Good Relations with R

David Meyer and Kurt Hornik

2007-06-24

Given k sets of objects $X_1, \ldots, X_k$, a k-ary relation R on $D(R) = (X_1, \ldots, X_k)$ is a subset $G(R)$ of the Cartesian product $X_1 \times \cdots \times X_k$. I.e., $D(R)$ is a k-tuple of sets and $G(R)$ is a set of k-tuples. We refer to $D(R)$ and $G(R)$ as the *domain* and the *graph* of the relation R, respectively (alternative notions are that of *ground* and *figure*, respectively).

Relations are a very fundamental mathematical concept: well-known examples include the linear order defined on the set of integers, the equivalence relation, notions of preference relations used in economics and political sciences, etc. Package **relations** provides data structures along with common basic operations for relations and also relation ensembles (collections of relations with the same domain), as well as various algorithms for finding suitable consensus relations for given relation ensembles. In addition, the package also includes support for sets and tuples of R objects upon which relations are built.

1 Sets and Tuples

There is only rudimentary support in base R for *sets*. Typically, they are represented using atomic or recursive vectors (lists), and one can use operations such as `union()`, `intersect()`, `setdiff()`, `setequal()`, and `is.element()` to emulate set operations. However, there are several drawbacks: first of all, quite a few other operations such as the Cartesian product, the power set, the subset predicate, etc., are missing. 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. Therefore, we decided to add more formalized and extended support for sets, and, because they are needed for Cartesian products, also for tuples.

The *tuple* functions in package **relations** represent basic infrastructure for handling tuples of general (R) objects. They are used, e.g., to correctly represent Cartesian products of sets, resulting in a set of tuples (see below). Although tuple objects should behave like “ordinary” vectors for the most common operations (see examples), some functions may yield unexpected results (e.g., `table()`) or simply not work (e.g., `plot()`) since tuple objects are in fact list objects internally. There are several constructors: `tuple()` for arbitrarily many objects, and `singleton()`, `pair()`, and `triple()` for tuples of lengths 1, 2 and 3, respectively. Note that tuple elements can be named.

```
> tuple(1, 2, 3, TRUE)

(1, 2, 3, TRUE)

> triple(1, 2, 3)

(1, 2, 3)

> pair(Name = "David", Height = 185)

(Name = David, Height = 185)
```

```
> tuple_is_triple(triple(1, 2, 3))
```

```
[1] TRUE
```

```
> tuple_is_ntuple(tuple(1, 2, 3, 4), 4)
```

```
[1] TRUE
```

```
> as.tuple(1:3)
```

```
(1, 2, 3)
```

```
> c(tuple("a", "b"), 1)
```

```
(a, b, 1)
```

```
> tuple(1, 2, 3) * tuple(2, 3, 4)
```

```
(2, 6, 12)
```

```
> rep(tuple(1, 2, 3), 2)
```

```
(1, 2, 3, 1, 2, 3)
```

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

```
> sum(tuple(1, 2, 3))
```

```
[1] 6
```

```
> range(tuple(1, 2, 3))
```

```
[1] 1 3
```

In addition, there is a `tuple_outer()` function to apply functions to all combinations of tuple elements. Note that `tuple_outer()` will also work for regular vectors and thus can really be seen as an extension of `outer()`:

```
> tuple_outer(pair(1, 2), triple(1, 2, 3))
```

```
  1 2 3
1 1 2 3
2 2 4 6
```

```
> tuple_outer(1:5, 1:4, "^")
```

```
  1  2   3   4
1 1  1   1   1
2 2  4   8  16
3 3  9  27  81
4 4 16  64 256
5 5 25 125 625
```

The basic constructor for creating sets is the `set()` function accepting an arbitrary number of R objects as arguments (which can be named). In addition, there is a generic `as.set()` for converting suitable objects to sets.

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

```
{1, 2, 3}

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

{one = 1, 2, three = 3}

> snamed[["one"]]

[1] 1

> set(c, "test", list(1, 2, 3))

{<<function>>, test, <<list(3)>>}

> set(set(), set(1))

{{}, {1}}

> s2 <- as.set(2:5)
> s2

{2, 3, 4, 5}
```

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

```
> set_is_empty(set())

[1] TRUE

> set_is_equal(set(1), set(1))

[1] TRUE

> set(1) == set(1)

[1] TRUE

> set(1) != set(2)

[1] TRUE

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

[1] TRUE

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

[1] TRUE

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

[1] TRUE

> set_is_proper_subset(set(1), set(1))

[1] FALSE

> set(1) < set(1)
```

```
[1] FALSE
> set(1, 2) > set(1)
[1] TRUE
> set_is_element(1, set(1, 2, 3))
[1] TRUE
> 1 %e% set(1, 2, 3)
[1] TRUE
> set_is_element(1:4, set(1, 2, 3))
[1]  TRUE  TRUE  TRUE FALSE
> 1:4 %e% set(1, 2, 3)
[1]  TRUE  TRUE  TRUE FALSE
```

`c()`, `+`, and `|` for the union, `-` for the 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] The `length` method for sets gives the cardinality. `set_combn()` returns the set of all subsets of specified length. Note that (currently) the `rep()` method for sets will just return its argument since set elements are unique.

```
> length(s)
[1] 3
> length(set())
[1] 0
> s - 1
{2, 3}
> s + set("a")
{1, 2, 3, a}
> s | set("a")
{1, 2, 3, a}
> s & s2
{2, 3}
> s %D% s2
{1, 2, 3, 4, 5}
> set(1, 2, 3) - set(1, 2)
```

[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.

```
{3}
> set_intersection(set(1, 2, 3), set(2, 3, 4), set(3, 4, 5))
{3}
> set_union(set(1, 2, 3), set(2, 3, 4), set(3, 4, 5))
{1, 2, 3, 4, 5}
> set_symdiff(set(1, 2, 3), set(2, 3, 4), set(3, 4, 5))
{1, 3, 5}
> s * s2
{(1, 2), (2, 2), (3, 2), (1, 3), (2, 3), (3, 3), (1, 4), (2, 4), (3,
 4), (1, 5), (2, 5), (3, 5)}
> s * s
{(1, 1), (2, 1), (3, 1), (1, 2), (2, 2), (3, 2), (1, 3), (2, 3), (3,
 3)}
> s^2
{(1, 1), (2, 1), (3, 1), (1, 2), (2, 2), (3, 2), (1, 3), (2, 3), (3,
 3)}
> s^3
{(1, 1, 1), (2, 1, 1), (3, 1, 1), (1, 2, 1), (2, 2, 1), (3, 2, 1), (1,
 3, 1), (2, 3, 1), (3, 3, 1), (1, 1, 2), (2, 1, 2), (3, 1, 2), (1, 2,
 2), (2, 2, 2), (3, 2, 2), (1, 3, 2), (2, 3, 2), (3, 3, 2), (1, 1, 3),
 (2, 1, 3), (3, 1, 3), (1, 2, 3), (2, 2, 3), (3, 2, 3), (1, 3, 3), (2,
 3, 3), (3, 3, 3)}
> 2^s
{{}, {1}, {2}, {3}, {1, 2}, {1, 3}, {2, 3}, {1, 2, 3}}
> set_combn(as.set(1:3), 2)
{{1, 2}, {1, 3}, {2, 3}}
```

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

```
> sum(s)
[1] 6
> range(s)
[1] 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
```

```
> X <- set_outer(set(1, 2), set(1, 2, 3), set)
> X[[2, 3]]
```

```
{2, 3}
```

```
> set_outer(2^set(1, 2, 3), set_is_subset)
```

```
             {}   {1}   {2}   {3} {1, 2} {1, 3} {2, 3} {1, 2, 3}
{}         TRUE  TRUE  TRUE  TRUE   TRUE   TRUE   TRUE      TRUE
{1}       FALSE  TRUE FALSE FALSE   TRUE   TRUE  FALSE      TRUE
{2}       FALSE FALSE  TRUE FALSE   TRUE  FALSE   TRUE      TRUE
{3}       FALSE FALSE FALSE  TRUE  FALSE   TRUE   TRUE      TRUE
{1, 2}    FALSE FALSE FALSE FALSE   TRUE  FALSE  FALSE      TRUE
{1, 3}    FALSE FALSE FALSE FALSE  FALSE   TRUE  FALSE      TRUE
{2, 3}    FALSE FALSE FALSE FALSE  FALSE  FALSE   TRUE      TRUE
{1, 2, 3} FALSE FALSE FALSE FALSE  FALSE  FALSE  FALSE      TRUE
```

Because set elements are unordered, it is not sensible 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
```

2 Relations and Relation Ensembles

2.1 Relations

For a k-ary relation R with domain $D(R) = (X_1, \dots, X_k)$, we refer to $s = (s_1, \dots, s_k)$, where each s_i gives the cardinality of X_i, as the *size* of the relation. Note that often, relations are identified with their graph; strictly speaking, the relation is the *pair* $(D(R), G(R))$. We say that a k-tuple t is *contained* in the relation R iff it is an element of $G(R)$. The *incidence* (array) $I(R)$ of R is a k-dimensional 0/1 array of size s whose elements indicate whether the corresponding k-tuples are contained in R or not.

Package **relations** implements finite relations as an S3 class which allows for a variety of representations (even though currently, only dense array representations of the incidences are employed). Other than by the generator `relation()`, relations can be obtained by coercion via the generic function `as.relation()`, which has methods for at least logical and numeric vectors, unordered and ordered factors, arrays including matrices, and data frames. Unordered factors are coerced to equivalence relations; ordered factors and numeric vectors are coerced to order relations. Logical vectors give unary relations (predicates). A (feasible) k-dimensional array is taken as the incidence of a k-ary relation. Finally, a data frame is taken as a relation table (object by attribute representation of the relation graph). Note that missing values will be propagated in the coercion.

```
> R <- relation(graph = data.frame(A = c(1, 1:3), B = c(2:4, 4)))
> relation_domain(R)
```

```
Relation domain:
A pair (A, B) with elements:
{1, 2, 3}
{2, 3, 4}

> relation_graph(R)

Relation graph:

A set with pairs (A, B):
(1, 2)
(1, 3)
(2, 4)
(3, 4)

> as.tuple(R)

(Domain = (A = {1, 2, 3}, B = {2, 3, 4}), Graph = {(1, 2), (1, 3), (2,
 4), (3, 4)})

> relation_incidence(R)

Incidences:
   B
A   2 3 4
  1 1 1 0
  2 0 0 1
  3 0 0 1

> R <- relation(graph = set(tuple(1, 2), tuple(1, 3), tuple(2,
+     4), tuple(3, 4)))
> relation_incidence(R)

Incidences:
  2 3 4
1 1 1 0
2 0 0 1
3 0 0 1

> R <- relation(domain = set(c, "test"), graph = set(tuple(c, c),
+     tuple(c, "test")))
> relation_incidence(R)

Incidences:
              X
X              <<function>> test
  <<function>>            1    1
  test                    0    0

> as.relation(1:3)

A binary relation of size 3 x 3.

> relation_graph(as.relation(c(TRUE, FALSE, TRUE)))
```

```
Relation graph:

A set with singletons:
(1)
(3)

> relation_graph(as.relation(factor(c("A", "B", "A"))))

Relation graph:

A set with pairs:
(1, 1)
(3, 1)
(2, 2)
(1, 3)
(3, 3)
```

The *characteristic function* f_R (sometimes also referred to as indicator function) of a relation R is the predicate (Boolean-valued) function on the Cartesian product $X_1 \times \cdots \times X_k$ such that $f_R(t)$ is true iff the k-tuple t is in $G(R)$. Characteristic functions can both be recovered from a relation via `relation_charfun()`, and be used in the generator for the creation. In the following, `R` represents "a divides b":

```
> divides <- function(a, b) b %% a == 0
> R <- relation(domain = list(1 : 10, 1 : 10), charfun = divides)
> R

A binary relation of size 10 x 10.

> "%|%" <- relation_charfun(R)
> 2 %|% 6

[1] TRUE

> c(2, 3, 4) %|% 6

[1]  TRUE  TRUE FALSE

> 2 %|% c(2, 3, 6)

[1]  TRUE FALSE  TRUE

> "%|%"(2, 6)

[1] TRUE
```

Quite a few `relation_is_`*`foo`*`()` predicate functions are available. For example, relations with arity 2, 3, and 4 are typically referred to as *binary*, *ternary*, and *quaternary* relations, respectively—the corresponding functions in package **relations** are `relation_is_binary()`, `relation_is_ternary()`, etc. For binary relations R, it is customary to write xRy iff (x, y) is contained in R. For predicates available on binary relations, see Table 1. An *endorelation* on X (or binary relation *over* X) is a binary relation with domain (X, X). Endorelations may or may not have certain basic properties (such as transitivity, reflexivity, etc.) which can be tested in **relations** using the corresponding predicates (see Table 2 for an overview). Some combinations of these basic properties have special names because of their widespread use (such as linear order, or preference), and can again be tested using the functions provided (see Table 3).

left-total	for all x there is at least one y such that xRy.
right-total	for all y there is at least one x such that xRy.
functional	for all x there is at most one y such that xRy.
surjective	the same as right-total.
injective	for all y there is at most one x such that xRy.
bijective	left-total, right-total, functional and injective.

Table 1: Some properties *foo* of binary relations—the predicates in **relations** are `relation_is_`*`foo`*`()` (with hyphens replaced by underscores).

reflexive	xRx for all x.
irreflexive	there is no x such that xRx.
coreflexive	xRy implies $x = y$.
symmetric	xRy implies yRx.
asymmetric	xRy implies that not yRx.
antisymmetric	xRy and yRx imply that $x = y$.
transitive	xRy and yRz imply that xRz.
complete	for all x and y, xRy or yRx.

Table 2: Some properties *bar* of endorelations—the predicates in **relations** are `relation_is_`*`bar`*`()`.

preorder	reflexive and transitive.
quasiorder	the same as preorder.
equivalence	a symmetric preorder.
weak order	complete and transitive.
preference	the same as weak order.
partial order	an antisymmetric preorder.
strict partial order	irreflexive, transitive and antisymmetric.
linear order	a complete partial order.
strict linear order	a complete strict partial order.
tournament	complete and antisymmetric.

Table 3: Some categories *baz* of endorelations—the predicates in **relations** are `relation_is_`*`baz`*`()` (with spaces replaced by underscores).

```
> R <- as.relation(1:5)
> relation_is_binary(R)

[1] TRUE

> relation_is_transitive(R)

[1] TRUE

> relation_is_partial_order(R)

[1] TRUE
```

Relations with the same domain can naturally be ordered according to their graphs. I.e., $R_1 \leq R_2$ iff $G(R_1)$ is a subset of $G(R_2)$, or equivalently, if every k-tuple t contained in R_1 is also contained in R_2. This induces a lattice structure, with meet (greatest lower bound) and join (least upper bound) the intersection and union of the graphs, respectively, also known as the *intersection* and *union* of the relations. The least element moves metric on this lattice is the *symmetric difference metric*, i.e., the cardinality of the symmetric difference of the graphs (the number of tuples in exactly one of the relation graphs). This "symdiff" dissimilarity between (ensembles of) relations can be computed by `relation_dissimilarity()`.

```
> x <- matrix(0, 3, 3)
> R1 <- as.relation(row(x) >= col(x))
> R2 <- as.relation(row(x) <= col(x))
> R3 <- as.relation(row(x) < col(x))
> relation_incidence(max(R1, R2))

Incidences:
  1 2 3
1 1 1 1
2 1 1 1
3 1 1 1

> relation_incidence(min(R1, R2))

Incidences:
  1 2 3
1 1 0 0
2 0 1 0
3 0 0 1

> R3 < R2

[1] TRUE

> relation_dissimilarity(min(R1, R2), max(R1, R2))

     [,1]
[1,]    6
```

The *complement* of a relation R is the relation with domain $D(R)$ whose graph is the complement of $G(R)$, i.e., which contains exactly the tuples not contained in R. For binary relations R and S with domains (X, Y) and (Y, Z), the *composition* of R and S is defined by taking xSz iff there is a y such that xRy and ySz. The *dual* (or *converse*) R^* of the relation R with domain (X, Y) is the relation with domain (Y, X) such that xR^*y iff yRx.

```
> relation_incidence(R1 * R2)

Incidences:
  1 2 3
1 1 1 1
2 1 1 1
3 1 1 1

> relation_incidence(!R1)

Incidences:
  1 2 3
1 0 1 1
2 0 0 1
3 0 0 0

> relation_incidence(t(R2))

Incidences:
  1 2 3
1 1 0 0
2 1 1 0
3 1 1 1
```

There is a `plot()` method for certain endorelations (currently, only complete or antisymmetric transitive relations are supported) provided that package **Rgraphviz** (Gentry and Long, 2007) is available, creating a Hasse diagram of the relation. The following code produces the Hasse diagram corresponding to the inclusion relation on the power set of $\{1, 2, 3\}$ which is a partial order (see Figure 1).

```
> ps <- 2^set("a", "b", "c")
> inc <- set_outer(ps, "<=")
> plot(relation(incidence = inc))
```

2.2 Relational Algebra

In addition to the basic operations defined on relations, the package provides functionality similar to the corresponding operations defined in relational algebra theory as introduced by Codd (1970). Note, however, that domains in database relations, unlike the concept of relations we use here, are unordered. In fact, a database relation ("table") is defined as a set of elements called "tuples", where the "tuple" components are named, but unordered. Thus, a "tuple" in this Codd sense is a set of mappings from the attribute names into the union of the attribute domains. The functions defined in **relations**, however, preserve and respect the column ordering.

The *projection* of a relation on a specified margin (i.e., a vector of domain names or indices) is the relation obtained when all tuples are restricted to this margin. As a consequence, duplicate tuples are removed. The corresponding function in package **relations** is `relation_projection()`.

```
> Person <- data.frame(Name = c("Harry", "Sally", "George", "Helena",
+     "Peter"), Age = c(34, 28, 29, 54, 34), Weight = c(80, 64,
+     70, 54, 80), stringsAsFactors = FALSE)
> Person <- as.relation(Person)
> relation_table(Person)

 Name   Age Weight
 Harry  34  80
```

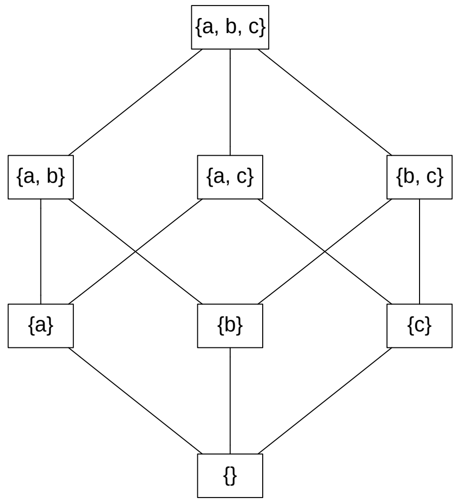

Figure 1: Hasse Diagram of the inclusion relation on the power set of $\{1, 2, 3\}$.

```
Peter  34  80
Sally  28  64
George 29  70
Helena 54  54
```

```
> relation_table(relation_projection(Person, c("Age", "Weight")))
```

```
Age Weight
34  80
28  64
29  70
54  54
```

(Note that Harry and Peter have the same age and weight.)

The *selection* of a relation is the relation obtained by taking a subset of the relation graph, defined by some logical expression. The corresponding function in **relations** is `relation_selection()`.

```
> relation_table(R1 <- relation_selection(Person, Age < 29))
```

```
Name  Age Weight
Sally 28  64
```

```
> relation_table(R2 <- relation_selection(Person, Age >= 34))
```

```
Name   Age Weight
Harry  34  80
Peter  34  80
Helena 54  54
```

```
> relation_table(R3 <- relation_selection(Person, Age == Weight))
```

```
Name   Age Weight
Helena 54  54
```

The *union* of two relations simply combines the graph elements of both relations; the *complement* of two relations X and Y removes the tuples of Y from X. One can use `-` as a shortcut for `relation_complement()`, and `%U%` or `|` for `relation_union()`. The difference between `%U%` and `|` is that the latter only works for identical domains.

```
> relation_table(R1 %U% R2)

 Name   Age Weight
 Harry  34  80
 Peter  34  80
 Sally  28  64
 Helena 54  54

> relation_table(R2 | R3)

 Name   Age Weight
 Harry  34  80
 Peter  34  80
 Helena 54  54

> relation_table(Person - R2)

 Name   Age Weight
 Sally  28  64
 George 29  70
```

The *intersection* (*symmetric difference*) of two relations is the relation with all tuples they have (do not have) in common. One can use `&` instead of `relation_intersection()` in case of identical domains.

```
> relation_table(relation_intersection(R2, R3))

 Name   Age Weight
 Helena 54  54

> relation_table(R2 & R3)

 Name   Age Weight
 Helena 54  54

> relation_table(relation_symdiff(R2, R3))

 Name  Age Weight
 Harry 34  80
 Peter 34  80
```

The *Cartesian product* of two relations is obtained by basically building the Cartesian product of all graph elements, but combining the resulting pairs into single tuples. A shortcut for `relation_cartesian()` is `%><%`.

```
> Employee <- data.frame(Name = c("Harry", "Sally", "George", "Harriet",
+     "John"), EmpId = c(3415, 2241, 3401, 2202, 3999), DeptName = c("Finance",
+     "Sales", "Finance", "Sales", "N.N."), stringsAsFactors = FALSE)
> Employee <- as.relation(Employee)
> relation_table(Employee)
```

```
 Name    EmpId DeptName
 Harry   3415  Finance
 George  3401  Finance
 Sally   2241  Sales
 Harriet 2202  Sales
 John    3999  N.N.
```

```
> Dept <- data.frame(DeptName = c("Finance", "Sales", "Production"),
+     Manager = c("George", "Harriet", "Charles"), stringsAsFactors = FALSE)
> Dept <- as.relation(Dept)
> relation_table(Dept)
```

```
 DeptName   Manager
 Finance    George
 Sales      Harriet
 Production Charles
```

```
> relation_table(Employee %><% Dept)
```

```
 Name    EmpId DeptName DeptName   Manager
 Harry   3415  Finance  Finance    George
 George  3401  Finance  Finance    George
 Sally   2241  Sales    Finance    George
 Harriet 2202  Sales    Finance    George
 John    3999  N.N.     Finance    George
 Harry   3415  Finance  Sales      Harriet
 George  3401  Finance  Sales      Harriet
 Sally   2241  Sales    Sales      Harriet
 Harriet 2202  Sales    Sales      Harriet
 John    3999  N.N.     Sales      Harriet
 Harry   3415  Finance  Production Charles
 George  3401  Finance  Production Charles
 Sally   2241  Sales    Production Charles
 Harriet 2202  Sales    Production Charles
 John    3999  N.N.     Production Charles
```

The *division* of relation X by relation Y is the reversed Cartesian product. The result is a relation with the domain unique to X and containing the maximum number of tuples which, multiplied by Y, are contained in X. The *remainder* of this operation is the complement of X and the division of X by Y. Note that for both operations, the domain of Y must be contained in the domain of X. The shortcuts for `relation_division()` and `relation_remainder()` are `%/%` and `%%`, respectively.

```
> Completed <- data.frame(Student = c("Fred", "Fred", "Fred", "Eugene",
+     "Eugene", "Sara", "Sara"), Task = c("Database1", "Database2",
+     "Compiler1", "Database1", "Compiler1", "Database1", "Database2"),
+     stringsAsFactors = FALSE)
> Completed <- as.relation(Completed)
> relation_table(Completed)
```

```
 Student Task
 Fred    Database1
 Eugene  Database1
 Sara    Database1
 Fred    Database2
```

```
 Sara    Database2
 Fred    Compiler1
 Eugene  Compiler1
```

```
> DBProject <- data.frame(Task = c("Database1", "Database2"), stringsAsFactors = FALSE)
> DBProject <- as.relation(DBProject)
> relation_table(DBProject)
```

```
 Task
 Database1
 Database2
```

```
> relation_table(Completed%/%DBProject)
```

```
 Student
 Fred
 Sara
```

```
> relation_table(Completed%%DBProject)
```

```
 Student Task
 Eugene  Database1
 Fred    Compiler1
 Eugene  Compiler1
```

The (natural) *join* of two relations is their Cartesian product, restricted to the subset where the elements of the common attributes do match. The left/right/full outer join of two relations X and Y is the union of $X/Y/(X$ and $Y)$, and the inner join of X and Y. The implementation of `relation_join()` uses `merge()`, and so the left/right/full outer joins are obtained by setting `all.x`/`all.y`/`all` to `TRUE` in `relation_join()`. The domains to be matched are specified using `by`. Alternatively, one can use the operators `%|><|%`, `%=><%`, `%><=%`, and `%=><=%` for the natural join, left join, right join, and full outer join, respectively.

```
> relation_table(Employee %|><|% Dept)
```

```
 Name    EmpId DeptName Manager
 Harry   3415  Finance  George
 George  3401  Finance  George
 Sally   2241  Sales    Harriet
 Harriet 2202  Sales    Harriet
```

```
> relation_table(Employee %=><% Dept)
```

```
 Name    EmpId DeptName Manager
 Harry   3415  Finance  George
 George  3401  Finance  George
 John    3999  N.N.     NA
 Sally   2241  Sales    Harriet
 Harriet 2202  Sales    Harriet
```

```
> relation_table(Employee %><=% Dept)
```

```
 Name    EmpId DeptName   Manager
 Harry   3415  Finance    George
 George  3401  Finance    George
 NA         NA  Production Charles
 Sally   2241  Sales      Harriet
 Harriet 2202  Sales      Harriet
```

```
> relation_table(Employee %=><=% Dept)

 Name    EmpId DeptName   Manager
 Harry   3415  Finance    George
 George  3401  Finance    George
 John    3999  N.N.       NA
 NA        NA  Production Charles
 Sally   2241  Sales      Harriet
 Harriet 2202  Sales      Harriet
```

The left (right) *semijoin* of two relations X and Y is the join of these, projected to the attributes of X (Y). Thus, it yields all tuples of X (Y) participating in the join of X and Y. Shortcuts for `relation_semijoin()` are `%|><%` and `%><|%` for left and right semijoin, respectively.

```
> relation_table(Employee %|><% Dept)

 Name    EmpId DeptName
 Harry   3415  Finance
 George  3401  Finance
 Sally   2241  Sales
 Harriet 2202  Sales

> relation_table(Employee %><|% Dept)

 DeptName Manager
 Finance  George
 Sales    Harriet
```

The left (right) *antijoin* of two relations X and Y is the complement of X (Y) and the join of both, projected to the attributes of X (Y). Thus, it yields all tuples of X (Y) *not* participating in the join of X and Y. Shortcuts for `relation_antijoin()` are `%|>%` and `%<|%` for left and right antijoin, respectively.

```
> relation_table(Employee %|>% Dept)

 Name EmpId DeptName
 John 3999  N.N.

> relation_table(Employee %<|% Dept)

 DeptName   Manager
 Production Charles
```

2.3 Relation Ensembles

"Relation ensembles" are collections of relations $R_i = (D, G_i)$ with the same domain D and possibly different graphs G_i. Such ensembles are implemented as suitably classed lists of relation objects (of class relation_ensemble and inheriting from tuple), making it possible to use `lapply()` for computations on the individual relations in the ensemble. Relation ensembles can be created via `relation_ensemble()`, or by coercion via the generic function `as.relation_ensemble()` which has methods for at least data frames (regarding each variable as a separate relation). Available methods for relation ensembles include those for subscripting, `c()`, `t()`, `rep()`, `print()`, and `plot()`. In addition, there are summary methods defined (`min()`, `max()`, and `range()`). Other operations work element-wise like on tuples due to the inheritance.

The Cetacea data set (Vescia, 1985) is a data frame with 15 variables relating to morphology, osteology, or behavior, with both self-explanatory names and levels, and a common zoological classification (variable `CLASS`) for 36 types of cetacea. We consider each variable an equivalence relation on the objects, excluding 2 variables with missing values, giving a relation ensemble of length 14 (number of complete variables in the data set).

```
> data("Cetacea")
> ind <- sapply(Cetacea, function(s) all(!is.na(s)))
> relations <- as.relation_ensemble(Cetacea[, ind])
> print(relations)
```

```
An ensemble of 14 relations of size 36 x 36.
```

Available methods for relation ensembles allow to determine duplicated (relation) entries, to replicate and combine, and extract unique elements:

```
> any(duplicated(relations))
```

```
[1] FALSE
```

```
> thrice <- c(rep(relations, 2), relations)
> all.equal(unique(thrice), relations)
```

```
[1] "names for current but not for target"
```

Note that `unique()` does not preserve attributes, and hence names. In case one wants otherwise, one can subscript by a logical vector indicating the non-duplicated entries:

```
> all.equal(thrice[!duplicated(thrice)], relations)
```

```
[1] TRUE
```

Relation (cross-)dissimilarities can be computed for relations and ensembles thereof:

```
> relation_dissimilarity(relations[1:2], relations["CLASS"])
```

```
                 CLASS
NECK               584
FORM_OF_THE_HEAD   330
```

To determine which single variable is "closest" to the zoological classification:

```
> d <- relation_dissimilarity(relations)
> sort(as.matrix(d)[, "CLASS"])[-1]
```

```
                        BLOW_HOLE                        DORSAL_FIN 
                              190                               240 
                     SET_OF_TEETH                          FLIPPERS 
                              288                               298 
                 FORM_OF_THE_HEAD                           FEEDING 
                              330                               382 
                          HABITAT                              BEAK 
                              398                               456 
                            COLOR LONGITUDINAL_FURROWS_ON_THE_THROAT 
                              494                               506 
               CERVICAL_VERTEBRAE                  SIZE_OF_THE_HEAD 
                              508                               542 
                             NECK 
                              584 
```

There is also an Ops group method for relation ensembles which works elementwise (in essence, as for tuples):

```
> complement <- !relations
> complement
```

```
An ensemble of 14 relations of size 36 x 36.
```

3 Consensus Relations

Consensus relations "synthesize" the information in the elements of a relation ensemble into a single relation, often by minimizing a criterion function measuring how dissimilar consensus candidates are from the (elements of) the ensemble (the so-called "optimization approach"), typically of the form $L(R) = \sum w_b d(R_b, R)$, where d is a suitable dissimilarity measure and w_b is the case weight given to element R_b of the ensemble (such consensus relations are called "central relations" in Régnier, 1965).

Consensus relations can be computed by `relation_consensus()`, which has the following built-in methods. Apart from Condorcet's, these are applicable to ensembles of endorelations only.

`"Borda"` the consensus method proposed by Borda (1781). For each relation R_b and object x, one determines the Borda/Kendall scores, i.e., the number of objects y such that yR_bx ("wins" in case of orderings). These are then aggregated across relations by weighted averaging. Finally, objects are ordered according to their aggregated scores.

`"Copeland"` the consensus method proposed by Copeland (1951) is similar to the Borda method, except that the Copeland scores are the number of objects y such that yR_bx, minus the number of objects y such that xR_by ("defeats" in case of orderings).

`"Condorcet"` the consensus method proposed by Condorcet (1785), in fact minimizing the criterion function L with d as symmetric difference distance over all possible relations. In the case of endorelations, consensus is obtained by weighting voting, such that xRy if the weighted number of times that xR_by is no less than the weighted number of times that this is not the case. Even when aggregating linear orders, this can lead to intransitive consensus solutions ("effet Condorcet").

`"SD/F"` an exact solver for determining the consensus relation by minimizing the criterion function L with d as symmetric difference distance ("SD") over a suitable class ("("Family")) of endorelations as indicated by F, with values:

- `E` equivalence relations: reflexive, symmetric, and transitive.
- `L` linear orders: complete (hence reflexive), antisymmetric, and transitive.
- `O` partial orders: reflexive, antisymmetric and transitive.
- `P` complete preorders (preference relations, "orderings"): complete (hence reflexive) and transitive.
- `T` tournaments: complete (hence reflexive) and antisymmetric.
- `C` complete relations.
- `A` antisymmetric relations.
- `S` symmetric relations.

These consensus relations are determined by reformulating the consensus problem as an integer program (for the relation incidences), which is solved via package **lpSolve**. See Hornik and Meyer (2007) for details.

In the following, we first show an example of computing a consensus equivalence (i.e., a consensus partition) of 30 felines repeating the classical analysis of Marcotorchino and Michaud (1982). The data comprises 10 morphological and 4 behavioral variables, taken here as different classifications of the same 30 animals:

```
> data("Felines")
> relations <- as.relation_ensemble(Felines)
```

Now fit an equivalence relation to this, and look at the classes:

```
> E <- relation_consensus(relations, "SD/E")
> ids <- relation_class_ids(E)
> split(rownames(Felines), ids)

$`1`
[1] "LION"  "TIGRE"

$`2`
[1] "JAGUAR"  "LEOPARD" "ONCE"    "PUMA"    "NEBUL"   "LYNX"

$`3`
[1] "GUEPARD"

$`4`
 [1] "SERVAL"   "OCELOT"   "CARACAL"  "VIVERRIN" "YAGUARUN" "CHAUS"
 [7] "DORE"     "MERGUAY"  "MARGERIT" "CAFER"    "CHINE"    "BENGALE"
[13] "ROUILLEU" "MALAIS"   "BORNEO"   "NIGRIPES" "MANUL"    "MARBRE"
[19] "TIGRIN"   "TEMMINCK" "ANDES"
```

Next, we demonstrate the computation of consensus preferences, using an example from Cook and Kress (1992, pp. 48ff). The input data is a "preference" matrix of paired comparisons, which we first transform into a tournament.

```
> pm <- matrix(c(0, 1, 0, 1, 1, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0,
+     0, 1, 0, 0, 0, 0, 1, 1, 0), nr = 5, byrow = TRUE, dimnames = list(letters[1:5],
+     letters[1:5]))
> R <- as.relation(1 - pm)
> relation_incidence(R)

Incidences:
  a b c d e
a 1 0 1 0 0
b 1 1 1 0 0
c 0 0 1 1 1
d 1 1 0 1 1
e 1 1 0 0 1

> relation_is_tournament(R)

[1] TRUE
```

Next, we seek a linear consensus order:

```
> L <- relation_consensus(R, "SD/L")
> relation_incidence(L)

Incidences:
  a b c d e
a 1 0 1 0 0
b 1 1 1 0 0
c 0 0 1 0 0
d 1 1 1 1 1
e 1 1 1 0 1
```

or perhaps more conveniently, the class ids sorted according to increasing preference:

```
> relation_class_ids(L)
```

```
a b c d e
4 3 5 1 2
```

Note, however, that this linear order is not unique; we can compute *all* consensus linear orders, and also produce a comparing plot (see Figure 2):

```
> L <- relation_consensus(R, "SD/L", control = list(all = TRUE))
> print(L)
```

```
An ensemble of 2 relations of size 5 x 5.
```

```
> if (require("Rgraphviz")) plot(L)
```

Finally, we compute the closest preference relation with at most 3 indifference classes:

```
> P3 <- relation_consensus(R, "SD/P", control = list(k = 3))
> relation_incidence(P3)
```

```
Incidences:
  a b c d e
a 1 0 0 0 0
b 1 1 0 0 1
c 1 1 1 1 1
d 1 1 1 1 1
e 1 1 0 0 1
```

```
> relation_class_ids(P3)
```

```
a b c d e
3 2 1 1 2
```

(Note again that this preference is not unique; there are 6 consensus preferences with $k = 3$ classes, which can be computed as above by adding `all = TRUE` to the `control` list.)

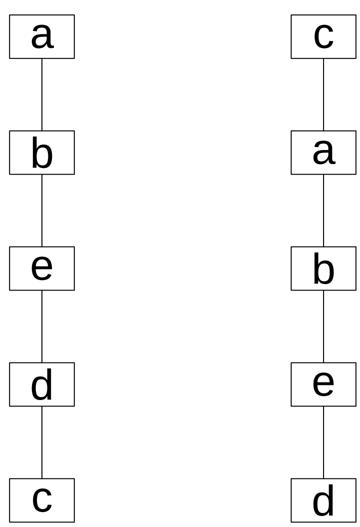

Figure 2: Hasse Diagram of all consensus relations (linear orders) for an example provided by Cook and Kress.

References

J. C. Borda. Mémoire sur les élections au scrutin. Histoire de l'Académie Royale des Sciences, 1781.

E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6): 377–387, 1970.

M. J. A. Condorcet. Essai sur l'application de l'analyse à la probabilité des décisions rendues à la pluralité des voix, 1785. Paris.

W. D. Cook and M. Kress. *Ordinal information and preference structures: decision models and applications*. Prentice-Hall, New York, 1992.

A. Copeland. A reasonable social welfare function. *mimeo*, 1951. University of Michigan.

J. Gentry and L. Long. **Rgraphviz***: Plotting Capabilities for R Graph Objects*, 2007. R package version 1.13.25.

K. Hornik and D. Meyer. Deriving consensus rankings from benchmarking experiments. In R. Decker and H.-J. Lenz, editors, *Advances in Data Analysis (Proceedings of the 30th Annual Conference of the Gesellschaft für Klassifikation e.V., Freie Universität Berlin, March 8–10, 2006*, Studies in Classification, Data Analysis, and Knowledge Organization, pages 163–170. Springer-Verlag, 2007.

F. Marcotorchino and P. Michaud. Agregation de similarites en classification automatique. *Revue de Statistique Appliquée*, 30(2):21–44, 1982. URL http://www.numdam.org/item?id=RSA_1982__30_2_21_0.

S. Régnier. Sur quelques aspects mathématiques des problèmes de classification automatique. *ICC Bulletin*, 4:175–191, 1965.

G. Vescia. Descriptive classification of cetacea: Whales, porpoises, and dolphins. In J. F. Marcotorchino, J. M. Proth, and J. Janssen, editors, *Data analysis in real life environment: ins and outs of solving problems*. Elsevier Science Publishers B.V., 1985.

Index