Journal of Statistical Software

MMMMMM YYYY, Volume VV, Issue II. http://www.jstatsoft.org/

Generalized and Customizable Sets in R

David Meyer
Wirtschaftsuniversität Wien

Kurt Hornik
Wirtschaftsuniversität Wien

Abstract

We present data structures and algorithms for sets and some generalizations thereof (fuzzy sets, multisets, and fuzzy multisets) available for R through the **sets** package. Fuzzy (multi-)sets are based on dynamically bound fuzzy logic families. Further extensions include user-definable iterators and matching functions.

Keywords: R, set, fuzzy logic, multiset, fuzzy set.

1. Introduction

Only few will deny the importance of sets and set theory, building the fundamentals of modern mathematics. For theory-building typically axiomatic approaches (e.g., Zermelo 1908; Fraenkel 1922) are used. However, even the primal, "naive" concept of sets representing "collections" of distinct objects (Cantor 1895) discarding order and count information seems both natural and practical. The main operation being "is-element-of", sets alone are of limited *practical* use—they most of the times serve as basic building blocks for more complex structures such as relations and generalized sets. A common way is to consider pairs (X, m) with set X ("Universe") and membership function $m : X \to D$ mapping each member to its "grade". The subset of X of elements with non-zero membership is called "support". In *multisets*, elements may appear more than once, i.e., $D = \mathbb{N}_0$ (m is also called the multiplicity function). There are many applications in computer science and other disciplines (For a survey, see, e.g., Singh, Ibrahim, Yohanna, and Singh 2007). In statistics, multisets appear as frequency tables. *Fuzzy sets* have become quite popular since their introduction by Zadeh (1965). Here, the membership function maps into the unit interval. An interesting characteristic of fuzzy sets is that the actual behavior of set operations depends on the underlying fuzzy logic employed, which can be chosen according to domain-specific needs. Fuzzy sets are actively used in fields such as machine learning, engineering and artificial intelligence.

COMMENT. Survey-Refs? Wie viele, welche Gebiete?

Fuzzy multisets (Yager 1986) combine both approaches by allowing each element to map to more than one fuzzy membership grade, i.e., D is the power set of multisets over the unit interval. Examples for the application of fuzzy multisets can be found in the field of information retrieval (e.g., Matthé, Caluwe, de Tré, Hallez, Verstraete, Leman, Cornelis, Moelants, and Gansemans 2006).

The use of sets and variants thereof is common in modern general purpose programming languages: Java and C++ provide corresponding abstract data types (ADTs) in their class libraries, Pascal and Python offer sets as native data type. Surprisingly enough, sets are not standard in many mathematical programming environments such as Matlab and Mathematica, and also R. Although the two latter offer set operations such as union and intersection, these are applied to ordered structures (lists and vectors, respectively), *interpreting* them as sets. When it comes to R, this emulation is far from complete, and occasionally leads to inconsistent behavior. First of all, the existing infrastructure has no clear concept of how to compare elements, leading to possibly confusing and inconsistent behavior when different data types are involved in the computations:

```
> s <- list(1, "1")
> union(s, s)

[[1]]
[1] 1

[[2]]
[1] "1"

> intersect(s, s)

[[1]]
[1] 1
```

The reason is that most of the existing operations rely on `match()` which automatically performs type conversions perturbing in this context. Also, quite a few other basic operations such as the Cartesian product, the power set, the subset predicate, etc., are missing, let alone more specialized operations such as the closure under union. Then, the current facilities do not make use of a class system, making extensions hard (if not impossible). Another consequence is that no distinction can be made between sequences (ordered collections of objects) and sets (unordered collections of objects), which is key for the definition of relations, where both concepts are combined. Also, there is no support for extensions such as fuzzy sets or multisets.

We therefore implemented the **sets** package (**?**) presented here. The main goal was to provide a flexible and customizable basic infrastructure for finite sets and the generalizations mentioned above. As a side effect, the package also provides basic operations for fuzzy logic. The remainder of the paper is structured as follows. In Section 2, we discuss the design rationale of data structures and core algorithms. Section 3 introduces the most important set operations. Section 4 starts with constructors and specific methods for generalized sets, followed by a more focused presentation of the fuzzy logic infrastructure. Section 5 shows how these can further be customized by specifying user-definable matching functions and iterators. Section 6 concludes.

2. Design issues

There are many ways of implementing sets. Choice and efficiency largely depend on the domain range (i.e., the number of possible values for each element). If the domain is relatively small, i.e. in the range of integral data types such as `byte`, `integer`, `word` etc., the probably most efficient representation is an array of bits representing the domain elements like in Pascal (Wirth 1983). Operations such as union and intersection can then straightforwardly be implemented using logical `OR` and `AND`, respectively. This approach obviously fails for intractably large domains (e.g., strings or recursive objects). Without further application knowledge, one needs to resort to generic container ADTs with efficient element access such as hash tables or search trees (for unique elements). Operations can then be implemented following the classical element-based definitions: Union by inserting all elements of the smaller set into the larger one; intersection by creating a new set with all elements of the smaller set also contained in the larger one; etc. Clearly, set comparison must be permutation invariant. Some care is needed for nested structures (e.g., sets of sets of ...). Assume, e.g., the comparison of $A = \{1, \{2, 3\}\}$ and $B = \{1, \{3, 2\}\}$ which clearly are identical. A matching operator would need to check if, e.g., all elements of A are contained in B. If elements were internally stored in their order during creation, the objects representing $\{2, 3\}$ and $\{3, 2\}$ would be different. Comparing two set elements for equality would thus require to recursively compare all elements down the nested structures, which can quickly become unfeasible computationally. This can be simplified by using a canonical ordering during set creation, guaranteeing that identical sets have identical physical representation as well. We chose to sort elements using their Unicode character representation, and to simply store them in a list.

> COMMENT. Mehr erklären?

For the sets package, further limitations are imposed by the extensions presented in Sections 4 and 5: Generalized sets require, for each element, the membership information, and we also support user-defined, high-level matching functions for comparing elements. Since operations defined for generalized sets basically operate on the memberships, it seems appropriate to store these as (generic) vectors. Thus, memberships of different sets can simply be combined element-wise.

A core operation is to match elements of two sets. This is conceptionally done by inserting the elements of the larger one into a hash table (we use environments), and to look up the elements of the smaller set in this table (Knuth 1973, p. 391). As hash key, we use the elements' character representation. Since different objects can have the same representation, we actually store the *indices* of the elements in the list, and match the actual objects using a simple linear search (Note that since the element list is sorted, elements with same representation are grouped, so the search will typically be fast). Objects for sets, generalized sets and customizable sets have `S3` classes `set`, `gset` and `cset`, respectively, with `set` inheriting from `gset` in turn inheriting from `cset`. Accordingly, all operations have `set_` / `gset_` / `cset_` prefixes to give the user the choice of up- or downcasts when objects of different class levels are involved in one computation.

> COMMENT. Kein überzeugendes Argument: warum nicht generische Operatoren und casting via `as.{c,g}set`?

3. Sets

The basic constructor for creating sets is the `set()` function accepting an arbitrary number of R objects as arguments.

```
> s <- set(1L, 2L, 3L)
> print(s)

{1L, 2L, 3L}

> set("test", c, set("a", 2.5), list(1, 2))

{"test", <<function>>, {"a", 2.5}, <<list(2)>>}
```

In addition, there is a generic `as.set()` function coercing suitable objects to sets.

```
> s2 <- as.set(2:4)
> print(s2)

{2L, 3L, 4L}
```

Elements can be named, allowing direct access and replacement not possible otherwise.

```
> snamed <- set(one = 1, 2, three = 3)
> print(snamed)

{one = 1, 2, three = 3}

> snamed[["one"]]

[1] 1
```

There are some basic predicate functions (and corresponding operators) defined for the (in)equality, (proper) sub-(super-)set, and element-of operations. Note that all the `set_is_`*`foo`*`()` functions are vectorized:

```
> set_is_empty(set())

[1] TRUE

> set_is_subset(set(1), set(1, 2))

[1] TRUE

> set(1) <= set(1, 2)

[1] TRUE
```

```
> set_contains_element(set(1, 2, 3), 1)

[1] TRUE

> 1:4 %e% set(1, 2, 3)

[1] FALSE FALSE FALSE FALSE
```

Other than these predicate functions and operators, one can use: `length()` for the cardinality, `c()`, `|` and `+` for the union, `-` for the difference (or complement), `&` for the intersection, `%D%` for the symmetric difference, `*` and `^`n for the (n-fold) Cartesian product (yielding a set of n-tuples), and `2^` for the power set. `set_union()`, `set_intersection()`, and `set_symdiff()` accept more than two arguments.[1]

> COMMENT. Evtl. `+` und `-` nicht erwähnen? Ist eigentlich direkte Summe und Differenz und macht für gsets was anderes.

```
> ## cardinality
> length(s)

[1] 3

> ## complement, union, intersection, symmetric difference:
> s - 1L

{2L, 3L}

> s | set("a") + "b"

{"a", "b", 1L, 2L, 3L}

> s & s2

{2L, 3L}

> s %D% s2

{1L, 4L}

> ## Cartesian product
> s * s2

{(1L, 2L), (1L, 3L), (1L, 4L), (2L, 2L), (2L, 3L), (2L, 4L), (3L, 2L),
 (3L, 3L), (3L, 4L)}
```

[1]The n-ary symmetric difference of a collection of sets consists of all elements contained in an odd number of the sets in the collection.

```
> s ^ 2L

{(1L, 1L), (1L, 2L), (1L, 3L), (2L, 1L), (2L, 2L), (2L, 3L), (3L, 1L),
 (3L, 2L), (3L, 3L)}

> ## power set
> 2L ^ s

{{}, {1L}, {2L}, {3L}, {1L, 2L}, {1L, 3L}, {2L, 3L}, {1L, 2L, 3L}}
```

`set_combn()` returns the set of all subsets of specified length. `closure()` and `reduction()` compute the closure and reduction under union and intersection for set *families* (i.e., set of sets).

```
> set_combn(s, 2L)

{{1L, 2L}, {1L, 3L}, {2L, 3L}}

> cl <- closure(set(set(1), set(2), set(3)))
> print(cl)

{{1}, {2}, {3}, {1, 2}, {1, 3}, {2, 3}, {1, 2, 3}}

> reduction(cl)

{{1}, {2}, {3}}
```

The `Summary()` methods will also work if defined for the elements:

```
> sum(s)

[1] 6

> range(s)

[1] 1 3
```

Because set elements are unordered, it is not allowed to use positional subscripting. However, it is possible to iterate over *all* elements using `for()` and `lapply()`/`sapply()`:

```
> sapply(s, sqrt)

[1] 1.000000 1.414214 1.732051

> for (i in s) print(i)
```

```
[1] 1
[1] 2
[1] 3
```

Using `set_outer()`, it is possible to apply a function on all factorial combinations of the elements of two sets. If only one set is specified, the function is applied to all pairs of this set.

```
> set_outer(set(1, 2), set(1, 2, 3), "/")

  1   2         3
1 1 0.5 0.3333333
2 2 1.0 0.6666667
```

4. Generalized sets

There are several extensions of sets such as *fuzzy sets* and *multisets*. Both can be be seen as special cases of *fuzzy multisets*. We present how they are constructed, and demonstrate the effect of choosing different fuzzy logic families.

4.1. Constructors and specific methods

Generalized sets are created using the `gset()` function, expecting support and membership information. This can be done in four ways:

1. Specify the support only (this yields a "classical" set).
2. Specify support and memberships.
3. Specify support and membership function.
4. Specify a set of elements along with their individual membership grades.

Note that for efficiency reasons, `gset()` will not store elements with zero memberships grades, i.e. really expects the support and not a domain (or universe in the fuzzy world sense).

```
> X <- c("A", "B", "C")
> ## set (X is converted to a set internally).
> gset(support = X)

{"A", "B", "C"}

> ## multiset
> multi <- 1:3
> gset(support = X, memberships = multi)

{"A" [1], "B" [2], "C" [3]}
```

```
> ## fuzzy set
> ms <- c(0.1, 0.3, 1)
> gset(support = X, memberships = ms)

{"A" [0.1], "B" [0.3], "C" [1]}

> ## fuzzy set using a membership function
> f <- function(x) switch(x, A = 0.1, B = 0.2, C = 1, 0)
> gset(universe = X, charfun = f)

{"A" [0.1], "B" [0.2], "C" [1]}
```

For fuzzy multisets, the membership argument expects a list of membership grades, either specified as vectors, or as multisets:

```
> ms2 <- list(c(0.1, 0.3, 0.4), c(1, 1), gset(support = ms, memberships = multi))
> gset(support = X, memberships = ms2)

{"A" [{0.1, 0.3, 0.4}], "B" [{1 [2]}], "C" [{0.1 [1], 0.3 [2], 1 [3]}]}
```

As for sets, the usual operations such as union, intersection, and complement are available. Additionally, the sum and the difference of sets are defined, which add and subtract multiplicities (or memberships for fuzzy sets):

```
> X <- gset(c("A", "B", "C"), 4:6)
> print(X)

{"A" [4], "B" [5], "C" [6]}

> Y <- gset(c("B", "C", "D"), 1:3)
> print(Y)

{"B" [1], "C" [2], "D" [3]}

> ## union vs. sum
> gset_union(X, Y)

{"A" [4], "B" [5], "C" [6], "D" [3]}

> gset_sum(X, Y)

{"A" [4], "B" [6], "C" [8], "D" [3]}

> ## intersection vs. difference
> gset_intersection(X, Y)
```

```
{"B" [1], "C" [2]}

> gset_difference(X, Y)

{"A" [4], "B" [4], "C" [4]}
```

4.2. Fuzzy logic and fuzzy sets

For fuzzy (multi-)sets, the user can choose the logic underlying the operations using the `fuzzy_logic()` function. Fuzzy logics are represented as named lists with four components `N`, `T`, `S`, and `I` containing the corresponding functions for negation, conjunction ("t-norm"), disjunction ("t-conorm"), and implication. The fuzzy logic is selected by calling `fuzzy_logic()` with a character string specifying the fuzzy logic "family", and optional parameters. The exported functions `.N.()`, `.T.()`, `.S.()`, and `.I.()` reflect the currently selected bindings. Available families include: `"Zadeh"` (default), `"drastic"`, `"product"`, `"Lukasiewicz"`, `"Fodor"`, `"Frank"`, `"Hamacher"`, `"Schweizer-Sklar"`, `"Yager"`, `"Dombi"`, `"Aczel-Alsina"`, `"Sugeno-Weber"`, `"Dubois-Prade"`, and `"Yu"` (see Appendix). A call to `fuzzy_logic()` without arguments returns the set fuzzy logic currently set.

```
> x <- 1:10/10
> y <- rev(x)
> .S.(x, y)

 [1] 1.0 0.9 0.8 0.7 0.6 0.6 0.7 0.8 0.9 1.0

> fuzzy_logic("Fodor")
> .S.(x, y)

 [1] 1 1 1 1 1 1 1 1 1 1
```

Fuzzy sets automatically use the fuzzy logic setting in performing set operations:

```
> X <- gset(c("A", "B", "C"), c(0.3, 0.5, 0.8))
> print(X)

{"A" [0.3], "B" [0.5], "C" [0.8]}

> Y <- gset(c("B", "C", "D"), c(0.1, 0.3, 0.9))
> print(Y)

{"B" [0.1], "C" [0.3], "D" [0.9]}

> ## Zadeh-logic (default)
> fuzzy_logic("Zadeh")
> gset_intersection(X, Y)
```

```
{"B" [0.1], "C" [0.3]}

> gset_union(X, Y)

{"A" [0.3], "B" [0.5], "C" [0.8], "D" [0.9]}

> gset_complement(X, Y)

{"B" [0.1], "C" [0.2], "D" [0.9]}

> !X

{"A" [0.7], "B" [0.5], "C" [0.2]}

> ## switch logic to Fodor
> fuzzy_logic("Fodor")
> gset_intersection(X, Y)

{"C" [0.3]}

> gset_union(X, Y)

{"A" [0.3], "B" [0.5], "C" [1], "D" [0.9]}

> gset_complement(X, Y)

{"D" [0.9]}

> !X

{"A" [0.7], "B" [0.5], "C" [0.2]}
```

The `cut()` method for generalized sets "filters" all elements with membership not less then a specified level—the result, thus, is a crisp (multi)set:

```
> cut(X, 0.5)

{"B", "C"}
```

Additionally, there is a `plot()` method for fuzzy (multi-)sets that produces a barplot for the membership vector (see Figure 1):

```
> plot(X)
```

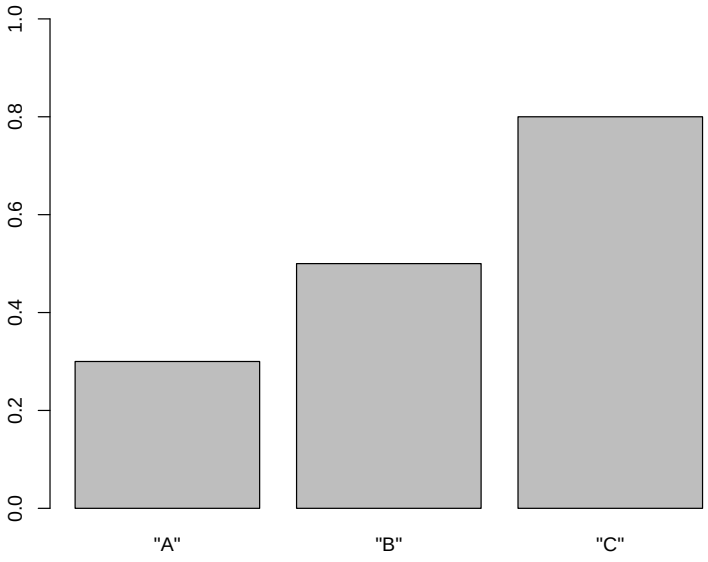

Figure 1: Membership plot for a fuzzy set.

5. User-definable extensions

We added *customizable sets* extending generalized sets in two ways: First, users can control the way elements are matched, i.e., define equivalence classes of elements. Second, arbitrary iteration orders can be specified.

5.1. Matching functions

By default, sets and generalized sets use `identical()` to match elements which is maximal restrictive. Note that this differs from the behavior of `"=="` or `match()` which perform implicit type conversions and thus confound, e.g., `1`, `1L` and `"1"`. In the following example, note that on most computer systems, $3.3 - 2.2$ will not be identical to 1.1 due to numerical issues.

```
> x <- set("1", 1L, 1, 3.3 - 2.2, 1.1)
> print(x)

{"1", 1L, 1, 1.1, 1.1}

> y <- set(1, 1.1, 2L, "2")
> print(y)

{"2", 2L, 1, 1.1}

> 1L %e% y
```

```
[1] FALSE

> x | y

{"1", "2", 1L, 2L, 1, 1.1, 1.1}
```

Customizable sets can be created using the `cset()` constructor, specifying the generalized set and some matching function.

```
> X <- cset(x, matchfun = match)
> print(X)

{"1", 1.1}

> Y <- cset(y, matchfun = match)
> print(Y)

{"2", 1, 1.1}

> 1L %e% Y

[1] TRUE

> X | Y

{"1", "2", 1.1}
```

The specified `foo(x, table)` function needs to be vectorized in the `table` argument. In order to make use of non-vectorized predicates such as `all.equal()`, the **sets** package provides `make_matchfun()` to generate one:

```
> FUN <- make_matchfun(function(x, y) isTRUE(all.equal(x, y)))
> X <- cset(x, matchfun = FUN)
> print(X)

{"1", 1L, 1.1}

> Y <- cset(y, matchfun = FUN)
> print(Y)

{"2", 2L, 1, 1.1}

> 1L %e% Y

[1] TRUE
```

```
> X | Y

{"1", "2", 1L, 2L, 1.1}
```

`set_options()` can be used to conveniently switch the default match and/or order function if a number of cset objects need to be created:

```
> sets_options("matchfun", match)
> cset(x)

{"1", 1.1}

> cset(y)

{"2", 1, 1.1}

> cset(1:3) <= cset(c(1,2,3))

[1] TRUE
```

5.2. Iterators

In addition to specifying matching functions, it is possible to change the order in which iterators such as `as.list()`

COMMENT. But not for(), argh!

process the elements. Note that the behavior of `as.list()` influences the labeling and print methods for customizable sets. Sets and generalized sets by default use the canonical internal ordering for iterations. With customizable sets, a "natural" ordering of elements can be kept by specifying either a permutation vector or an order function.

```
> cset(letters[1:5], orderfun = 5:1)

{"e", "d", "c", "b", "a"}

> FUN <- function(x) order(as.character(x), decreasing = TRUE)
> Z <- cset(letters[1:5], orderfun = FUN)
> print(Z)

{"e", "d", "c", "b", "a"}

> as.character(Z)

[1] "a\n" "b\n" "c\n" "d\n" "e\n"
```

Note that converters for ordered factors keep the order:

```
> o <- ordered(c("a", "b", "a"), levels = c("b", "a"))
> as.set(o)

{b, a}

> as.cset(o)

{a [1], b [2]}
```

Converters for other data types are order-preserving only if elements are unique:

```
> as.cset(c("A", "quick", "brown", "fox"))

{"A", "quick", "brown", "fox"}

> as.cset(c("A", "quick", "brown", "fox", "quick"))

{"A" [1], "brown" [1], "fox" [1], "quick" [2]}
```

6. Conclusion

In this paper, we described the **sets** package for R, providing infrastructure for sets and generalizations thereof such as fuzzy sets, multisets and fuzzy multisets. The fuzzy variants make use of a dynamic fuzzy logic infrastructure offering several fuzzy logic families. Generalized sets are further extended to allow for user-defined iterators and matching functions. Current work focuses on data structures and algorithms for relations, an important application of sets.

A. Available fuzzy logic families

Let us refer to $N(x) = 1 - x$ as the *standard* negation, and, for a t-norm T, let $S(x, y) = 1 - T(1 - x, 1 - y)$ be the *dual* (or complementary) t-conorm. Available specifications and corresponding families are as follows, with the standard negation used unless stated otherwise.

COMMENT. Refs?

`"Zadeh"` Zadeh's logic with $T = \min$ and $S = \max$. Note that the minimum t-norm, also known as the Gödel t-norm, is the pointwise largest t-norm, and that the maximum t-conorm is the smallest t-conorm.

`"drastic"` the drastic logic with t-norm $T(x, y) = y$ if $x = 1$, x if $y = 1$, and 0 otherwise, and complementary t-conorm $S(x, y) = y$ if $x = 0$, x if $y = 0$, and 1 otherwise. Note that the drastic t-norm and t-conorm are the smallest t-norm and largest t-conorm, respectively.

`"product"` the family with the product t-norm $T(x, y) = xy$ and dual t-conorm $S(x, y) = x + y - xy$.

`"Lukasiewicz"` the Lukasiewicz logic with t-norm $T(x, y) = \max(0, x + y - 1)$ and dual t-conorm $S(x, y) = \min(x + y, 1)$.

`"Fodor"` the family with Fodor's *nilpotent minimum* t-norm given by $T(x, y) = \min(x, y)$ if $x + y > 1$, and 0 otherwise, and the dual t-conorm given by $S(x, y) = \max(x, y)$ if $x + y < 1$, and 1 otherwise.

`"Frank"` the family of Frank t-norms T_p, $p \geq 0$, which gives the Zadeh, product and Lukasiewicz t-norms for $p = 0$, 1, and ∞, respectively, and otherwise is given by $T(x, y) = \log_p(1 + (p^x - 1)(p^y - 1)/(p - 1))$.

`"Hamacher"` the three-parameter family of Hamacher, with negation $N_\gamma(x) = (1-x)/(1+\gamma x)$, t-norm $T_\alpha(x, y) = xy/(\alpha + (1 - \alpha)(x + y - xy))$, and t-conorm $S_\beta(x, y) = (x + y + (\beta - 1)xy)/(1 + \beta xy)$, where $\alpha \geq 0$ and $\beta, \gamma \geq -1$. This gives a deMorgan triple iff $\alpha = (1 + \beta)/(1 + \gamma)$.

The following parametric families are obtained by combining the corresponding families of t-norms with the standard negation and complementary t-conorm.

`"Schweizer-Sklar"` the Schweizer-Sklar family T_p, $-\infty \leq p \leq \infty$, which gives the Zadeh (minimum), product and drastic t-norms for $p = -\infty$, 0, and ∞, respectively, and otherwise is given by $T_p(x, y) = \max(0, (x^p + y^p - 1)^{1/p})$.

`"Yager"` the Yager family T_p, $p \geq 0$, which gives the drastic and minimum t-norms for $p = 0$ and ∞, respectively, and otherwise is given by $T_p(x, y) = \max(0, 1 - ((1 - x)^p + (1 - y)^p)^{1/p})$.

`"Dombi"` the Dombi family T_p, $p \geq 0$, which gives the drastic and minimum t-norms for $p = 0$ and ∞, respectively, and otherwise is given by $T_p(x, y) = 0$ if $x = 0$ or $y = 0$, and $T_p(x, y) = 1/(1 + ((1/x - 1)^p + (1/y - 1)^p)^{1/p})$ if both $x > 0$ and $y > 0$.

`"Aczel-Alsina"` the family of t-norms T_p, $p \geq 0$, introduced by Aczél and Alsina, which gives the drastic and minimum t-norms for $p = 0$ and ∞, respectively, and otherwise is given by $T_p(x, y) = \exp(-(|\log(x)|^p + |\log(y)|^p)^{1/p})$.

`"Sugeno-Weber"` the family of t-norms T_p, $-1 \leq p \leq \infty$, introduced by Weber with dual t-conorms introduced by Sugeno, which gives the drastic and product t-norms for $p = -1$ and ∞, respectively, and otherwise is given by $T_p(x, y) = \max(0, (x+y-1+pxy)/(1+p))$.

`"Dubois-Prade"` the family of t-norms T_p, $0 \leq p \leq 1$, introduced by Dubois and Prade, which gives the minimum and product t-norms for $p = 0$ and 1, respectively, and otherwise is given by $T_p(x, y) = xy/\max(x, y, p)$.

`"Yu"` the family of t-norms T_p, $p \geq -1$, introduce by Yu, which gives the product and drastic t-norms for $p = -1$ and ∞, respectively, and otherwise is given by $T(x, y) = \max(0, (1 + p)(x + y - 1) - pxy)$.

References

Cantor G (1895). "Beiträge zur Begründung der transfiniten Mengenlehre." In "Mathematische Annalen," volume 46, pp. 481–512.

Fraenkel AA (1922). "Über die Grundlagen der Cantor-Zermeloschen Mengenlehre." In "Mathematische Annalen," volume 86, pp. 230–237.

Knuth DE (1973). *The Art of Computer Programming*, volume 3. Addison-Wesley, Reading.

Matthé T, Caluwe RD, de Tré G, Hallez A, Verstraete J, Leman M, Cornelis O, Moelants D, Gansemans J (2006). "Similarity Between Multi-valued Thesaurus Attributes: Theory and Application in Multimedia Systems." In "Flexible Query Answering systems," Lecture Notes in Computer Science, pp. 331–342. Springer.

Singh D, Ibrahim A, Yohanna T, Singh J (2007). "An overview of the applications of multisets." *Novi Sad Journal of Mathematics*, **37**(3), 73–92.

Wirth N (1983). *Algorithmen und Datenstrukturen.* Teubner, Stuttgart.

Yager RR (1986). "On the theory of bags." *International Journal of General Systems*, **13**, 23–37.

Zadeh LA (1965). "Fuzzy sets." *Information and Control*, **8**(3), 338–353.

Zermelo E (1908). "Untersuchungen über die Grundlagen der Mengenlehre." In "Mathematische Annalen," volume 65, pp. 261–281.

Affiliation:

David Meyer
Department of Information Systems and Operations
E-mail: David.Meyer@wu-wien.ac.at
URL: http://wi.wu-wien.ac.at/~meyer/

Kurt Hornik
Department of Statistics and Mathematics
E-mail: Kurt.Hornik@wu-wien.ac.at
URL: http://statmath.wu-wien.ac.at/~hornik/

Wirtschaftsuniversität Wien
Augasse 2–6
1090 Wien, Austria

Journal of Statistical Software
MMMMMM YYYY, Volume VV, Issue II.
http://www.jstatsoft.org/

Submitted: yyyy-mm-dd
Accepted: yyyy-mm-dd