$is_probabilistic()`
- `ModVar$description()`
- `ModVar$units()`
- `ModVar$distribution()`
- `ModVar$mean()`
- `ModVar$mode()`

- ModVar$SD()
- ModVar$quantile()
- ModVar$r()
- ModVar$set()
- ModVar$get()
- ModVar$clone()

Method `new()`**:** Create an object of type `ModVar`.

Usage:

```
ModVar$new(description, units, D = NULL, k = 1L)
```

Arguments:

`description` A character string description of the variable and its role in the model. This description will be used in a tabulation of the variables linked to a model.

`units` A character string description of the units, e.g. `"GBP"`, `"per year"`.

`D` The distribution representing the uncertainty in the variable. Should inherit from class `Distribution`, or NULL if none is defined.

`k` The index of the dimension of the multivariate distribution that applies to this model variable.

Details: A `ModVar` is associated with an uncertainty distribution (a "has-a" relationship in object-oriented terminology). There can be a 1-1 mapping of `ModVars` to `Distributions`, or several model variables can be linked to the same distribution in a many-1 mapping, e.g. when each transition probability from a Markov state is represented as a `ModVar` and each can be linked to the `k` dimensions of a common multivariate Dirichlet distribution.

Returns: A new `ModVar` object.

Method `is_expression()`**:** Is this `ModVar` an expression?

Usage:

```
ModVar$is_expression()
```

Returns: `TRUE` if it inherits from `ExprModVar`, `FALSE` otherwise.

Method `is_probabilistic()`**:** Is the model variable probabilistic?

Usage:

```
ModVar$is_probabilistic()
```

Details: Tests whether the model variable is probabilistic, i.e. a random variable that follows a distribution, or an expression involving random variables, some of which follow distributions.

Returns: `TRUE` if probabilistic

Method `description()`**:** Accessor function for the description.

Usage:

```
ModVar$description()
```

Returns: Description of model variable as character string.

Method `units()`**:** Accessor function for units.

Usage:

```
ModVar$units()
```

Returns: Description of units as character string.

Method `distribution()`**:** Name and parameters of the uncertainty distribution.

Usage:

```
ModVar$distribution()
```

Details: If $K > 1$ the dimension of the distribution associated with this model variable is appended, e.g. `Dir(2,3)[1]` means that the model variable is associated with the first dimension of a 2D Dirichlet distribution with alpha parameters 2 and 3.

Returns: Distribution name as character string.

Method `mean()`**:** Mean value of the model variable.

Usage:

```
ModVar$mean()
```

Returns: Mean value as a numeric value.

Method `mode()`**:** The mode of the variable.

Usage:

```
ModVar$mode()
```

Details: By default returns `NA`, which will be the case for most `ModVar` variables, because arbitrary distributions are not guaranteed to be unimodal.

Returns: Mode as a numeric value.

Method `SD()`**:** Standard deviation of the model variable.

Usage:

```
ModVar$SD()
```

Returns: Standard deviation as a numeric value

Method `quantile()`**:** Quantiles of the uncertainty distribution.

Usage:

```
ModVar$quantile(probs)
```

Arguments:

`probs` Numeric vector of probabilities, each in range [0,1].

Returns: Vector of numeric values of the same length as `probs`.

Method `r()`**:** Draw a random sample from the model variable.

Usage:

```
ModVar$r()
```

Details: The same random sample will be returned until `set` is called to force a resample.

Returns: A sample drawn at random.

Method `set()`**:** Sets the value of the `ModVar`.

Usage:

```
ModVar$set(what = "random", val = NULL)
```

Arguments:

`what` Until `set` is called again, subsequent calls to `get` will return a value determined by the `what` parameter as follows:

- `"random"` a random sample is drawn from the uncertainty distribution;
- `"expected"` the mean of the uncertainty distribution;
- `"q2.5"` the lower 95% confidence limit of the uncertainty distribution, i.e. the 2.5th percentile;
- `"q50"` the median of the uncertainty distribution;
- `"q97.5"` the upper 95% confidence limit of the uncertainty distribution, i.e. the 97.5th percentile;
- `"current"` leaves the `what` parameter of method `set` unchanged, i.e. the call to `set` has no effect on the subsequent values returned by `get`. It is provided as an option to help use cases in which the `what` parameter is a variable;
- `"value"` sets the value explicitly to be equal to parameter `val`. This is not recommended for normal usage because it allows the model variable to be set to an implausible value, based on its defined uncertainty. An example of where this may be needed is in threshold finding.

`val` A numeric value, only used with `what="value"`, ignored otherwise.

Returns: Updated `ModVar`.

Method `get()`**:** Get the value of the `ModVar`.

Usage:

```
ModVar$get()
```

Details: Returns the value defined by the most recent call to `set()`.

Returns: Value determined by last `set()`.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
ModVar$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Author(s)

Andrew Sims <andrew.sims@newcastle.ac.uk>

Node *A node in a graph*

Description

An R6 class representing a node in a graph.

Details

A base class to represent a single node in a graph.

Methods

Public methods:

- `Node$new()`
- `Node$label()`
- `Node$type()`
- `Node$clone()`

Method `new()`**:** Create new `Node` object.

Usage:

```
Node$new(label = "")
```

Arguments:

`label` An optional label for the node.

Returns: A new `Node` object.

Method `label()`**:** Return the label of the node.

Usage:

```
Node$label()
```

Returns: Label as a character string.

Method `type()`**:** node type

Usage:

```
Node$type()
```

Returns: `Node` class, as character string.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
Node$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Author(s)

Andrew Sims <andrew.sims@newcastle.ac.uk>

`NormalDistribution` *A parametrized Normal distribution*

Description

An R6 class representing a parametrized Normal distribution.

Details

A Normal distribution with hyperparameters mean (`mu`) and standard deviation (`sd`). Inherits from class `Distribution`.

Super class

`rdecision::Distribution` -> `NormalDistribution`

Methods

Public methods:

- `NormalDistribution$new()`
- `NormalDistribution$distribution()`
- `NormalDistribution$sample()`
- `NormalDistribution$mean()`
- `NormalDistribution$SD()`
- `NormalDistribution$quantile()`
- `NormalDistribution$clone()`

Method `new()`**:** Create a parametrized normal distribution.

Usage:

```
NormalDistribution$new(mu, sigma)
```

Arguments:

`mu` Mean of the Normal distribution.

`sigma` Standard deviation of the Normal distribution.

Returns: A `NormalDistribution` object.

Method `distribution()`**:** Accessor function for the name of the distribution.

Usage:

```
NormalDistribution$distribution()
```

Returns: Distribution name as character string.

Method `sample()`**:** Draw a random sample from the model variable.

Usage:

```
NormalDistribution$sample(expected = FALSE)
```

Arguments:

`expected` If TRUE, sets the next value retrieved by a call to `r()` to be the mean of the distribution.

Returns: A sample drawn at random.

Method `mean()`**:** Return the mean value of the distribution.

Usage:

```
NormalDistribution$mean()
```

Returns: Expected value as a numeric value.

Method `SD()`**:** Return the standard deviation of the distribution.

Usage:

```
NormalDistribution$SD()
```

Returns: Standard deviation as a numeric value

Method `quantile()`**:** Return the quantiles of the Normal uncertainty distribution.

Usage:

```
NormalDistribution$quantile(probs)
```

Arguments:

`probs` Vector of probabilities, in range [0,1].

Returns: Vector of quantiles.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
NormalDistribution$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Author(s)

Andrew J. Sims <andrew.sims@newcastle.ac.uk>

`NormModVar` *A model variable whose uncertainty follows a Normal distribution*

Description

An R6 class representing a model variable with Normal uncertainty.

Details

A model variable for which the uncertainty in its point estimate can be modelled with a Normal distribution. The hyperparameters of the distribution are the mean (`mu`) and the standard deviation (`sd`) of the uncertainty distribution. The value of `mu` is the expected value of the variable. Inherits from class `ModVar`.

Super class

rdecision::ModVar -> NormModVar

Methods

Public methods:

- NormModVar$new()
- NormModVar$is_probabilistic()
- NormModVar$clone()

Method `new()`**:** Create a model variable with normal uncertainty.

Usage:

```
NormModVar$new(description, units, mu, sigma)
```

Arguments:

`description` A character string describing the variable.

`units` Units of the quantity; character string.

`mu` Hyperparameter with mean of the Normal distribution for the uncertainty of the variable.

`sigma` Hyperparameter equal to the standard deviation of the normal distribution for the uncertainty of the variable.

Returns: A `NormModVar` object.

Method `is_probabilistic()`**:** Tests whether the model variable is probabilistic.

Usage:

```
NormModVar$is_probabilistic()
```

Returns: `TRUE` if probabilistic.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
NormModVar$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Author(s)

Andrew J. Sims <andrew.sims@newcastle.ac.uk>

Reaction *A reaction (chance) edge in a decision tree*

Description

An R6 class representing a reaction (chance) edge in a decision tree.

Details

A specialism of class Arrow which is used in a decision tree to represent edges whose source nodes are ChanceNodes.

Super classes

rdecision::Edge -> rdecision::Arrow -> Reaction

Methods

Public methods:

- Reaction$new()
- Reaction$modvars()
- Reaction$set_probability()
- Reaction$p()
- Reaction$set_cost()
- Reaction$cost()
- Reaction$set_benefit()
- Reaction$benefit()
- Reaction$clone()

Method new(): Create an object of type Reaction. A probability must be assigned to the edge. Optionally, a cost and a benefit may be associated with traversing the edge. A *pay-off* (benefit-cost) is sometimes used in edges of decision trees; the parametrization used here is more general.

Usage:

```
Reaction$new(source_node, target_node, p, cost = 0, benefit = 0, label = "")
```

Arguments:

source_node Chance node from which the reaction leaves.

target_node Node which the reaction enters.

p Conditional probability of traversing the reaction edge.

cost Cost associated with traversal of this edge (numeric or ModVar).

benefit Benefit associated with traversal of the edge (numeric or ModVar).

label Character string containing the reaction label.

Returns: A new Reaction object.

Method `modvars()`**:** Find all the model variables of type `ModVar` that have been specified as values associated with this Action. Includes operands of these `ModVars`, if they are expressions.

Usage:

```
Reaction$modvars()
```

Returns: A list of `ModVars`.

Method `set_probability()`**:** Set the probability associated with the reaction edge.

Usage:

```
Reaction$set_probability(p)
```

Arguments:

`p` Conditional probability of traversing the reaction edge. Of type numeric or `ModVar`. If numeric, `p` must be in the range [0,1].

Returns: Updated `Reaction` object.

Method `p()`**:** Return the current value of the edge probability, i.e. the conditional' probability of traversing the edge.

Usage:

```
Reaction$p()
```

Returns: Numeric value in range [0,1].

Method `set_cost()`**:** Set the cost associated with the reaction edge.

Usage:

```
Reaction$set_cost(c = 0)
```

Arguments:

`c` Cost associated with traversing the reaction edge. Of type numeric or `ModVar`.

Returns: Updated `Reaction` object.

Method `cost()`**:** Return the cost associated with traversing the edge.

Usage:

```
Reaction$cost()
```

Returns: Cost.

Method `set_benefit()`**:** Set the benefit associated with the reaction edge.

Usage:

```
Reaction$set_benefit(b = 0)
```

Arguments:

`b` Benefit associated with traversing the reaction edge. Of type numeric or `ModVar`.

Returns: Updated `Action` object.

Method `benefit()`**:** Return the benefit associated with traversing the edge.

Usage:

```
Reaction$benefit()
```

Returns: Benefit.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
Reaction$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Author(s)

Andrew J. Sims <andrew.sims@newcastle.ac.uk>

`SemiMarkovModel` *A semi-Markov model for cohort simulation*

Description

An R6 class representing a semi-Markov model for cohort simulation.

Details

A class to represent a continuous time semi-Markov chain, modelled using cohort simulation. As interpreted in **rdecision**, semi-Markov models may include temporary states and transitions are defined by per-cycle probabilities. Although used widely in health economic modelling, the differences between semi-Markov models and Markov processes introduce some caveats for modellers:

- If there are temporary states, the result will depend on cycle length.
- Transitions are specified by their conditional probability, which is a *per-cycle* probability of starting a cycle in one state and ending it in another; if the cycle length changes, the probabilities should change, too.
- Probabilities and rates cannot be linked by the Kolmogorov forward equation, where the per-cycle probabilities are given by the matrix exponential of the transition rate matrix, because this equation does not apply if there are temporary states. In creating semi-Markov models, it is the modeller's task to estimate probabilities from published data on event rates.
- The cycle time cannot be changed during the simulation.

Graph theory

A Markov model is a directed multidigraph permitting loops (a loop multidigraph), optionally labelled, or a *quiver*. It is a multidigraph because there are potentially two edges between each pair of nodes {A,B} representing the transition probabilities from A to B and *vice versa*. It is a directed graph because the transition probabilities refer to transitions in one direction. Each edge can be optionally labelled. It permits self-loops (edges whose source and target are the same node) to represent patients that remain in the same state between cycles.

Transition rates and probabilities

Why semi-Markov?: Beck and Pauker (1983) and later Sonnenberg and Beck (1993) proposed the use of Markov processes to model the health economics of medical interventions. Further, they introduced the additional concept of temporary states, to which patients who transition remain for exactly one cycle. This breaks the principle that Markov processes are memoryless and thus the underlying mathematical formalism, first developed by Kolmogorov, is not applicable. For example, ensuring that all patients leave a temporary state requires its transition rate to be infinite. Hence, such models are usually labelled as semi-Markov processes.

Rates and probabilities: Miller and Homan (1994) and Fleurence & Hollenbeak (2007) provide advice on estimating probabilities from rates. Jones (2017) and Welton (2005) describe methods for estimating probabilities in multi-state, multi-transition models, although those methods may not apply to semi-Markov models with temporary states. In particular, note that the "simple" equation, $p = 1 - e^{-rt}$ (Briggs 2006) applies only in a two-state, one transition model.

Uncertainty in rates: In semi-Markov models, the conditional probabilities of the transitions from each state are usually modelled by a Dirichlet distribution. In **rdecision**, create a Dirichlet distribution for each state and optionally create model variables for each conditional probability (ρ_{ij}) linked to an applicable Dirichlet distribution.

Super classes

`rdecision::Graph` -> `rdecision::Digraph` -> `SemiMarkovModel`

Methods

Public methods:

- `SemiMarkovModel$new()`
- `SemiMarkovModel$set_probabilities()`
- `SemiMarkovModel$transition_probabilities()`
- `SemiMarkovModel$transition_cost()`
- `SemiMarkovModel$get_statenames()`
- `SemiMarkovModel$reset()`
- `SemiMarkovModel$get_populations()`
- `SemiMarkovModel$get_elapsed()`
- `SemiMarkovModel$tabulate_states()`
- `SemiMarkovModel$cycle()`
- `SemiMarkovModel$cycles()`
- `SemiMarkovModel$modvars()`
- `SemiMarkovModel$modvar_table()`
- `SemiMarkovModel$clone()`

Method `new()`**:** Creates a semi-Markov model for cohort simulation.

Usage:

```
SemiMarkovModel$new(
  V,
  E,
  tcycle = as.difftime(365.25, units = "days"),
  discount.cost = 0,
  discount.utility = 0
)
```

Arguments:

`V` A list of nodes (`MarkovStates`).

`E` A list of edges (`Transitions`).

`tcycle` Cycle length, expressed as an R `difftime` object.

`discount.cost` Annual discount rate for future costs. Note this is a rate, not a probability (i.e. use 0.035 for 3.5%).

`discount.utility` Annual discount rate for future incremental utility. Note this is a rate, not a probability (i.e. use 0.035 for 3.5%).

Details: A semi-Markov model must meet the following conditions:

1. It must have at least one node and at least one edge.
2. All nodes must be of class `MarkovState`;
3. All edges must be of class `Transition`;
4. The nodes and edges must form a digraph whose underlying graph is connected;
5. Each state must have at least one outgoing transition (for absorbing states this is a self-loop);
6. For each state the sum of outgoing conditional transition probabilities must be one. For convenience, one outgoing transition probability from each state may be set to NA when the probabilities are defined. Typically, probabilities for self loops would be set to NA. Transition probabilities in Pt associated with transitions that are not defined as edges in the graph are zero. Probabilities can be changed between cycles.
7. No two edges may share the same source and target nodes (i.e. the digraph may not have multiple edges). This is to ensure that there are no more transitions than cells in the transition matrix.
8. The node labels must be unique to the graph.

Returns: A `SemiMarkovModel` object. The population of the first state is set to 1000 and from each state there is an equal conditional probability of each allowed transition.

Method `set_probabilities()`**:** Sets transition probabilities.

Usage:

```
SemiMarkovModel$set_probabilities(Pt)
```

Arguments:

`Pt` Per-cycle transition probability matrix. The row and column labels must be the state names and each row must sum to one. Non-zero probabilities for undefined transitions are not allowed. At most one `NA` may appear in each row. If an NA is present in a row, it is replaced by 1 minus the sum of the defined probabilities.

Returns: Updated `SemiMarkovModel` object

Method `transition_probabilities()`**:** Per-cycle transition probability matrix for the model.

Usage:

```
SemiMarkovModel$transition_probabilities()
```

Returns: A square matrix of size equal to the number of states. If all states are labelled, the dimnames take the names of the states.

Method `transition_cost()`**:** Return the per-cycle transition costs for the model.

Usage:

```
SemiMarkovModel$transition_cost()
```

Returns: A square matrix of size equal to the number of states. If all states are labelled, the dimnames take the names of the states.

Method `get_statenames()`**:** Returns a character list of state names.

Usage:

```
SemiMarkovModel$get_statenames()
```

Returns: List of the names of each state.

Method `reset()`**:** Resets the model counters.

Usage:

```
SemiMarkovModel$reset(
  populations = NULL,
  icycle = 0L,
  elapsed = as.difftime(0, units = "days")
)
```

Arguments:

`populations` A named vector of populations for the start of the state. The names should be the state names. Due to the R implementation of matrix algebra, `populations` must be a numeric type and is not restricted to being an integer. If NULL, the population of all states is set to zero.

`icycle` Cycle number at which to start/restart.

`elapsed` Elapsed time since the index (reference) time used for discounting as an R `difftime` object.

Details: Resets the state populations, next cycle number and elapsed time of the model. By default the model is returned to its ground state (zero people in the all states; next cycle is labelled zero; elapsed time (years) is zero). Any or all of these can be set via this function. `icycle` is simply an integer counter label for each cycle, `elapsed` sets the elapsed time in years from the index time from which discounting is assumed to apply.

Returns: Updated `SemiMarkovModel` object.

Method `get_populations()`**:** Gets the occupancy of each state

Usage:

```
SemiMarkovModel$get_populations()
```

Returns: A numeric vector of populations, named with state names.

Method `get_elapsed()`**:** Gets the current elapsed time.

Usage:

```
SemiMarkovModel$get_elapsed()
```

Details: The elapsed time is defined as the difference between the current time in the model and an index time used as the reference time for applying discounting. By default the elapsed time starts at zero. It can be set directly by calling `reset`. It is incremented after each call to `cycle` by the cycle duration to the time at the end of the cycle (even if half cycle correction is used). Thus, via the `reset` and `cycle` methods, the time of each cycle relative to the discounting index and its duration can be controlled arbitrarily.

Returns: Elapsed time as an R `difftime` object.

Method `tabulate_states()`**:** Tabulation of states

Usage:

```
SemiMarkovModel$tabulate_states()
```

Details: Creates a data frame summary of each state in the model.

Returns: A data frame with the following columns:

Name State name

Cost Annual cost of occupying the state

Utility Incremental utility associated with being in the state.

Method `cycle()`**:** Applies one cycle of the model.

Usage:

```
SemiMarkovModel$cycle(hcc.pop = TRUE, hcc.cost = TRUE)
```

Arguments:

`hcc.pop` Determines the state populations returned by this function and for calculating incremental utility, and the time at which the utility discount is applied. If FALSE, the end of cycle populations and time apply; if TRUE the mid-cycle populations and time apply. The mid-cycle populations are taken as the mean of the start and end populations and the discount time as the mid-point. The value of this parameter does not affect the state populations or elapsed time passed to the next cycle or available via `get_populations`; those are always the end cycle values.

`hcc.cost` Determines the state occupancy costs returned by this function and the time at which the cost discount is applied to the occupancy costs and the entry costs. If FALSE, the end of cycle populations and time apply; if TRUE the mid-cycle populations and time apply, as per `hcc.pop`. The value of this parameter does not affect the state populations or elapsed time passed to the next cycle or available via `get_populations`; those are always the end cycle values.

Returns: Calculated values, one row per state, as a data frame with the following columns:

`State` Name of the state.

`Cycle` The cycle number.

`Time` Clock time in years of the end of the cycle.

`Population` Populations of the states; see `hcc.pop`.

`OccCost` Cost of the population occupying the state for the cycle. Discounting and half cycle correction is applied, if those options are set. The costs are normalized by the model population. The cycle costs are derived from the annual occupancy costs of the `MarkovStates`.

`EntryCost` Cost of the transitions *into* the state during the cycle. Discounting is applied, if the option is set. The result is normalized by the model population. The cycle costs are derived from `Transition` costs.

`Cost` Total cost, normalized by model population.

`QALY` Quality adjusted life years gained by occupancy of the states during the cycle. Half cycle correction and discounting are applied, if these options are set. Normalized by the model population.

Method `cycles()`**:** Applies multiple cycles of the model.

Usage:

```
SemiMarkovModel$cycles(ncycles = 2L, hcc.pop = TRUE, hcc.cost = TRUE)
```

Arguments:

`ncycles` Number of cycles to run; default is 2.

`hcc.pop` Determines the state populations returned by this function and for calculating incremental utility, and the time at which the utility discount is applied. If FALSE, the end of cycle populations and time apply; if TRUE the mid-cycle populations and time apply. The mid-cycle populations are taken as the mean of the start and end populations and the discount time as the mid-point. The value of this parameter does not affect the state populations or elapsed time passed to the next cycle or available via `get_populations`; those are always the end cycle values.

`hcc.cost` Determines the state occupancy costs returned by this function and the time at which the cost discount is applied to the occupancy costs and the entry costs. If FALSE, the end of cycle populations and time apply; if TRUE the mid-cycle populations and time apply, as per `hcc.pop`. The value of this parameter does not affect the state populations or elapsed time passed to the next cycle or available via `get_populations`; those are always the end cycle values.

Details: The starting populations are redistributed through the transition probabilities and the state occupancy costs are calculated, using function `cycle`. The end populations are then fed back into the model for a further cycle and the process is repeated. For each cycle, the state populations and the aggregated occupancy costs are saved in one row of the returned data frame, with the cycle number. If the cycle count for the model is zero when called, the first cycle reported will be cycle zero, i.e. the distribution of patients to starting states.

Returns: Data frame with cycle results, with the following columns:

`Cycle` The cycle number.

`Years` Elapsed time at end of cycle, years

`Cost` Cost associated with occupancy and transitions between states during the cycle.

`QALY` Quality adjusted life years associated with occupancy of the states in the cycle.

`<name>` Population of state `<name>` at the end of the cycle.

Method `modvars()`**:** Find all the model variables in the Markov model.

Usage:

```
SemiMarkovModel$modvars()
```

Details: Returns variables of type `ModVar` that have been specified as values associated with transition rates and costs and the state occupancy costs and utilities.

Returns: A list of `ModVars`.

Method `modvar_table()`**:** Tabulate the model variables in the Markov model.

Usage:

```
SemiMarkovModel$modvar_table(expressions = TRUE)
```

Arguments:

`expressions` A logical that defines whether expression model variables should be included in the tabulation.

Returns: Data frame with one row per model variable, as follows:

`Description` As given at initialization.

`Units` Units of the variable.

`Distribution` Either the uncertainty distribution, if it is a regular model variable, or the expression used to create it, if it is an `ExprModVar`.

`Mean` Mean; calculated from means of operands if an expression.

`E` Expectation; estimated from random sample if expression, mean otherwise.

`SD` Standard deviation; estimated from random sample if expression, exact value otherwise.

`Q2.5` p=0.025 quantile; estimated from random sample if expression, exact value otherwise.

`Q97.5` p=0.975 quantile; estimated from random sample if expression, exact value otherwise.

`Est` TRUE if the quantiles and SD have been estimated by random sampling.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
SemiMarkovModel$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Author(s)

Andrew J. Sims <andrew.sims@newcastle.ac.uk>

References

Beck JR and Pauker SG. The Markov Process in Medical Prognosis. *Med Decision Making*, 1983;**3**:419–458.

Briggs A, Claxton K, Sculpher M. Decision modelling for health economic evaluation. Oxford, UK: Oxford University Press; 2006.

Fleurence RL and Hollenbeak CS. Rates and probabilities in economic modelling. *PharmacoEconomics*, 2007;**25**:3–6.

Jones E, Epstein D and García-Mochón L. A procedure for deriving formulas to convert transition rates to probabilities for multistate Markov models. *Medical Decision Making* 2017;**37**:779–789.

Miller DK and Homan SM. Determining transition probabilities: confusion and suggestions. *Medical Decision Making* 1994;**14**:52-58.

Sonnenberg FA, Beck JR. Markov models in medical decision making: a practical guide. *Medical Decision Making*, 1993:**13**:322.

Welton NJ and Ades A. Estimation of Markov chain transition probabilities and rates from fully and partially observed data: uncertainty propagation, evidence synthesis, and model calibration. *Medical Decision Making*, 2005;**25**:633-645.

`Stack` *A stack*

Description

An R6 class representing a stack of objects of any type.

Details

Conventional implementation of a stack. Used extensively in graph algorithms and offered as a separate class for ease of programming and to ensure that implementations of stacks are optimized. By intention, there is only minimal checking of method arguments. This is to maximize performance and because the class is mainly intended for use internally to **rdecision**.

Methods

Public methods:

- `Stack$new()`
- `Stack$push()`
- `Stack$pop()`
- `Stack$size()`
- `Stack$as_list()`
- `Stack$clone()`

Method `new()`**:** Create a stack.

Usage:

```
Stack$new()
```

Returns: A new `Stack` object.

Method `push()`**:** Push an item onto the stack.

Usage:

```
Stack$push(x)
```

Arguments:

`x` The item to push onto the top of the stack. It should be of the same class as items previously pushed on to the stack. It is not checked.

Returns: An updated `Stack` object

Method `pop()`**:** Pop an item from the stack. Stack underflow and raises an error.

Usage:

```
Stack$pop()
```

Returns: The item previously at the top of the stack.

Method `size()`**:** Gets the number of items on the stack.

Usage:

```
Stack$size()
```

Returns: Number of items.

Method `as_list()`**:** Inspect items in the stack.

Usage:

```
Stack$as_list()
```

Returns: A list of items.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
Stack$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Author(s)

Andrew Sims <andrew.sims@newcastle.ac.uk>

`Transition` *A transition in a semi-Markov model*

Description

An R6 class representing a transition in a semi-Markov model.

Details

A specialism of class `Arrow` which is used in a semi-Markov model to represent a transition between two `MarkovStates`. The transition is optionally associated with a cost. The transition probability is associated with the model (`SemiMarkovModel`) rather than the transition.

Super classes

`rdecision::Edge` -> `rdecision::Arrow` -> `Transition`

Methods

Public methods:

- `Transition$new()`
- `Transition$modvars()`
- `Transition$set_cost()`
- `Transition$cost()`
- `Transition$clone()`

Method `new()`**:** Create an object of type `MarkovTransition`.

Usage:

```
Transition$new(source_state, target_state, cost = 0, label = "")
```

Arguments:

`source_state` `MarkovState` from which the transition starts.

`target_state` `MarkovState` to which the transition ends.

`cost` Cost associated with the transition.

`label` Character string containing a label for the transition (the name of the event).

Returns: A new `Transition` object.

Method `modvars()`**:** Find all the model variables.

Usage:

```
Transition$modvars()
```

Details: Find variables of type `ModVar` that have been specified as values associated with this `MarkovTransition`. Includes operands of these `ModVars`, if they are expressions.

Returns: A list of `ModVars`.

Method `set_cost()`**:** Set the cost associated with the transition.

Usage:

```
Transition$set_cost(c = 0)
```

Arguments:

`c` Cost associated with the transition.

Returns: Updated `Transition` object.

Method `cost()`**:** Return the cost associated with traversing the edge.

Usage:

```
Transition$cost()
```

Returns: Cost.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
Transition$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Author(s)

Andrew J. Sims <andrew.sims@newcastle.ac.uk>

Index