Karl-Franzens-Universität Graz

SRBT Tools User Manual

Susanne Pötzi and Gudrun Wesiak

Berichte

aus dem

Institut für Psychologie

A-8010 Graz, Universitätsplatz 2/III

Institutsbericht Nr. 2004/1

SRBT Tools User Manual[*]

Susanne Pötzi and Gudrun Wesiak[†]

Institut für Psychologie
Karl–Franzens–Unversität Graz, Austria

April 16, 2004

[*]The described tools were developed within the Project *Surmise Relations between Tests*. The project was financed by the Austrian National Science Fund (FWF) under the project number P12726–SOZ granted to Dietrich Albert and Wilhelm Schappacher. The current version of this document is online available at `http://wundt.uni-graz.at/projects/srbt`.

[†]Correspondence concerning this manual should be mailed to `gudrun.wesiak@uni-graz.at`, `cord.hockemeyer@uni-graz.at`, or `dietrich.albert@uni-graz.at`

Contents

Preface

This user manual describes the tools that were developed within the Project *Surmise relations between tests*. The programs were written accompanying the mathematical developments and experimental data analysis. Thus, some tools are rather demand specific, while others are more general. Furthermore, the manual also includes some tools for working with surmise relations between items. We refer the reader to the KST Tools user manual by Cord Hockemeyer (2001) for further tools working on an item level.

As already mentioned above, the manual is the result of an interdisciplinary project on surmise relations between tests, which was granted by the Austrian Science Fund (FWF) to Dietrich Albert and Wilhelm Schappacher. The mathematical part of the project, which had a great influence on the development of the programs, was mainly worked out by Silke Brandt. Furthermore, a few of the programs were written by Alexander Wenzl and Ali Ünlü gave some valuable comments to the actual realization of the tools.

In order to use the tools, you can either apply for a user account at `http://wundt.uni-graz.at/ePsyt/` (limited WWW–version) or write an Email to `KST@wundt.uni-graz.at` (for the full version running on a solaris platform).

The manual is the first attempt to give an overview of the developed software and to explain the functions of each tool to new users. We therefore invite all our readers to give us their feedback on the tools themenselves, their descriptions, or eventual bugs. We are thankful for all comments.

1 Introduction[1]

1.1 Overview

The theory of knowledge space, originally developed by Doignon and Falmagne (1985; 1999; Falmagne et al., 1990), is a non–numerical approach to the representation and efficient diagnosis of knowledge in a given domain. In all knowledge domains or psychometric tests, the set of items varies with respect to difficulty. By considering these implicit dependencies (the so called surmise relation) among a set of problems, the correct or incorrect solutions to a subset of items can be inferred from previously obtained responses. As an example, one might imagine a person who is capable of multiplying fractions. Assuming that the same person will also be capable of multiplying real numbers, it would be inefficient to present problems containing this type of task. Hence, by taking advantage of the implicit structure relating a set of problems, it is possible to reduce the number of items presented in a test.

If the dependencies among items are specified by varying problem demands, it is furthermore possible to obtain precise information on the testee's knowledge state, i. e. of the problem requirements he or she is able to meet. Thus, the knowledge space theory provides a framework for more efficient testing procedures, which diagnose a person's knowledge state by specifying the problem demands the person is able to meet and/or the underlying skills the person possesses.

The following sections give a short introduction into the concepts of a surmise relation between items and a surmise relation between tests. The subsequent chapters contain the manpages for tools used for working with surmise relations. The tools are devided into six main categories. The manual starts with a description of the file formats required by the programs (Chapter 2), which is followed by tools for generating knowledge spaces (Chapters 3) and tools for data evaluation and validation (Chapter 4). All of the tools described in Chapters 3 and 4 refer to surmise relations between items. Chapter 5 contains two programs for the simulation of answer patterns and in Chapter 6 tools for working with surmise relations between tests are introduced. Finally, Chapter 7 describes some general tools for the transformation of different file types into each other.

[1]This chapter is a reduced version of the introductory section on knowledge spaces in the doctoral thesis of the second author (Wesiak, 2003)

Note, that this user manual is based on an online manual, which is also linked to the *KST tools user manual* by Hockemeyer (2001). References to the KST Tools manual are indicated by (1K) or (5K).

1.2 Surmise Relations between Items

Doignon and Falmagne (1985, 1999) defined the *knowledge state* of a person as the set of items in a specified domain Q this individual is able to master under ideal conditions. Accordingly, items of a specified domain are ordered on the basis of surmise or prerequisite relations. A surmise relation is a binary relation on a set of items with the following interpretation: whenever a subject solves an item $x \in Q$ and we can surmise from this performance that this subject is able to solve item $y \in Q$, as well, we say that the pair (y, x) is in a *surmise relation* (we note $y \preceq x$, ySx, $(y, x) \in S$, or $S \subseteq Q \times Q$). Surmise relations are reflexive and transitive but not necessarily connex, i. e. they are quasi or partial orders. The relation can be depicted as a Hasse diagram, where a descending line signifies that a correct solution of the lower item is surmisable from a correct solution of the upper item (see Figure 1.1a).

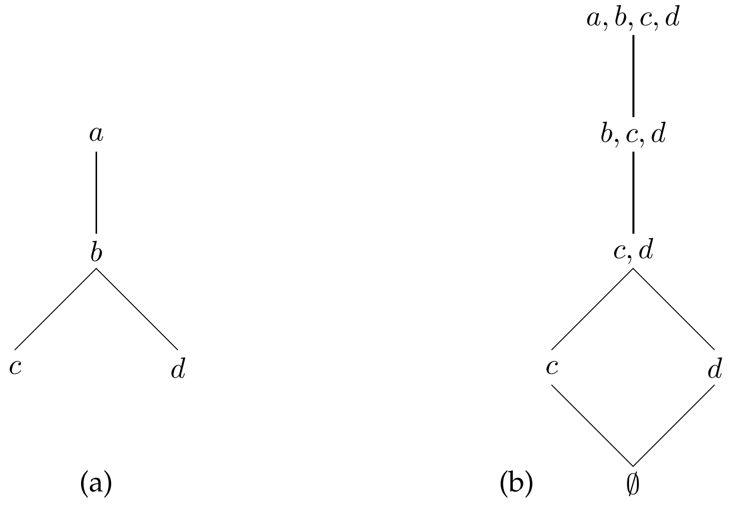

Figure 1.1: Surmise relation (a) and its corresponding knowledge space (b) on a set $Q = \{a, b, c, d\}$ of four items

This type of relationship is also called a *prerequisite relation*, meaning that mastering item y is prerequisite for mastering item x.

Each item combination in accordance with a given surmise relation is called a *quasi ordinal knowledge state*. Formally, a knowledge state K is defined by

$$K \subseteq Q \Leftrightarrow (\forall x, y \in Q, ySx \wedge x \in K \Rightarrow y \in K). \qquad (1.1)$$

Now, the set of all knowledge states is called *knowledge structure* ($\mathcal{K}$). A knowledge structure $\mathcal{K}$ which is closed under union and intersection is called a quasi ordinal *knowledge space* (see Figure 1.1b). This means that for any two knowledge states K and L, their union $\cup$ and their intersection $\cap$ are also knowledge states. A quasi ordinal knowledge space consists of the family of all knowledge states including the empty set and the complete set of items. According to the Birkhoff (1937), quasi orders on a set of items establish a one–to–one correspondence between a surmise relation and its corresponding quasi ordinal knowledge space. Thus, the quasi ordinal relation can be directly inferred from the quasi ordinal space, and vice versa. In general, the advantage of organizing knowledge according to surmise or prerequisite relations is that the number of possible item combinations, i. e. the powerset 2^Q of all items, can be reduced to a subset $\mathcal{K} \subseteq 2^Q$ of knowledge states.

1.3 Surmise Relations between Tests

So far, we have referred to single tests. However, in common psychological assessment procedures we often deal with a set $\mathcal{T}$ of different tests that are related. The common conception of the relations between tests is based on correlations. On the background of Doignon and Falmagne's framework, Albert (1995, Ünlü et al.,2004; Brandt et al.; Brandt, 1999; 2000; 2003) extended the concept of the surmise relation between items (i. e. *within* tests) to a surmise relation *between* tests ($\dot{\mathcal{S}} \subseteq \mathcal{T} \times \mathcal{T}$ or $\dot{\mathcal{S}}_{\mathcal{T}\times\mathcal{T}}$). The advantage of the concept of surmise relations between tests is that we can specify prerequisite relations not only between single items but between subsets of items, i. e. entire tests, as for example, between tests of cognitive or developmental functioning where the possession of one ability may be prerequisite for some other ability.

The interpretation of a surmise or prerequisite relation between tests, i. e. $(B, A) \in \dot{\mathcal{S}}$ or $B\ \dot{\mathcal{S}}\ A$, is that two tests $A, B \in \mathcal{T}$ are in surmise relation from A to B, if one can surmise from the correct solution of at least one item in test A the correct solution of a non-empty subset of items in test B (see Figure 1.2). In other words, solving the item(s) in test B (e. g. item b_3 in Figure 1.2) is a prerequisite for the solution of a given set of items in test A (e. g. item a_1 in Figure 1.2). Formally, the relation $\dot{\mathcal{S}} \subseteq \mathcal{T} \times \mathcal{T}$ is defined by

$$B\ \dot{\mathcal{S}}\ A \Leftrightarrow \exists a \in A : B_a \neq \emptyset \quad \forall A, B \in \mathcal{T} \quad \text{with } B_a = B \cap \bigcap \mathcal{K}_a. \tag{1.2}$$

$\mathcal{K}_a$ is the set of all knowledge states containing item a.

For a set of tests $\mathcal{T} = \{A, B, C, \ldots\}$, a surmise relation between tests has the property of reflexivity but not necessarily transitivity, i. e. in general, it is not a quasi order. However, there are special cases for which transitivity holds, namely left– and right–covering surmise relations. In these cases, the transitive surmise relation between tests

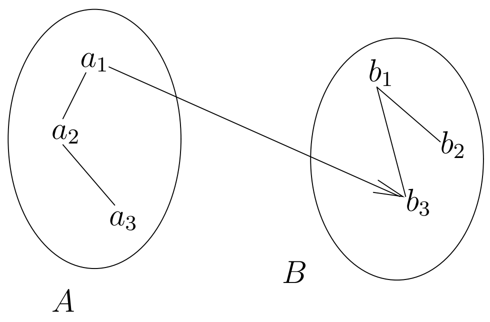

Figure 1.2: Two tests A and B are in surmise relation from A to B ($B \; \dot{\mathcal{S}} \; A$)

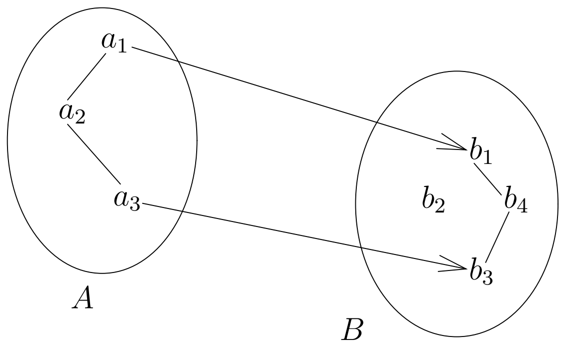

Figure 1.3: Left–covering surmise relation from test A to test B ($B \; \dot{\mathcal{S}}_l \; A$)

can be inferred from the corresponding test knowledge structure (see below). However, the reverse inference is not valid for a set of tests (Ünlü et al., 2004).

The interpretation of a *left–covering surmise relation* ($B \; \dot{\mathcal{S}}_l \; A$, see Figure 1.3) is that for each item $a \in A$ there exists a nonempty subset of prerequisites in test B, i. e. a person who doesn't solve any item in B will not be able to solve any item in A, either. There is no need to test this person on test A. Formally, we say that two tests $A, B \in \mathcal{T}$ are in left–covering surmise relation from test A to test B. The relation is defined by

$$B \; \dot{\mathcal{S}}_l \; A \Leftrightarrow \forall a \in A : B_a \neq \emptyset \qquad \forall A, B \in \mathcal{T} \tag{1.3}$$

Right–covering means, that for each item $b \in B$, there exists at least one item $a \in A$ for which b is a prerequisite ($B \; \dot{\mathcal{S}}_r \; A$), i. e. failing to solve any item in test B implies a failure on a subset of items in test A (see Figure 1.4). In other words, a person who solves all items in test A is also able to solve all items in test B. Hence, there is no further need to test the person on test B. Formally, we say that two tests $A, B \in \mathcal{T}$ are in right–covering surmise relation from test A to test B. The relation is defined by

$$B \; \dot{\mathcal{S}}_r \; A \Leftrightarrow \bigcup_{a \in A} B_a = B \qquad \forall A, B \in \mathcal{T} \tag{1.4}$$

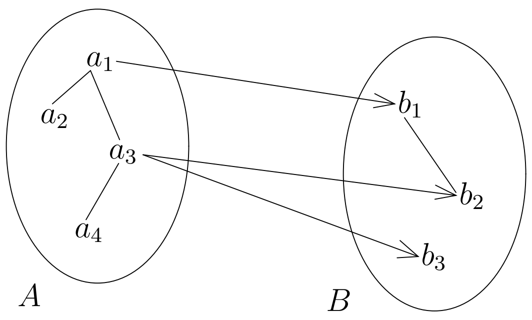

Figure 1.4: Right–covering surmise relation from test A to test B ($B\ \dot{\mathcal{S}}_r\ A$)

Finally, we speak of a *total–covering* surmise relation, if all items in test B are prerequisite for some item $a \in A$ and all items in A have a prerequisite $b \in B$, i. e. the surmise relation is left– as well as right–covering.

Aside of the surmise relation between tests, it is necessary to differentiate between various subsets of the surmise relation on the entire set Q of items. S_{QxQ} denotes the surmise relation on the whole set of items and is referred to as the surmise relation *between items (SRbI)*. The disjoint subsets of the surmise relation S_{QxQ} on two tests A and B are denoted S_{AxA}, S_{BxB}, S_{AxB}, and S_{BxA}. The sets S_{AxA} and S_{BxB} are called surmise relations *within tests (SRwT)*, the sets S_{AxB} and S_{BxA} surmise relations *across tests (SRxT)*. Note that the number of pairs within and across tests equals the number of pairs between items, when reflexive pairs are not taken into account. Each subset is defined as follows:

$$\begin{aligned}
S_{QxQ} &= \{(y,x)\,|\,x,y \in Q \wedge ySx\}\\
S_{AxA} &= \{(a_i,a_j)\,|\,a_i,a_j \in A \wedge a_iSa_j\}\\
S_{BxB} &= \{(b_i,b_j)\,|\,b_i,b_j \in B \wedge b_iSb_j\}\\
S_{AxB} &= \{(a,b)\,|\,a \in A, b \in B \wedge aSb\}\\
S_{BxA} &= \{(b,a)\,|\,a \in A, b \in B \wedge bSa\} \qquad (1.5)
\end{aligned}$$

If a surmise relation fulfills either the condition S_{AxB} or S_{BxA}, we speak of a surmise relation between tests ($\dot{\mathcal{S}}_{\mathcal{T}x\mathcal{T}}$ or $SRbT$) as it is defined in terms of Equation 1.2. If S_{AxB} or S_{BxA} fulfill the conditions specified in Equation 1.3 or 1.4, we speak of a left– respectively right–covering surmise relation. Surmise relations between tests which fulfill both conditions are called total–covering.

Extending the concepts of Doignon and Falmagne's approach, a *test knowledge state* $\dot{K}_i$ is defined as the combination of item subsets per test $(A_i, B_i, \ldots)$ person i is capable of mastering. The collection of all test knowledge states $\dot{\mathcal{K}}$ is called the *test knowledge structure*, which is defined as the pair $(\mathcal{T}, \dot{\mathcal{K}})$, with $\mathcal{T}$ denoting the set of tests $\{A, B, C, \ldots\}$. If a test knowledge structure is closed under union $\cup$ it is called *test knowledge space*, if it is also closed under intersection $\cap$ we speak of a *quasi ordinal test kowledge space*. As for the surmise relation we differentiate between the knowledge space between items, within, across, and between tests.

References

Albert, D. (1995). *Surmise relations between tests*. Talk at the 28th Annual Meeting of the Society for Mathematical Psychology, University of California, Irvine, August.

Birkhoff, G. (1937). Rings of sets. *Duke Mathematical Journal*, *3*, 443–454.

Brandt, S. (2000). *Surmise relations between tests*. Unpublished documentation.

Brandt, S., Albert, D., & Hockemeyer, C. (1999). Surmise relations between tests - preliminary results of the mathematical modelling. *Electronic Notes in Discrete Mathematics*, *2*.

Brandt, S., Albert, D., & Hockemeyer, C. (2003). Surmise relations between tests - mathematical considerations. *Discrete Applied Mathematics*, *127*(2), 221–239.

Doignon, J.-P. & Falmagne, J.-C. (1985). Spaces for the assessment of knowledge. *International Journal of Man-Machine Studies*, *23*, 175–196.

Doignon, J.-P. & Falmagne, J.-C. (1999). *Knowledge spaces*. Berlin: Springer–Verlag.

Falmagne, J.-C., Koppen, M., Villano, M., Doignon, J.-P., & Johannesen, L. (1990). Introduction to knowledge spaces: How to build, test and search them. *Psychological Review*, *97*, 201–224.

Hockemeyer, C. (2001). *Tools and utilities for knowledge spaces* (2nd ed.). Unpublished technical report, Institut für Psychologie, Karl–Franzens–Universität Graz, Austria.

Ünlü, A., Brandt, S., & Albert, D. (2004). *Test surmise relations, test knowledge structures, and their characterizations*. Submitted for publication.

Wesiak, G. (2003). *Ordering inductive reasoning tests for adaptive knowledge assessments: An application of surmise relations between tests*. Unpublished doctoral dissertation, Karl–Franzens–Universität Graz, Graz, Austria.

2 File Formats

2.1 Filetypes — SRBT File formats

Synopsis

#SRBT `[version] [structtype] [encoding] [endian] [wordsize] [comment]`

Description

This manpage describes the general file formats, as they are required by the SRBT tools. Users, who are used to the old file formats (e.g. spacefile (5K) or basisfile (5K)), note that the main extensions are the introduction of header-lines and the definition of additional content types.

Usage

#SRBT `[version] [structtype] [encoding] [endian] [wordsize] [comment]`

`#SRBT` This string at the very beginning of the first line of a file denotes that the file is for use with the `libsrbt` or `libsrbi` libraries.

version Version number of the file format. Currently: `v2.0`

structtype This string describes the type of data contained in the file. Ten types of files are currently supported (see below for more detailed descriptions of the various file tyes):
`basis`
`space`
`structure`
`relation` (surmise relation between items)

`data`
`disjpartition` (disjoint partition)
`partition`
`patterns`
`testrelation` (surmise relation between tests)
`i_hypothesis` (partial hypothesis on a surmise relation between items)

Bases, relations, patterns, and partitions are always stored as ASCII text files, whereas data, spaces, and structures can be stored in either ASCII or binary format.

encoding The `encoding` information specifies whether the data are stored in `ASCII` or in `binary` form.

endian The `endian` information specifies (for binary files only) whether the storing computer has a `LITTLE` or a `BIG` endian processor. Default is `BIG` endian.

wordsize States are stored as a multiple of `wordsize` bits (for binary files only). The default value is 32. Specification of `wordsize` requires a specification of `endian`.

comment The possibility to specify a comment is primarily provided for use with the ASCII files. The first line may contain a comment separated by a hash sign (#). Subsequent lines beginning with a hash sign (#) may also contain comments.

For **binary** files, the format header line and optional comments are closed by a `NUL` (i.e. `0x00`) character.

Additional Information for Items and Students/Tests

It is possible to store non-ambiguous identification numbers for each item and each student or test in the file. These numbers will be called 'item information numbers' and 'line information numbers' in the following. The storage of these numbers is useful especially in cases, in which the order of items is changed, items are removed, or items are combined. The information lines require the following structure (for a detailed description of the structure and interpretation of the below mentioned matrices see the following sections on the respective filetypes):

Item information:

`#*` (column in the structure matrix): non-ambiguous item number(s), separated by a blank.

Line information:

`#+` (line in the structure matrix): non-ambiguous line number(s), separated by a blank.

Examples:

```
#* 4:  4
#* 5:  6
#* 6:  7
```

means, that the 4th column of the matrix contains the relationships for item number 4, the 5th column for item number 6, and the 6th column for item number 7. Item number 5 has been removed.

```
 #* 1:  2 3
```

means, that the first column of the matrix contains the relationships for items number 2 and 3 (e.g. if the items 2 and 3 are equivalent).

For line numbers, the same structure applies.

ATTENTION: Counting of lines and columns starts with 0!!!

If a file is created automatically, an additional comment line is added, which specifies the number of each item. For matrices with large numbers of items, this comment line should simplify the counting of columns.

Remarks

The new *SRBT* file format was developed in order to ensure that users do not mingle different file type specifications. This used to become a major problem with users having only little computer experience. However, the tools should also be able to read files in the old format.

If *SRBT* files are created manually, one should provide as many header information as possible.

If a file of the wrong type (in the new format) is passed to a program, it may either be rejected (with an appropriate error message) or converted (optionally issuing an additional warning that a file of wrong type was passed over).

See also

basisfile (Sect. 2.2, p. 16), patternfile (Sect. 2.7, p. 20), srbifile (Sect. 2.9, p. 22), spacefile (Sect. 2.8, p. 21)

2.2 basisfile — Format of basisfiles (v2.0)

Description

A `basisfile` is an ASCII file describing the basis of a knowledge space. It has the following format:

The first line is the header-line containg information about version and filetype (see Filetypes (Sect. 2.1, p. 13)):

`#SRBT` v2.0 basis

The second line contains the number of items in the knowledge domain.

The third line contains the number of basis elements.

After these three lines you can add comments and identification numbers for items and states in the basis (for a detailed description see Filetypes (Sect. 2.1, p. 13)).

The following lines contain the basis elements building a matrix where the columns describe the items and the rows describe the basis elements. In each cell of this matrix there is a '0' if the basis element does not contain the item, a '1' if it is a clause for the item, and a '2' otherwise.

For any set of knowledge states, a basis according to this specification can be computed with the basis (1K) program.

Version information

This manpage describes version v2.0 of the `basisfile` format. In version v1.0 no headerlines and no additional comments and information to items and state numbers are possible. Files of the old (v1.0) and new (v2.0) filetypes can be converted into each other with the programs new2old (Sect. 7.2, p. 61) and old2new (Sect. 7.3, p. 62).

See also

basis (1K), spacefile (Sect. 2.8, p. 21), new2old (Sect. 7.2, p. 61), old2new (Sect. 7.3, p. 62)

2.3 datafile — Format of datafiles (v2.0)

Description

`Datafiles` can be stored in either ASCII or binary file format.

ASCII file format

The first line is the header-line containing information about version and filetype (see Filetypes (Sect. 2.1, p. 13)):

#`SRBT` v2.0 data ASCII

The second line contains the number of items in the knowledge domain.

The third line contains the number of answer patterns.

After these three lines you can add comments and identification numbers for items and students in the data structure (for a detailed description see Filetypes (Sect. 2.1, p. 13)).

The following lines contain the answer patterns: In each cell of this matrix there is a '1' if the student answered the respective item (in column *i*) correctly and a '0', otherwise.

Binary file format

The first line is the header-line containing information about version and filetype (see Filetypes (Sect. 2.1, p. 13)). The file contains a sequence of `long integer` numbers. The first two numbers give the number of items and the number of answer patterns. The following `long integers` build bitsets, one per knowledge state. A bitset consists of as many `long integers` as are needed to represent the item set. This number of `long integers` needed can be computed as `(ItemNo + BitsPerLong - 1) / BitsPerLong` where `BitsPerLong` is the machine

specific number of bits used to store a `long integer` number.

In binary files you cannot add any comment lines. If you convert ASCII to binary files, all additional information about items and lines is lost.

Warning

Using a binary `datafile` on different hardware platforms may produce unexpected results, because there may be different byte orders and therefore different bit orders.

Version information

This manpage describes version v2.0 of the `datafile` format. The format changes from v1.0 include additional format and meta information header lines (see spacefile (5K)).

See also

structurefile (Sect. 2.10, p. 23), spacefile (Sect. 2.8, p. 21), new2old (Sect. 7.2, p. 61), old2new (Sect. 7.3, p. 62)

2.4 disjpartitionfile — Format of disjoint partitionfiles (v2.0)

Description

This manpage describes the format and header of a disjoint partition of items into tests.

The first line is the header-line containing information about version and filetype (Filetypes (Sect. 2.1, p. 13):

`#SRBT` v2.0 disjpartition

The second line contains the number of items in the domain.

The third line contains the number of tests in the partition.

After these three lines you can add comments and identification numbers for items and tests in the partition (for a detailed description see Filetypes (Sect. 2.1, p. 13)).

The following lines contain the partition of items into different tests. Each line of the matrix corresponds to one test, in each cell of this matrix there is a '1' if the item belongs to the test in the respective line, and a '0', otherwise. In disjoint partition files each item has to belong to exactly one test. It is not possible, that items belong to more than one test or that items do not belong to any test at all.

See also

partitionfile (Sect. 2.6, p. 19)

2.5 i_hypothesis — File formats for partial input of relations between items

This manpage describes the format and header of files for surmise relations between items, if not all relationships in the investigated domain are known.

The first line is the header-line containing information about version and filetype (see Filetypes (Sect. 2.1, p. 13)):

`#SRBT` v2.0 i_hypothesis

The second line contains the number of items.

After these two lines you can add comments and identification numbers for the items in the file.

The following matrix describes a surmise relation between items, for which not all relationships between the items are known. The files have the same structure as srbifiles (see `srbifile`, Sect. 2.9), where a '1' in line *i* and column *j* indicates, the the pair (*i*,*j*) is an element of the surmise relation *S* (*iSj*) and a '0' indicates that the pair is not an element of *S*. In addition to `srbifiles`, the `i_hypothesis` file also supports the entry 'n', which indicates, that you do not know, whether or not the pair (*i*,*j*) is element of *S*.

Example: You know, that the pair (*i*,*j*) is not element of *S*, but you do not know, whether or not the pair (*j*,*i*) is element of *S*: you enter a '0' in the *j*-th column and *i*-th line of the matrix and an 'n' on the *j*-th line and *i*-th column of the matrix.

Warning

Currently this filetype is not supported by any program of this software package.

See also

Filetypes (Sect. 2.1, p. 13), srbifile (Sect. 2.9, p. 22)

2.6 partitionfile — Format of partitionfiles (v2.0)

Description

The first line is the header-line containing information about version and filetype (see Filetypes (Sect. 2.1, p. 13)):

#SRBT v2.0 partition

The second line contains the number of items in the domain.

The third line contains the number of tests in the partition.

After these three lines you can add comments and identification numbers for items and tests in the partition (for a detailed description see Filetypes (Sect. 2.1, p. 13)).

The following lines contain the partition of items into different tests. Each line of the matrix corresponds to a test, in each cell of the matrix there is a '1' if the item belongs to the test in the respective line and a '0', if the item does not belong to the test.

In `partitionfiles` it is possible, that items belong to more than one test or that items do not belong to any test at all.

See also

disjpartitionfile (Sect. 2.4, p. 18)

2.7 patternfile — Format of patternfiles (v2.0)

Description

A `patternfile` stores the answers of students to a set of items, distinguishing between correct, incorrect, and not answered (e.g. skipped) items.

The first line is the header-line containg information about version and filetype (see Filetypes (Sect. 2.1, p. 13)):

#SRBT v2.0 patterns

The second line contains the number of items in the knowledge domain.

The third line contains the number of answer patterns.

After these three lines you can add comments and identification numbers for items and students in the pattern structure (for a detailed description see Filetypes (Sect. 2.1, p. 13)).

The following lines contain the answer patterns of students: In each cell of this matrix there is a '1' if the student answered the respective item (in column *i*) correctly, a '0', if the answer is incorrect, and an 'x', if the student did not answer the item at all.

See also

datafile (Sect. 2.3, p. 16), pat2data (Sect. 7.4, p. 63)

2.8 spacefile — Format of spacefiles (v2.0)

Description

A `spacefile` can be stored in either ASCII or binary file format. Both types describe knowledge spaces in a very similar manner:

ASCII file format

The first line is the header-line containing information about version and filetype (see Filetypes (Sect. 2.1, p. 13)):

`#SRBT` v2.0 space ASCII

The second line contains the number of items in the knowledge domain.

The third line contains the number of states in the knowledge space.

After these three lines you can add comments and identification numbers for items and states in the space (for a detailed description see Filetypes (Sect. 2.1, p. 13)).

The following lines contain the knowledge states building a matrix where the columns describe the items and the rows describe the knowledge states. In each cell of this matrix there is a '1' if the knowledge state does contain the respective item, and a '0' otherwise.

Binary file format

The first line is the header-line containing information about version and filetype (see Filetypes (Sect. 2.1, p. 13)). The file contains a sequence of `long integer` numbers. The first two numbers give the number of items and the number of knowledge states. The following `long integers` build bitsets, one per knowledge state. A bitset consists of as many `long integers` as are needed to represent the item set. This number of `long integers` needed can be computed as `(ItemNo + BitsPerLong - 1) / BitsPerLong` where `BitsPerLong` is the machine specific number of bits used to store a `long integer` number.

In binary files you cannot add any comment lines. If you convert ASCII to binary files, all additional information about items and lines is lost.

Warning

Using a binary `spacefile` on different hardware platforms may produce unexpected results, because there may be different byte orders and therefore different bit orders.

Version information

This manpage describes version v2.0 of the `spacefile` format. The format changes from v1.0 include additional format and meta information header lines (see spacefile (5K)).

See also

basisfile (Sect. 2.2, p. 16), new2old (Sect. 7.2, p. 61), old2new (Sect. 7.3, p. 62)

2.9 srbifile — File formats for relationfiles between items

This manpage describes the format and header of files for surmise relations between items (in this documentation they are either called `relationfile` or `srbifile`).

The first line is the header-line containing information about version and filetype (see Filetypes (Sect. 2.1, p. 13)):

`#SRBT` v2.0 relation

The second line contains the number of items in the knowledge domain.

After these two lines you can add comments and identification numbers for items in the `srbifile` (for a detailed description see Filetypes (Sect. 2.1, p. 13)).

The following lines describe the surmise relation between items in form of a matrix: The matrix includes a '1' in column *i* and line *j*, iff item *i* is in surmise relation with item *j*, so that *j* is prerequisite for *i* (*jSi*) and a '0' otherwise. Because a surmise relation is reflexive, the main diagonal contains always '1's. If you want to complete a surmise relation matrix in order to achieve transitivity, use the complete-srbi (Sect. 3.3, p. 29).

Warning

Please note that the introductionary string `#SRBT` is used for `srbifiles`, too.

See also

Filetypes (Sect. 2.1, p. 13), complete-srbi (Sect. 3.3, p. 29)

2.10 structurefile — Format of structurefiles (v2.0)

Description

This manpage describes the format and header of files for storage of arbitrary structures for working with knowledge spaces (parts of spaces, data, etc.).

A `structurefile` can be stored in either ASCII or binary file format.

ASCII file format

The first line is the header-line containing information about version and filetype (see Filetypes (Sect. 2.1, p. 13)):

`#SRBT` v2.0 structure ASCII

The second line contains the number of items in the knowledge domain.

The third line contains the number of states/data sets... in the structure.

After these two lines you can add identification numbers for items and/or states (for a detailed description of the format of the information lines, see Filetypes (Sect. 2.1, p. 13)).

The following lines contain the knowledge states building a matrix where the columns describe the items and the rows describe the knowledge states. In each cell of this matrix there is a '1' if the knowledge state does contain the item, and a '0' otherwise.

Binary file format

The file contains a sequence of `long integer` numbers. The first two numbers give the number of items and the number of knowledge states. The following `long inte-`

`gers` build bitsets, one per knowledge state. A bitset consists of as many `long integers` as are needed to represent the item set. This number of `long integers` needed can be computed as `(ItemNo + BitsPerLong - 1) / BitsPerLong` where `BitsPerLong` is the machine

specific number of bits used to store a `long integer` number.

In binary files you cannot add any comment lines. If you convert ASCII to binary files, all additional information about items and lines is lost.

Warning

Using a binary `structurefile` on different hardware platforms may produce unexpected results, because there may be different byte orders and therefore different bit orders.

Version information

This manpage describes version v2.0 of the `structurefile` format. The format changes from v1.0 include additional format and meta information header lines (see spacefile (5K)).

See also

Filetypes (Sect. 2.1, p. 13), spacefile (Sect. 2.8, p. 21).

2.11 testrelation — File formats for surmise relations between tests

This manpage describes the format and header of files for surmise relations between tests. The file includes the general, the left-covering, and the right-covering surmise relation between tests.

The first line is the header-line containing information about version and filetype (see Filetypes (Sect. 2.1, p. 13)):

`#SRBT` v2.0 testrelation

The second line contains the number of tests.

After these two lines you can add comments and identification numbers for the tests in the relationfile (for a detailed description see Filetypes (Sect. 2.1, p. 13)).

In this file, three different matrices are stored:
The first matrix describes the general surmise relation between tests: The matrix includes a '1' in line *A* and column *B*, iff the tests are in a surmise relation from test *B* to test *A* (*ASB*) and a '0' otherwise. Because the relation is reflexive, the main diagonal contains always '1's.
The second matrix describes the right-covering, the third matrix the left-covering surmise relation between tests. Both matrices follow the same structure as described above. If the `testrelationfile` is created automatically, there are comment lines included between the three matrices.

See also

Filetypes (Sect. 2.1, p. 13), srbifile (Sect. 2.9, p. 22), srbi-part2srbt (Sect. 6.4, p. 56)

3 SRBI Specific Tools

3.1 combine-equ-items — Combine equivalent items

Synopsis

combine-equ-items `srbifile outputfile`

Description

`combine-equ-items` combines equivalent items in a relation matrix for a surmise relation between items. Two items *i* and *j* are called equivalent, iff *iSj* and *jSi* (*S* denotes the surmise relation between items).

If, for example, the first and the second item in a srbi-matrix are equivalent, the second line and the second column in the matrix are removed. In the 'item information lines' the information is added, that the first line/column contains the relationships for the (original) first and second item. Before combining equivalent items, the surmise relation is tested for reflexitivity and transitivity properties, and completed, if necessary (see complete-srbi (Sect. 3.3, p. 29)).
The outputfile is a srbifile (Sect. 2.9, p. 22), in which all equivalent items have been combined.

Usage

combine-equ-items `srbifile outputfile`

See also

srbifile (Sect. 2.9, p. 22), complete-srbi (Sect. 3.3, p. 29)

3.2 combine-two-items — Combine two items

Synopsis

combine-two-items `srbi-file outputfile item1 item2`

Description

`combine-two-items` combines two items in a surmise relation between items *S* (srbi-matrix). You can specify, which items should be combined, but the given items have to be either parallel or equivalent.

Two items *i* and *j* are called equivalent here, iff *iSj* and *jSi* (see also combine-equ-items (Sect. 3.1, p. 27)).

Two items *i* and *j* are called parallel, iff there exist two other items *x* and *y*, so that *xSi*, *xSj* and *iSy*, *jSy* (i.e. *i* and *j* have the same lower and upper neighbours). If, for example, the first and the second item in a srbi-matrix are equivalent/parallel and should be combined, the second line and the second column in the matrix are removed. In the 'item information lines' the information is added, that the first line/column contains the relationships for the (original) first and second item. Before combining the specified items, the surmise relation is tested for reflexitivity and transitivity properties, and completed, if necessary (see complete-srbi (Sect. 3.3, p. 29)).
The outputfile is a `srbifile`, in which the specified equivalent/parallel items have been combined.

Usage

combine-two-items `srbi-file outputfile item1 item2`
`item1` and `item2` are integer numbers between '0' and 'number of items - 1'. These two items have to be either parallel or equivalent.

See also

srbifile (Sect. 2.9, p. 22), complete-srbi (Sect. 3.3, p. 29), combine-equ-items (Sect. 3.1, p. 27)

3.3 complete-srbi — Complete a srbi-matrix because of reflexivity and transitivity properties

Synopsis

complete-srbi `srbi-file outputfile`

Description

`complete-srbi` completes a surmise relation between items to fulfill the properties of (1) reflexivity and (2) transitivity. This means, (1) that the main diagonal of the matrix contains only '1's, because each item is in a surmise relationship with itself (*iSi*). For (2), any transitive surmise relation *S* containing the pairs (*i*,*j*) and (*j*,*k*) also contains the pair (*i*,*k*). Thus, if there is a '1' in column *j* and line *i* (*iSj*) and a '1' in column *k* and line *j* (*jSk*), the program sets a '1' in column *k* and line *i*, too (because *iSk* is implied by *iSj* and *jSk*). The outputfile is a srbifile (Sect. 2.9, p. 22) that contains the completed matrix.

Usage

complete-srbi `srbi-file outputfile`

See also

srbifile (Sect. 2.9, p. 22)

3.4 gs-closure — Compute the closure under intersection

Synopsis

gs-closure `[Options] structurefile outputfile`

Description

`gs-closure` computes the closure under intersection of a family of sets.

`gs-closure` uses the algorithm developed by B. Ganter. The program is equivalent to the program s-closure (5K), but with large spaces and a large number of items it has the advantage, that it needs less computer memory. Because not all calculated states have to be kept in memory, the algorithm needs only one state to calculate the next in a lexicografic order. The algorithm should especially be used for large numbers of states and items.

Usage

gs-closure `[Options] structurefile outputfile`

The `outputfile` will be a `structurefile`.
Options are:

- `-a|-ascii:` Use ASCII format for structurefile (Sect. 2.10, p. 23).
- `-b|-binary:` Use binary format for structurefile (Sect. 2.10, p. 23).
- `-v|-verbose:` Select informative output.

See also

datafile (Sect. 2.3, p. 16), structurefile (Sect. 2.10, p. 23), s-closure (5K)
Ganter, B. and Reuter, K. (1991). Finding all closed sets: A general approach. *Order*, *8*, 283-290.

3.5 g-constr — Compute the closure under union

Synopsis

g-constr `[Options] structurefile spacefile`

Description

`g-constr` computes the closure under union of a family of sets. The `inputfile` can be a basisfile (Sect. 2.2, p. 16) or an arbitrary structure (see structurefile (Sect. 2.10, p. 23)). The result is a knowledge space stored in a spacefile (Sect. 2.8, p. 21). `g-constr` uses the algorithm developed by B. Ganter. Its main usage is the construction of knowledge spaces from their bases. The program is equivalent to the program constr (5K), but with large spaces and a large number of items it has the advantage, that it needs much less memory.

Usage

g-constr `[Options] structurefile spacefile`
Options are:

- `-a:` Use ASCII format for spacefile (Sect. 2.8, p. 21).
- `-b:` Use binary format for spacefile (Sect. 2.8, p. 21).
- `-v:` Select informative output.

See also

basisfile (Sect. 2.2, p. 16), datafile (Sect. 2.3, p. 16), spacefile (Sect. 2.8, p. 21), structurefile (Sect. 2.10, p. 23), constr (5K)
Ganter, B. and Reuter, K. (1991). Finding all closed sets: A general approach. *Order*, *8*, 283-290.

3.6 leeuwe — Compute the basis for a set of answer patterns

Synopsis

leeuwe `[Options] datafile tolerance-level`

Description

`leeuwe` computes the basis of a quasi ordinal knowledge space from a given `datafile`. It uses the algorithm developed by Held and Korossy (1998), which is based on van Leeuwe (1974). The program investigates for each pair of items *i* and *j*, whether the pair (*j*,*i*) is an element of the surmise relation *S* on the set of items (*jSi*). In the strictest case (the tolerance-level equals 0), a pair (*j*,*i*) is added to the surmise relation, if there is no answer pattern with a correct answer to item *i*, but an incorrect answer to item *j* (*j* is prerequisite for *i*). Setting the tolerance-level to a value greater than 0 allows for the specified percentage of contradicting answer patterns. For example, assume a data set of 100 answer patterns and a tolerance level of 5. In this case, all item pairs with 5 or less contradicting answer patterns (i.e. a correct answer to item *i* and an incorrect answer to item *j*) are added to the surmise relation. The results are written to the screen and the corresponding basis can be stored automatically in a basisfile (Sect. 2.2, p. 16).

The basis can be created with or without transitive closure (with tolerance levels greater 0, intransitive triplets may occur). In the case of intransitive triplets, `leeuwe` looks for the smallest transitive closure covering the empirical relation (i.e. pairs are added to the relation in order to achieve transitivity).

Optionally, additional information such as the prerequisites for each item, the number and kind of intransitive triplets, the Correlational Agreement coefficient (CA, see van Leeuwe, 1974), or 2x2-item-relation-tables can be provided.

Usage

leeuwe `[options] datafile tolerance-level`

Options are:

`-r:` Put out the pairs in the surmise relation in the format "i->j", with *j* being a prerequisite for *i*.

`-rc:` Put out the prerequisites for each item in the format "prerequisites for item i:j,k,l".

`-t:` Put out the frequency of correct/correct (aij), correct/incorrect (bij), incorrect/correct (cij), incorrect/incorrect (dij) cases for each item pair (*j*,*i*), as well as the values Tau (t=(b+1)/(c+1), see Brandt, 2000) and CA (van Leeuwe, 1974).

`-b:` Put out the corresponding bases of the surmise relation and store it in a file named `datafile.ibas` and `datafile.bas` for the empirical relation and the relation closed under transitivity respectively (`datafile` matches the name of the input-file without extension).

`-i:` Put out the intransitive triplets and the total number of triplets (separated by trivial and non-trivial triplets)

`-pj:` Use only the correct answers to *j* as basis for the tolerance check (pairs which are element of the surmise relation without the option -pj are written in brackets).

`-v1:` Force "format 1" for output

`-v0:` Force "format 0" for output

`-d1:` The inputfile has "format 1"

The programm accepts 3 different formats for input and output:
- Format 0: a data matrix without header lines.
- Format 1: corresponds to the old file format (see Filetypes (Sect. 2.1, p. 13)), i.e. the

number of items in the first line, the number of subjects in the second line, followed by the data matrix.
- Format 2: corresponds to the new file format (see datafile (Sect. 2.3, p. 16)), i.e. the `#SRBT` header line, the number of items in the second line, the number of subjects in the third line, followed by the data matrix.

"Format 2" is default for output data. There is an auto-detection for input-data which is able to distinguish between format 0 and 2. For input-data in format 1 use the -d1 flag. Each row of the data matrix contains an answer pattern ("0" for incorrect, "1" for correct answers). For all three file formats, there must not be any spaces between the answers.

Remarks

The program `leeuwe` was first developed by T. Held and extended by A. Wenzl and C. Hockemeyer.

See also

datafile (Sect. 2.3, p. 16), basisfile (Sect. 2.2, p. 16)

Brandt S. (2000). *Surmise Relations between Tests*. Unpublished documentation [Chapter 8].

Held, T. and Korossy, K. (1998). Data analysis as a heuristic for establishing theoretically founded item structures. *Zeitschrift fuer Psychologie*, *206*, 169-188.

van Leeuwe, J. F. J. (1974). Item tree analysis. *Nederlands Tijdschrift voor de Psychologie*, *29*, 475-484.

3.7 parallel-items — Investigate a surmise relation matrix for parallel items

Synopsis

parallel-items `srbifile`

Description

`parallel-items` investigates a surmise relation between items S (srbi-matrix) for parallel items. Two items i and j are called parallel, iff there exist two other items x and y, so that xSi, xSj and iSy, jSy (i.e. i and j have the same lower and upper neighbours).
The results are printed to stdout.

Usage

parallel-items `srbifile`

Bugs

The definition of parallel items does not include special cases. Give a new definition!

See also

srbifile (Sect. 2.9, p. 22), combine-two-items (Sect. 3.2, p. 28)

4 Tools for data evaluation and validation

4.1 count-data — Count the frequencies of answer patterns in a given data matrix

Synopsis

count-data `datafile new-datafile [spacefile]`

Description

`count-data` counts the frequencies of answer patterns in a given data matrix. The data matrix must be stored in a datafile (Sect. 2.3, p. 16) in either binary or ASCII format. The frequencies of each line in the data matrix are printed to stdout. Optionally, if a spacefile (Sect. 2.8, p. 21) is given, a tag shows, if the pattern is element of the space. In the `new-datafile` all repeated answer patterns are deleted and each pattern is stored together with the respective line number(s) in the original `datafile`.

Example:
Assume the following 4 answer patterns:
100
110
100
111
The `outputfile` includes the line information with the number(s) of the original answer patterns for each line (attention: counting starts with ’0’!) and the answer patterns in the following form:
```
#* 0:  0 2
#* 1:  1
#* 2:  3
```
100
110
111
In this example, the answer pattern in the first line (No. 0) occured twice in the original

file (in the 1st and 3rd line), the two remaining patterns occured only once.

Usage

count-data `datafile new-datafile [spacefile]`
`spacefile` is an optional parameter.

Bugs

Currently only ASCII output of the `new-datafile` is possible.

See also

datafile (Sect. 2.3, p. 16), spacefile (Sect. 2.8, p. 21), count-patterns (Sect. 4.3, p. 37)

4.2 count-data-rm — Count the frequencies of answer patterns in a given data matrix and delete trivial patterns

Synopsis

count-data-rm `datafile new-datafile [spacefile]`

Description

`count-data-rm` counts the frequencies of answer patterns in a given data matrix. The program works analogously to the program count-data (Sect. 4.1, p. 35), but all trivial answer patterns (i.e. patterns where either all or none of the items are answered correctly) are stored only once.

Example:
Assume the following 5 answer patterns:
100
110
111
111
100

The `outputfile` includes the line information with the number(s) of the original answer patterns for each line (attention: counting starts with ′0′!) and the answer patterns in the following form: The trivial patterns are stored only once:

```
#* 0:  0 4
#* 1:  1
#* 2:  2
100
110
111
```

In this example, the answer pattern in the first line (No. 0) occured twice in the original file (in the 1st and 5th line) and the pattern in the second line (No. 1) occured only once (2nd line in the original file). The trivial pattern in the third line (No. 2) occured twice (3rd and 4th line in the original file), but is stored only once (line 4 of the original file has been removed).

Usage

count-data-rm `datafile new-datafile [spacefile]`
`spacefile` is an optional parameter.

Bugs

Currently only ASCII output of the `new-datafile` is possible.

See also

datafile (Sect. 2.3, p. 16), spacefile (Sect. 2.8, p. 21), count-data (Sect. 4.1, p. 35)

4.3 count-patterns — Count the frequencies of answer patterns in a given pattern matrix

Synopsis

count-patterns `patternfile new-patternfile [spacefile]`

Description

`count-patterns` counts the frequencies of answer patterns in a given matrix of answer patterns. The matrix has to be stored in form of a patternfile (Sect. 2.7, p. 20). The frequencies of all different answer patterns are printed to stdout. Optionally, if a spacefile (Sect. 2.8, p. 21) is given, a tag shows, if the pattern is a potential element of the space (for patterns containing 'x's we do not know, whether or not the pattern corresponds to a knoweldge state). In the `new-patternfile` all repeated patterns are deleted and each pattern is stored together with the respective line number(s) in the original `patternfile` (cf. count-data (Sect. 4.1, p. 35)).

Usage

count-patterns `patternfile new-patternfile [spacefile]`
`spacefile` is an optional parameter.

See also

patternfile (Sect. 2.7, p. 20), spacefile (Sect. 2.8, p. 21), count-data (Sect. 4.1, p. 35)

4.4 del-equ-data — Delete equal answer patterns

Synopsis

del-equ-data `datafile outputfile`

Description

`del-equ-data` deletes equal answer patterns out of a `datafile`. Item numbers of the original file are kept, but the line information is removed. The outputfile is a datafile (Sect. 2.3, p. 16), where each answer patterns occurs only once. If you want to keep the line information, use the program count-data (Sect. 4.1, p. 35).

Usage

del-equ-data `datafile outputfile`

Bugs

Currently only ASCII output is possible.

See also

datafile (Sect. 2.3, p. 16), count-data (Sect. 4.1, p. 35)

4.5 delete-not-ans — Delete patterns with missing answers to certain items

Synopsis

delete-not-ans `patternfile datafile no_items`

Description

`delete-not-ans` calculates for each item, how many students gave a (correct or incorrect) answer to the item ('0' or '1' in the matrix of answer patterns, but not 'x'). Then the program deletes as many items as specified, starting with the item that has been answered by the smallest number of students. Then the item with the second smallest number of answers is deleted, etc. Afterwards, the patterns, which do not contain answers to all of the remaining items, are deleted. The result is a datafile (Sect. 2.3, p. 16), from which the specified number of items (`no_items`) and the incomplete answer patterns have been removed.

Usage

delete-not-ans `patternfile datafile no_items`

See also

datafile (Sect. 2.3, p. 16), patternfile (Sect. 2.7, p. 20), patt-statistics (Sect. 4.11, p. 44)

4.6 elim — Eliminate answer patterns that contradict the closure under union and intersection

Synopsis

elim `[Options] datafile outputfile`

Description

`elim` eliminates answer patterns in a datafile (Sect. 2.3, p. 16), that contradict the closure under union and intersection, which is an important property of quasi ordinal knowledge spaces. `elim` looks, which answer patterns in the datafile do most frequently contradict the assumption, that the union (intersection) of two arbitrary answer patterns is again an answer pattern. The algorithm eliminates the patterns for which the quotient "number of contradictions / frequency of the pattern" is a maximum. It works recursive, until no contradictions to the closure under union and intersection are left. Before using the program `elim`, the frequencies of the different patterns have to be counted. This happens automatically by means of the program count-data (Sect. 4.1, p. 35).

`elim` prints to stdout, which answer patterns have been removed. The remaining patterns are stored in the `outputfile`, which is again a datafile (Sect. 2.3, p. 16).

Usage

elim `[Options] datafile outputfile`

Options are:

`-a:` Use ASCII format for spacefile (Sect. 2.8, p. 21).

`-b:` Use binary format for spacefile (Sect. 2.8, p. 21).

`-v:` Select informative output.

See also

datafile (Sect. 2.3, p. 16), count-data (Sect. 4.1, p. 35), elim-frequ (Sect. 4.7, p. 41), elim-perc (Sect. 4.8, p. 42), elim-stud (Sect. 4.9, p. 43), elim4pat (Sect. 4.10, p. 43)

Brandt S. (2000). *Surmise Relations between Tests*. Unpublished documentation [Chapter 7].

4.7 elim-frequ — Eliminate answer patterns that contradict the closure under union and intersection

Synopsis

elim-frequ `datafile outputfile frequency-percentage`

Description

`elim-frequ` eliminates answer patterns in a datafile (Sect. 2.3, p. 16), that contradict the closure under union and intersection, which is an important property of quasi ordinal knowledge spaces. The algorithm of this program is the same as in the program elim (Sect. 4.6, p. 40) with the additional feature, that answer patterns exceeding a specified percentage of occurance are not eliminated. This means, that the answer patterns, which occur more often than the given percentage of all answer patterns are not eliminated, even if they include most contradictions.
For example, assume a data set containing 100 answer patterns and a specified `frequency-percentage` of 5. Then all of the patterns, which occur more often than 5 times are not eliminated.
Before using the program `elim-frequ`, the frequencies of the different patterns have to be counted. This happens automatically by means of the program count-data (Sect. 4.1, p. 35).

`elim-frequ` prints to stdout, which answer patterns have been removed. The remaining patterns are stored in the `outputfile`, which is again a datafile (Sect. 2.3, p. 16).

Usage

elim-frequ `datafile outputfile frequency-percentage`

See also

datafile (Sect. 2.3, p. 16), count-data (Sect. 4.1, p. 35), elim (Sect. 4.6, p. 40), elim-perc (Sect. 4.8, p. 42), elim-stud (Sect. 4.9, p. 43), elim4pat (Sect. 4.10, p. 43)

Brandt S. (2000). *Surmise Relations between Tests.* Unpublished documentation [Chapter 7].

4.8 elim-perc — Eliminate answer patterns that contradict the closure under union and intersection

Synopsis

elim-perc `datafile outputfile contradiction-percentage`

Description

`elim-perc` eliminates answer patterns in a datafile (Sect. 2.3, p. 16), that contradict the closure under union and intersection, which is an important property of quasi ordinal knowledge spaces. The algorithm of this program is the same as in the program elim (Sect. 4.6, p. 40) with the additional feature, that it terminates as soon as a specified (maximal) percentage of contradictions is reached. This means that the algorithm does not eliminate answer patterns until no contradictions are left, but terminates the elimination process, when the number of contradictions becomes smaller than the given percentage of acceptable contradictions.
For example, assume a data set containing 100 answer patterns and a specified `contradiction-percentage` of 5. The algorithm terminates the elimination process, if the remaining patterns lead to less than 5 contradictions with regard to a union and intersection closed knowledge space.
Before using the program `elim-perc`, the frequencies of the different patterns have to be counted. This happens automatically by means of the program count-data (Sect. 4.1, p. 35).

`elim-perc` prints to stdout, which answer patterns have been removed. The remaining patterns are stored in the `outputfile`, which is again a datafile (Sect. 2.3, p. 16).

Usage

elim-perc `datafile outputfile contradiction-percentage`

See also

datafile (Sect. 2.3, p. 16), count-data (Sect. 4.1, p. 35), elim (Sect. 4.6, p. 40), elim-frequ (Sect. 4.7, p. 41), elim-stud (Sect. 4.9, p. 43), elim4pat (Sect. 4.10, p. 43)

Brandt S. (2000). *Surmise Relations between Tests*. Unpublished documentation [Chapter 7].

4.9 elim-stud — Eliminate answer patterns that contradict the closure under union and intersection

Synopsis

elim-stud `datafile outputfile no_patterns-left`

Description

`elim-stud` eliminates answer patterns in a datafile (Sect. 2.3, p. 16), that contradict the closure under union and intersection, which is an important property of quasi ordinal knowledge spaces. The algorithm of this program is the same as in the program elim (Sect. 4.6, p. 40) with the additional feature, that it terminates as soon as (approximately) the specified (minimal) number of different patterns is reached. This means that the algorithm does not eliminate answer patterns until no contradictions are left, but terminates the elimination process, when the number of different answer patterns falls below the specified value for `no_patterns-left`. Before using the program `elim-stud`, the frequencies of the different patterns have to be counted. This happens automatically by means of the program count-data (Sect. 4.1, p. 35).

`elim-stud` prints to stdout, which answer patterns have been removed. The remaining patterns are stored in the `outputfile`, which is again a datafile (Sect. 2.3, p. 16).

Usage

elim-stud `datafile outputfile no_patterns-left`

See also

datafile (Sect. 2.3, p. 16), count-data (Sect. 4.1, p. 35), elim (Sect. 4.6, p. 40), elim-frequ (Sect. 4.7, p. 41), elim-perc (Sect. 4.8, p. 42), elim4pat (Sect. 4.10, p. 43)

Brandt S. (2000). *Surmise Relations between Tests.* Unpublished documentation [Chapter 7].

4.10 elim4pat — Eliminate answer patterns that contradict the closure under union and intersection

Synopsis

elim4pat `patternfile outputfile`

Description

`elim4pat` eliminates answer patterns in a patternfile (Sect. 2.7, p. 20), that contradict the closure under union and intersection, which is an important property of quasi ordinal knowledge spaces. `elim4pat` works with the same algorithm as the program elim (Sect. 4.6, p. 40), but it works with answer patterns that include the possibility of correct, incorrect, and not answered questions (see patternfile (Sect. 2.7, p. 20)).

For this program, the union (u) and intersection (s) of patterns are defined as follows:

(111 000 xxx) u (10x 10x 10x) = (111 10x 1xx)
(111 000 xxx) s (10x 10x 10x) = (10x 000 x0x)

Before using the program `elim4pat`, the frequencies of the different patterns have to be counted. This happens automatically by means of the program count-patterns (Sect. 4.3, p. 37).

`elim4pat` prints to stdout, which answer patterns have been removed. The remaining patterns are written to the `outputfile`, which is again a patternfile (Sect. 2.7, p. 20).

Usage

elim4pat `patternfile outputfile`

See also

patternfile (Sect. 2.7, p. 20), count-patterns (Sect. 4.3, p. 37), elim (Sect. 4.6, p. 40)

Brandt S. (2000). *Surmise Relations between Tests*. Unpublished documentation [Chapter 7].

4.11 patt-statistics — Calculate how many students gave an answer to subsets of items in Q

Synopsis

patt-statistics `patternfile`

Description

`patt-statistics` counts, how many students gave an answer (correct or incorrect) to various subsets of items in the set Q of all items under investigation. The program starts with counting the number of complete answer patterns, i.e. patterns with all items answered (denoted by a '0' or '1', but not an 'x' in the patternfile (Sect. 2.7, p. 20)). Then, the item that has been answered by the fewest students is deleted and the program recounts the number of complete answer patterns for the remaining items. Once again the item, which has been answered by the fewest students is deleted and the number of complete answer patterns is recounted. This process continues until there are only complete patterns left for the remaining set of items.

Finally, the program calculates how many items have to be deleted to get (1) the maximal number of students, that have answered all items, (2) the maximal number of items, that have been answered by all students, and (3) the maximum of the product 'items*students' with all items answered.

The results are printed to stdout.
The program can be used to find the appropriate item number for the program delete-not-ans (Sect. 4.5, p. 39).

Usage

patt-statistics `patternfile`

See also

patternfile (Sect. 2.7, p. 20), delete-not-ans (Sect. 4.5, p. 39)

4.12 selpat — Select patterns with a given minimal frequency of occurance

Synopsis

selpat `input-patternfile limit output-patternfile`

Description

`selpat` counts how often each answer pattern occurs and stores all patterns with the specified minimal frequency of occurance (i.e. at least the given `limit`) in an outputfile. Input and output files can either be datafiles or patternfiles (see Filetypes (Sect. 2.1, p. 13)). Each pattern is stored as often as it occured in the original file (`input-patternfile`). If you want to store each pattern just once, use the program count-data (Sect. 4.1, p. 35) or count-patterns (Sect. 4.3, p. 37) (depending on whether the inputfile is a datafile (Sect. 2.3, p. 16) or a patternfile (Sect. 2.7, p. 20)).

Usage

selpat `input-patternfile limit output-patternfile`

Remarks

The program `selpat` was written by A. Wenzl.

See also

datafile (Sect. 2.3, p. 16), patternfile (Sect. 2.7, p. 20), count-data (Sect. 4.1, p. 35), count-patterns (Sect. 4.3, p. 37)

4.13 valid — Validate the fit of a surmise relation to a set of answer patterns

Synopsis

valid `srbifile datafile [-t]`

Description

`valid` validates the fit of a surmise relation to a set of answer patterns. Two indices are used to indicate the quality of the fit: the Correlational Agreement coefficient (CA, van Leeuwe, 1974; see also Held and Korossy, 1998) and the Gamma index (Goodman and Kruskal, 1954). Additionally to the global indices, gamma values are given for each item pair (*i,j*) and for each subject. Optionally, the frequencies of the four possible correct/incorrect combinations (a, b, c, and d) are provided for each item pair. 'a' represents the case incorrect/incorrect, 'b' stands for incorrect/correct, 'c' for correct/incorrect, and 'd' for correct/correct. Furthermore, the value 'tau' (t=b/c, see Brand, 2000), is given.

Usage

valid `srbifile datafile [-t]`

Options:

`-t:` Put out the abcd-tables for each pair of items

Remarks

The program `valid` was written by A. Wenzl.

See also

datafile (Sect. 2.3, p. 16), srbifile (Sect. 2.9, p. 22)

Brandt S. (2000). *Surmise Relations between Tests*. Unpublished documentation [Chapter 8].

Goodman. L.A. and Kruskal. W.H. (1972). Measures of association for cross classifications. *Journal of the American Statistical Association*, *67*, 415-421.

Held, T. and Korossy, K. (1998). Data analysis as a heuristic for establishing theoretically founded item structures. *Zeitschrift fuer Psychologie*, *206*, 169-188.

van Leeuwe, J. F. J. (1974). Item tree analysis. *Nederlands Tijdschrift voor de Psychologie*, *29*, 475-484.

5 Simulating student answers

5.1 learning-sim — Simulating students using the learning path model

Synopsis

learning-sim `basisfile outputfile probabilitiesfile no_students`

Description

`learning-sim` simulates the knowledge states of students using the learning path model (Brandt 2000). From a given basis, the possible learning paths are calculated. The `probabilitiesfile` includes (1) the probabilities for the number of items learned (e.g. the probabilitiy for learning half of the items is higher than the probability for learning all items) and (2) the probabilities for learning item *i* rather than item *j* (for an explanation of the exact meaning of the probabilities see Brandt, 2000, Chapter 7).

The `probabilitiesfile` has the following form:

prob. for learning no item
prob. for learning 1 item
prob. for learning 2 items
....
prob. for learning all items
if you learn an item: prob. for learning item 1
prob. for learning item 2
....
prob. for learning item q

Let q be the number of items in the basis, then the first q+1 probabilities have to sum up to '1' and the second q probabilities have to sum up to '1'. The randomly (by means of the given probabilities) chosen learning paths are written to stdout. The knowledge states for all simulated students are stored in the outputfile, which is a datafile (Sect. 2.3,

p. 16). The resulting set of states is either equal to the space or a subset of the space. Regarding the subsets of the space, those states occur more often, which include items that are more probably learned. Furthermore, the items that are more probably learned, occur more often in the learning paths. Careless errors and lucky guesses are not included in the file. If you want to include careless errors and/or lucky guesses, use the program noisy-learn-sim (Sect. 5.2, p. 50).

Usage

learning-sim `basisfile outputfile probabilitiesfile no_students`

The outputfile will be a datafile in ASCII format.

Bugs

Currently only ASCII output of the new datafile is possible.

See also

datafile (Sect. 2.3, p. 16), basisfile (Sect. 2.2, p. 16), noisy-learn-sim (Sect. 5.2, p. 50)

Brandt S. (2000). *Surmise Relations between Tests*. Unpublished documentation [Chapter 7].

5.2 noisy-learn-sim — Simulating students making careless errors and lucky guesses using the learning path model

Synopsis

noisy-learn-sim `basisfile outputfile probabilitiesfile no_students beta eta`

Description

`noisy-learn-sim` simulates the knowledge states of students using the learning path model (Brandt, 2000). From a given basis, the possible learning paths are calculated. The `probabilitiesfile` includes (1) the probabilities for the number of items learned (e.g. the probabilitiy for learning half of the items is higher than the probability for learning all items) and (2) the probabilities for learning item *i* rather than item *j* (for an explanation of the exact meaning of the probabilities see Brandt, 2000, Chapter 7).

The `probabilitiesfile` has the following form:

prob. for learning no item
prob. for learning 1 item
prob. for learning 2 items
....
prob. for learning all items
if you learn an item: prob. for learning item 1
prob. for learning item 2
....
prob. for learning item q

Let q be the number of items in the basis, then the first q+1 probabilities have to sum up to '1' and the second q probabilities have to sum up to '1'. The randomly (by means of the given probabilities) chosen learning paths are written to stdout. After choosing the states, in which the students are according to the given probabilities in the probability file, careless errors and lucky guesses are simulated. The value beta is the probability for a lucky guess, the value eta for a careless error (both numbers have to be values between 0 and 1). The simulated answer patterns are stored in the `outputfile`, which is a datafile (Sect. 2.3, p. 16).

Usage

noisy-learn-sim `basisfile outputfile probabilitiesfile no_students beta eta`
The outputfile will be a datafile in ASCII format.

Bugs

Currently only ASCII output of the new datafile is possible.

See also

datafile (Sect. 2.3, p. 16), basisfile (Sect. 2.2, p. 16), learning-sim (Sect. 5.1, p. 49)

Brandt S. (2000). *Surmise Relations between Tests*. Unpublished documentation [Chapter 7].

6 Tools for partitions and tests

6.1 connex-part — Produce a connex disjoint partition of items into tests

Synopsis

connex-part `srbifile outputfile`

Description

`connex-part` creates a connex disjoint partition of items into tests (given a surmise relation on the set of items). The program tries to generate as few tests as possible (only one item per test would be a trivial solution that is always possible).
A test is called 'connex' here, if for each item i in the test there exists another item j in the same test, so that iSj or jSi (S denotes the surmise relation between items). A partition is called connex, if all tests in the partition are connex.
The `outputfile` is a disjoint partition file (see disjpartitionfile (Sect. 2.4, p. 18)).

Usage

connex-part `srbifile outputfile`

See also

disjpartitionfile (Sect. 2.4, p. 18), srbifile (Sect. 2.9, p. 22)

6.2 part-properties — Investigate the properties of a partition of items into tests

Synopsis

part-properties `srbifile partitionfile`

Description

`part-properties` investigates the properties of a partition of items into tests using the surmise relation between items.

The following properties are investigated:

- Is the whole partition connex?
- Is the whole partition transitive?
- Is the whole partition antisymmetric?

For a definition of the properties, see Brandt (2000, Chapter 5). The results are printed to stdout.

Usage

part-properties `srbifile partitionfile`

Bugs

The properties right-, left-, and total-coveringness of the whole partition have to be defined theoretically and included into the program.

See also

srbifile (Sect. 2.9, p. 22), partitionfile (Sect. 2.6, p. 19), tests-properties (Sect. 6.7, p. 58)

Brandt S. (2000). *Surmise Relations between Tests*. Unpublished documentation [Chapter 5].

6.3 random-part — Produce a random disjoint partition of items into tests

Synopsis

random-part `partition-kind no_items no_tests outputfile`

Description

`random-part` produces a random disjoint partition of items into tests. There are five possibilities for choosing the properties of the resulting disjoint partition:

- Random selection of the number of tests
- Specification of a maximal number of tests
- Specification of an exact number of tests
- Equal partitioning of the items into tests (only possible, if the number of items is a multiple of the number of tests)
- Specification of a minimal number of items per test

The `outputfile` is a disjoint partition file (see disjpartitionfile (Sect. 2.4, p. 18)).

Usage

random-part `partition-kind no_items no_tests outputfile`

Options for the kind of partition (`partition-kind`) are: Create a random disjoint partition with

- `-r:` a randomly chosen number of tests.
- `-m:` a given maximal number of tests.
- `-t:` a given number of tests.
- `-e:` an equal number of items per test.
- `-i<n>:` a minimal number of 'n' items per test.

Bugs

Option ′-r′ requires a value for the number of tests, which is, however, not used in the program.

See also

disjpartitionfile (Sect. 2.4, p. 18)

6.4 srbi-part2srbt — Calculate the surmise relations between tests out of a surmise relation between items and a partition into tests

Synopsis

srbi-part2srbt `srbifile partitionfile testrelation`

Description

`srbi-part2srbt` calculates the general, right-, and left-covering surmise relations between tests out of a given surmise relation between items and a partition of items into tests. The results are printed to stdout, and the resulting surmise relations are stored in a testrelation (Sect. 2.11, p. 24) file. The relations are written in form of three matrices, one for the general surmise relation, one for the right-, and one for the left-covering surmise relation: A ′1′ in line A and column B means, that there is a surmise relation from Test B to Test A ($A\ S\ B$ with S denoting the surmise relation between tests). The same goes for right- and leftcovering surmise relations.

Usage

srbi-part2srbt `srbifile partitionfile testrelation`

See also

testrelation (Sect. 2.11, p. 24), srbifile (Sect. 2.9, p. 22), partitionfile (Sect. 2.6, p. 19), tests-properties (Sect. 6.7, p. 58)

Brandt S. (2000). *Surmise Relations between Tests*. Unpublished documentation [Chapter 2].

6.5 srwt — Look for surmise relations within tests

Synopsis

srwt `srbifile partitionfile outputfile`

Description

`srwt` uses a surmise relation between items (srbifile (Sect. 2.9, p. 22)) and a partition of these items into tests (partitionfile (Sect. 2.6, p. 19)) to extract the relationships between items that belong to the same test. The program looks for all relationships within tests (SRwT: both items belong to the same test). The results are printed to stdout. The `outputfile` contains a srbi-matrix, in which all relationships across tests (SRxT: the items belong to different tests) are set to ’0’.

Usage

srwt `srbifile partitionfile outputfile`

See also

srbifile (Sect. 2.9, p. 22), partitionfile (Sect. 2.6, p. 19), srxt (Sect. 6.6, p. 57)

6.6 srxt — Look for surmise relations across tests

Synopsis

srxt `srbifile partitionfile outputfile`

Description

`srxt` uses a surmise relation between items (srbifile (Sect. 2.9, p. 22)) and a partition of these items into tests (partitionfile (Sect. 2.6, p. 19)) to extract the relationships between items that belong to different test. The program looks for all surmise relations across tests (SRxT: the items belong to different tests). The results are printed to stdout. The `outputfile` contains a srbi-matrix, in which all relationships within tests (SRwT: the items belong to the same test) are set to ’0’.

Usage

srxt `srbifile partitionfile outputfile`

See also

srbifile (Sect. 2.9, p. 22), partitionfile (Sect. 2.6, p. 19), srwt (Sect. 6.5, p. 57)

6.7 tests-properties — Investigate the properties of the tests of a given partition

Synopsis

tests-properties `srbifile partitionfile`

Description

`tests-properties` investigates the properties of tests in a given partition using a given surmise relation between items and a partition of the items into tests. For all possible combinations of tests, the following properties are investigated:

- Is there a general, left-, right-, and/or total-covering surmise relation between tests? For each pair of tests *A* and *B*, ’TA S TB’ means, that there is a surmise relation from Test *B* to Test *A* and ’TA !S TB’ means, that there is no surmise relation from Test *B* to Test *A*. The same goes for left-, right-, and total-covering surmise relations, which are denoted by ’Sl, Sr, St’ respectively.
- Are the two tests antisymmetric?
- Is a given test connex?

For a definition of the properties, see Brandt (2000, Chapters 2 and 5). The results are printed to stdout.

Usage

tests-properties `srbifile partitionfile`

See also

srbifile (Sect. 2.9, p. 22), partitionfile (Sect. 2.6, p. 19), part-properties (Sect. 6.2, p. 53)

Brandt S. (2000). *Surmise Relations between Tests*. Unpublished documentation [Chapters 2 and 5].

7 General Tools

7.1 bas2srbi — Convert a basis into a surmise relation

Synopsis

bas2srbi `basisfile srbifile`

Description

`bas2srbi` converts a basisfile (Sect. 2.2, p. 16) into a surmise relation file (srbifile (Sect. 2.9, p. 22)).

Usage

bas2srbi `basisfile srbifile`

Remarks

The program `bas2srbi` was written by A. Wenzl.

See also

basisfile (Sect. 2.2, p. 16), srbifile (Sect. 2.9, p. 22)

7.2 new2old — Change a file in new file format (v2.0) into a file in old format

Synopsis

new2old filename-new-format filename-old-format

Description

`new2old` changes a file in new file format (containing a header line and information numbers, see Filetypes (Sect. 2.1, p. 13)) to a file in old format (see e.g. spacefile (5K)). Information on the type of file as well as information numbers of items and lines are lost with this transformation. `new2old` is currently used to work with KST-tools, which only accept the old file format.

Usage

new2old filename-new-format filename-old-format

See also

Filetypes (Sect. 2.1, p. 13), old2new (Sect. 7.3, p. 62)

7.3 old2new — Change a file in old file format into a file in new format (v2.0)

Synopsis

old2new filename-old-format filename-new-format filetype

Description

`old2new` changes a file in old file format into a file in new format (containing a header line with information on the filetype) (see Filetypes (Sect. 2.1, p. 13)).

Usage

old2new filename-old-format filename-new-format filetype
filetype is one of the types of files frequently used with knowledge space theory (space, basis, data,..., see Filetypes (Sect. 2.1, p. 13)).

See also

Filetypes (Sect. 2.1, p. 13), new2old (Sect. 7.2, p. 61)

7.4 pat2data — Change a patternfile into a datafile

Synopsis

pat2data `patternfile datafile`

Description

`pat2data` changes a patternfile (Sect. 2.7, p. 20) into a datafile (Sect. 2.3, p. 16). All missing answers (denoted by an 'x' in the `patternfile`) are viewed as incorrect answers (denoted by '0' ind the new `datafile`).

Usage

pat2data `patternfile datafile`

See also

patternfile (Sect. 2.7, p. 20), datafile (Sect. 2.3, p. 16)