# Package ‘rPref’

October 14, 2022

**Version** 1.3

**Date** 2019-02-15

**Title** Database Preferences and Skyline Computation

**Author** Patrick Roocks <mail@p-roocks.de>

**Maintainer** Patrick Roocks <mail@p-roocks.de>

**Description** Routines to select and visualize the maxima for a given strict partial order. This especially includes the computation of the Pareto frontier, also known as (Top-k) Skyline operator (see Börzsönyi, et al. (2001) <doi:10.1109/ICDE.2001.914855>), and some generalizations known as database preferences (see Kießling (2002) <doi:10.1016/B978-155860869-6/50035-4>).

**URL** https://www.p-roocks.de/rpref

**Depends** R (>= 3.1.2)

**Imports** Rcpp (>= 1.0.0), RcppParallel (>= 4.3.6), dplyr (>= 0.8.0), igraph (>= 1.0.1), lazyeval (>= 0.2.1), methods, utils

**SystemRequirements** C++11, GNU make, Windows: cmd.exe and cscript.exe

**License** GPL (>= 2)

**LazyData** true

**LinkingTo** Rcpp, RcppParallel

**Suggests** testthat, graph, Rgraphviz (>= 2.16.0), knitr, ggplot2, rmarkdown

**Collate** 'rPref.r' 'RcppExports.R' 'pref-classes.r' 'base-pref.r' 'base-pref-macros.r' 'complex-pref.r' 'general-pref.r' 'pref-eval.r' 'show-pref.r' 'visualize.r' 'pred-succ.r'

**VignetteBuilder** knitr

**RoxygenNote** 6.1.1

**Encoding** UTF-8

**NeedsCompilation** yes

**Repository** CRAN

**Date/Publication** 2019-02-16 10:00:03 UTC

## R topics documented:

## `base_pref` *Base Preferences*

### Description

Base preferences are used to describe the different goals (dimensions, in case of a Skyline query) of a preference query.

### Usage

```
low(expr, df = NULL)

low_(expr, df = NULL)

high(expr, df = NULL)

high_(expr, df = NULL)

true(expr, df = NULL)

true_(expr, df = NULL)

is.base_pref(x)
```

### Arguments

| | |
|---|---|
| `expr` | A numerical/logical expression which is the term to evaluate for the current preference. The objective is to search for minimal/maximal values of this expression (for `low`/`high`) or for logical `TRUE` values (for `true`). For `low_`, `high_` and `true_`, the argument must be an expression, a call or a string. |

| | |
|---|---|
| `df` | (optional) A data frame, having the same structure (i.e., columns) like that data frame, where this preference is evaluated later on. Causes a partial evaluation of the preference and the preference is associated with this data frame. See below for details. |
| `x` | An object to be tested if it is a base preference. |

## Details

Mathematically, all base preferences are strict weak orders (irreflexive, transitive and negative transitive).

The three fundamental base preferences are:

`low(a)`, `high(a)` Search for minimal/maximal values of `a`, i.e., the induced order is the "smaller than" or "greater than" order on the values of `a`. The values of `a` must be numeric values.

`true(a)` Search for true values in logical expressions, i.e., `TRUE` is considered to be better than `FALSE`. The values of `a` must be logical values. For a tuplewise evaluation of a complex logical expression one has to use the `&` and `|` operators for logical AND/OR (and not the `&&` and `||` operators).

The term `expr` may be just a single attribute or may contain an arbitrary expression, depending on more than one attribute, e.g., `low(a+2*b+f(c))`. There `a`, `b` and `c` are columns of the addressed data set and `f` has to be a previously defined function.

Functions contained in `expr` are evaluated over the entire data set, i.e., it is possible to use aggregate functions (`min`, `mean`, etc.). Note that all functions (and also variables which are not columns of the data set, where `expr` will be evaluated on) must be defined in the same environment (e.g., environment of a function or global environment) as the base preference is defined.

The function `is.base_pref` returns `TRUE` if `x` is a preference object and `FALSE` otherwise.

## Using Expressions in Preferences

The `low_`, `high_` and `true_` preferences have the same functionality as `low`, `high` and `true` but expect an expression, a call or a string as argument. For example, `low(a)` is equivalent to `low_(expression(a))` or `low_("a")`. Lazy expressions (see the lazyeval package) are also possible.

This is helpful for developing your own base preferences. Assume you want to define a base Preference `false` as the dual of `true`. A definition like `false <- function(x) -true(x)` is the wrong approach, as `psel(data.frame(a = c(1,2)), false(a == 1))` will result in the error "object 'a' not found". This is because `a` is considered as a variable and not as an (abstract) symbol to be evaluated later. By defining

```
false <- function(x, ...) -true_(substitute(x), ...)
```

one gets a preference which behaves like a "built-in" preference. Additional optional parameters (like `df`) are bypassed. The object `false(a == 1)` will output `[Preference] -true(a == 1)` on the console and `psel(data.frame(a = c(1,2)), false(a==1))` returns correctly the second tuple with `a==2`.

There is a special symbol `df__` which can be used in preference expression to access the given data set `df`, when `psel` is called on this data set. For example, on a data set where the first column has the name `A` the preference `low(df__[[1]])` is equivalent to `low(A)`.

### Partial Evaluation and Associated Data Frames

If the optional parameter `df` is given, then the expression is evaluated at the time of definition as far as possible. All variables occurring as columns in `df` remain untouched. For example, consider

```
f <- function(x) 2*x
p <- true(cyl == f(1), mtcars)
```

Then `p` is equivalent to the preference `true(cyl == 2)` as the variable `cyl` is a column in `mtcars`. Additionally the data set `mtcars` is associated with the preference `p`, implying that the preference selection can be done with `peval`. See `assoc.df` for details on associated data sets.

The preference selection, i.e., `psel(mtcars, p)` can be invoked without the partial evaluation. But this results in an error, if the function `f` has meanwhile removed from the current environment. Hence it is safer to do an early partial evaluation of all preferences, as far as they contain user defined functions.

The partial evaluation can be done manually by `partial.eval.pref`.

## See Also

See `complex_pref` how to compose complex preferences to retrieve e.g., the Skyline. See `general_pref` for functions applying to all kind of preferences. See `base_pref_macros` for more base preferences.

## Examples

```
# define a preference with a score value combining mpg and hp
p1 <- high(4 * mpg + hp)
# perform the preference selection
psel(mtcars, p1)

# define a preference with a given function
f <- function(x, y) (abs(x - mean(x))/max(x) + abs(y - mean(y))/max(y))
p2 <- low(f(mpg, hp))
psel(mtcars, p2)

# use partial evaluation for weighted scoring
p3 <- high(mpg/sum(mtcars$mpg) + hp/sum(mtcars$hp), df = mtcars)
p3
# select Pareto optima
peval(p3)
```

---

# `base_pref_macros` *Useful Base Preference Macros*

---

## Description

In addition to the base preferences, rPref offers some macros to define preferences where a given interval or point is preferred.

## Usage

```
around(expr, center, df = NULL)

between(expr, left, right, df = NULL)

pos(expr, pos_value, df = NULL)

layered(expr, ..., df = NULL)
```

## Arguments

| | |
|---|---|
| `expr` | A numerical expression (for `around` and `between`) or an arbitrary expression (for `pos` and `layered`). The objective are those tuples where `expr` evaluates to a value within the preferred interval, layer, etc. Regarding attributes, functions and variables, the same requirements as for `base_pref` apply. |
| `center` | Preferred numerical value for `around`. |
| `df` | (optional) Data frame for partial evaluation and association of preference and data set. See `base_pref` for details. |
| `left` | Lower limit (numerical) of the preferred interval for `between`. |
| `right` | Upper limit (numerical) of the preferred interval for `between`. |
| `pos_value` | A vector containing the preferred values for a `pos` preference. It has to be of the same type (numeric, logical, character, ...) as `expr`. |
| `...` | Layers (sets) for a `layered` preference. Each parameter corresponds to a layer and the first one characterizes the most preferred values. |

## Definition of the Preference Macros

`between(expr, left, right)` Those tuples are preferred where `expr` evaluates to a value between `left` and `right`. For values not in this interval, the values nearest to the interval are preferred.

`around(expr, center)` Same as `between(expr, center, center)`.

`pos(expr, pos_value)` Those tuples are preferred, where `expr` evaluates to a value which is contained in `pos_value`.

`layered(expr, layer1, layer2, ..., layerN)` For the most preferred tuples `expr` must evaluate to a value in `layer1`. The second-best tuples are those where `expr` evaluates to a value in `layer2` and so forth. Values occurring in none of the layers are considered worse than those in `layerN`. Technically, this is realized by a prioritization chain (lexicographical order) of `true` preferences.

Note that only the argument `expr` may contain columns from the data frame, all other variables must evaluate to explicit values. For example `around(mpg, mean(mpg))` is not allowed. In this case, one can use `around(mpg, mean(mtcars$mpg))` instead. Or alternatively, without using the base preference macros, `low(abs(mpg - mean(mpg)))` does the same. There, the actual mean value of `mpg` is calculated just when the preference selection via `psel` is called.

### Examples

```
# search for cars where mpg is near to 25
psel(mtcars, around(mpg, 25))

# cyl = 2 and cyl = 4 are equally good, cyl = 6 is worse
psel(mtcars, layered(cyl, c(2, 4), 6))
```

## complex_pref *Complex Preferences*

### Description

Complex preferences are used to compose different preference orders. For example the Pareto composition (via operator `*`) is the usual operator to compose the preference for a Skyline query. The Skyline is also known as Pareto frontier. All complex preferences are mathematically strict partial orders (irreflexive and transitive).

### Usage

```
## S4 method for signature 'preference,preference'
e1 * e2

## S4 method for signature 'preference,preference'
e1 & e2

## S4 method for signature 'preference,preference'
e1 | e2

## S4 method for signature 'preference,preference'
e1 + e2

reverse(p)

is.complex_pref(x)
```

### Arguments

| | |
|---|---|
| `p, e1, e2` | Preference objects (they can be either base preferences, see `base_pref`, or complex preferences) |
| `x` | An object to be tested if it is a complex preference. |

### Skylines

The most important preference composition operator is the Pareto operator (`p1 * p2`) to formulate a Skyline query. A tuple t1 is better than t2 w.r.t. `p1 * p2` if it is strictly better w.r.t. one of the preferences p1, p2 and is better or equal w.r.t. the other preference.

The syntactical correspondence to other query languages supporting Skylines/preferences to rPref is given as follows:

- A query in the syntax from Borzsonyi et. al (2001) like
  `"... SKYLINE OF a MAX, b MIN, c MAX"`
  corresponds in rPref to the preference
  `high(a) * low(b) * high(c)`.
- A query in the syntax from Kiessling (2002) like
  `"... PREFERRING a LOWEST AND (b HIGHEST PRIOR TO c LOWEST)"`
  corresponds in rPref to
  `low(a) * (high(b) & low(c))`.
- A query in the syntax of the "Skyline" feature of the commercial database "EXASOL EXASolution 5" like
  `"... PREFERRING LOW a PLUS (b = 1 PRIOR TO LOW c))"`
  corresponds in rPref to
  `low(a) * (true(b == 1) & low(c))`.

Note that preferences in rPref can be translated to some of this query dialects by `show.query`.

## Definition of Additional Preference Operators

Additionally, rPref supports the following preference composition operators:

`p1 & p2` Prioritization (lexicographical order): A tuple t1 is better than t2 w.r.t. `p1 & p2` if it is strictly better w.r.t. `p1` or is equal w.r.t. `p1` and is better w.r.t. `p2`.

`p1 | p2` Intersection preference: A tuple t1 is better than t2 w.r.t. `p1 | p2` if it is strictly better w.r.t. both preferences. This is a stricter variant of the Pareto operator. The evaluation of `psel(df, p1 | p2)` is always a subset of `psel(df, p1 * p2)`.

`p1 + p2` Union preference: A tuple t1 is better than t2 w.r.t. `p1 + p2` if it is strictly better w.r.t. to one of the preferences. Note that this can violate the strict partial order property, if the domains (the tuples on which `p1` and `p2` define better-than-relationships) of the preferences are not disjoint.

`reverse(p1)` **or** `-p1` Reverse preference (converse relation): A tuple t1 is better than t2 w.r.t. `-p1` if t2 is better than t1 w.r.t. `p1`. The unary minus operator, i.e. `-p1`, is a short hand notation for `reverse(p1)`.

The function `is.complex_pref` returns `TRUE` if `x` is a complex preference object (i.e., was constructed by one of these binary operators or the unary operator `reverse`) and `FALSE` otherwise.

## Associated Data Sets

If one of the preferences for a binary operator are associated with a data set (see `base_pref`), then this association is propagated. For example, the preference

`p <- high(mpg, df = mtcars) * high(hp)`

as well as

`p <- high(mpg) * high(hp, df = mtcars)`

both result in the same complex preference which is associated with `mtcars`. A partial evaluation is also invoked for all preferenced which are added. For example, using this `p`, `p <- p * true(cyl == max(mtcars$cyl))`

generates the following console output:

```
[Preference] high(mpg) * high(hp) * true(cyl == 8)
* associated data source: data.frame "mtcars" [32 x 11]
```

We see that the association with the data set is propagated and `max(mtcars$cyl)` is partially evaluated.

## References

S. Borzsonyi, D. Kossmann, K. Stocker (2001): The Skyline Operator. In Data Engineering (ICDE '01), pages 421-430.

W. Kiessling (2002): Foundations of Preferences in Database Systems. In Very Large Data Bases (VLDB '02), pages 311-322.

S. Mandl, O. Kozachuk, M. Endres, W. Kiessling (2015): Preference Analytics in EXASolution. 16th Conference on Database Systems for Business, Technology, and Web.

## See Also

See `base_pref` for the construction of base preferences. See `general_pref` for functions applicable to all kind of preferences. See `psel` for the evaluation of preferences.

## Examples

```
# define preference for cars with low consumption (high mpg-value)
# and simultaneously high horsepower
p1 <- high(mpg) * high(hp)

# perform the preference search
psel(mtcars, p1)

# alternative way: create preference with associated data set
p2 <- high(mpg, df = mtcars) * high(hp)
peval(p2)
```

---

## general_pref *Utility Functions for Preferences*

---

## Description

Collection of some useful functions which are applicable to all preference objects.

## Usage

```
empty()

is.empty_pref(x)

## S4 method for signature 'preference'
length(x)

is.preference(x)

## S4 method for signature 'preference'
as.expression(x, ...)

## S4 method for signature 'preference'
assoc.df(x)

## S4 replacement method for signature 'preference'
assoc.df(x) <- value
```

## Arguments

| | |
|---|---|
| `x` | A preference, or, for `is.preference`, an object to be tested if it is an (empty) preference. |
| `...` | Optional arguments passed to `as.expression`. |
| `value` | A data frame to associate with a preference object. |

## Details

The empty preference `empty()` is a neutral element for the complex preference compositions `{*, &, +}`. It holds that `empty() * p` and `empty() & p` is equal to `p` for all preferences `p`.

The function `length(p)` returns the term length of the preference term `p` which is defined as the number of base preferences in a complex preference term. The empty preference `empty()` has length 0, and all base preferences have length 1.

With `as.expression(p)` for a preference `p` the call to the preference is constructed. This means, `eval(as.expression(p))` returns the preference `p`, evaluated in the current environment.

The function `is.empty_pref` returns `TRUE` if `x` is the empty preference object `empty()` and `FALSE` otherwise.

With `assoc.df` the associated data frame of a preference can be retrieved or set. Setting the associated data frame means that a partial evaluation based on this data frame is done. See `show.pref` for details on partial evaluation of preferences. Next, the preference is linked to that data frame, such that `peval(p)` can be used instead of `psel(df, p)`. It returns `NULL` if no data frame is associated. Use `set.assoc.df(NULL)` to delete an associated data frame.

## See Also

See `base_pref` for the construction of base preferences, and `complex_pref` for the construction of complex preferences. See `show.pref` for string output and partial evaluation of preference terms.

## Examples

```
# Same as low(a) * low(b)
p <- low(a) * low(b) * empty()

# returns 2, as empty() does not count
length(p)

# the preference expression (without empty())
as.expression(p)
```

---

## get_hasse_diag *Adjacency List of Hasse diagramm*

---

### Description

Returns the adjacency list of the Hasse diagram of a preference as an (n x 2) matrix. This is the transitive reduction of the preference relation.

### Usage

```
get_hasse_diag(df, pref)
```

### Arguments

| | |
|---|---|
| df | A data frame. |
| pref | A preference on the columns of df, see psel for details. |

### Details

A row (i, j) in the resulting matrix means that `df[i,]` is better than `df[j,]` with regard to the preference `p`. The matrix is the transitive reduction (Hasse diagram) of the induced relations, i.e., if (1,2) and (2,3) occur in the result, then (1,3) will not be contained. The number of rows in the result depends on the number of non-transitive Better-Than-Relationships in `df` w.r.t. `p`.

### See Also

`get_btg` to plot the Hasse diagram.

### Examples

```
get_hasse_diag(mtcars, low(mpg))
```

## plot_btg *Better-Than-Graphs*

### Description

Returns/plots a Hasse diagram of a preference order (also called the Better-Than-Graph, short BTG) on a given data set. Ploting within R relies on the igraph package or the Rgraphviz package. Alternatively, a dot file for an external graphviz/dot interpreter can be generated.

### Usage

```
plot_btg(df, pref, labels = 1:nrow(df), flip.edges = FALSE,
  levelwise = TRUE, use_dot = "Rgraphviz" %in%
  rownames(installed.packages()))

get_btg(df, pref, flip.edges = FALSE, use_dot = "Rgraphviz" %in%
  rownames(installed.packages()))

get_btg_dot(df, pref, labels = 1:nrow(df), flip.edges = FALSE,
  levelwise = TRUE, file = NULL)
```

### Arguments

| | |
|---|---|
| `df` | A data frame. |
| `pref` | A preference on the columns of `df`, see `psel` for details. |
| `labels` | (optional) Labels for the vertices. Default values are the row indices. |
| `flip.edges` | (optional) Flips the orientation of edges, if `TRUE` than arrows point from worse nodes to better nodes. |
| `levelwise` | (optional) Only relevant is the dot layouter is used. If `TRUE`, all tuples from the same level are placed on one row. If `FALSE`, the row arrangement is subject to the dot layouter. |
| `use_dot` | (optional) If `TRUE`, the dot layouter from Rgraphviz is used. If `FALSE`, igraph is used. By default this is `TRUE` iff Rgraphviz is available. |
| `file` | (optional) If specified, then `get_btg_dot` writes the graph specification to given file path. If not specified, the graph specification is returned as a string. |

### Details

The Hasse diagram of a preference visualizes all the better-than-relationships on a given data set. All edges which can be retrieved by transitivity of the order are omitted in the graph.

The functions `get_btg` and `plot_btg` either use the igraph package (if `use_dot = FALSE`) or the dot layouter from the Rgraphviz package (if `use_dot = TRUE`). If Rgraphviz is available it is used by default, otherwise the igraph Package is used. Note that Rgraphviz is only available on BioConductor and not on CRAN.

The dot layouter from Rgraphviz is more appropriate for Better-Than-Graphs than the igraph layouter, as all edges will be directed in the same direction (rank based ordering). Using `levelwise = TRUE` (the default), all tuples of the same level are placed on the same row.

## BTGs with igraph

If used with `use_dot = FALSE`, the function `get_btg` returns a list `l` with the following list entries:

`l$graph` An igraph object, created with the `igraph` package.

`l$layout` A typical Hasse diagram layout for plotting the graph, also created with igraph.

To plot the resulting graph returned from `get_btg`, use the `plot` function as follows:

`plot(l$graph, layout = l$layout)`

For more details, see `igraph.plotting`.

## BTGs with Rgraphviz

If used with `use_dot = FALSE`, the function `get_btg` returns a `graphNEL` object from the graph-package (Rgraphviz is build on top of that package). This object can also be plotted using `plot(...)`.

## Direct Plotting

In both cases (wheter Rgraphviz is used or not), the function `plot_btg` directly plots the Better-Than-Graph. There is an additional parameter `labels`, specifying the node labels. The default are the row numbers (not the `rownames` of the data frame), ranging from `"1"` to `as.character(nrow(df))`.

## Dot (Graphviz) String Output

The function `get_btg_dot` produces the source code of the Better-Than-Graph in the dot language of the Graphviz software. This is useful for an external dot interpreter. Depending on the `file` parameter the output is either written to a file (if a file path is given) or returned as a string (if `file = NULL`).

## Additional Parameters

By default, the directed edges in the diagram point from better to worse nodes w.r.t. the preference. This means an arrow can be read as "is better than". If `flip.edges = TRUE` is set, then the arrows point from worse nodes to better nodes ("is worse than"). In any case, the better nodes are plotted at the top and the worse nodes at the bottom of the diagram.

If Rgraphviz is used for `plot_btg` and for `get_btg_dot`, the option `levelwise` controls if all nodes of the same level are placed in one row. If this parameter is `FALSE`, then the vertical arrangement is subject to the dot layouter.

## Examples

```
# pick a small data set and create preference and BTG
df <- mtcars[1:10,]
pref <- high(mpg) * low(wt)
```

```
# directly plot the BTG with row numbers as labels
# uses Rgraphviz if available and igraph otherwise
plot_btg(df, pref)

# plot the graph with labels with relevant values
labels <- paste0(df$mpg, "; ", df$wt)
plot_btg(df, pref, labels)

# show lattice structure of 3-dimensional Pareto preference
df <- merge(merge(data.frame(x = 1:3), data.frame(y = 1:3)), data.frame(z = 1:2))
labels <- paste0(df$x, ",", df$y, ",", df$z)
plot_btg(df, low(x) * low(y) * low(z), labels)

# Create a graph with external Graphviz (requires installed Graphviz)
## Not run:
# creates tmpgraph.dot in the current working directoy
get_btg_dot(df, pref, labels, file = "tmpgraph.dot")
# convert to diagram tmpgraph.png using Graphviz
shell(paste0('"C:/Program Files (x86)/Graphviz2.38/bin/dot.exe"',
             ' -Tpng tmpgraph.dot -o tmpgraph.png'))
# open resulting image
shell("tmpgraph.png")
## End(Not run)
```

## `plot_front` *Pareto Front Plot*

### Description

Connects the points of a Pareto front (also known as Pareto frontier) and hence visualizes the dominance region of a Skyline.

### Usage

```
plot_front(df, pref, ...)
```

### Arguments

| | |
|---|---|
| `df` | The data frame for which the Pareto front is plotted. This may be already a maximal set w.r.t. the preference `pref`, but anyway the maximal set is recalculated via `psel(df, pref)`. |
| `pref` | The preference representing the Skyline goals. This must be a Pareto composition (`p1 * p2`) or intersection composition (`p1 \| p2`) of two `low` or `high` preferences. |
| `...` | Additional graphic parameters which are passed to the `segments` function (internally used to plot the front). |

## Details

`plot_front` assumes that there is an existing plot, where the value of the first preference was plotted as x-coordinate and the value of the second preference as y-coordinate.

Note that `plot_front` is only recommended if you want to use the plotting functionality from base R. If you prefer to use ggplot2, we recommend using `geom_step` for plotting the Pareto front. See `vignette("visualization", package = "rPref")` for examples.

## Examples

```
# plots Pareto fronts for the hp/mpg values of mtcars
show_front <- function(pref) {
  plot(mtcars$hp, mtcars$mpg)
  sky <- psel(mtcars, pref)
  plot_front(mtcars, pref, col = rgb(0, 0, 1))
  points(sky$hp, sky$mpg, lwd = 3)
}

# do this for all four combinations of Pareto compositions
show_front(low(hp)  * low(mpg))
show_front(low(hp)  * high(mpg))
show_front(high(hp) * low(mpg))
show_front(high(hp) * high(mpg))

# compare this to the front of a intersection preference
show_front(high(hp) | high(mpg))
```

---

# `pred_succ` *Predecessor and Successor Functions*

---

## Description

Function for traversing the BTG (Better-Than-Graph or Hasse diagram) of a preference.

## Usage

```
init_pred_succ(p, df = NULL)

hasse_pred(p, v, intersect = FALSE)

hasse_succ(p, v, intersect = FALSE)

all_pred(p, v, intersect = FALSE)

all_succ(p, v, intersect = FALSE)
```

## Arguments

| | |
|---|---|
| `p` | A preference. Worse tuples in the induced order are successors and better tuples are predecessors. |
| `df` | (optional) A data frame characterizing the set wherein predecessors/successors are searched. If `df` is `NULL` then the data frame associated with the preference is used. Causes an error if `df == NULL` and no data frame is associated. |
| `v` | A numeric vector of indices in `df`. This represents the set of tuples for which predecessors/successors are searched. |
| `intersect` | (optional) Logical value. If it is `FALSE` (by default) the union of all predecessors/successors of `v` is returned. For `intersect = TRUE` the intersection of those values is returned. |

## Details

These functions return the predecessors and successors in the Better-Than-Graph of a preference. Note that the successors/predecessors can can be plotted via `get_btg`. Before any of the successor/predecessor functions can be used the initialization has to be called as follows:

```
init_pred_succ(p, df)
```

There `p` is a preference object and `df` a data frame. When this done, the data frame `df` is associated with `p`, i.e., implicitly `assoc.df` is called. If the preference has already an associcated data frame, `df` can be omitted. For example

```
p <- low(mpg, df = mtcars)
init_pred_succ(p)
```

does the initialization of the preference `low(mpg)` on the data set `mtcars`.

The `init_pred_succ` function calculates the Better-Than-Relation on `df` w.r.t. `p`. Afterwards the predecessor and successor functions, as subsequently described, can be called. The value of `v` is a numeric vector within `1:nrow(df)` and characterizes a subset of tuples in `df`. The return value of these functions is again a numeric vector referring to the row numbers in `df` and it is always ordered ascending, independently of the order of the indices in `v`.

`all_pred(p, v)` Returns all predecessors of `v`, i.e., indices of better tuples than `v`.

`all_succ(p, v)` Returns all successors of `v`, i.e., indices of worse tuples than `v`.

`hasse_pred(p, v)` Returns the direct predecessors of `v`, i.e., indices of better tuples than `v` where the better-than-relation is contained in the transitive reduction.

`hasse_succ(p, v)` Returns the direct successors of `v`, i.e., indices of worse tuples than `v` where the better-than-relation is contained in the transitive reduction.

If `v` has length 1, then the value of `intersect` does not matter, as there is nothing to intersect or join. For scalar values `x` and `y` the following identities hold, where `f` is one of the predecessor/successor functions:

```
f(p, c(x, y), intersect = FALSE) == union(f(p, x), f(p, y))
```

```
f(p, c(x, y), intersect = TRUE) == intersect(f(p, x), f(p, y))
```

## Examples

```
# preference on mtcars for high mpg and low weight
p <- high(mpg) * low(wt)
init_pred_succ(p, mtcars)

# helper to show mpg/hp values
show_vals <- function(x) mtcars[x,c('mpg','wt')]

# pick some tuple "in the middle"
show_vals(10)

# show (direct) predecessors/successors of tuple 10
show_vals(hasse_pred(p, 10)) # Next better car
show_vals(hasse_succ(p, 10)) # Next worse car
show_vals(all_pred(p, 10))   # All better cars
show_vals(all_succ(p, 10))   # All worse cars
```

---

`psel` *Preference Selection*

---

## Description

Evaluates a preference on a given data set, i.e., returns the maximal elements of a data set for a given preference order.

## Usage

```
psel(df, pref, ...)

psel.indices(df, pref, ...)

peval(pref, ...)
```

## Arguments

| | |
|---|---|
| `df` | A data frame or, for a grouped preference selection, a grouped data frame. See below for details. |
| `pref` | The preference order constructed via `complex_pref` and `base_pref`. All variables occurring in the definition of `pref` must be either columns of the data frame `df` or variables/functions of the environment where `pref` was defined. |
| `...` | Additional (optional) parameters for top(-level)-k selections: <br> `top` A `top` value of k means that the k-best tuples of the data set are returned. This may be non-deterministic, see below for details. <br> `at_least` An `at_least` value of k returns the top-k tuples and additionally all tuples which are not dominated by the worst tuple (i.e. the minima) of the Top-k set. The number of tuples returned is greater or equal than `at_least`. In contrast to top-k, this is deterministic. |

`top_level` A `top_level` value of k returns all tuples from the k-best levels. See below for the definition of a level.

`and_connected` Logical value, which is only relevant if more than one of the above {`top`, `at_least`, `top_level`} values is given, otherwise it will be ignored. Then `and_connected = TRUE` (which is the default) means that all top-conditions must hold for the returned tuples: Let `cond1` and `cond2` be top-conditions like `top=2` or `top_level=3`, then `psel([...], cond1, cond2)` is equivalent to the intersection of `psel([...], cond1)` and `psel([...], cond2)`. If we have `and_connected = FALSE`, these conditions are or-connected. This corresponds to the union of `psel([...], cond1)` and `psel([...], cond2)`.

`show_level` Logical value. If `TRUE`, a column `.level` is added to the returned data frame, containing all level values. If at least one of the {`top`, `at_least`, `top_level`} values are given, then `show_level` is `TRUE` by default for the `psel` function. Otherwise, and for `psel.indices` in all cases, this option is `FALSE` by default.

## Details

The difference between the three variants of the preference selection is:

- The `psel` function returns a subset of the data set which are the maxima according to the given preference.
- The function `psel.indices` returns just the row indices of the maxima (except top-k queries with `show_level = TRUE`, see top-k preference selection). Hence `psel(df, pref)` is equivalent to `df[psel.indices(df, pref),]` for non-grouped data frames.
- Finally, `peval` does the same like `psel`, but assumes that `p` has an associated data frame which is used for the preference selection. Consider `base_pref` to see how base preferences are associated with data sets or use `assoc.df` to explicitly associate a preference with a data frame.

## Top-k Preference Selection

For a given `top` value of k the k best elements and their level values are returned. The level values are determined as follows:

- All the maxima of a data set w.r.t. a preference have level 1.
- The maxima of the remainder, i.e., the data set without the level 1 maxima, have level 2.
- The n-th iteration of "Take the maxima from the remainder" leads to tuples of level n.

By default, `psel.indices` does not return the level values. By setting `show_level = TRUE` this function returns a data frame with the columns '.indices' and '.level'. Note that, if none of the top-k values {`top`, `at_least`, `top_level`} is set, then all level values are equal to 1.

By definition, a top-k preference selection is non-deterministic. A top-1 query of two equivalent tuples (equivalence according to `pref`) can return on both of these tuples. For example, a `top=1` preference selection on the tuples (a=1, b=1), (a=1, b=2) w.r.t. `low(a)` preference can return either the 'b=1' or the 'b=2' tuple.

On the contrary, a preference selection using at_least is deterministic by adding all tuples having the same level as the worst level of the corresponding top-k query. This means, the result is filled with all tuples being not worse than the top-k result. A preference selection with top-level-k returns all tuples having level k or better.

If the top or at_least value is greater than the number of elements in df (i.e., nrow(df)), or top_level is greater than the highest level in df, then all elements of df will be returned without further warning.

## Grouped Preference Selection

Using psel it is also possible to perform a preference selection where the maxima are calculated for every group separately. The groups have to be created with group_by from the dplyr package. The preference selection preserves the grouping, i.e., the groups are restored after the preference selection.

For example, if the summarize function from dplyr is applied to psel(group_by(...), pref), the summarizing is done for the set of maxima of each group. This can be used to e.g., calculate the number of maxima in each group, see the examples below.

A {top, at_least, top_level} preference selection is applied to each group separately. A top=k selection returns the k best tuples for each group. Hence if there are 3 groups in df, each containing at least 2 elements, and we have top = 2, then 6 tuples will be returned.

## Parallel Computation

On multi-core machines the preference selection can be run in parallel using a divide-and-conquer approach. Depending on the data set, this may be faster than a single-threaded computation. To activate parallel computation within rPref the following option has to be set:

options(rPref.parallel = TRUE)

If this option is not set, rPref will use single-threaded computation by default. With the option rPref.parallel.threads the maximum number of threads can be specified. The default is the number of cores on your machine. To set the number of threads to the value of 4, use:

options(rPref.parallel.threads = 4)

## See Also

See complex_pref on how to construct a Skyline preference.

## Examples

```
# Skyline and top-k/at-least skyline
psel(mtcars, low(mpg) * low(hp))
psel(mtcars, low(mpg) * low(hp), top = 5)
psel(mtcars, low(mpg) * low(hp), at_least = 5)

# preference with associated data frame and evaluation
p <- low(mpg, df = mtcars) * (high(cyl) & high(gear))
peval(p)
```

```
# visualize the skyline in a plot
sky1 <- psel(mtcars, high(mpg) * high(hp))
plot(mtcars$mpg, mtcars$hp)
points(sky1$mpg, sky1$hp, lwd=3)

# grouped preference with dplyr
library(dplyr)
psel(group_by(mtcars, cyl), low(mpg))

# return size of each maxima group
summarise(psel(group_by(mtcars, cyl), low(mpg)), n())
```

## rPref *Summary of the rPref Package*

### Description

rPref contains routines to select and visualize the maxima for a given strict partial order. This especially includes the computation of the Pareto frontier, also known as (Top-k) Skyline operator, and some generalizations (database preferences).

### Preference Composition/Selection

- Preferences are primarily composed from base preferences (see `base_pref`) and complex preferences (see `complex_pref`), where especially the Pareto operator for Skylines is such a complex preference.
- Some utility functions for preferences are collected in `general_pref`.
- Additionally some base preference macros are provided in `base_pref_macros`.
- The (top(-level)-k) preference selection `psel` allows to retrieve the maxima of a preference (or Pareto frontier, Skyline), constructed with the functions above, on a given data set.

### Visualization and Analysis of Preferences

- The visualization of the preference order in a Better-Than-Graph (Hasse diagram) is possible via `plot_btg`.
- The adjacency list of the Hasse diagram can be accessed via `get_hasse_diag`.
- Predecessors/successors in the Hasse diagram are calculated with the `pred_succ` functions.
- The Pareto frontier can be plotted using the `plot_front` function.

### String Output of Preferences

- The preference query for some preference-supporting DBMS can be given by `show.query`.
- A preference is partially evaluated and printed with `show.pref`.

### Vignettes

To learn the basics of rPref, start with the vignettes:

- A general introduction and some examples are given in
  `vignette("introduction", package = "rPref")`
- The visualization of preferences is explained in
  `vignette("visualization", package = "rPref")`

### Further Information

The rPref homepage is http://www.p-roocks.de/rpref. To submit bugs, feature requests or other comments, feel free to write a mail to me.

### Author(s)

Patrick Roocks, `<mail@p-roocks.de>`

---

## `show.pref` *Partial Evaluation and String Output of Preferences*

---

### Description

Functions to substitute variables and functions in preferences which can be calculated before the preference is evaluated on a data frame and character output of preferences.

### Usage

```
show.pref(p, df = NULL)

## S4 method for signature 'preference'
as.character(x, ...)

pref.str(p, df = NULL)

partial.eval.pref(p, df = NULL)
```

### Arguments

| | |
|---|---|
| `p, x` | The preference to be shown or partially evaluated. |
| `df` | (optional) A data frame on which the preference operates. Used for partial evaluation. |
| `...` | Optional arguments passed to `as.character`. |

## Details

The function pref.str (or as.character(p) for a preference p) returns the preference string while show.pref outputs it directly to the console, preceded by '[Preference]'. If df is specified, then a partial evaluation of the preference is done before converting it to a string.

The function partial.eval.pref (with given data frame df) partially evaluates the internal preference expression and returns again a preference object. All expressions in p are evaluated in the environment where p was defined, except the the column names in df (which are potential attributes in p) and except the special variable df__, which accesses the entire data set (see psel). The content of the data frame df does not matter; only names(df) is used to get the "free variables" in p.

If p has already an associated data frame (see assoc.df), then a partial evaluation was already done when the data frame was associated. In this case, the df parameter should not be used. The association will not be changed if one of these function are called with a given data frame on a preference object having an associated data frame.

## Partial Evaluation Before String Output

The functions show.pref and pref.str have the optional parameter df. If this parameter is given, these functions call partial.eval.pref before they output or return the preference string. The following equalities hold:

- as.character(partial.eval.pref(p, df)) == pref.str(p, df)
- show(partial.eval.pref(p, df)) produces the same console output as show.pref(p, df)

## See Also

See general_pref for more utility functions for preferences.

## Examples

```
f <- function(x) 2*x
p <- true(cyl == f(1))

# prints 'true(cyl == f(1))'
p

# prints 'true(cyl == 2)'
show.pref(p, mtcars)
partial.eval.pref(p, mtcars)
```

---

show.query *Show Preferences in Database Query Languages*

---

## Description

For a given preference this shows the PREFERRING clause of a database query in different SQL dialects which support preferences.

## Usage

```
show.query(p, dialect = "EXASOL", df = NULL)
```

## Arguments

| | |
|---|---|
| `p` | A preference. |
| `dialect` | The preference query dialect, which determines the syntax of the returned query. This has to be one of the following (not case sensitive): **`'EXASOL'`:** Syntax of the "Skyline" feature of the commercial database EXASOL EXASolution 5. **`'Preference SQL'` or `'PSQL'`:** Syntax of the Preference SQL system. This is a research prototype developed at the Chair of Databases and Information Systems of the University of Augsburg. See references for details. |
| `df` | Optional parameter to specify a data frame on which the preference operates causing a partial evaluation. See `show.pref` for details. |

## Details

There are few database systems supporting Skyline queries. A Skyline query consists of a usual SQL query followed by a `PREFERRING`-clause (in some rarely used dialects also `SKYLINE OF`). For example consider a database table r(a,b). The preference selection `psel(r, low(a) * high(b))` can be expressed by (in the Exasol dialect):

`SELECT * FROM r PREFERRING LOW a PLUS HIGH b`

The `show.query` function generates just the `PREFERRING`-clause, i.e. `show.query(low(a) * high(b))` returns

`PREFERRING LOW a PLUS HIGH b`

As usual in SQL queries, all keywords are not case sensitive, i.e., `PLUS` or `plus` does not make any difference.

## References

W. Kiessling, M. Endres, F. Wenzel (2011): The Preference SQL System - An Overview. IEEE Data Engineering Bulletin, Vol. 34 No. 3, pages 12-19.

S. Mandl, O. Kozachuk, M. Endres, W. Kiessling (2015): Preference Analytics in EXASolution. 16th Conference on Database Systems for Business, Technology, and Web.

## Examples

```
show.query(low(a) * high(b))

show.query(low(a) * high(b), dialect = 'Preference SQL')
```

# Index