Contents

1 Additional Functions Dealing with Bitsets ..................... 2
2 Additional Functions for Surmise Relations between Items 9
3 Additional Functions for Spaces .............................. 13
   3.5 ganter — Class for computing the closure under union and
       intersection for structures ............................... 15
4 Working with Information Structures .......................... 19
5 Working with Item Hypotheses ................................. 22
6 Working with Answer Patterns ................................. 25
7 Additional Functions for Working with Patterns .............. 29
8 Equivalence Properties of Items ............................... 37
9 Data Structures for Working with Tests ....................... 40
   9.1 partition — Partition of a set of items into tests ........ 40
   9.2 disjpartition — Partition of items into disjoint tests ..... 44
   9.3 srbt — Surmise relations between tests .................. 47
10 Investigating Equal Structures ............................... 51
11 Working with (Disjoint) Partitions ........................... 53
12 Surmise Relations Within and Across Tests .................. 58
13 Creating Different Kinds of Partitions ....................... 60
14 Surmise Relations between Tests .............................. 65
15 Properties of Partitions ...................................... 69
16 Functions for Equivalence Properties of Tests .............. 72
17 Functions for working with vectors of integer numbers .... 76
18 Functions for working with Isotonic Probabilistic Models . 86
   Class Graph ............................................... 92

This page has been automatically generated with DOC++
DOC++ is ©1995 by Roland Wunderling & Malte Zöckler

January 8, 2002
Additional Functions Dealing with Bitsets (a,b)

Names

1.1 double minv (double* vector, int len)  
minimal entry of a vector .................................. 3

1.2 int minv (int* vector, int len)  
minimal entry of a vector .................................. 3

1.3 double maxv (double* vector, int len)  
maximal entry of a vector .................................. 4

1.4 int maxv (int* vector, int len)  
maximal entry of a vector .................................. 4

1.5 int min_pos (float* vector, int len)  
position of the minimal element in a vector .................... 4

1.6 int min_pos (int* vector, int len)  
position of the minimal element in a vector .................... 5

1.7 int max_pos (float* vector, int len)  
position of the maximal element in a vector .................... 5

1.8 int max_pos (int* vector, int len)  
position of the maximal element in a vector .................... 6

1.9 int is_number_el (double item, double* set, int set_size)  
is a number element of a set of numbers .................... 6

1.10 int is_number_el (int item, int* set, int set_size)  
is a number element of a set of numbers .................... 6

1.11 int is_bitset_el (bitset element, bitset* set, int q_size, int s_size)  
is a bitset element of a set of bitsets? ..................... 7

1.12 int is_bitset_el (bitset element, structure* st)
1.13 int* set (int* array, int len, int value=0)

set all elements of an array the same value

---

1.1 double minv (double* vector, int len)

minimal entry of a vector

Compute the minimal entry of a vector of real numbers.

Return Value: minimal entry of a vector of real numbers

Parameters:
- vector — vector of (real) numbers
- len — length of vector

---

1.2 int minv (int* vector, int len)

minimal entry of a vector

Compute the minimal entry of a vector of integer numbers.

Return Value: minimal entry of a vector of integer numbers

Parameters:
- vector — vector of (real) numbers
- len — length of vector
1.3

\begin{verbatim}
double maxv (double* vector, int len)
\end{verbatim}

maximal entry of a vector

maximal entry of a vector. Compute the maximal entry of a vector of real numbers.

Return Value: maximal entry of a vector of numbers
Parameters: vector — vector of (real) numbers
len — length of vector

1.4

\begin{verbatim}
int maxv (int* vector, int len)
\end{verbatim}

maximal entry of a vector

maximal entry of a vector. Compute the maximal entry of a vector of integer numbers.

Return Value: maximal entry of a vector of numbers
Parameters: vector — vector of (real) numbers
len — length of vector

1.5

\begin{verbatim}
int min_pos (float* vector, int len)
\end{verbatim}

position of the minimal element in a vector

position of the minimal element in a vector. Compute the position of the minimal entry in a vector of real numbers. Attention: The first entry of the vector is on position '0'!

Return Value: position of the minimum in the vector
1 Additional Functions Dealing with Bitsets

Parameters:
vector — vector of real numbers
len — length of vector

1.6

```c
int min_pos (int* vector, int len)
```

position of the minimal element in a vector. Compute the position of the minimal entry in a vector of integer numbers. Attention: The first entry of the vector is on position '0'!

Return Value: position of the minimum in the vector

Parameters:
vector — vector of integer numbers
len — length of vector

1.7

```c
int max_pos (float* vector, int len)
```

position of the maximal element in a vector. Compute the position of the maximal entry in a vector of real numbers. Attention: The first entry of the vector is on position '0'!

Return Value: position of the maximum in the vector

Parameters:
vector — vector of real numbers
len — length of vector
1.8

int max_pos (int* vector, int len)

position of the maximal element in a vector

position of the maximal element in a vector. Compute the position of the maximal entry in a vector of integer numbers. Attention: The first entry of the vector is on position '0'!

Return Value: position of the maximum in the vector
Parameters:
vector — vector of integer numbers
len — length of vector

1.9

int is_number_el (double item, double* set, int set_size)

is a number element of a set of numbers

is a number element of a set of numbers. Look, if a single real number is element of a set of real numbers.

Return Value: yes (1) or no (0)
Parameters:
item — number to look for
set — pointer to set of real numbers
set_size — size of set of numbers

1.10

int is_number_el (int item, int* set, int set_size)

is a number element of a set of numbers

is a number element of a set of numbers. Look, if a single integer number is element of a set of integer numbers.

Return Value: yes (1) or no (0)
Parameters: 

- `item` — number to look for
- `set` — pointer to set of integer numbers
- `set_size` — size of set of numbers

1.11

```
int is_bitset_el (bitset element, bitset* set, int q_size, int s_size)
```

is a bitset element of a set of bitsets?

is a bitset element of a set of bitsets?. Look, if a bitset (meaning also a vector of integer numbers of length `q_size`) is element of a set of bitsets (or a set of integer numbers).

**Return Value:**

- yes (1) or no (0)

**Parameters:**

- `element` — bitset to look for
- `set` — pointer to set of bitsets
- `q_size` — number of items in each bitset
- `s_size` — number of bitsets in the set

1.12

```
int is_bitset_el (bitset element, structure* st)
```

is a bitset element of an arbitrary structure?

is a bitset element of an arbitrary structure?. Look, if a bitset is element of the given structure (or space, data...).

**Return Value:**

- yes (1) or no (0)

**Parameters:**

- `element` — bitset to look for
- `set` — pointer to set of bitsets
- `q_size` — number of items in each bitset
- `s_size` — number of bitsets in the set
### 1.13

**int* set (int* array, int len, int value=0)**

*set all elements of an array the same value*

set all elements of an array the same value. Set all elements of the given array to the value of the given parameter. If no `value` parameter is given, all entries of the array are set to 0.

**Return Value:** pointer to the array

**Parameters:**
- `array` — array of integers
- `len` — length of array
- `value` — value to be set, default is 0
Additional Functions for Surmise Relations between Items

Names

2.1 srbi* copy_srbi (srbi* sr) 
* make a copy of a surmise relation matrix .......... 9

2.2 srbi* change_items (srbi* sr, int nr1, int nr2 )
* change items .................. 10

2.3 srbi* remove_item_sr (srbi* sr, int item )
* remove an item ................. 10

2.4 int count_equ_items (srbi* sr)
* count equivalent items ........... 11

2.5 srbi* delete_equ_items (srbi* sr)
* delete equivalent items .......... 11

2.6 srbi* close_reflex_srbi (srbi* sr)
* complete surmise relations because of reflexivities ............... 11

2.7 srbi* close_trans_srbi (srbi* sr)
* complete transitivities in surmise relation between items .......... 12

2.8 int is_item_sr (int i, int j, srbi* sr)
* is there a surmise relation between two items? ...................... 12

ATTENTION: In all following functions the numbering of items starts with '0'!
This means, the first item has the number '0', the last 'q.size-1'.

2.1

srbi* copy_srbi (srbi* sr)

* make a copy of a surmise relation matrix
2 Additional Functions for Surmise Relations between Items

Return Value: pointer to copied srbi-structure
Parameters: sr — surmise relation structure to be copied

2.2

srbi* change_items (srbi* sr, int nr1, int nr2)

change items. Change the position of two items in a srbi-matrix. The information numbers of the items is also changed.
Return Value: Surmise relation matrix, where items number i and j changed their position in the matrix.
Parameters: sr — matrix with surmise relation
nr1 — number of first item
nr2 — number of second item

2.3

srbi* remove_item_sr (srbi* sr, int item)

remove an item. Remove one item in a surmise-relation matrix. For example: you want to eliminate item nr. i - the function eliminates the i-th line and the i-th column in the matrix. The informations for this item also is deleted.
Return Value: new srbi with deleted item.
Parameters: sr — surmise relation between items
item — number of the item to be deleted
2 Additional Functions for Surmise Relations between Items

2.4

```c
int count_equ_items (srbi* sr)
```

count equivalent items. Count, how many items are equivalent in the given surmise relation. Comment: two items \( a \) and \( b \) are equivalent, if \( aSb \) and \( bSa \) (\( S \) denotes the surmise relation between items).

Return Value: number of equivalent items
Parameters: \( sr \) — surmise relation between items

2.5

```c
srbi* delete_equ_items (srbi* sr)
```

delete equivalent items. Delete one of two equivalent items. Comment: two items \( a \) and \( b \) are equivalent, if \( aSb \) and \( bSa \).

Return Value: new srbi with deleted equivalent items.
Parameters: \( sr \) — matrix with surmise relations

2.6

```c
srbi* close_reflex_srbi (srbi* sr)
```

complete surmise relations because of reflexivities. For each item 'a' we always have: a is in surmise relation with a. Therefore in the matrix with surmise relations there have to be all '1's in the main diagonal. If they are missing, they are completed by this function.

Return Value: new srbi with added reflexivities.
Parameters: \( sr \) — matrix with surmise relations
2.7

srbi* close_trans_srbi (srbi* sr)

*complete transitivity in surmise relation between items*

complete transitivity in surmise relation between items. Complete a surmise relation matrix between items by regarding transitivity properties. Example: 'S' denotes the surmise relation between Items, a,b,c are Items; if aSb and bSc, the function will set aSc in the srbi-structure.

Return Value: complete srbi-structure with all surmise-relations
Parameters: sr — relation to be completed

2.8

int is_item_sr (int i, int j, srbi* sr)

*is there a surmise relation between two items?*

is there a surmise relation between two items?. Look if item number i is in surmise relation with item number j.

Return Value: yes or no (1/0)
Parameters: i — number of first item
ej — number of second item
sr — surmise relations between the items
3

Additional Functions for Spaces

Names
3.1 int write_space (space* s)  write a space to stdout ............ 13
3.2 structure* states_with_x (space* s, int item_x)  states with item x ............... 14
3.3 data* transpose_matrix (data* input)  transpose a matrix ............... 14
3.4 int* count_data (data* d)  count the frequencies of answer patterns in a data matrix ...... 14
3.5 class ganter  Class for computing the closure under union and intersection for structures ................. 15

3.1

int write_space (space* s)

write a space to stdout

write a space to stdout. Comment: this function can be used for other structs with the same internal structure (data, structure...) just by making a typecasting.

Return Value: 1 if an error occurred, 0 else.

3.2

structure* states_with_x (space* s, int item_x)

states with item x
states with item x. Compute all states of a knowledge space containing an item x.

**Return Value:** structure containing all the states that contain 'item x', NULL if an error occurred.

**Parameters:**
- s — knowledge space
- item_x — number of item

### 3.3

```c
data* transpose_matrix (data* input)
```

transpose a matrix. Transpose a given matrix in a data-structure by changing \( a[i,j] \) to \( a[j,i] \). Comment: use this functions also for spaces, structures and partitions just making a typecasting.

**Return Value:** pointer to a data-structure, containing the transposed matrix.

**Parameters:**
- input — data-structure including the matrix to be transposed

### 3.4

```c
int* count_data (data* d)
```

count the frequencies of answer patterns in a data matrix. Count the frequencies of different answer patterns in a given data matrix, where the line-infos are complete (they already include, which answer patterns occur how often and they include all different patterns only once.

**Return Value:** vector with the frequencies of answer patterns

**Parameters:**
- d — data-structure with line-infos
Class for computing the closure under union and intersection for structures

Public Members

3.5.1  space*  u_closure (basis* b)  
   close a basis under union ........  16

3.5.2  int  u_closure (basis* b, filetype mode,  
         const char filename[])  
   close a basis under union ........  16

3.5.3  space*  u_closure (structure* su)  
   close a structure under union ...  16

3.5.4  int  u_closure (structure* su, filetype mode,  
         const char filename[])  
   close a structure under union ...  17

3.5.5  structure*  s_closure (structure* su)  
   close a structure under intersection  ......................  17

3.5.6  int  s_closure (structure* su, filetype mode,  
         const char filename[])  
   close a structure under intersection  ......................  18

Class for computing the closure under union and intersection for structures.
The functions in this class use the Ganter algorithm to compute the closure  
under union and intersection. A great advantage with this algorithm is, that  
not the whole structure/space has to be kept in memory. The next state in  
the resulting structure/space is computed out of the last. Especially with large  
structures, it is recommendable to use the versions of closure, that store the  
resulting space/structure directly to a file.

3.5.1  space*  u_closure (basis* b)  
   close a basis under union
3.5.2

```c
int u_closure (basis* b, filetype mode, const char filename[])
```

close a basis under union. Compute the closure of a basis under union using Ganter’s algorithm. 

Return Value: resulting space, NULL, if an error occurred.

Parameters: 

- `b` — basis to be closed

3.5.3

```c
space* u_closure (structure* su)
```

close a structure under union. Compute the closure under union of a given structure using Ganter’s algorithm. The resulting space is directly written to a file.

Return Value: number of states in the space.

Parameters: 

- `b` — basis to be closed.
- `mode` — file format.
- `filename` — filename for resulting space.

Parameters: 

- `st` — structure to be closed
### 3.5.4

```c
int u_closure (structure* su, filetype mode, const char filename[])
```

*close a structure under union*

Close a structure under union. Compute the closure under union of a structure using Ganter’s algorithm. The resulting space is directly written to a file.

**Return Value:** number of states in the space.

**Parameters:**
- `b` — basis to be closed.
- `mode` — file format.
- `filename` — filename for resulting space.

### 3.5.5

```c
structure* s_closure (structure* su)
```

*close a structure under intersection*

Close a structure under intersection. Compute the closure under intersection of a given structure using Ganter’s algorithm.

**Return Value:** resulting structure closed under intersection, NULL if an error occurred.

**Parameters:**
- `st` — structure to be closed.

### 3.5.6

```c
int s_closure (structure* su, filetype mode, const char filename[])
```

*close a structure under intersection*

This page has been automatically generated with DOC++

DOC++ is ©1995 by Roland Wunderling & Malte Zöckler
close a structure under intersection. Compute the closure under intersection of a structure using Ganter’s algorithm. The resulting structure is directly written to a file.

Return Value: number of states in the structure.

Parameters:
- \( b \) — basis to be closed.
- \( \text{mode} \) — file format.
- \( \text{filename} \) — filename for resulting structure.
### Names

| 4.1 | int | **add_info** (int** info, int position, int number) |
|     |     | *add identification numbers* |
| 4.2 | int | **change_two_infos** (int** info, int pos1, int pos2, int nr) |
|     |     | *change the positions of two lines of informations* |
| 4.3 | int* | **remove_info** (int pos, int* line) |
|     |     | *remove an information number* |
| 4.4 | int | **is_number_in_info** (int num, int* line) |
|     |     | *is a number element of a set of information numbers?* |
| 4.5 | int* | **copy_info_line** (int* info) |
|     |     | *make a copy of an info line* |
| 4.6 | void | **set_info_line** (int* set, int* orig) |
|     |     | *make a copy of an info line* |

ATTENTION: In all following functions the numbering of items starts with '0'! This means, the first item has the number '0', the last 'q.size-1'. The same applies for patterns or states.

#### 4.1

**int add_info (int** info, int position, int number)**

*add identification numbers.* In a set of information vectors, an additional information number is added at a certain position.  

**Return Value:** 1 if an error occurred, 0 else.  

**Parameters:** 
- **info** — pointer to matrix of information numbers  
- **position** — position of the information vector in the information structure, where the number should be added  
- **number** — number to be added
4.2

```c
int change_two_infos (int** info, int pos1, int pos2, int nr)
```

*change the positions of two lines of informations*

change the positions of two lines of informations. The information numbers from the position 'pos1' are written to 'pos2' and vice versa.

**Return Value:** -1 if an error occurred, 0 else

**Parameters:**
- `info` — pointer to matrix of information numbers
- `pos1` — first position of info-numbers to be changed
- `pos2` — second position of info-numbers to be changed
- `nr` — number of info-lines possible in the info-struct

4.3

```c
int* remove_info (int pos, int* line)
```

*remove an information number*

remove an information number. Remove a single information number out of a vector containing informations for a special position in the information matrix.

**Return Value:** new information line

**Parameters:**
- `pos` — position of the info-number to be removed
- `line` — line in the information matrix, which should be removed

4.4

```c
int is_number_in_info (int num, int* line)
```

*is a number element of a set of information numbers?*
Return Value: yes (1) or no (0)
Parameters: num — number to look for
            line — pointer to vector of information numbers

4.5

int* copy_info_line (int* info)

make a copy of an info line

make a copy of an info line. Allocate the necessary memory and
make a copy of an already existing line of information numbers.
Return Value: pointer to copied info vector
Parameters: info — vector of information numbers to be copied

4.6

void set_info_line (int* set, int* orig)

make a copy of an info line

make a copy of an info line. Make a copy of an already ex-
isting info line, if the necessary memory already is available.
Parameters: set — pointer to copied info line
            orig — info line to be copied
Working with Item Hypotheses

Names

5.1 ihypoth* **load_ihypoth**(const char filename[])

load an item hypothesis from a file

................................. 22

5.2 int **write_ihypoth**(FILE *f, ihypoth *ih)

write an item hypothesis ........... 23

5.3 ihypoth* **new_ihypoth**(int q_size)

allocate memory for a new item hypothesis ............................ 23

5.4 ihypoth* **copy_ihypoth**(ihypoth* ih)

make a copy of an item hypothesis structure ............................ 23

5.5 void **free_ihypoth**(ihypoth **r)

return memory to the system ... 24

5.6 srbi** ihypoth2srbi**(ihypoth* h)

item-hypothesis to surmise relation ................................. 24

---

5.1

ihypoth* **load_ihypoth**(const char filename[])

load an item hypothesis from a file.

This function loads an item hypothesis from a file. It determines automatically which type of data is stored in the file.

**Return Value:** pointer to resulting item hypothesis.

**Parameters:** filename[] — Name of file to be loaded
5.2

```c
int write_ihypoth (FILE *f, ihypoth *ih)
```

write an item hypothesis

write an item hypothesis. This function writes an item hypothesis to a given file (or alternatively) to stdout or stderr.

Return Value: -1 if an error occurred, 0 else

Parameters:
- `f` — File to be written to
- `ih` — Pointer to item hypothesis

5.3

```c
ihypoth* new_ihypoth (int q_size)
```

allocate memory for a new item hypothesis

Return Value: Pointer to the new item hypothesis struct

Parameters:
- `q_size` — Number of items

5.4

```c
ihypoth* copy_ihypoth (ihypoth* ih)
```

make a copy of an item hypotheses structure

Return Value: Pointer to the resulting item hypothesis structure

Parameters:
- `ih` — Original item hypothesis
5.5

```c
void free_ihypo (ihypo **r)
```

return memory to the system

return memory to the system. Return memory used by an item hypothesis struct to the system.

5.6

```c
srbi* ihypo2srbi (ihypo* h)
```

*item-hypothesis to surmise relation*

*item-hypothesis to surmise relation*. Change an hypothesis on surmise relations between some items to an surmise relation structure, all fields, which were not known in the hypotheses (all fields with ".") are set '0', the relation is completed because of transitivity and reflexivity properties.
6 Working with Answer Patterns

Names

6.1 int **write_patterns** (patterns* pa)
   write a set of answer-patterns to stdout

6.2 patterns* **copy_patterns** (patterns* orig)
   make a copy of a set of patterns

6.3 patterns* **load_patterns** (const char filename[])
   load a pattern set from a file

6.4 int **save_patterns** (patterns* pa, filetype mode, const char filename[])
   write a pattern set to a file

6.5 patterns* **new_patterns** (int q_size, int p_size)
   allocate memory for a pattern set

6.6 void **free_patterns** (patterns** p)
   Return memory used by a pattern set to the system

6.7 data* **patterns2data** (patterns* p)
   convert pattern set to data set

6.8 patterns* **data2patterns** (data* d)
   convert data set to pattern set

6.1 int **write_patterns** (patterns* pa)

write a set of answer-patterns to stdout. Write a set of answer patterns to stdout, using '1' for a correct solved item, '0' for a wrong solution and 'x' for a not answered item.

Return Value: -1, if an error occurred, 0 else.
6.2 patterns* \texttt{copy\_patterns} (patterns* orig)

\textit{make a copy of a set of patterns}

make a copy of a set of patterns. The necessary memory is allocated.

\textbf{Return Value:} pointer to copied patterns.

\textbf{Parameters:} \texttt{orig} — original patterns to be copied

6.3 patterns* \texttt{load\_patterns} (const char filename[])

\textit{load a pattern set from a file}

load a pattern set from a file. This function loads a pattern set from a file.

\textbf{Return Value:} pointer to resulting pattern set.

\textbf{Parameters:} \texttt{filename[]} — name of the file to be loaded

6.4 int \texttt{save\_patterns} (patterns *pa, filetype mode, const char filename[])

\textit{write a pattern set to a file}

write a pattern set to a file. This function writes a set of answer patterns to a file using the new patternfile format.

\textbf{Return Value:} error code.
6.5  

patterns*  **new_patterns** (int q_size, int p_size)

allocate memory for a pattern set

allocate memory for a pattern set.  **Parameters:**  
q_size — number of items  
p_size — number of patterns

6.6  

void  **free_patterns** (patterns **p)

Return memory used by a pattern set to the system

**Parameters:**  
p — patterns set

6.7  

data*  **patterns2data** (patterns *p)

convert pattern set to data set

convert pattern set to data set.  This function takes a pattern set and converts it to a data set assuming that all un-answered items are not mastered.

**Return Value:**  
pointer to resulting data set

**Parameters:**  
p — patterns structure
6.8 patterns* \textbf{data2patterns} (data* d)

\begin{itemize}
\item \textit{convert data set to pattern set} This function is mere a cast operator.
\item In the resulting pattern set, all items are considered to be answered, i.e. the un-answered matrix is set to zero.
\end{itemize}

\textbf{Return Value:} pointer to resulting patterns set

\textbf{Parameters:} \hspace{1cm} d — data set
Names

7.1 patterns* \texttt{remove\_item} (patterns* \texttt{pa}, int \texttt{itemnr})

remove a single item  

\hspace{1cm} \rightarrow 30

7.2 patterns* \texttt{remove\_items} (patterns* \texttt{pa}, int* \texttt{nrs}, int \texttt{number})

remove given items out of an answer-pattern  

\hspace{1cm} \rightarrow 31

7.3 patterns* \texttt{remove\_pattern} (patterns* \texttt{orig}, int \texttt{number})

remove a pattern  

\hspace{1cm} \rightarrow 31

7.4 patterns* \texttt{remove\_patterns} (patterns* \texttt{orig}, int \texttt{number}, int* \texttt{pat\_nrs})

remove a set of answer-patterns  

\hspace{1cm} \rightarrow 31

7.5 double* \texttt{percent\_items\_answered} (patterns* \texttt{p})

how many percent of students answered each item?  

\hspace{1cm} \rightarrow 32

7.6 double* \texttt{percent\_items\_correct} (patterns* \texttt{p})

how many percent of students gave a correct answer to each item?  

\hspace{1cm} \rightarrow 32

7.7 double* \texttt{percent\_pattern\_answered} (patterns* \texttt{p})

how many items were answered in each pattern?  

\hspace{1cm} \rightarrow 33

7.8 int* \texttt{number\_items\_answered} (patterns* \texttt{pa})

how many persons gave an answer to an item?  

\hspace{1cm} \rightarrow 33

7.9 int \texttt{persons\_all\_items\_answered} (patterns* \texttt{pa})

how many persons answered all the items?  

\hspace{1cm} \rightarrow 33

7.10 patterns* \texttt{delete\_percent\_ua\_patterns} (patterns* \texttt{pa}, int \texttt{percent})

delete patterns in which less than a given percentage of items was answered  

\hspace{1cm} \rightarrow 34

7.11 patterns* \texttt{delete\_percent\_ua\_items} (patterns* \texttt{pa}, int \texttt{percent})
7 Additional Functions for Working with Patterns

Delete items that have been answered less often than a given percentage

7.12 patterns* remove_pat_with ua (patterns* pa)
   remove all patterns, where one or more items are not answered

7.13 int is_pattern_el (bitset sol, bitset unans, patterns* pat)
   is a pattern element of a set of patterns?

7.14 int is_pattern_el (bitset sol, bitset unans, bitset* orig_sol, bitset* orig_una, int q, int num)
   is a pattern element of a set of patterns?

7.15 int patt_in_space (bitset sol, bitset unans, space* sp)
   Is it possible, that an answer pattern is element of a given space?

ATTENTION: In all following functions the numbering of items starts with '0'!
This means, the first item has the number '0', the last 'q.size-1'. The same applies for patterns.

7.1 patterns* remove_item (patterns* pa, int itemnr)
remove a single item

Remove a single item out of a patterns-structure.

Return Value: new patterns-structure with deleted item

Parameters: pa — pointer to patterns struct
            itemnr — number of item to be deleted
7.2

patterns* remove_items (patterns* pa, int* nrs, int number)

remove given items out of an answer-pattern

remove given items out of an answer-pattern. Remove the given items out of an answer-pattern.

Return Value: patterns with removed item, 'NULL', if an error occurred

Parameters:
- pa — patterns, where the items should be removed
- nrs — numbers of the items to be removed
- number — how many items should be removed

7.3

patterns* remove_pattern (patterns* orig, int number)

remove a pattern

remove a pattern. Remove a single pattern out of a patterns-structure.

Return Value: new patterns-structure with removed pattern

Parameters:
- pa — pointer to patterns-structure
- number — number of pattern to be removed

7.4

patterns* remove_patterns (patterns* orig, int number, int* pat_nrs)

remove a set of answer-patterns

remove a set of answer-patterns. Remove a set of answer-patterns
out of a patterns-structure. Attention: Currently the numbers of the patterns to be deleted have to be entered in increasing order!

**Return Value:** new patterns-structure with patterns removed.

**Parameters:**
- `orig` — patterns-structure.
- `number` — number of patterns to be removed.

### 7.5

```c
double* percent_items_answered (patterns* p)
```

**Return Value:** pointer to vector which includes the percentages for each item.

### 7.6

```c
double* percent_items_correct (patterns* p)
```

**Return Value:** pointer to vector which includes the percentages for each item.
7.7

```c
double* percent_pattern_answered (patterns* p)
```

*how many items were answered in each pattern?*

how many items were answered in each pattern?. Calculate for each pattern how many percent of the items have been answered (no difference, if the answer was correct or not).

**Return Value:** pointer to vector which includes the percent-numbers for each pattern.

7.8

```c
int* number_items_answered (patterns* pa)
```

*how many persons gave an answer to an item?*

how many persons gave an answer to an item?. Calculate for each item how many subject answered the item (making no difference, if the answer was correct or not).

**Return Value:** pointer to vector with number of subjects, that answered item i on the i-th position.

7.9

```c
int persons_all_items_answered (patterns* pa)
```

*how many persons answered all the items?*

how many persons answered all the items?. Calculate, how many persons answered all the given items (making no difference, if the answer was correct or not).

**Return Value:** number of persons, that answered all items.
7.10

patterns* delete_percent ua_patterns (patterns* pa, int percent)

*delete patterns in which less than a given percentage of items was answered*

delete patterns in which less than a given percentage of items was answered. All these patterns in a patterns struct are deleted, where less than the given percentage of the items was answered (no matter, if the answer was correct or not).

Return Value: patterns-structure with deleted patterns.

Parameters:
- *pa* — pointer to patterns struct
- *percent* — percentage of items that must be answered, if the patterns should not be deleted

7.11

patterns* delete_percent ua_items (patterns* pa, int percent)

*delete items that have been answered less often than a given percentage*

Return Value: patterns-structure with deleted items.

Parameters:
- *pa* — pointer to patterns struct
- *percent* — percentage of students that must have answered the item, if the item should not be deleted

7.12

patterns* remove_pat_with ua (patterns* pa)

*remove all patterns, where one or more items are not answered*
Return Value: pointer to patterns-structure without the patterns, where no answer to one or more items was given

### 7.13

```c
int is_pattern_el (bitset sol, bitset unans, patterns* pat)
```

is a pattern element of a set of patterns?

Return Value: smallest number of the pattern in the patterns struct, which is equal to the given pattern or 0, if the given pattern is no element of the structure

Parameters:
- `sol` — bitset including the correct solved items the pattern
- `unans` — bitset including the unanswered items in the pattern
- `pat` — pointer to patterns struct

### 7.14

```c
int is_pattern_el (bitset sol, bitset unans, bitset* orig_sol,
                  bitset* orig_una, int q, int num)
```

is a pattern element of a set of patterns?. Look, if a single pattern is element of a set of patterns, which is given in form of two bitsets (one for the correctly solved items, one for the unanswered items) here.

Return Value: smallest number of the pattern in the patterns struct, which is equal to the given pattern or 0, if the given pattern is no element of the structure
Parameters:

sol — bitset including the correct solved items the pattern
unans — bitset including the unanswered items in the pattern
orig_sol — pointer to set of bitsets including correctly solved items
orig_una — pointer to set of bitsets including unanswered items
q — number of items per bitset/pattern
num — number of patterns in the set of patterns

7.15

int patt_in_space (bitset sol, bitset unans, space* sp)

Is it possible, that an answer pattern is element of a given space?

Is it possible, that an answer pattern is element of a given space?. Look, if a pattern does not contradict any of the states in a space.

Return Value:

yes (1) or no (0)

Parameters:

sol — bitset including the correct solved items the pattern
unans — bitset including the unanswered items in the pattern
sp — knowledge space
Equivalence Properties of Items

Names

8.1 int is_equivalent_items (int item1, int item2, srbi* sr)
   are two items equivalent? ........ 37

8.2 int is_up_parallel (int item1, int item2, srbi* sr)
   are two items 'up-parallel'? ...... 38

8.3 int is_down_parallel (int item1, int item2, srbi* sr)
   are two items 'down-parallel'? .. 38

8.4 int is_parallel (int item1, int item2, srbi* sr)
   are two items parallel? .......... 38

ATTENTION: In all following functions the numbering of items starts with '0'!
This means, the first item has the number '0', the last 'q_size-1'. ATTENTION:
We have to find new and better names for the different levels of parallelility and
equivalence.

8.1

int is_equivalent_items (int item1, int item2, srbi* sr)

are two items equivalent?.

are two items equivalent? Look, if two items a and b are equiva-

lent. Comment: two items a and b are called equivalent here,
if aSb and bSa ('S' denotes the surmise relation between items).

Return Value: yes or no (1 or 0), -1 if an error occured

Parameters:  item1 — first item
             item2 — second item
             sr — surmise relation between items

This page has been automatically generated with DOC++
DOC++ is ©1995 by Roland Wunderling & Malte Zöckler  January 8, 2002 37
8.2

```c
int is_up_parallel (int item1, int item2, srbi* sr)
```

are two item 'up-parallel'?.

Comment: two items a and b are called 'up-parallel' here, if for all items c, c!=a and c!=b, in the knowledge-space we have: cSa => cSb

Return Value: yes or no (1 or 0), -1 if an error occurred

Parameters:
- `item1` — first item
- `item2` — second item
- `sr` — surmise relation between items

8.3

```c
int is_down_parallel (int item1, int item2, srbi* sr)
```

are two items 'down-parallel'?.

Comment: two items are called 'down-parallel' here, if for all items c, c!=a and c!=b, in the knowledge-space we have: aSc => bSc

Return Value: yes or no (1 or 0), -1 if an error occurred

Parameters:
- `item1` — first item
- `item2` — second item
- `sr` — surmise relation between items

8.4

```c
int is_parallel (int item1, int item2, srbi* sr)
```

are two item parallel?.

Comment: two items a and b
are called 'parallel' here, iff they are up and down parallel

Return Value: yes or no (1 or 0), -1 if an error occurred

Parameters:

item1 — first item
item2 — second item
sr — surmise relation between items
Data Structures for Working with Tests

Names

9.1 class partition \( \text{Partition of a set of items into tests} \) 40

9.2 class disjoint partition \( \text{Partition of items into disjoint tests} \) 44

9.3 class srbt \( \text{Surmise relations between tests} \) 47

9.1 class partition

\( \text{Partition of a set of items into tests} \)

Inheritance

9.1 partition

\hline

9.2 disjoint partition

Public Members

\begin{align*}
\text{int} & \quad \text{q.size} & \text{number of items} \\
\text{int} & \quad \text{t.size} & \text{number of tests} \\
\text{bitset.basis} & \quad \text{matrix} & \text{binary matrix (which item belongs to which test)} \\
\text{int}^{**} & \quad \text{item.info} & \text{information numbers for each item} \\
\text{int}^{**} & \quad \text{test.info} & \text{information numbers for each test}
\end{align*}
9.1.1 partition ()  standard constructor  ............  42
9.1.2 partition (int q_size, int t_size)  'new'-constructor  ............  42
9.1.3 partition (partition* orig)  'copy'-constructor  ............  42
9.1.4 partition (const char filename[])  'load'-constructor  ............  42
9.1.5 partition (char* buffer, int q_size, int t_size)  constructor for reading from a buffer  ............  43
9.1.6 virtual ~partition ()  destructor  ............  43
9.1.7 int save (filetype mode, const char filename[])  save a partition to a file  ............  43
9.1.8 void write ()  write a partition  ............  44

Protected Members

int wordq  number of words needed to store q_size bits
int storage  storage needed for the binary matrix
structtype stype  type of structure (partition or disjoint partition) - for internal use
int get_memory ()  allocate memory for the partition, according to the given number of items and test  ............  44

Partition of a set of items into tests. The class partition includes the basic structure for the partitioning of a set of items into tests. The binary matrix includes a '1', if an items belongs to a certain test, '0' else.

9.1.1 partition ()

standard constructor
standard constructor. This constructor is called from the inherited class.

9.1.2

\begin{boxedmath}
\text{partition} \ (\text{int} \ q\_\text{size}, \ \text{int} \ t\_\text{size})
\end{boxedmath}

'new'-constructor

'new'-constructor. This constructor allocates memory for a new partition. If not enough memory is available, the class destructor is called.

Parameters:
- \(q\_\text{size}\) — number of items
- \(t\_\text{size}\) — number of tests

9.1.3

\begin{boxedmath}
\text{partition} \ (\text{partition}* \ \text{orig})
\end{boxedmath}

'copy'-constructor

'copy'-constructor. Make a copy of an existing partition

Parameters:
- \(\text{orig}\) — original partition

9.1.4

\begin{boxedmath}
\text{partition} \ (\text{const} \ \text{char} \ \text{filename}[])\n\end{boxedmath}

'load'-constructor

'load'-constructor. Load a partition from file. If the file cannot be opened or not free memory is available, the class destructor is called. If the type of file is not 'partition' or 'disjoint partition', no partition is loaded, the class destructor is called.

Parameters:
- \(\text{filename}[]\) — name of inputfile
9.1.5

partition (char* buffer, int q_size, int t_size)

constructor for reading from a buffer

Read a partition with defined size from a buffer.

Parameters:
- buffer — bitset matrix including the partition
- q_size — number of items
- t_size — number of tests

9.1.6

virtual ~partition ()

destructor

destructor. Return used memory to the system, set and structype variable to UNKNOWN and the variables q_size and t_size to '-1'.

9.1.7

int save (filetype mode, const char filename[])

save a partition to a file

Return Value: '1' if an error occured, '0' else
Parameters:
- mode — mode of the file to be written (binary or ASCII, partition or dispartition)
- filename — name of output file
9.1.8

```c
void write ()
```

write a partition

write a partition. Write a partition in form of a matrix to stdout

9.1.9

```c
int get_memory ()
```

allocate memory for the partition, according to the given number of items and test

Return Value: '1' if an error occurred, '0' else.

9.2

```c
class disjpartition : public partition
```

Partition of items into disjoint tests

Inheritance

```
```

This page has been automatically generated with DOC++

DOC++ is ©1995 by Roland Wunderling & Malte Zickler
Public Members

9.2.1 \textit{disjpartition (const char filename[])}

'load-constructor'  \ldots \ldots \ldots 45

9.2.2 \textit{disjpartition (int q\_size, int t\_size)}

'new-constructor'  \ldots \ldots \ldots 45

9.2.3 \textit{disjpartition (disjpartition* dp)}

'copy-constructor'  \ldots \ldots \ldots 46

9.2.4 \textit{disjpartition (char* buffer, int q, int t)}

constructor for reading from a buffer  \ldots \ldots \ldots \ldots \ldots \ldots \ldots 46

9.2.5 int is\_disjpartition ()

is a given partition disjoint? \ldots 46

Partition of items into disjoint tests. The class disjpartition is derived from partition. It has one additional property: Each item in the partition must belong exactly to one test.

9.2.1 \textbf{disjpartition (const char filename[])}

'load-constructor'. Load a disjoint partition from a file. If 'type of file' is 'partition', it is tested, if each item belongs exactly to one test. If 'type of file' is neither 'partition' nor 'disjoint partition', no partition is loaded, the class destructor is called.

\textbf{Parameters:} \hspace{1em} \texttt{filename[]} — name of input file

9.2.2 \textbf{disjpartition (int q\_size, int t\_size)}

'new-constructor'. Allocate memory for a new disjoint partition of given size.
9.2.3

**disjpartition** (disjpartition* dp)

'copy-constructor'. Make a copy of a disjoint partition. The necessary memory is allocated.

**Parameters:**
- dp — original disjoint partition

9.2.4

**disjpartition** (char* buffer, int q, int t)

Constructor for reading from a buffer. Read a matrix meaning a disjoint partition from a buffer.

**Parameters:**
- buffer — vector including the disjoint partition
- q — number of items
- t — number of tests

9.2.5

**is_disjointpartition** ()

Is a given partition disjoint?

**Return Value:**
- '1', if the partition is disjoint, '0' else
9.3

class srbt

Surmise relations between tests

Public Members

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>char**</td>
<td>smatrix</td>
<td>surmise relation between tests</td>
</tr>
<tr>
<td>char**</td>
<td>slmatrix</td>
<td>left-covering surmise relation</td>
</tr>
<tr>
<td>char**</td>
<td>srmatrix</td>
<td>right-covering surmise relation</td>
</tr>
<tr>
<td>int**</td>
<td>test_info</td>
<td>information numbers for each test</td>
</tr>
</tbody>
</table>

srbt ()
standard constructor

9.3.1
srbt (int t_size)
‘new’-constructor Allocate memory for a new srbt class with given number of tests ................. 48

9.3.2
srbt (const char filename[])
load-constructor Load a srbt-structure from a file .......... 48

9.3.3
srbt (srbt* sr)
‘copy’-constructor Make a copy of an existing surmise relation .... 49

9.3.4
srbt (partition* p, srbi* si)
constructor for calculating the surmise relations between tests out of surmise relation between items and the partition of items into tests ....................... 49

9.3.5
srbt ()
destructor ....................... 49

9.3.6
int save (const char filename[])
save a srbt-structure to a file ... 50

9.3.7
void write (FILE* f)
write all three kinds of surmise relations ......................... 50

Private Members

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>stype</td>
<td>type of structure</td>
</tr>
</tbody>
</table>
Surmise relations between tests. In this class three different forms of surmise relations between tests can be stored: 'normal' surmise relation between tests, right-covering surmise relation, left-covering surmise relation. All these relations are coded in form of a matrix, which has as many lines and columns as the number of tests. Writing a '1' in the i-th column and the j-th line means, that tests i is in surmise relation to tests j, writing a '0' on this position means, that these tests are not in surmise relation. The order of storage in the file is: surmise relation, right-covering surmise relation, left-covering surmise relation. A comment line with an appropriate headline is stored for each matrix.

9.3.1

\texttt{srbt (int t\_size)}

\textit{'new'-constructor} Allocate memory for a new srb class with given number of tests

\textbf{Parameters:} \texttt{t\_size} — number of tests.

9.3.2

\texttt{srbt (const char filename[])}

\textit{load-constructor} Load a srbt-structure from a file

load-constructor Load a srbt-structure from a file. If an error occurs (file cannot be opened, or type of file is not srbt, the class destructor is called).
9.3.3

srbt (srbt* sr)

'copy'-constructor

Make a copy of an existing surmise relation.

The necessary memory is allocated.

Parameters:

sr — surmise relations between tests to be copied

9.3.4

srbt (partition* p, srbi* si)

constructor for calculating the surmise relations between tests out of surmise relation between items and the partition of items into tests

Parameters:

p — partition of items into tests
si — surmise relation between items

9.3.5

~srbt ()

destructor

destructor. The destructor for the srbt-class returns used memory to the system, sets the structtype variable to UNKNOWN and the number of tests to '-1'.
9.3.6

```c
int save (const char filename[])
```

*save a srbt-structure to a file*

save a srbt-structure to a file. All three matrices for surmise relation between tests, right-and leftcovering surmise relation are written to a file together with appropriate headlines for each relation matrix and the according header for srbt-files.

**Parameters:**
- `filename[]` — name of outputfile

9.3.7

```c
void write (FILE* f)
```

*write all three kinds of surmise relations*

write all three kinds of surmise relations. All three matrices (surmise relation, right-and leftcovering surmise relations) are written to a file.

**Parameters:**
- `f` — name of file (also possible: stdout/stderr)
Investigating Equal Structures

Names

```c
int is_equal_struct ( structure* s1 , structure* s2 )
look, if two structures are equal
int is_equal_partition ( partition* p1 , partition* p2 )
look, if two partitions are equal
int is_equal_disjoint_partition ( disjoint_partition* p1 , disjoint_partition* p2 )
look, if two disjoint partitions are equal.
int is_equal_space ( space* s1 , space* s2)
look, if two spaces are equal.
int is_equal_data ( data* d1, data* d2)
look, if two data structs are equal.
```

10.1 `int is_equal_test (partition* p, int test1, int test2)`
look, if two tests in a partition are equal

The functions "is_equal_struct" work for structures, partitions, disjoint partitions, data and spaces. In all these functions two structs are called equal, iff they have the same number of tests/patterns/states, the same number of items and if each line (=test/pattern/state) in "struct1" is element of "struct2" and vice versa. This function does not consider any form of equivalence or parallelity between items.

Return Value: 1 if the structs are equal, 0 else.

```
int is_equal_test (partition* p, int test1, int test2)
look, if two tests in a partition are equal
```
if two tests in a given partition contain the same items. It does not consider any form of equivalence or parallelity between items.

Return Value: yes or no (1 or 0), -1 if an error occurred.

Parameters:
- `p` — partition into tests
- `test1` — number of first test
- `test2` — number of second test
Working with (Disjoint) Partitions

Names

11.1 bitset  \textbf{get\_test (partition* p, int position)}  
\textit{get a single test from a partition}  
54

11.2 bitset  \textbf{get\_test (partition* p, int position)}  
\textit{get a single test from a disjoint partition}  
54

11.3 int  \textbf{set\_test (partition* p, int position, bitset value)}  
\textit{set a test in a partition}  
54

11.4 int  \textbf{set\_test (disjpartition* p, int position, bitset value)}  
\textit{set a test in a disjoint partition}  
55

11.5 partition*  \textbf{remove\_tests (partition* p, int start, int end=-1)}  
\textit{remove tests within a partition}  
55

11.6 disjpartition*  \textbf{remove\_tests (disjpartition* p, int start, int end=-1)}  
\textit{remove tests from a disjoint partition}  
56

11.7 partition*  \textbf{testunion (partition* p, int test1, int test2)}  
\textit{union of two tests in a partition}  
56

11.8 int  \textbf{is\_disj\_partition (partition* p)}  
\textit{is a partition disjoint?}  
56

11.9 int  \textbf{count\_item\_test (partition* p, int testnr)}  
\textit{count, how many items are in the given test}  
57

11.10 srbi*  \textbf{order\_items (partition* p, srbi* sr)}  
\textit{order items}  
57

11.1 bitset  \textbf{get\_test (partition* p, int position)}  
\textit{get a single test from a partition}
11.2  

bitset  get_test (partition* p, int position)

get a single test from a disjoint partition

11.3  

int  set_test (partition* p, int position, bitset value)

set a test in a partition

Return Value:  bitset with requested test
Parameters:  
p — partition into tests
position — index of the test to be copied

Return Value:  bitset with requested test
Parameters:  
p — disjoint partition into tests
position — index of the test to be copied

Return Value:  -1, if an error occurred, 0 else
Parameters:  
p — partition into tests
position — index of the test to be set
value — new value of the test
11.4

```c
int set_test (disjpartition* p, int position, bitset value)
```

set a test in a disjoint partition

**Return Value:** -1, if an error occured, 0 else

**Parameters:**
- `p` — partition into tests
- `position` — index of the test to be set
- `value` — new value of the test

11.5

```c
partition* remove_tests (partition* p, int start, int end=-1)
```

remove tests within a partition

remove tests within a partition. Remove a set of tests from a partition, starting with the test on the 'start' position, ending with the test on the 'end' position, replace the tests on these positions with the last tests in the partition. Comment: the 'end' parameter is a default parameter - it can be left out, if only one test should be removed.

**Return Value:** partition with removed test

**Parameters:**
- `p` — partition into tests
- `start` — position of the first test to be removed
- `end` — position of the last test to be removed, can be left out

11.6

```c
disjpartition* remove_tests (disjpartition* p, int start, int end=-1)
```

This page has been automatically generated with DOC++

DOC++ is ©1995 by Roland Wunderling & Malte Zöckler

January 8, 2002 55
remove tests from a disjoint partition

remove tests from a disjoint partition. Remove a set of tests from a partition or disjoint partition, starting with the test on the 'start' position, ending with the test on the 'end' position, replace the tests on these positions with the last tests in the partition. Comment: the 'end' parameter is a default parameter - it can be left out, if only one test should be removed.

Return Value: disjoint partition with removed test

Parameters: p — disjoint partition into tests
             start — position of the first test to be removed
             end — position of the last test to be removed, can be left out

11.7

partition* testunion (partition* p, int test1, int test2)

union of two tests in a partition

union of two tests in a partition. Calculate the union of two given tests in a partition and return a new partition which includes the union of the two given tests instead of the two tests.

Return Value: new partition with one test less

Parameters: p — partition into tests
             test1 — number of first test
             test2 — number of second test

11.8

int is_disj_partition (partition* p)

is a partition disjoint?

is a partition disjoint?. Look, if each item in the partition belongs to exactly one test.
11 Working with (Disjoint) Partitions

Return Value: 1 if the partition is disjoint, 0 else
Parameters: \( p \) — pointer to the partition

11.9

\[
\text{int count_item_test (partition* p, int testnr)}
\]

\text{count, how many items are in the given test}

Return Value: number of items in the requested test
Parameters: \( p \) — pointer to the partition
\( \text{testnr} \) — number of test in the partition

11.10

\[
\text{srbi* order_items (partition* p, srbi* sr)}
\]

\text{order items}

order items. Order the in a given matrix of surmise relations in a way, that the first \( n \) items belong to the first test in the partition, the next \( m \) items belong to the second test and so on. The identity numbers for each item are also changed.

Return Value: ordered surmise relation between items
Parameters: \( p \) — partition
\( sr \) — surmise relations between items
Surmise Relations Within and Across Tests

Names

12.1 srbi* write\_SRwT\_matrix (partition* p, srbi* sr)
write all surmise relations within tests in form of a matrix ...... 58

12.2 srbi* write\_SRwT\_rel (partition* p, srbi* sr)
write all surmise relations within tests in form of item pairs ...... 59

12.3 srbi* write\_SRxT\_matrix (partition* p, srbi* sr)
write all surmise relations across tests in form of a matrix ...... 59

12.4 srbi* write\_SRxT\_rel (partition* p, srbi* sr)
write all surmise relations across tests in form of item pairs ...... 59

12.1

srbi* write\_SRwT\_matrix (partition* p, srbi* sr)

write all surmise relations within tests in form of a matrix. Only the surmise relations within tests, meaning the surmise relations between items of the same test, are regarded. Surmise relations for items which are in two different tests are all set to '0'. The resulting matrix is written to stdout.

Return Value: surmise relations for items within tests
Parameters: p — partition into tests sr — surmise relations between items
12.2

srbi* write_SRwT_rel (partition* p, srbi* sr)

write all surmise relations within tests in form of item pairs

write all surmise relations within tests in form of item pairs. All surmise relations for items between tests are written to stdout in form of item pairs, e.g. item1 $ item2.

Return Value: surmise relations for items within tests
Parameters:  
p — partition
sr — matrix with surmise relations

12.3

srbi* write_SRxT_matrix (partition* p, srbi* sr)

write all surmise relations across tests in form of a matrix

write all surmise relations across tests in form of a matrix. Only the surmise relations for items in different tests are regarded, the resulting matrix is written to stdout.

Return Value: surmise relations for items across tests
Parameters:  
p — partition
sr — surmise relations between tests

12.4

srbi* write_SRxT_rel (partition* p, srbi* sr)

write all surmise relations across tests in form of item pairs

write all surmise relations across tests in form of item pairs. Only the surmise relations for items in different tests are regarded, the resulting relations are written to stdout in form of item pairs.

Return Value: surmise relations for items across tests
Parameters:  
p — partition
sr — surmise relations between items
Creating Different Kinds of Partitions

Names

13.1 disjpartition*

random_part (int q_size, int max_t_size=0)
create a random disjoint partition

13.2 disjpartition*

random_part_t (int q_size, int t_size)
create a random partition with a
given number of tests

13.3 disjpartition*

equal_part (int q_size, int t_size)
create a random disjoint partition
where each test has the same num-
er of items

13.4 disjpartition*

min_item_part (int q_size, int min_item_no,
int t_size)
create a random disjoint partition
into tests, where each test has at
least a given minimal number of
items

13.5 disjpartition*

trans_part (int max_t_size, srbi* sr)
create a transitiv disjoint partition
into tests

13.6 disjpartition*

antisym_part (int max_t_size, srbi* sr)
create an antisymmetric disjoint
partition into tests

13.7 disjpartition*

connex_part (srbi* sr)
create a disjoint partition into
connex tests

13.8 disjpartition*
13 Creating Different Kinds of Partitions

**left_cover_part** (int max_t_size, srbi* sr)
create a left-covering disjoint partition

13.9 disjpartition*

**right_cover_part** (int max_t_size, srbi* sr)
create a right-covering disjoint partition

13.10 partition* **make_partition** (int q_size, int t_size)
create a partition

13.1

**disjpartition*** **random_part** (int q_size, int max_t_size=0)
create a random disjoint partition.
Create a random disjoint partition, where the number of tests is selected randomly, it will be between 1 and max_t_size. Comment: the maximal number of tests is a default parameter, default is q_size/2.

Return Value: resulting disjoint partition
Parameters:
q_size — number of items
max_t_size — maximal number of tests

13.2

**disjpartition*** **random_part_t** (int q_size, int t_size)
create a random partition with a given number of tests

Return Value: resulting disjoint partition
Parameters:
q_size — number of items
t_size — number of tests
13.3 disjpartition* **equal_part** (int q_size, int t_size)

create a random disjoint partition where each test has the same number of items

Return Value: resulting disjoint partition, NULL if q_size modulo t_size not zero
Parameters: 
- q_size — number of items
- t_size — number of tests

13.4 disjpartition* **min_item_part** (int q_size, int min_item_no, int t_size)

create a random disjoint partition into tests, where each test has at least a given minimal number of items

Return Value: resulting disjoint partition
Parameters: 
- q_size — number of items
- t_size — number of tests
- min_item_no — minimal number of items per test

13.5 disjpartition* **trans_part** (int max_t_size, srbi* sr)

create a transitive disjoint partition into tests
create a transitiv disjoint partition into tests. WARNING: Not implemented! The mathematical solutions are still missing! You have to enter a maximal size of tests, because the trivial solution (one item per test) is always possible.

**Return Value:** resulting disjoint partition  
**Parameters:**  
- `max_t_size` — maximal number of tests to be created  
- `sr` — surmise relations between items

### 13.6

```c
disjpartition* antisym_part (int max_t_size, srbi* sr)
```

create an antisymmetric disjoint partition into tests

create an antisymmetric disjoint partition into tests. WARNING: Not implemented! The mathematical solutions are still missing! You have to enter a maximal size of tests, because the trivial solution (one item per test) is always possible.

**Return Value:** resulting disjoint partition  
**Parameters:**  
- `max_t_size` — maximal number of tests to be created  
- `sr` — surmise relations between items

### 13.7

```c
disjpartition* connex_part (srbi* sr)
```

create a disjoint partition into connex tests

create a disjoint partition into connex tests. A disjoint partition into connex tests is generated, the number of tests will be as small as possible.

**Return Value:** resulting disjoint partition  
**Parameters:**  
- `sr` — surmise relations between items
13.8 disjpartition* **left_cover_part** (int max_t_size, srbi* sr)

*create a left-covering disjoint partition*

create a left-covering disjoint partition. WARNING: Not implemented! The mathematical solutions are still missing! You have to enter a maximal size of tests, because the trivial solution (one item per test) is always possible.

**Return Value:** resulting disjoint partition  
**Parameters:**  
  - `max_t_size` — maximal number of tests to be created  
  - `sr` — surmise relations between items

13.9 disjpartition* **right_cover_part** (int max_t_size, srbi* sr)

*create a right-covering disjoint partition*

create a right-covering disjoint partition. WARNING: Not implemented! The mathematical solutions are still missing! You have to enter a maximal size of tests, because the trivial solution (one item per test) is always possible.

**Return Value:** resulting disjoint partition  
**Parameters:**  
  - `max_t_size` — maximal number of tests to be created  
  - `sr` — surmise relations between items

13.10 partition* **make_partition** (int q_size, int t_size)

*create a partition*

create a partition. The user can enter which item belongs to each test manually; he is asked for each test, which items he wants to set to the test.

**Return Value:** created partition  
**Parameters:**  
  - `q_size` — number of items  
  - `t_size` — number of tests
Surmise Relations between Tests

Names

14.1 int is_test_sr (partition* p, int test1, int test2, srbi* sr)
   is there a surmise relation between two tests? 66

14.2 int is_test_leftsr (partition* p, int test1, int test2, srbi* sr)
   is there a left-covering surmise relation between two tests? 66

14.3 int is_test_rightsr (partition* p, int test1, int test2, srbi* sr)
   is there a right-covering surmise relation between two tests? 66

14.4 int is_test_totalsr (partition* p, int test1, int test2, srbi* sr)
   is there a total-covering surmise relation? 67

14.5 int is_test_transitive (partition* p, int test1, int test2, int test3, srbi* sr)
   is a group of three tests in a partition transitive? 67

14.6 int is_test_antisymmm (partition* p, int test1, int test2, srbi* sr)
   are two tests antisymmetric? 68

14.7 int is_connex_test (int testnr, partition* p, srbi* sr)
   is a given test in a partition conex? 68

ATTENTION: In all following functions the numbering of tests starts with '0'!
This means, the first test in a partition has the number '0', the last 't_size-1'.

14.1 int is_test_sr (partition* p, int test1, int test2, srbi* sr)
is there a surmise relation between two tests?

Look, if test 1 is in surmise relation with test 2.

Return Value: yes or no (1/0)

Parameters:
- `p` — partition into tests
- `test1` — number of first test
- `test2` — number of second test
- `sr` — surmise relations between items

14.2

```
int is_test_leftsr (partition* p, int test1, int test2, srbi* sr)
```

is there a left-covering surmise relation between two tests?

Look, if test 1 is in left-covering surmise relation with test 2.

Return Value: yes or no (1/0)

Parameters:
- `p` — partition into tests
- `test1` — number of first test
- `test2` — number of second test
- `sr` — surmise relations between items

14.3

```
int is_test_rightsr (partition* p, int test1, int test2, srbi* sr)
```

is there a right-covering surmise relation between two tests?

Look, if test 1 is in right-covering surmise relation with test 2.

Return Value: yes or no (1/0)
Surmise Relations between Tests

Parameters:

\( p \) — partition into tests
\( \text{test1} \) — number of first test
\( \text{test2} \) — number of second test
\( \text{sr} \) — surmise relations between items

14.4

\[
\text{int } \text{is_test_totalsr} (\text{partition}^* p, \text{int } \text{test1}, \text{int } \text{test2}, \text{srbi}^* \text{sr})
\]

is there a total-covering surmise relation?

is there a total-covering surmise relation?. Look, if
the two tests are in total-covering surmise relation, that
means they are in left- and right-covering surmise relation.

Return Value:

yes or no (1 or 0)

Parameters:

\( p \) — partition into tests
\( \text{test1} \) — number of first test
\( \text{test2} \) — number of second test
\( \text{sr} \) — surmise relations between items

14.5

\[
\text{int } \text{is_test_transitive} (\text{partition}^* p, \text{int } \text{test1}, \text{int } \text{test2}, \text{int } \text{test3}, \text{srbi}^* \text{sr})
\]

is a group of three tests in a partition transitive?

is a group of three tests in a partition transitive?. This function looks, if a
group of three tests in a partition is transitive, it is important to give the
tests in the correct order: if \( \text{test1} \ S \ \text{test2} \) and \( \text{test2} \ S \ \text{test3} \) and \( \text{test1} \ S \ \text{test3} \),
the three tests are transitive (\( S \) denotes the surmise relation between tests).

Return Value:

2, if not \( \text{test1} \ S \ \text{test2} \) or \( \text{test2} \ S \ \text{test3} \), 1 if the three tests
are transitive, 0 if they are not, -1 if an error occurred
Parameters: p — partition into tests
test1 — number of first test
test2 — number of second test
test3 — number of third test
sr — surmise relation between items

14.6

int is_test_antisym (partition* p, int test1, int test2, srbi* sr)

are two tests antisymmetric?.

are two tests antisymmetric?.

Look, if two tests in a partition are antisymmetric (meaning: test1 S test2, but test2 not S test1, S meaning the surmise relation between tests).

Return Value: yes or no (1 or 0);

Parameters:
p — partition into tests
test1 — number of first test
test2 — number of second test
sr — surmise relations between items

14.7

int is_connex_test (int testnr, partition* p, srbi* sr)

is a given test in a partition connex?

is a given test in a partition connex?.

Look, if the given test in a partition is connex, which means, that for each item exists a prerequisite in this test or it is a prerequisite of another item.

Return Value: yes or no (1 or 0)

Parameters: testnr — number of test to be investigated
p — partition to be investigated
sr — surmise relations between items
Properties of Partitions

Names

15.1 int is_part_connex (partition* p, srbi* sr)

is a given partition connex? .... 69

15.2 int is_part_leftsr (partition* p, srbi* sr)
are all test in a given partition in a left-covering surmise relation? 70

15.3 int is_part_rightsr (partition* p, srbi* sr)
are all test in a given partition in a right-covering surmise relation?

............................... 70

15.4 int is_part_totalsr (partition* p, srbi* sr)
are all test in a given partition in a total-covering surmise relation?

............................... 70

15.5 int is_transitive_part (partition* p, srbi* sr)
is the partition into tests transitive? ................. 71

15.6 int is_antisyymm_part (partition* p, srbi* sr)
is a partition into tests antisymmetric? ............... 71

15.1 int is_part_connex (partition* p, srbi* sr)

is a given partition connex?. This function investigates, if for each test A in the partition exists another test B so that either A S B or B S A (‘S’ denotes the surmise relation between tests.)

Return Value: yes or no (1 or 0)

Parameters: p — partition into tests to be investigated

sr — surmise relations between items
15.2

```c
int is_part_leftsr (partition* p, srbi* sr)
```

are all test in a given partition in a left-covering surmise relation?

Return Value: yes or no (1 or 0)
Parameters:  
  p — partition into tests to be investigated  
  sr — surmise relations between items

15.3

```c
int is_part_rightsr (partition* p, srbi* sr)
```

are all test in a given partition in a right-covering surmise relation?

Return Value: yes or no (1 or 0)
Parameters:  
  p — partition into tests to be investigated  
  sr — surmise relations between items

15.4

```c
int is_part_totalsr (partition* p, srbi* sr)
```

are all test in a given partition in a total-covering surmise relation?

Return Value: yes or no (1 or 0)
Parameters:  
  p — partition into tests to be investigated  
  sr — surmise relations between items
15.5

```c
int is_transitive_part (partition* p, srbi* sr)
```

is the partition into tests transitive?

is the partition into tests transitive? Look, if transitivity holds for all tests in a partition.

Return Value: yes or no (1/0)

Parameters:
- `p` — partition into tests
- `sr` — surmise relation between items

15.6

```c
int is_antisymm_part (partition* p, srbi* sr)
```

is a partition into tests antisymmetric?

Return Value: yes or no (1 or 0);

Parameters:
- `p` — partition into tests
- `sr` — surmise relation between items
Functions for Equivalence Properties of Tests

Names

16.1 int is_weak_parallel (partition* p, int testA, int testB, srbi* sr)
   are two tests 'weak parallel'?

16.2 int is_leftc_parallel (partition* p, int testA, int testB, srbi* sr)
   are two tests 'left-covering parallel'?

16.3 int is_rightc_parallel (partition* p, int testA, int testB, srbi* sr)
   are two tests 'right-covering parallel'?

16.4 int is_totalc_parallel (partition* p, int testA, int testB, srbi* sr)
   are two tests 'total-covering parallel'?

16.5 int is_equivalent_test (partition* p, int testA, int testB, srbi* sr=NULL)
   are two tests in a partition equivalent?

16.6 int is_equivalent_test (partition* p, int testA, int testB, space* s)
   are two tests in a partition equivalent?

ATTENTION: Better names for the different forms of parallelity must be found.
are two tests 'weak parallel'? Look, if two tests $A$ and $B$ in a partition are 'weak parallel', meaning that $ASB$ and $BSA$ ($S$ denoting the surmise relation between tests).

**Return Value:** yes (1) or no (0).

**Parameters:**
- $p$ — partition into tests
- $testA$ — first test
- $testB$ — second test
- $sr$ — matrix with surmise relations between items

16.2

```c
int is_leftc_parallel (partition* p, int testA, int testB, srbi* sr)
```

are two tests 'left-covering parallel'?

are two tests 'left-covering parallel'? Look, if two tests $A$ and $B$ in a partition are 'left-covering parallel', meaning $A S_l B$ and $B S_l A$ ($S_l$ denoting the left-covering surmise relation).

**Return Value:** yes (1) or no (0).

**Parameters:**
- $p$ — partition into tests
- $testA$ — first test
- $testB$ — second test
- $sr$ — matrix with surmise relations between items

16.3

```c
int is_rightc_parallel (partition* p, int testA, int testB, srbi* sr)
```

are two tests 'right-covering parallel'?

are two tests 'right-covering parallel'? Look, if two tests $A$ and $B$ in a partition are 'right-covering parallel', meaning that
16. Functions for Equivalence Properties of Tests

\( AS, B \) and \( BS, A \) (\( S_r \) denoting the right-covering surmise relation).

**Return Value:** yes (1) or no(0).

**Parameters:**
- \( p \) — partition into tests
- \( \text{testA} \) — first test
- \( \text{testB} \) — second test
- \( sr \) — matrix with surmise relations between items

### 16.4

```
int is_totalc_parallel (partition* p, int testA, int testB,
                       srbi* sr)
```

are two tests 'total-covering parallel'?

### 16.5

```
int is_equivalent_test (partition* p, int testA, int testB,
                        srbi* sr=NULL)
```

are two tests in a partition equivalent?

### 16.4

are two tests 'total-covering parallel'? Look, if two tests A and B in a partition are 'total-covering parallel', meaning that \( AS, B \) and \( BS, A \) (\( S_t \) denoting the total-covering surmise relation).

**Return Value:** yes (1) or no(0).

**Parameters:**
- \( p \) — partition into tests
- \( \text{testA} \) — first test
- \( \text{testB} \) — second test
- \( sr \) — matrix with surmise relations between items

### 16.5

are two tests in a partition equivalent? Look, if two tests in a partition contain the same items or - if the matrix with the surmise relations between items is given - if two test contain equivalent items.

**Return Value:** yes or no (1 or 0), -1 if an error occurred
Parameters:  

- $p$ — partition into tests  
- $testA$ — first test  
- $testB$ — second test  
- $sr$ — matrix with surmise relations between items, default parameter

16.6

```c
int is_equivalent_test (partition* p, int testA, int testB, space* s)
```

are two tests in a partition equivalent?  

The function tests the equivalency of two tests by the following definition: In a given knowledge space exists for each item 'a' in testA an item 'b' in testB so, that the set of states containing item $a$ is equal to the set of all states containing item $b$.  

Return Value:  

- $s$ yes or no (1 or 0), -1 if an error occurred  

Parameters:  

- $p$ — partition into tests  
- $testA$ — first test  
- $testB$ — second test  
- $s$ — knowledge space
**Functions for working with vectors of integer numbers**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>17.1</strong> data_v* new_data_v (int len)</td>
<td>allocate memory for a new data vector of integer numbers</td>
<td>78</td>
</tr>
<tr>
<td><strong>17.2</strong> void free_data_v (data_v* d)</td>
<td>return memory used by a vector to the system</td>
<td>78</td>
</tr>
<tr>
<td><strong>17.3</strong> data_v* copy_data_v (data_v* orig)</td>
<td>make a copy of a vector</td>
<td>78</td>
</tr>
<tr>
<td>void write_data_v (data_v* d)</td>
<td>write a data vector of integer numbers to stdout</td>
<td></td>
</tr>
<tr>
<td><strong>17.4</strong> int is_zero (data_v* d)</td>
<td>are all entries in a data vector equal zero?</td>
<td>79</td>
</tr>
<tr>
<td><strong>17.5</strong> int equal (data_v* d1, data_v* d2)</td>
<td>are two data vectors equal?</td>
<td>79</td>
</tr>
<tr>
<td><strong>17.6</strong> data_v* minus (data_v* d1, data_v* d2)</td>
<td>calculate the coordinatewise difference of two data vectors</td>
<td>79</td>
</tr>
<tr>
<td><strong>17.7</strong> data_v* minus (bitset a, bitset b, int set_size)</td>
<td>calculate the coordinatewise difference of two bitsets</td>
<td>80</td>
</tr>
<tr>
<td><strong>17.8</strong> data_v* sgn_minus (data_v* v1, data_v* d2)</td>
<td>calculate the signum of the coordinatewise difference of two data vectors</td>
<td>80</td>
</tr>
<tr>
<td><strong>17.9</strong> data_v* larger_matrix (data_v* v)</td>
<td>calculate a matrix out of a vector by calculating the product of the vector with himself and the operation 'minus'</td>
<td>80</td>
</tr>
<tr>
<td><strong>17.10</strong> data_v* product (data_v*, data_v*)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
17 Functions for working with vectors of integer numbers

17.11 data_v*  
**product** (bitset a, bitset b, int set_size)

*calculate the coordinatewise product of two bitsets, which are interpreted as '01'-vectors here*

17.12 int  
**plus_number** (data_v* d)

*count the number of positive entries in a given data vector*

17.13 int  
**minus_number** (data_v* d)

*count the number of negative entries in a given data vector*

17.14 int  
**vz_search** (data_v* d)

*look, if the entries of the data vector have different algebraic signs*

17.15 int  
**summ** (data_v* d)

*sum the entries of a vector*

17.16 data_v*  
**bitset2data_v** (bitset b, int set_size)

*transform a bitset to a data vector including the same entries*

17.17 int  
**is_negative_entry** (data_v* s)

*does the data vector include a negative entry?*

17.18 void  
**split_vector** (data_v* s1, data_v* s2, data_v* s, disjpartition* p)

*split a data vector in two vectors*

17.19 data_v**  
**transpose_data_v** (data_v** d, int n)

*transpose a matrix of polytomous answer patterns*

17.20 data_v**  
**count_data_v** (data_v** in_data, int* count, int* num)

*count the number of different data vectors in a structure of 'num' data vectors*

17.1  

**data_v* new_data_v** (int len)
17 Functions for working with vectors of integer numbers

allocate memory for a new data vector of integer numbers

Return Value: pointer to new vector
Parameters: len — length of new vector

void free_data_v (data_v* d)

return memory used by a vector to the system

Parameters: d — pointer to the data vector of integer numbers

data_v* copy_data_v (data_v* orig)

make a copy of a vector

Return Value: pointer to copied vector
Parameters: orig — pointer to original vector
17 Functions for working with vectors of integer numbers

17.4

int is_zero (data_v* d)

are all entries in a data vector equal zero?

Return Value: yes (1) or no (0)
Parameters: d — pointer to data vector

17.5

int equal (data_v* d1, data_v* d2)

are two data vectors equal?

Return Value: yes(1) or no(0)
Parameters: d1 — pointer to first vector
d2 — pointer to second vector

17.6

data_v* minus (data_v* d1, data_v* d2)

calculate the coordinatewise difference of two data vectors

calculate the coordinatewise difference of two data vectors. Entry nr. i of the difference vector is calculated by subtracting entry nr. i of the second vector from entry nr. i of the first vector (d1[i] - d2[i])

Return Value: pointer to difference vector
Parameters: d1 — first data vector
d2 — second data vector, which is subtracted from d1
17 Functions for working with vectors of integer numbers

17.7

\[ \text{data_v* \ minus (bitset a, bitset b, int set_size)} \]

calculate the coordinatewise difference of two bitsets

calculate the coordinatewise difference of two bitsets. The co-
ordinatewise difference of two bitsets is calculated, the result is
written to a data vector (the elements of this vector can be -1,0,1). The second bitset given is subtracted from the first.

Return Value: pointer to difference vector

Parameters:
- \( a \) — first bitset
- \( b \) — second bitset, to be subtracted from \( a \)

17.8

\[ \text{data_v* \ sgn\_minus (data_v* v1, data_v* d2)} \]

calculate the signum of the coordinatewise difference of two data vectors

Return Value: pointer to the signum vector of the difference vector

17.9

\[ \text{data_v* \ larger\_matrix (data_v* v)} \]

calculate a matrix out of a vector by calculating the product of the vector with himself and the operation 'minus'

calculate a matrix out of a vector by calculating the product of the vector with himself and the operation 'minus'. This function calculates a quadratic matrix out of a vector (line number of the matrix = length
17 Functions for working with vectors of integer numbers

of the vector). Call the given vector \( v \), the length of \( v \) \( \text{len} \), the resulting matrix \( m \), then for all \( i, j < \text{len} \): \( m[j][i] = v[i] - v[j] \). The resulting matrix is returned in form of a data vector of length \( (\text{len} \times \text{len}) \).

**Return Value:** pointer to resulting matrix, NULL if an error occurred

**Parameters:** \( v \) — pointer to data vector

17.10

\[
\text{data}_v^* \quad \text{product} \quad (\text{data}_v^*, \text{data}_v^*)
\]

*calculate the coordinatewise product of two data vectors*

**Return Value:** pointer to the resulting product vector of the same length as the input vectors, NULL if an error occurred

17.11

\[
\text{data}_v^* \quad \text{product} \quad (\text{bitset} \ a, \text{bitset} \ b, \text{int} \ \text{set}\_\text{size})
\]

*calculate the coordinatewise product of two bitsets, which are interpreted as '01'-vectors here*

**Return Value:** pointer to the resulting product vector of the same length as the bitsets, NULL if an error occurred

**Parameters:**
- \( a \) — first bitset
- \( b \) — second bitset
- \( \text{set}\_\text{size} \) — length of the bitsets \( a \) and \( b \)
17 Functions for working with vectors of integer numbers

17.12

```c
int plus_number (data_v* d)
```

*count the number of positive entries in a given data vector*

**Return Value:** number of positive entries in the data vector

17.13

```c
int minus_number (data_v* d)
```

*count the number of negative entries in a given data vector*

**Return Value:** number of negative entries in the data vector

17.14

```c
int vz_search (data_v* d)
```

*look, if the entries of the data vector have different algebraic signs*

**Return Value:** yes(1) or no(0), -1 if an error occured.
17 Functions for working with vectors of integer numbers

17.15

int summ (data_v* d)

summ the entries of a vector

Return Value: summed entries

17.16

data_v* bitset2data_v (bitset b, int set_size)

transform a bitset to a data vector including the same entries

Return Value: pointer to data vector including the same entries
Parameters: 
  b — bitset to be transformed
  set_size — length of bitset

17.17

int is_negative_entry (data_v* s)

does the data vector include a negative entry?

Return Value: s -1, if the vector includes a negative entry, +1, if all entries in the vector are positive (or all are zero), 0 if no data vector was given.
17 Functions for working with vectors of integer numbers

17.18

```c
void split_vector (data_v* s1, data_v* s2, data_v* s, disj-partition* p)
```

split a data vector in two vectors

**Parameters:**
- `s` — original data vector
- `p` — has the form of a disjoint partition with 2 tests: which item of the original vector will belong to the first, which item will belong to the second vector?
- `s1` — first part of the original vector, corresponding to the first ‘test’ in `p`
- `s2` — second part of the original vector

17.19

```c
data_v** transpose_data_v (data_v** d, int n)
```

transpose a matrix of polytomous answer patterns

transpose a matrix of polytomous answer patterns. A matrix of polytomous answer patterns is given in form of `n` data vectors of the same length `l`. The result of this function is a pointer to `l` data vectors of length `n`, which correspond to the transposed matrix of the original structure.

**Return Value:**
- `s` pointer to `l` data vectors of length `n` which correspond to the transposed matrix of `d`

**Parameters:**
- `d` — pointer to a structure of `n` data vectors of length `l`
- `n` — number of data vectors
17 Functions for working with vectors of integer numbers

17.20

data_v** count_data_v (data_v** in_data, int* count, int* num)

count the number of different data vectors in a structure of 'num' data vectors

Return Value: s pointer to a set of data vectors, where each vector occurs only once.

Parameters:
in_data — pointer to num data vectors of the same length
count — vector of integer numbers; will include, how often each pattern occurs
Functions for working with Isotonic Probabilistic Models

Names

18.1  int  similar_discord (int* sim, int* dis, data* dat, int* count)
      count the number of similar and discordantly ordered pairs of vectors in a data structure ... 87

18.2  int  similar_discord (int* sim, int* dis, data v** dat, int* count, int num)
      count the number of similar and discordantly ordered pairs of vectors in a set of polytomous response vectors ............... 88

18.3  double pred (data* d, int* count = NULL)
      calculate the Index of Predictability of a data set of dichotomous response variables .................. 88

18.4  double pred (data v** d, int num, int* count)
      calculate the Index of Predictability of a set of data vectors of polytomous response variables ... 88

18.5  double prede (int q, int vpn, int* probabil)
      calculate the Index of Predictability of a set of all possible answer patterns of a given number of items and students .................. 89

18.6  double predset (data* d, dispartition* p, int* count = NULL)
      calculate the Set-Predictability of a criterion and predictor sets of items ......................... 89

18.7  double predset (data v** d, int num, dispartition* p, int* count)
      calculate the Set-Predictability of a criterion and predictor sets of items ......................... 90

18.8  double* w0lit (data* d, int* count = NULL)
Functions for working with Isotonic Probabilistic Models

Direct test of Axiom (W1) of ISOP for all items .......................... 90

18.9 double* wo2vekt (data* d, int* count = NULL)

Direct test of Axiom (W2) of ISOP for all response vectors ...

Currently, each of the functions is written twice, once for the usage with 'traditional' data structures consisting of bitsets and including only dichotomous answer possibilities, and once for a structure on data vectors including also polytomous answer variables.

### 18.1

```c
int similar_discord (int* sim, int* dis, data* dat, int*
                    count)
```

`count the number of similar and discordantly ordered pairs of vectors in a data structure`

**Return Value:** s -1, if an error occurred, 0 else.

**Parameters:**

- `sim` — returns the number of similar pairs
- `dis` — returns the number of discordantly pairs
- `dat` — pointer to data structure
- `count` — vector of integer numbers including how often each answer pattern occurs in the data structure.

### 18.2

```c
int similar_discord (int* sim, int* dis, data_v** dat, int*
                    count, int num)
```

`count the number of similar and discordantly ordered pairs of vectors in a set of polytomous response vectors`
18Functions for working with Isotonic Probabilistic Models

Return Value:  
s -1, if an error occurred, 0 else.

Parameters:  
- `sim` — returns the number of similar pairs
- `dis` — returns the number of discordantly ordered pairs
- `dat` — pointer to a set of polytomous data vectors
- `num` — number of data vectors
- `count` — vector of integer numbers including how often each answer pattern occurs in the data structure.

18.3

```c
double pred (data* d, int* count = NULL)
```

calculate the Index of Predictability of a data set for dichotomous response variables

calculate the Index of Predictability of a data set for dichotomous response variables. The numbers of similar and discordantly ordered pairs of vectors are printed to stdout together with the Indices of Isotonicity, Predictability and the standard deviation of the Index of Predictability.

Return Value:  
s Index of Predictability

Parameters:  
- `d` — data structure
- `count` — integer vector, which includes how often each answer pattern in the data set occurs. This parameter is a default parameter, if no count-vector is given, it is calculated in the function

18.4

```c
double pred (data** d, int num, int* count)
```

calculate the Index of Predictability of a set of data vectors of polytomous response variables

calculate the Index of Predictability of a set of data vectors of polytomous response variables. The numbers of similar and discordantly ordered
pairs of vectors are printed to stdout together with the Indices of Isotonic-
ity, Predictability and the standard deviation of the Index of Predictability.

**Return Value:**

- $s$ index of predictability

**Parameters:**

- $d$ — pointer to a set of data vectors
- $num$ — number of vectors in the data set
- $count$ — integer vector, which includes how often each answer pattern in the data set occurs.

### 18.5

```c
double prede (int q, int vpn, int* probabil)
```

**Calculate the Index of Predictability of a set of all possible answer patterns of a given number of items and students.**

Calculate the Index of Predictability of a set of all possible answer patterns of a given number of items and students. The function calculates the expected frequency of answer patterns out of the given probabilities: Call the number of items $q$, there are $2^q$ possible answer patterns. In the probability vector the user stores the probabilities for a wrong answer to each item. The frequencies of answer patterns are calculated out of this probabilities.

**Return Value:**

- $s$ index of predictability of item pairs

**Parameters:**

- $q$ — number of items
- $vpn$ — number of answer patterns
- $probabil$ — integer vector of probabilities for a wrong answer to each item.

### 18.6

```c
double predset (data* d, disjpartition* p, int* count = NULL)
```

**Calculate the Set-Predictability of a criterion and predictor sets of items.**

This page has been automatically generated with DOC++

DOC++ is ©1995 by Roland Wunderling & Malte Zöckler

January 8, 2002 89
18Functions for working with Isotonic Probabilistic Models

Return Value: s Index of Set-Predictability
Parameters:
- d — data structure
- p — Disjoint partition of items in a criterion and predictor set (corresponds to a disjoint partition with 2 tests
- count — integer vector, which includes how often each answer pattern in the data set occurs. This parameter is a default parameter, if no count-vector is given, it is calculated in the function

18.7

```c
double predset (data_v** d, int num, disjpartition* p, int* count)
```

*calculate the Set-Predictability of a criterion and predictor sets of items*

Return Value: s Index of Set-Predictability
Parameters:
- d — pointer to a set of data vectors
- num — number of data vectors
- p — Disjoint partition of items in a criterion and predictor set (corresponds to a disjoint partition with 2 tests
- count — integer vector, which includes how often each answer pattern in the data set occurs

18.8

```c
double* w0lit (data* d, int* count = NULL)
```

*Direct test of Axiom (W1) of ISOP for all items*

Return Value: s vector with indices w1 for each item
18 Functions for working with Isotonic Probabilistic Models

Parameters: 
- \( d \) — data structure 
- \( \text{count} \) — integer vector, which includes how often each answer pattern in the data set occurs. This parameter is a default parameter, if no count-vector is given, it is calculated in the function

18.9

\[
\text{double}^* \quad \text{wo2vekt} \quad (\text{data}^* \ d, \ \text{int}^* \ \text{count} = \text{NULL})
\]

Direct test of Axiom (W2) of ISOP for all response vectors

Return Value: 
- \( s \) vector with indices \( w2 \) for each response vector

Parameters: 
- \( d \) — data structure 
- \( \text{count} \) — integer vector, which includes how often each answer pattern in the data set occurs. This parameter is a default parameter, if no count-vector is given, it is calculated in the function
Class Graph

3.5 ganter .................................................. 15

9.1 partition .................................................. 40

9.2 disjpartition .............................................. 44

9.3 srbt ........................................................ 47