Knowledge Space Theory

Christina Stahl Cord Hockemeyer

2018-01-03

Abstract

This document explains algorithms and basic operations of knowledge structures and knowledge spaces available in R through the **kst** package.

Knowledge Space Theory (Doignon and Falmagne, 1999) is a set- and order-theoretical framework, which proposes mathematical formalisms to operationalize knowledge structures in a particular domain. The most basic assumption of knowledge space theory is that every knowledge *domain* can be represented in terms of a set of domain problems or items. Moreover, knowledge space theory assumes dependencies between these items in that knowledge of a given item or a subset of items may be a prerequisite for knowledge of another, more difficult or complex item. These prerequisite *relations* are realized by surmise relations, which create a quasi-order between different items. One advantage of these surmise relations is that they reduce the quantity of all possible solution patterns to a more manageable amount of knowledge states. Each of these knowledge states represents the subset of items an individual is capable of solving. The collection of all knowledge states captures the organization of the domain and is referred to as *knowledge structure*.

1 Knowledge Structures

The `kstructure()` function in package **kst** is the basic constructor for knowledge structures. It takes an endorelation representing a surmise relation or a set of sets each representing one knowledge state (e.g., one clause of a surmise system) and returns the corresponding knowledge structure:

```
> # An endorelation representing a surmise relation
> kst <- endorelation(graph=set(tuple(1,1), tuple(2,2), tuple(3,3),
+   tuple(4,4), tuple(2,1), tuple(3,1), tuple(4,1), tuple(3,2), tuple(4,2)))
> kstructure(kst)

{{}, {"3"}, {"4"}, {"2", "3", "4"}, {"1", "2", "3", "4"}}

> # A set of sets representing knowledge states (e.g., clauses of a surmise system)
> kst <- kstructure(set(set("a"), set("a","b"), set("a","c"), set("d","e"),
+    set("a","b","d","e"), set("a","c","d","e"), set("a","b","c","d","e")))
> kst

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

Note that by default the quotes indicate the fact that the items are represented by characters. For displaying purposes, these quotes may be turned off:

```
> sets_options("quote",FALSE)
> kst
```

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

On the resulting knowledge structure several operations can be performed. Firstly, the knowledge *domain* of the knowledge structure can be determined by means of the `kdomain()` function:

```
> kdomain(kst)
```

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

Secondly, the *notions* of the knowledge structure can be determined by means of the `knotions()` function. A notion is a set of items always jointly contained in all knowledge states. Consequently, these items carry the same information and may therefore be considered equivalent:

```
> knotions(kst)
```

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

Thirdly, the *atoms* of the knowledge structure can be determined by means of the `katoms()` function. For any item of the knowledge domain, an atom is a minimal knowledge state containing the respective item, where minimal refers to the fact that the respective knowledge state is not the union of any other knowledge states:

```
> katoms(kst, items=set("a","b","c"))
```

```
$a
{{a}}

$b
{{a, b}}

$c
{{a, c}}
```

Forthly, the *trace* of the knowledge structure can be determined by means of the `ktrace()` function. The trace of a knowledge structure on a set of items is the substructure of the knowledge structure on these items, i.e., the substructure resulting from restricting the knowledge structure to the specified item(s):

```
> ktrace(kst, items=set("c","d","e"))
```

```
{{}, {c}, {d, e}, {c, d, e}}
```

Fifthly, the `kneighbourhood()` function computers the neighbourhood of a knowledge state.

```
> kneighbourhood(kst, state=set("a", "b"))
```

```
{{a}}
```

Finally, the `kfringe()` function allows for determining the fringe of a knowledge state. Fringes determine the symmetric difference between a given knowledge state and its neighbouring states.

```
> kfringe(kst, state=set("a", "b"))
```

```
{b}
```

In addition, several properties of knowledge structures may be tested. Currently, only the functions `kstructure_is_wellgraded()` and the `kstructure_is_space()` are implemented (see Section 2 for the latter).

A knowledge structure is considered *well-graded* if any two of its states are connected by a bounded path, i.e., each knowledge state (except the empty set {}) has at least one predecessor state that contains the same domain items with the exception of exactly one and each knowledge state (except the state for the full set of domain problems Q) has at least one immediate successor state that comprises the same domain items plus exactly one.

```
> kstructure_is_wellgraded(kst)
```

```
[1] FALSE
```

Apart from these basic operations, the **kst** package also provides plotting functionalities for knowledge structures (see README for details). The `plot()` method takes an arbitrary knowledge structure and plots a Hasse Diagram of the respective knowledge structure (see Figure 1):

```
> if(requireNamespace("Rgraphviz")) {Rgraphviz::plot(kst)}
```

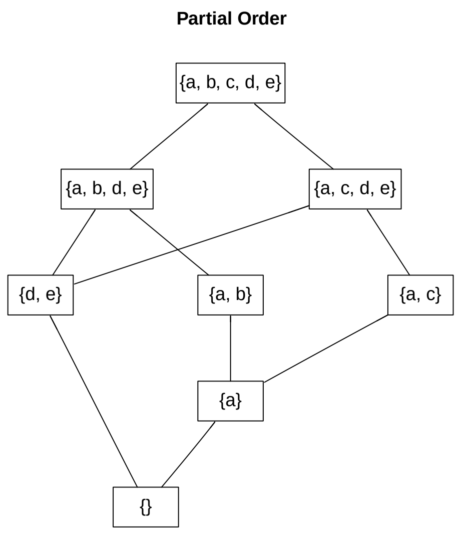

Figure 1: Knowledge Structure

In order to allow for plotting the surmise relation underlying a knowledge structure, the **kst** package provides the `as.relation()` method, which computes its underlying surmise relation, i.e., the set of item pairs corresponding to the knowledge dependencies. Antisymmetric and transitive surmise relations may then be plotted as a Hasse diagram:

```
> as.relation(kst)
```

```
A binary relation of size 5 x 5.
```

In those cases where individuals' response patterns are available, they may be used to assess individuals or validate a knowledge structure.

The `kassess()` function assigns individuals to their corresponding knowledge state in a knowledge structure. Currently only "deterministic" assessment is implemented. Assessing individuals based on a deterministic procedure starts by determining an item a, which is contained in approximately half of the available knowledge states. If the individual being assessed has successfully solved the respective item a, all knowledge states that do not contain item a are removed from the set of potential knowledge states of the individual. If, on the other hand, the individual has not solved the respective item a, all knowledge states that do contain item a are removed from the set of potential knowledge states of the individual. From the remaining knowledge states an item b, which again is contained in approximately half of the still available knowledge states, is selected. If the individual has successfully solved the respective item b, all knowledge states that do not contain item b are removed from the set of potential knowledge states of the individual. If, on the other hand, the individual has solved the respective item b, all knowledge states that do contain item b are removed from the set of potential knowledge states of the individual. This procedure is repeated until only one knowledge state is left. This is the knowledge state the individual is currently located in.

```
> rp <- data.frame(a=c(1,1,0,1,1,1,1,0,0,0),b=c(0,1,0,1,0,1,0,1,0,0),
+    c=c(0,0,0,0,1,1,1,0,1,0),d=c(0,0,1,1,1,1,0,0,0,1), e=c(0,0,1,1,1,1,0,0,0,0))
> kassess(kst, rpatterns=rp)

$Respondent1
{a}

$Respondent2
{a, b}

$Respondent3
{d, e}

$Respondent4
{a, b, d, e}

$Respondent5
{a, c, d, e}

$Respondent6
{a, b, c, d, e}

$Respondent7
{a, c}

$Respondent8
{}

$Respondent9
{}

$Respondent10
{d, e}
```

The `kvalidate()` function on the other hand calculates validity coefficients for prerequisite relations and knowledge structures. The γ-*Index* (Goodman and Kruskal, 1972) validates the prerequisite relation underlying a knowledge structure and assumes that not every response pattern is represented by a prerequisite relation. For this purpose it compares the number of response patterns that are represented by a prerequisite relation (i.e., concordant pairs) with the number of response patterns that are not represented by a prerequisite relation (i.e., discordant pairs). Formally, the γ-Index is defined as

$$\gamma = \frac{N_c - N_d}{N_c + N_d}$$

where N_c is the number of concordant pairs and N_d the number of discordant pairs. Generally, a positive γ-value supports the validity of prerequisite relations.

The validation method *percent* likewise validates prerequisite relations and assumes that more difficult or complex items are solved less frequently than less difficult or complex items. For this purpose it calculates the relative solution frequency for each of the items in the domain.

The *Violational Coefficient* (Schrepp, Held, and Albert, 1999) also validates prerequisite relations. For this purpose, the number of violations (i.e., the earlier mentioned discordant pairs) against a prerequisite relation are calculated. Formally, the VC is defined as

$$VC = \frac{1}{n(|S| - m)} \sum_{x,y} v_{xy}$$

where n denotes the number of response vectors, $|S|$ refers to the number of pairs in the relation, m denotes the number of items, and v_{xy} again refers to the number of discordant pairs. Generally, a low VC supports the validity of prerequisite relations.

In contrast to the other three indices, the *Distance Agreement Coefficient* (Schrepp, 1999) validates the resulting knowledge structure. For this purpose it compares the average symmetric distance between the knowledge structure and respone patterns (referred to as *ddat*) to the average symmetric distance between the knowledge structure and the power set of response patterns (referred to as *dpot*). By calculating the ratio of *ddat* and *dpot*, the DA is determined. Generally, a lower DA-value indicates a better fit between a knowledge structure and a set of response patterns.

```
> kvalidate(kst, rpatterns=rp, method="gamma")

[1] 0.4

> kvalidate(kst, rpatterns=rp, method="percent")

   %
a 60
b 40
c 40
d 50
e 40

> kvalidate(kst, rpatterns=rp, method="VC")

[1] 0.075

> kvalidate(kst, rpatterns=rp, method="DA")

$ddat
[1] 0.3

$ddat_dist
Distances
0 1
7 3

$dpot
[1] 0.96875

$dpot_dist
Distances
 0  1  2
 8 17  7

$DA
[1] 0.3096774
```

Apart from these kst-specific functions, the **kst** package also provides general set-related methods. In particular, these include methods pertaining to the *closure* and *reduction* of sets.

The `closure()` method for objects of class `kstructure()` performs the closure of a knowledge structure by computing the union or intersection of any two knowledge states. `union` is also used as a basis for the `kspace()` function (see next section).

```
> closure(kst, operation="union")
```

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

The `reduction()` method performs the reduction of a knowledge structure by computing the minimal subset having the same closure as the knowledge structure. Additionally, it allows for computing the *discriminative* reduction of a knowledge structure. Such a discriminative reduction is a knowledge structure in which each notion contains a single item.

```
> reduction(kst, operation="discrimination")
```

```
{{}, {a}, {de}, {a, b}, {a, c}, {a, b, de}, {a, c, de}, {a, b, c, de}}
```

2 Knowledge Spaces

Apart from knowledge structures, knowledge space theory also suggests the concept of *knowledge spaces*. A knowledge structure is considered a knowledge space if it includes one state for the empty set {}, one state for the full set of domain items, and a state for the union of any two knowledge states (i.e., the closure under union). The basic constructor for creating knowledge spaces is the `kspace()` function. It takes an arbitrary knowledge structure and returns the corresponding knowledge space:

```
> ksp <- kspace(kst)
> ksp
```

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

In order to test for the space property of a knowledge structure, the **kst** package provides the function `kstructure_is_space()`:

```
> kstructure_is_kspace(ksp)
```

```
[1] TRUE
```

Apart from the functions described in the previous section, which can likewise be performed on knowledge spaces, the package **kst** provides the additional function `kbase()`, which is only applicable to knowledge spaces. The `kbase()` function takes an arbitrary knowledge space and computes its *base*. A base for a knowledge space is a minimal family of knowledge states spanning the knowledge space, i.e., the base includes the minimal states sufficient to reconstruct the full knowledge space. A knowledge structure has a base only if it is a knowledge space.

```
> kbase(ksp)
```

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

3 Learning Paths

Both, knowledge structures and knowledge spaces, involve the concept of *learning paths.* A learning path is a maximal sequence of knowledge states, which allows learners to gradually traverse a knowledge structure or space from the empty set {} (or any other bottom state) to the full set of domain problems Q. The basic constructor for creating learning paths is the `lpath()` function. It takes an arbitrary knowledge structure or space and computes all possible learning paths in the respective knowledge structure or space. The result is a list where each element represents one learning path:

```
> lp <- lpath(ksp)
> lp

[[1]]
({}, {a}, {a, b}, {a, b, c}, {a, b, c, d, e})

[[2]]
({}, {a}, {a, c}, {a, b, c}, {a, b, c, d, e})

[[3]]
({}, {a}, {a, b}, {a, b, d, e}, {a, b, c, d, e})

[[4]]
({}, {a}, {a, d, e}, {a, b, d, e}, {a, b, c, d, e})

[[5]]
({}, {d, e}, {a, d, e}, {a, b, d, e}, {a, b, c, d, e})

[[6]]
({}, {a}, {a, c}, {a, c, d, e}, {a, b, c, d, e})

[[7]]
({}, {a}, {a, d, e}, {a, c, d, e}, {a, b, c, d, e})

[[8]]
({}, {d, e}, {a, d, e}, {a, c, d, e}, {a, b, c, d, e})
```

A learning path is considered a *gradation* if each state in the respective learning path differs from its predecessor and/or successor state by a single item/notion. The `lpath_is_gradation()` function allows for testing the gradation property of a list of learning paths:

```
> lpath_is_gradation(lp)

[[1]]
[1] FALSE

[[2]]
[1] FALSE

[[3]]
[1] FALSE

[[4]]
[1] FALSE
```

```
[[5]]
[1] FALSE

[[6]]
[1] FALSE

[[7]]
[1] FALSE

[[8]]
[1] FALSE
```

References

J.-P. Doignon and J.-C. Falmagne. *Knowledge Spaces*. Springer Verlag, Heidelberg, 1999.

L. A. Goodman and W. H. Kruskal. Measures of association for cross classification. *Journal of the American Statistical Association*, 67:415–421, 1972.

M. Schrepp. An empirical test of a process model for letter series completion problems. In D. Albert and J. Lukas, editors, *Knowledge Spaces: Theories, Empirical Research, Applications*. Lawrence Erlbaum Associates, 1999.

M. Schrepp, T. Held, and D. Albert. Component-based construction of surmise relations for chess problems. In D. Albert and J. Lukas, editors, *Knowledge Spaces: Theories, Empirical Research, Applications*. Lawrence Erlbaum Associates, 1999.

Index