AALBORG UNIVERSITY

An Environment for Graphical Models

Part III

Xlisp+CoCo — A Tool for Graphical Models

Jens Henrik Badsberg

INSTITUTE OF ELECTRONIC SYSTEMS
DEPARTMENT OF MATHEMATICS AND COMPUTER SCIENCE
Fredrik Bajers Vej 7 — DK 9220 Aalborg — Denmark
Phone: +45 98 15 85 22 — Telefax +45 98 15 81 29

An Environment for Graphical Models

Part III:

Xlisp+CoCo —
A Tool for Graphical Models

Jens Henrik Badsberg

March 1995

A thesis submitted to the Faculty of Technology and Science at Aalborg University for the degree of Doctor of Philosophy.

INSTITUTE FOR ELECTRONIC SYSTEMS
DEPARTMENT OF MATHEMATICS AND COMPUTER SCIENCE
Fredrik Bajers Vej 7 — DK 9220 Aalborg Øst — Denmark
Tel.: +45 98 15 85 22 — TELEX 69 790 aub dk

This book has been produced with LaTeX by the author and has been typeset in Times Roman typeface. The book has been prepared on Sun Sparcstations running SunOS (UNIX) and OpenWindows and it has been printed in PostScript on a Hewlett-Packard LaserJet 4m.

Preface

This guide is about CoCo within XLISP-STAT. The system "Xlisp+CoCo — A Tool for Graphical Models" is obtained by loading CoCo into XLISP-STAT. With this graphical user interface a model selection on graphical models can be performed by mouse interaction with a plot of an independence graph.

The layout of the graph, the association diagram, can be edited by dragging the vertices with the mouse, and the edges will then follow the vertices. New association diagrams can be created from a diagram by adding or dropping edges by the mouse. Tests of two variables conditionally independent can be performed by clicking the edge between the two variables by the mouse. Edges are then drawn with a width proportional to the significance of the edges. Causal models can be handled in the association diagrams. Methods for backward elimination and forward selection of edges in association diagrams are implemented. In the backward elimination of edges the edges are redrawn with a width proportional to the significance of the edges as they are visited. The edge will be drawn with a color depending on whether the edge is fitted or not. If the test for an edge is not computed, then the edge will be drawn with a color depending on whether the edge is fixed, the edge is rejected by coherence, removing the edge will result in a non-decomposable model, etc.

Besides describing the association diagram this guide also introduces a tool for dumping TeX picture-code of a graph, the Model Dynamic Spin Plot and the Model Manager. The Model Dynamic Spin Plot is a spin-plot, where the values in the plot are updated when models are modified. The model search by association diagrams may generate numerous windows with graphs, and to handle those windows the Model Manager has been made. The Model Manager is a *overview graph*, where each point in the graph is a model. Tests can be performed by dragging edges between points in the Model Manager.

The purpose of loading CoCo into XLISP-STAT is besides "Xlisp+CoCo — A Tool for Graphical Models" to be able to do further computations on fitted table values and statistics, plot output from CoCo by high-resolution graphics or, e.g., do a model search by a strategy implemented by the user in Lisp.

Standalone CoCo

CoCo is a program for estimation, test and model search among hierarchical interaction models for large complete contingency tables. The name CoCo is derivated of “Co”mplete “Co”ntingency tables, since the initial program could only handle complete tables, but the program has been enhanced to handle incomplete tables.

CoCo works especially efficiently on *graphical models*, and some of the commands are designed to handle graphical models.

Graphical models are log-linear interaction models for contingency tables that can be represented by a simple undirected graph with as many vertices as the table has dimension. Further all these models can be given an interpretation in terms of conditional independence and the interpretation can be read directly off the graph in the form of a Markov property. The class of graphical model is a proper subclass of the *hierarchical models*, but the class strictly contains the *decomposable models*, e.g., Haberman (1974).

See Darroch, Lauritzen & Speed (1980) for how graphical models are defined by the close connection between the theory of Markov fields and that of log-linear interaction models for contingency tables.

Besides incomplete tables and tables with incomplete observations can be handled and exact tests between any two nested decomposable models computed.

CoCo is a program designed to perform estimation and tests in large contingency tables. By using graph-theoretical results (Rose, Tarjan & Lueker 1976, Tarjan & Yannakakis 1984, Tarjan 1985, Leimer 1993) the hierarchical log-linear interaction models are decomposed. See also chapter 3 of Part I. The IPS-algorithm is not used on the full table, but only on the non-decomposable irreducible components (chapter 2 of Part I). Furthermore, the optimized version of the IPS-algorithm of Jiroušek (1991) is used on these non-decomposable atoms.

If one model is tested against another and the two models have common decompositions, then the test can be partitioned in tests on smaller tables (Goodman 1971). Collapsibility (Asmussen & Edwards 1983) of models and tests is used. If two large sparse models (many factors but few interactions) have no common decompositions, they can be tested against each other by computing the deviance for each model, not by summing over all cells in the full table, but by summing over only cells in sufficient marginal tables (chapter 2 of Part I).

Tests between models with an unlimited number of factors can then be performed on a PC, if the largest clique in the fill-in graphs of non-decomposable atoms of graphs for the models has no more than 14 binary vertices (24 on workstations).

The likelihood ratio test statistic, Pearson's χ^2 test statistic and the power divergence statistics can be computed. In sparse tables an adjusted number of degrees of freedom is computed. Exact tests between any two nested decomposable models can be performed.

Tables with "structural zeros" (Incomplete tables) can be declared, and a modified version of the IPS-algorithm is used on these.

Commands for interactive model search among graphical models are implemented. A semi-automatic model search is possible by backward elimination and forward selection. The model search from Edwards & Havránek (1985) is included in the program.

Besides functions for tests and model search CoCo also has some procedures for model control. The test of one decomposable model against another decomposable model can be factorized into a sequence of tests with one edge (Sundberg 1975, Frydenberg & Lauritzen 1989), and the test between any two submodels can be factorized in tests with one interaction. Functions for producing these factorizations are available in CoCo. CoCo also has a function for computing a wide range of measures of associations on 2-dimensional tables.

Observed counts, estimated counts and probabilities and residual (absolute, adjusted, standard, Freeman-Tukey, etc.) can be listed, plotted pairwise, printed in tables and given an univariate description with mean, variance, median, range, etc.

Finally, to make the use of CoCo somewhat easier and more flexible some data selection is possible and cases with missing values (incomplete observations) can be excluded at various levels in CoCo or the EM-algorithm applied. CoCo can exclude all observations with any variable marked as unobserved, when reading the observations, exclude observations with variables unobserved in a given set, after reading the observations, or exclude observations with relevant variables for a given test marked as unobserved, when performing the test.

CoCo can besides be loaded into XLISP-STAT also be loaded into New S (S-Plus) and XLISP-STAT. Functions for returning statistics and residuals from CoCo, e.g., high-resolution plotting are provided for New S.

The program is originally designed to test methods described in the dissertation Badsberg (1986).

Size of Tables

Tiny tables: 2 and 3 dimensional tables;

In these tables a wide range of measures of associations on the 2-dimensional tables given other variables can be computed.

Small tables: tables with up to between 7 and 10 variables;

On small tables the global model search procedure of Edwards & Havránek (1985) is useful, and will terminate after an acceptable computing time. Also exact tests by Monte Carlo simulation and the EM-algorithm are useful on these tables.

Before any computation all the marginal tables are found to reduce the computing time (unless some cases have unobserved variables).

Medium tables: tables with up to 20 variables;

Any test between two hierarchical models can be performed. The procedures for backward elimination and forward selection of edges in graphical models are useful.

In these tables only sufficient tables of observed counts and tables of estimated probabilities for few tables can be stored internally in the computer.

Large tables: tables with more than 20 variables;

We do not say that all models on large tables are large, e.g., a model with only main effects is called a *small* model. If all the tables of the sufficient marginals of a model cannot fit in the computer memory at one time together with the state spaces of all the non-decomposable components of the model, then the model is *large*. A model is *very large*, if any of the sufficient marginal tables cannot fit in the computer memory. We cannot handle very large models, if we cannot at least store the state spaces of the non-decomposable components of the model one by one. Such models are called *huge* models. Thus, in very large models the largest clique of the fill-in graphs of non-decomposable atoms of the graph of the very large model cannot have more than 14 binary vertices (24 on workstations).

One model can be tested against another by using partitioning and collapsibility of tests, or by computing the deviance for each model by summing over only non-zero cells in the sufficient marginal tables as described in chapter 2 . These tests can be performed between any two large (or very large) models, but if the parts of the test involve tests on large tables, then only the deviance can be computed, and the adjusted degrees of freedom cannot be computed, if the parts involve large models.

The procedures for backward elimination and forward selection of edges in graphical models are still useful on large tables. (Various runs of model selection by respectively forward selection and backward elimination has been performed on the 121 dimensional table with 10.000 cases of Wedelin (1993).)

In large tables the observations are placed as a case list on an internal file.

This crude classification of tables follows the classification of datasets by Huber (1994): Tiny datasets are suitable for black-boards, a small set can be printed on a few pages, a medium set fits on a floppy disk, a large dataset requires a tape, and a huge dataset requires many tapes.

Other programs

Other programs for analysis of log-linear interaction models for contingency tables are DIGRAM (Kreiner 1989) and MIM (Edwards 1989). CoCo finds closed form expression for estimates in decomposable models, handles incomplete tables and tables with incomplete observations, computes exact tests between any two nested decomposable models, handles larger tables and has commands for semi-automatic and automatic search. DIGRAM handles recursive graphical models on contingency tables. MIM also handles continuous variables in mixed interaction models, CG-distributions. Both DIGRAM and MIM runs only on Personal Computers, and are on these machines able to present graphical models as graphs.

Prerequisites and Where to use CoCo

The statistical background assumed for this guide is familiarity with log-linear models. Also some experience with the standalone version of CoCo will be of value. No prior knowledge of Lisp and XLISP-STAT is assumed. At some point, where you want to do more advanced things with Lisp, it might be a good idea to get a hold on (Tierney 1990), which by the way, for a statistician is an excellent book from which to learn Lisp.

The program CoCo is not only useful for working statisticians and other scientists analyzing discrete data by contingency tables, but also in courses teaching analysis of discrete data and graphical models. The guide to CoCo is probably useful in such courses, but the guide to CoCo is not a text-book on contingency tables, and should only be used in courses assisted by text-books such as, e.g., Lauritzen (1982), Whittaker (1990) and Christensen (1990).

Xlisp+CoCo should be able to run at a SUN 3 workstation, but little more than a SPARC-station 1 with 16 Mbytes of RAM would be a good idea.

Using this Tutorial

This guide is the third and last volume of the thesis "A Environment for Graphical Models". Volume 2 of the thesis is "A Guide to CoCo". This second volume will in this guide be referred to as "A Guide to CoCo".

The "Introduction" will show you how to start XLISP-STAT and how to create a plot of an independence graph: a Tutorial Introduction to Association Diagrams and CoCo.

You can then go directly to chapter 9 about the association diagram. For more details on entering data etc. consult part II and "A Guide to CoCo". The chapters 2 to 8 in part II will describe how to enter data from Lisp into CoCo, describe options for tests in model selection in CoCo, describe model selection in CoCo be stepwise edge/interaction selection and elimination and describe the model selection by the EH-procedure.

Part II will only give a very short description of the procedures in CoCo. For further details, see "A Guide to CoCo". Changes to CoCo made since the first edition of "A Guide to CoCo" is described in this guide without stating that they are new features. Some of these changes are around valid data (reals in case lists), factor names longer than one character, printing of tables, initial values for the IPS-algorithm, ordinal tests, lots of test statistics on 2-dimensional tables and both the incremental model selection and the EH-procedure. And since model selection is a main aspect in this guide, a new description of model selection in the standalone version of CoCo is given in this guide.

Part III will give a very detailed description of the association diagram, a programmer's guide. All methods will be mentioned, also those of little interest to the ordinary user. There is a chapter on association diagrams, a chapter on model selection by and in association diagrams, a chapter on causal models and a chapter on extending the association diagrams.

The appendix contains an Installation Guide and a Quick Reference Card. The part of the Quick Reference Card about messages to CoCo objects is ordered as the Quick Reference Card for the standalone version of CoCo.

Availability

The source code in C, executable for Sun 4 (Sparc) and executable of the standalone version of CoCo for PC's and Macintosh for this version of CoCo is available free of charge for non-commercial use.

The source code may only be read and edited for the purpose of porting CoCo to other machines. No new features may be added to CoCo and no parts of the program may be included in other systems or new interface-procedures to New S, S-Plus, XLISP-STAT or any other extendable system may be made without the written permission from the author.

The source code in C and executable for Sun 4 (Sparc), both with Lisp-code for the graphics, and executable of the standalone version of CoCo for PC's and Macintosh can be obtained by anonymous *ftp* over internet from *ftp.iesd.auc.dk*, or by *WWW* from *http://www.iesd.auc.dk/pub/packages/CoCo*. Or CoCo is available on disks (3.5" or 5.25" High density). You should, however, by prepared to bear the costs of copying, e.g., by supplying a disk or tape and a stamped mailing envelope. This guide and the guide to CoCo is also available in TeX and postscript code from the ftp site or printed from Aalborg University.

Disclaimer

CoCo is an experimental program. It has not been extensively tested. The author of CoCo, the University of Aalborg and any other part assisting the author of CoCo in distributing CoCo take no responsibility for losses or damages resulting directly or indirectly from the use of this program.

CoCo is an evolving system. Over the time new features will be introduced, and existing features that do not work may be changed. Every effort will be made to keep CoCo consistent with the information in this guide, but if this is not possible, the help information in CoCo and XLISP-STAT should give accurate information about the current use of a command.

Acknowledgments

Many thanks go to Professor Steffen Lilholt Lauritzen who inspired the creation of CoCo. Also thanks to David Edwards and Svend Kreiner for useful comments and suggestions to new features in CoCo. Thanks to my sister Annette Badsberg for helping with the English language of (earlier versions of) this guide. Thanks to Luke Tierney for making XLISP-STAT available.

Aalborg, Danmark, March 15th 1995,

Jens Henrik Badsberg

Contents

Part I

Introduction

Chapter 1

Introduction

This guide to "LISP-STAT + CoCo" describes the first attempt to implement "Xlisp+CoCo — A Tool for Graphical Models". The current implementation is in the "XLISP-STAT" version of "LISP-STAT".

The purpose of loading CoCo dynamically into XLISP-STAT is besides "Xlisp+CoCo — A Tool for Graphical Models" to be able to do further computations on fitted table values and statistics, plot output from CoCo by high-resolution graphics or, e.g., do a model search by a strategy implemented in Lisp. Data (specification of table and the table of counts) and models can be sent to the CoCo object. Tables and test-values can be returned.

The system "Xlisp+CoCo — A Tool for Graphical Models" is on UNIX systems obtained by loading CoCo dynamically into XLISP-STAT. This feature is only tested on Sparc. When CoCo is loaded into XLISP-STAT the system demands more memory and is somewhat less stable than any of the 2 programs alone, although it is a long time since the author unexpected has crashed the system. If you are, e.g., going to do a lot of computation in XLISP-STAT without using CoCo-functions or you are doing model search in CoCo, run XLISP-STAT or CoCo respectively without linking the two programs together.

The CoCo object file of CoCo can also be linked together with S-Plus. For S-Plus only a subset, the most useful, of the interface procedures is implemented.

1.1 A Tutorial Example

The data of table 1.1 from Reiniš, Pokorný, Bašiká, Tišerová, Goričan, Horáková, Stuchliková, Havránek & Hrabovský (1981) is also used in Edwards & Havránek (1985).

			B	No		Yes	
			A	No	Yes	No	Yes
F	E	D	C				
Negative	<3	<140	No	44	40	112	67
			Yes	129	145	12	23
		>140	No	35	12	80	33
			Yes	109	67	7	9
	>3	<140	No	23	32	70	66
			Yes	50	80	7	13
		>140	No	24	25	73	57
			Yes	51	63	7	16
Positive	<3	<140	No	5	7	21	9
			Yes	9	17	1	4
		>140	No	4	3	11	8
			Yes	14	17	5	2
	>3	<140	No	7	3	14	14
			Yes	9	16	2	3
		>140	No	4	0	13	11
			Yes	5	14	4	4

A, smoking; B, strenuous mental work; C, strenuous physical work;
D, systolic blood pressure; E, ratio of α to β lipoproteins;
F, family anamnesis of coronary heart disease.

Table 1.1: *Risk factors for coronary heart disease.*

Start XLISP-STAT with the script `xlisp+coco`[1] to set up some environment variables needed by CoCo and to load the Lisp functions for interfacing CoCo. To be able to edit and redo entered lines in XLISP-STAT you may want to use `fep`:

```
$ fep xlisp+coco
```

To create a CoCo-object, use the function `(make-coco)`:

```
(def reinis-coco-object (make-coco))
```

By using the function `(def)` instead of `(setf)` the created objects, variables, etc. can be listed by the function `(variables)`.

[1]Make sure that you or your system administrator has done step 4) and 5) of the installation of CoCo, see appendix A, and that XLISP-STAT is available on your system. XLISP-STAT must be compiled with `FOREIGN_FLAG = -DFOREIGNCALL` to make the dynamic loading working. See the ‘Makefile’ for XLISP-STAT.

The specification of the table is then sent to the CoCo-object by the method `:enter-names`:

```
(send reinis-coco-object :enter-names
      '("A" "B" "C" "D" "E" "F")
      '(2 2 2 2 2 2) '(0 0 0 0 0 0))
```

The method `:enter-names` has three arguments: A list with the names, text string, of the variables, a list of integers with the number of unmarked levels for each factor and an optional list of integers with the number of levels to be marked as missing.

The variable n to hold the table of observed counts in a list is defined by:

```
(def n '(
 44  40    112  67    129 145    12  23
 35  12     80  33    109  67     7   9
 23  32     70  66     50  80     7  13
 24  25     73  57     51  63     7  16
  5   7     21   9      9  17     1   4
  4   3     11   8     14  17     5   2
  7   3     14  14      9  16     2   3
  4   0     13  11      5  14     4   4))
```

Instead of `'( ... )` the form `(list ... )` could have been used. But, if the list is long, you will get a stack-error.

The list n containing the table of counts is then sent to the CoCo-object by:

```
(send reinis-coco-object :enter-table n)
```

The following three messages will read the 3 models into the CoCo-objects and create model-objects for these:

```
(def model-1 (send reinis-coco-object :make-model "*"))
(def model-2 (send reinis-coco-object :make-model "ABCF,ACDE"))
(def model-3 (send reinis-coco-object :make-model "ACE,ADE,BC,F"))
```

and graphs for the 1st and 3rd model is created by:

```
(def graph-1 (send model-1 :make-graph
                   :title "Reinis CoCo Graph"))

(def graph-3 (send Model-3 :make-graph :location (list 700 50)
                   :title "Reinis: ACE,ADE,BC,F"))
```

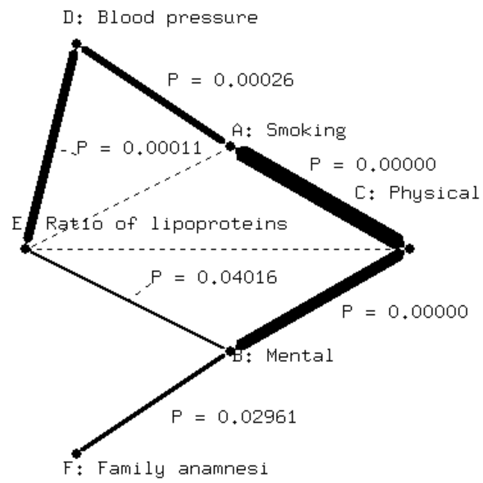


Figure 1.1: The result of a "Headlong Backward" on Graph-1 with added p-values. The bit-mat dump is created by Xlisp+CoCo.

You can now move vertices, set variable labels, drop and add edges etc. with the mouse and perform, e.g., a model search by selecting items from the menu. See chapter 10.

To set labels on the vertices in the 1st graph:

```
(send graph-1 :vertex-label "A" "A: Smoking")
(send graph-1 :vertex-label "B" "B: Mental")
(send graph-1 :vertex-label "C" "C: Physical")
(send graph-1 :vertex-label "D" "D: Blood pressure")
(send graph-1 :vertex-label "E" "E: Ratio of lipoproteins")
(send graph-1 :vertex-label "F" "F: Family anamnesi")
```

and move the vertices in the graph:

```
(send graph-1 :vertex-position "A" (list   0 -20) :redraw nil)
(send graph-1 :vertex-position "B" (list   0  20) :redraw nil)
(send graph-1 :vertex-position "C" (list  35   0) :redraw nil)
(send graph-1 :vertex-position "D" (list -30 -40) :redraw nil)
(send graph-1 :vertex-position "E" (list -40   0) :redraw nil)
(send graph-1 :vertex-position "F" (list -30  40) :redraw T)
```

Change the color on the vertex-labels for all graphs:

```
(send graph-1 :item-color 'vertex-label 'red)
```

Move the vertex-labels for vertex ‘C’ and ‘E’:

```
(send graph-1 :label-position "E" (list -3 -4))
(send graph-1 :label-position "C" (list -11 -2 0) :redraw T)
```

Add control buttons to graph-1:

```
(send graph-1 :add-controls)
```

Let ‘graph-3’ have same vertex-positions as ‘graph-1’:

```
(send graph-3 :slot-value 'positions
      (send graph-1 :slot-value 'positions))
```

A Model Manager is created by:

```
(setf manager (return-manager))
```

The following will create a “Model Dynamic Spin Plot” for ‘graph-1’. The values in the plot are re-evaluated and the plot is redrawn when the graph for the spin-plot receives one of the two messages `:make-graph-current-model` and `:make-graph-base-model`:

```
(let ((a '(send *this-graph* :return-vector 'adjusted "*"
                :model 'current))
      (b '(send *this-graph* :return-vector 'adjusted "*"
                :model 'base))
      (c '(send *this-graph* :return-vector 'observed "*"
                :model nil)))
  (def graph-1-spin
       (send graph-1 :return-dynamic-coco-spin-plot (list a b c)))
  (send graph-1-spin :location 700 500)
  )
```

A ‘Headlong Backward Elimination’ at ‘graph-1’ can be performed by:

```
(send graph-1 :drop-least-significant-edge
      :p-accepted (send graph-1 :graph-p-accepted)
      :p-rejected (send graph-1 :graph-p-rejected)
      :random-order (send graph-1 :graph-random-order)
      :coherent (send graph-1 :graph-coherent)
      :headlong T :recursive T :x-move 0)
```

The “Model Dynamic Spin Plot” and ‘Headlong Backward’ can also be made by selecting the **Recursive backward** and **Dynamic Spin-plot** items from the graph menu.

Or the total example of the tutorial can be performed by the four lines

```
(load (concatenate 'string *xlisp-cocohome*
                            "/Examples/Testgraph"))
(setf manager (return-manager))
(load (concatenate 'string *xlisp-cocohome*
                            "/Examples/TestDynamic"))
(load (concatenate 'string *xlisp-cocohome*
                            "/Examples/TestHeadlong"))
```

Consider also

```
(load (concatenate 'string *xlisp-cocohome*
                            "/Examples/TestBlock"))
(load (concatenate 'string *xlisp-cocohome*
                            "/Examples/TestBootstrap"))
(load (concatenate 'string *xlisp-cocohome*
                            "/Examples/TestTeX"))
(load (concatenate 'string *xlisp-cocohome*
                            "/Examples/Krippendorf"))
```

Part II

XLISP-STAT + CoCo

Chapter 2

CoCo Objects and Messages for Entering of Data

This chapter will describe how to make a CoCo-object, how to enter data into the object, do dataselection, set initial values for the IPS-algorithm and how to handle missing values. Also a short description of how to handle objects in XLISP-STAT is given.

2.1 Creating a CoCo-object

`(make-coco &key (`⟨*n*⟩` 65535) (`⟨*p*⟩` 65535) (`⟨*q*⟩` 1023) (`⟨*title*⟩` nil))`

The function `(make-coco)` will return a *CoCo object*, if the command succeeds:

```
(setf a-coco-object (make-coco))
```

`(make-coco)` may be given the four optional keyword arguments ⟨*n 65535*⟩, ⟨*p 65535*⟩, ⟨*q 1023*⟩ and ⟨*title*⟩ for setting the initial size of the N-array, P-array, Q-array and for setting an optional title on the object respectively. E.g., `(make-coco :n 512 :p 128 :q 16 :title "A CoCo object")`.

Some versions of CoCo, the object file, loaded into XLISP-STAT allow for handling more than one CoCo object in the same XLISP-STAT session. At the same time in XLISP-STAT, CoCo can then be used on different datasets. To test whether the version of Xlisp+CoCo you a running is able to handle more than one CoCo-object at the same time, simply create two CoCo-objects by the function `(make-coco)` and see if there is an error-message.

The result returned by `(make-coco)` is an XLISP-STAT *object*. To use an object, you have to send it *messages*. This is done by using the `(send)` function, as in the expression

`(send` ⟨*object*⟩ ⟨*message selector*⟩ ⟨*argument a*⟩ ⟨*argument b*⟩ `... )`

2.2 Sending the Specification of a Table to a CoCo-Object

`:enter-names` ⟨*list of names*⟩ ⟨*list of number of levels*⟩ `&optional`
(⟨*list of number of levels to be marked as missing*⟩ `nil`)

The expression

```
(send a-coco-object :enter-names '("A" "B") '(2 2))
```

sends a specification of a table to the CoCo object. The ⟨*message selector*⟩ is `:enter-names`, the lists `'("A" "B")` and the list `'(2 2)` are arguments. The message consists of the *selector* and the *arguments*. Message selectors are always Lisp keywords; that is, they are symbols that begin with a colon.

The message `:enter-names` has three arguments:

`:enter-names` ⟨*list of names*⟩ ⟨*list of number of levels*⟩ `&optional`
(⟨*list of number of levels to be marked as missing*⟩ `nil`)

The argument ⟨*names*⟩ is a list with the factor names: a text-string for each factor. If some of the factor names is longer then one character, then <u>all</u> the factor names should start with the character `:`. E.g.,

```
(send a-coco-object :enter-names '(":Smoking" ":Sex") '(2 2))
```

The argument ⟨*levels*⟩ is a vector of integers giving the number of unmarked levels for each factor. Finally, the *optional argument* ⟨*missing-levels*⟩ is an optional vector of integers giving the number of levels to be marked as missing. See "A Guide to CoCo" for details about the specification and the number of levels to be marked as missing.

The message `:enter-names` has no *keyword arguments*. When a message has keyword arguments and keyword arguments are given, then the argument values have to be preceded by the keywords, see, e.g., the function `(make-coco)` and the method `:enter-list`. If a method have both optional arguments and keyword arguments, then all the optional arguments has to be given, if some of the keyword arguments are given.

If the message `:enter-names` is successful, then the value `TRUE` is returned, else `nil` is returned and an error flag is printed. This is the convention for messages to CoCo-objects not returning values[1]: If the message is successful, the value `TRUE` is returned, else `nil` is returned and the error flag is printed. For messages expected to return values, the value is returned, if the message is successful, else `nil` is returned and an error flag is printed. For messages returning booleans, the value `nil` is also returned for the value `FALSE`. The error flag is besides printed also pushed into the list `*coco-errors*`.

[1] Instead it could have been chosen to return `nil`, if the message was successful, else the error-message.

2.3 Declaring Ordinal Variables

`:set-ordinal` ⟨*set*⟩

With this message the subset ⟨*set*⟩ of the factors is declared to be ordinal. Then when two factors are tested conditionally independent Goodman and Kruskal's Gamma coefficient is also computed. The argument ⟨*set*⟩ has to be a text-string with the names of the ordinal factors.

2.4 Sending Observations as a Table of Counts to a CoCo-Object

A table of counts is entered by the message

`:enter-table` ⟨*counts: list of integers*⟩

The argument ⟨*table*⟩ is a vector of integers giving the cell counts ordered as when reading a table of counts into CoCo in the normal set. The first declared factor is alternating the fastest. If `:enter-table` is successful, then the value `TRUE` is returned, else `nil` is returned and an error flag is printed.

(If the message `:set-read 'subset` ⟨*subset*⟩ has been used or data selection has been done between the messages `:enter-names` and `:enter-table`, the fully declared table still has to be entered.)

The count `-1` can be entered for structural zero cells, see also section 2.10.

2.5 Sending Observations as a List of Cases to the CoCo-Object

`:enter-list` ⟨*cases: list of integers*⟩ `&key (`⟨*accumulated*⟩ `nil)`
`(`⟨*ncol*⟩ `nil) (`⟨*select-case-fun*⟩ `nil set) (`⟨*columns*⟩ `nil subset)`
⟨*select-case-fun*⟩ ⟶ { `function on a case (list of integers):`
`returning the value 'True' for cases to be selected.` }

This message is used to enter a list of cases into the CoCo object. The list ⟨*cases: list of integers*⟩ is a list of integers. The list contains the cases: for each case without separators the level of each declared factor.

Please note that the number of variables entered must be the number of variables declared in the CoCo-object (plus one, if an accumulated list is entered); When the arguments ⟨*columns*⟩ or ⟨*accumulated*⟩ are not given, the length of the case list must be the number of cases times the number of declared factors, also when the message `:set-read 'subset` ⟨*subset*⟩ has been used.

When the keyword argument ⟨*columns*⟩ is given, the case list is split into lists of length ⟨*ncol*⟩, and the values are selected form each sublist according to the argument ⟨*columns*⟩. The argument ⟨*columns*⟩ has to be a list with the indexes of the values to be selected. For the method `:enter-list` to be able to split the case list, when the keyword argument ⟨*columns*⟩ is given, the keyword argument ⟨*ncol*⟩ also has to be given. The number of variables selected by ⟨*columns*⟩ must be the number of variables declared in the CoCo-object (plus one, if an accumulated list is entered).

If ⟨*accumulated*⟩ is `TRUE`, then an accumulated list is expected and the number of values for each case must be one greater than the number of declared factors in the CoCo-object, and the first value for each case must be the count. If keyword argument ⟨*columns*⟩ is given together with ⟨*accumulated*⟩, then the first selected column must be the case count, and the number of selected variables must be one greater than the number of declared variables in the CoCo-object.

The keyword argument ⟨*select-case-fun*⟩ has to be a function taking a list of length ⟨*ncol*⟩ as argument and returning `TRUE`, if the case is to be entered into the CoCo-object. The function ⟨*select-case-fun*⟩ is used on each case before selecting variables, columns.

The message `:enter-names-and-list` will declare factors, enter cases and select a subset of the cases and of the factors, all by this one message. See a later section.

Factors can be recoded during entering the list of cases by the message `:cutpoints`, and cases selected or rejected by the messages `:select-cases` and `:reject-cases`.

But selection and recoding are done more elegantly and more generally in Lisp. See the following section. (Selection and recoding is faster in CoCo.)

2.5.1 Selecting Cases from the List and/or Entering only a Subset of the Columns from the Case List

Consider the following variables in the "The Glostrup Study":

F: FEV, Forced Ejection Volume - lung function; continuous

K: Cholesterol; continuous

H: Hypertension; binary (no/yes)

B: BMI, Body Mass Index; continuous, Normal: 20.0-24.9, Moderately obese: 25.0-29.9, Severe obese: 30.0+

S: Smoking; binary (not smoking/smoking)

A: Alcohol consumption; binary (seldom/frequently)

W: Work; binary (working/not working)

G: Gender; binary (male/female)

Y: Cohort - survey year; binary (1967/1984)

The observations of the 9 variables for 1082 cases are put into the list `ksl` in Lisp by the following code[2]:

```
(def ksl '(
   252  680 2    3237  2  2  2  1  2
   205  669 2    2671  2  1  2  2  2
    66  636 2    2260  2  1  2  2  2
   255  669 2    2486  2  1  2  1  2
   135  555 1    3227  2  1  2  2  2
   314  475 2    2191  1  1  2  1  2
  9999  825 1    2439  2 -1  2  2  2
   173  634 2   99999  1 -1  2  2  2
   235  657 1    2395  3  2  2  1  2
   111  619 1    1890  3  2  1  2  2

;  << 1070 lines deleted >>

   180  853 1    2962  2  1  2  2  1
   120 1008 1    2272  1  1  2  2  1 ))
```

To show how missing values are coded, some of the observations have been changed to missing values: `9999` for 'FEV', `99999` for 'BMI', `3` for 'Smoking' and `-1` for 'Alcohol'. Use `'-1'` or some large value for missing values. The characters `*` and `.` cannot be used as missing values in XLISP-STAT.

From the list `ksl` an array `ksl-array` is created and the columns with the three continuous variables 'FEV', 'Cholesterol' and 'BMI' are extracted into the lists `f`, `k` and `b`:

```
(setf ksl-array (make-array '(1082 9)
                            :initial-contents (list ksl)))
(def f (coerce (element-seq (select ksl-array T 0)) 'list))
(def k (coerce (element-seq (select ksl-array T 1)) 'list))
(def b (coerce (element-seq (select ksl-array T 3)) 'list))
```

A case list with recoded values for the three continuous variables added is then created:

```
(def ksl-new
     (mapcar #'(lambda (case f1 k1 b1)
                 (concatenate 'list case (list f1 k1 b1)))
             (split-list ksl 9)
             (mapcar #'(lambda (i)
                         (length
```

[2]Instead of `'( ... )` the form `(list ... )` could have been used. But, if the list is long, you will get a stack-error.

```
                    (which (> i (list 0 150 200 999)))))
              f)
      (mapcar #'(lambda (i)
                   (length
                    (which (> i (list 0 600 750 1999)))))
              k)
      (mapcar #'(lambda (i)
                   (length
                    (which
                     (>= i
                         (list 0 2000 2500 3000 4999)))))
              b)))
```

'FEV' is recoded into four groups: below 150 (150 included), between 150 and 200 (200 included), above 200 and a group for missing values. 'Cholesterol' is also recoded into four groups: below 600 (600 included), between 600 and 750 (750 included), above 750 and a group for missing values. 'BMI' is recoded into 5 groups: below 2000 (2000 not included, Below normal), between 2000 and 2500 (2500 not included, Normal), between 2500 and 3000 (3000 not included, Moderately obese), above 3000 (Severe obese) and a group for missing values.

A CoCo-object is then created and the specification of the factors is entered into the object by[3]:

```
(setf ksl-coco-object (make-coco))
(send ksl-coco-object :enter-names
      '("F" "K" "H" "B" "S" "A" "W" "G" "Y")
      '(3 3 2 4 2 2 2 2 2) '(1 1 0 1 1 1 0 0 0))
```

or

```
(setf ksl-coco-object (make-coco))
(send ksl-coco-object :enter-names
      '(":FEV" ":Cholesterol" ":Hypertension" ":BMI"
        ":Smoking" ":Alcohol" ":Work" ":Gender" ":Year")
      '(3 3 2 4 2 2 2 2 2) '(1 1 0 1 1 1 0 0 0))
```

The recoded case list is then entered by:

```
(send ksl-coco-object :enter-list (element-seq ksl-new)
      :ncol 12
      :columns '(9 10 2 11 4 5 6 7 8))
```

[3] At the time of writing this, the list entered by `:enter-list` should be a list of integers. In the standalone version a list of reals is permitted

With the three recoded variables added, each case has 12 variables. The number of variables entered must be the number of variables declared in the CoCo-object (plus one, if an accumulated list is entered). In this situation the 9 relevant variables are selected by `:columns '(9 10 2 11 4 5 6 7 8)`. The numbers in the list given by the keyword argument ⟨*columns*⟩ are the indexes, the column-number for the columns to be selected minus 1. For the method `:enter-list` to be able to split the case list, when the keyword argument ⟨*columns*⟩ is given, the keyword argument ⟨*ncol*⟩ also has to be given.

Cases with the coded value for 'Work' (the 7th variable) equal the coded value for 'Year' (the 9th variable), `'(= (nth 6 case) (nth 8 case))'`, and with the coded value for 'Alcohol' (the 6th variable) different from the coded value for 'Gender' (the 8the variable), `'(not (= (nth 5 case) (nth 7 case)))'`, are selected by:

```
(flet ((x-select (case)
                 (and (= (nth 6 case) (nth 8 case))
                      (not (= (nth 5 case) (nth 7 case))))))
      (send ksl-coco-object :enter-list (element-seq ksl-new)
            :accumulated nil
            :ncol 12
            :select-case-fun #'x-select
            :columns '(9 10 2 11 4 5 6 7 8)))
```

2.6 Read a Subset of Declared Factors

`:set-read` ⟨*code*⟩ `&optional` ⟨*subset*⟩
    ⟨*argument*⟩ ⟶ { `'all` | `'subset` ⟨*string*⟩ }

The message `:set-read 'subset` ⟨*subset*⟩ is used between `:enter-names` (evt. `:read-names` or `:read-factors`) and `:enter-table` or `:enter-list` (evt. `:read-table` or `:read-list`) to select only a subset of the declared factors. Observations for all declared factors still have to be entered, so the same data can be reused without changes. See "A Guide to CoCo" for details.

2.7 Sending the Specification, the Observations as a List and do Case-Selection by only One Message

`:enter-names-and-list` ⟨*case-list*⟩ ⟨*names*⟩ ⟨*levels*⟩
  `&key` (⟨*missing levels*⟩ `nil`) (⟨*select-case-fun*⟩ `nil set`)
  (⟨*columns*⟩ `nil subset`)

This message will declare factors and enter a case list while select a subset of the factors and a subset of the cases. The method is implemented in XLISP-STAT code with the use of the methods `:enter-names` and `:enter-list`. Thus one more method is added without much additional functionality, and making the system harder to learn to use. But anyway, the method might become handy, e.g., for split models.

Each case in the case list must consist of the variables declared in the list of names. Only the variables selected by ⟨*columns*⟩ are declared in the CoCo-object and only observation of these variables are read into the CoCo-object. Thus at this message the length of the list given as the keyword argument ⟨*columns*⟩ can be shorter than the length of the list of names, whereas at the message `:enter-list` the length of the list given as the keyword argument ⟨*columns*⟩ should be equal to the number of declared factors in the CoCo-object (plus one, if an accumulated list is entered).

The specification and observations from the previous section is then entered by:

```
(flet ((x-select (case)
                 (and (= (nth 6 case) (nth 8 case))
                      (not (= (nth 5 case) (nth 7 case))))))
      (send ksl-coco-object :enter-names-and-list
            (element-seq ksl-new)
            '("F" "K" "H" "B" "S" "A" "W" "G" "Y" "f" "k" "h")
            '(0 0 2 0 2 2 2 2 2 3 3 4)
            :missing-levels '(1 1 0 1 1 1 0 0 0 1 1 1)
            :select-case-fun #'x-select
            :columns '(9 10 2 11 4 5 6 7 8)))
```

The function ⟨*select-case-fun*⟩ is used on each case before selecting variables.

If the keyword argument ⟨*accumulated*⟩ is set to `TRUE`, then the selected sublist of variables is expected to be an accumulated list, the first integer for each 'case' is the number of cases with that configuration of variables. If the keyword argument ⟨*columns*⟩ is given together with ⟨*accumulated*⟩, then the first selected column must be the count, else, if ⟨*accumulated*⟩ is given without ⟨*columns*⟩, then the first variable must be the count, and this variable is not entered into the CoCo-object.

2.8 Returning Specification and Data

`:return-name-list &key (`⟨*full*⟩` nil)`
`:return-level-list &key (`⟨*full*⟩` nil)`
`:return-missing-list &key (`⟨*full*⟩` nil)`
`:return-names &key (`⟨*full*⟩` nil)`

These messages will return the specification of the table in the CoCo-object. If the keyword argument ⟨*full*⟩ is set to `TRUE`, then the full specification is returned ignoring a `:set-read 'subset` ⟨*subset*⟩ message.

The method `:return-names` is used to return the specification of the table in the format used in the association diagram.

2.9 Replacing Observations with a Random Data Set

`:substitute`

This message will replace the table of observed counts with a random table of counts, where the sufficient marginals for the **current** model are unchanged.

2.10 Initial Values for the IPS-Algorithm and Structural Zeros

`:enter-q-table` ⟨*set*⟩ ⟨*table: list of integers*⟩
`:enter-q-list` ⟨*set*⟩ ⟨*cells: list of integers*⟩

Initial values (integers) to the IPS-algorithm are entered by `:enter-q-table`[4]. Zero, `0`, is entered for cells to be zero by structure. Cells to be zero by structure can also be entered by the method `:enter-q-list`. See also "A Guide to CoCo".

Remove Cases in Cells Zero by Structure

`:clean-data`

This message will remove cases from cells to be zero by structure. See also "A Guide to CoCo".

2.11 Old Functions for Recoding Variables and Case Selection

The messages in this section is implemented for comparability with the standalone version of CoCo.

The selection of cases and recoding of variables are done more elegantly and more generally in XLISP-STAT although the selection done by the messages in this section might be faster.

[4]At the time of writing this guide, the list entered by `:enter-q-table` should be a list of integers. This should not be much of a restriction, since the table can just be multiplied with a factor.

2.11.1 Declaring Cutpoints for Variables in the List of Cases

`:redefine-factor` ⟨*name*⟩ ⟨*levels*⟩ ⟨*missing levels*⟩
`:cutpoints` ⟨*name*⟩ ⟨*cutpoints*⟩

The message `:cutpoints` will declare cutpoints for variables to be entered by `:enter-list`[5] and for variables to be read from a file, see later sections. The argument ⟨*name*⟩ is the name of the factor for which to enter cutpoints. The argument ⟨*cutpoints*⟩ is a list with the cutpoints. The cutpoints entered by this method must be integers. The length of the list must be one less than the total number of levels declared for the factor ⟨*name*⟩. The total number of levels is the sum of the number of unmarked levels (⟨*levels*⟩) and the number of marked levels (⟨*missing*⟩). The message `:redefine-factor` can be used to redefine the number of levels for a declared factor. See also "A Guide to CoCo".

2.11.2 Declaring Data Selection for the List of Cases

`:select-cases` ⟨*set*⟩ `&optional` ⟨*cell*⟩
`:or-select-cases` ⟨*set*⟩ `&optional` ⟨*cell*⟩
`:reject-cases` ⟨*set*⟩ `&optional` ⟨*cell*⟩
`:or-reject-cases` ⟨*set*⟩ `&optional` ⟨*cell*⟩

With the messages `:select-cases` and `:or-select-cases` between entering the specification of the table and entering the observations only cases in particular cells in one or more marginal tables are entered. The message `:select-cases` disposes of previous select-statements. Use the message `:select-cases "."` to select all cases not rejected.

Cases in particular cells can be skipped with the messages `:reject-cases` and `:or-reject-cases`. As with the message `:select-cases` the message `:reject-cases` disposes of previous reject-statements. All cases selected are entered after the message `:reject-cases "."`.

Note that selection will result in zero cells in the entered data, if the factors done selection on is entered, i.e., if not the method `:set-read` is used. Cases with a particular level at a non-binary factor can be excluded without making zero cells by entering a case-list with the factor twice, doing selection on one copy of the factor, and grouping the other copy of the factor by the message `:cutpoints`. If a factor is recoded and selection is done on the factor, then the selection is done on the result of the recoding, i.e., result of the of the messages `:redefine-factor` and `:cutpoints`.

See also "A Guide to CoCo".

Observations entered previously in this chapter can then be entered by:

[5] At the time of writing this, the list entered by `:enter-list` should be a list of integers. Case lists with reals can be read from a file by `:read-list`.

```
(setf ksl-coco-object (make-coco))
(send ksl-coco-object :enter-names
      '("F" "K" "H" "B" "S" "A" "W" "G" "Y")
      '(3 3 2 4 2 2 2 2 2) '(1 1 0 1 1 1 0 0 0))

(send ksl-coco-object :select-cases "YW" '(1 1))
(send ksl-coco-object :or-select-cases "YW" '(2 2))
(send ksl-coco-object :reject-cases "GA" '(1 1))
(send ksl-coco-object :or-reject-cases "GA" '(2 2))

(send ksl-coco-object :cutpoints "F" '( 150 200 999))
(send ksl-coco-object :cutpoints "K" '( 600 750 1999))
(send ksl-coco-object :cutpoints "B" '( 2001 2501 3001 4999))

(send ksl-coco-object :enter-list ksl)
```

2.12 Missing Values

`:skip-missing`
`:exclude-missing &optional (`⟨*code*⟩ `'flop) (`⟨*set*⟩ `";")`
    ⟨*argument*⟩ ⟶ { `'on` | `'off` | `'in` ⟨*string*⟩ }
`:em-on`

Missing values might in CoCo be handled in one of three ways:

- Skip all cases with missing values.
  This is selected by the message `:skip-missing`. Cases with missing values among the variables to be read (`:set-read 'subset` ⟨*subset*⟩) are then skipped during the reading of cases (by `:enter-list` or when reading cases from a file).

- Use all cases without missing values for relevant factors.
  After the message `:exclude-missing 'on` all cases without missing values for relevant factors are used in each test. This is turned off again by the message `:exclude-missing 'off` and all cases are used. The message `:exclude-missing 'in` ⟨*string*⟩ will exclude cases with missing values for some of the variables in the set ⟨*string*⟩ of factors. The method is used after entering data.

- Use the EM-algorithm to estimate the values of unobserved factors.
  This is selected by the message `:em-on` after entering data. The current implementation is tested on Fuchs (1982) for situations, where factors at random are unobserved, and on Dawid & Skene (1979) with a latent variable, i.e., a variable unobserved for all cases.

See "A Guide to CoCo" for details.

2.13 Selecting Data Structure

`:set-datastructure &optional (`⟨*code*⟩` 'all)`

    ⟨*code*⟩ ⟶ { `'all` | `'necessary` | `'file` }

This message will select the data structure used in CoCo. Do not worry about selecting data structure for CoCo, since CoCo has been become quite good at doing this. See “A Guide to CoCo” for details.

2.14 Reading CoCo-data-files

By the messages in this section specification and observations can be read from files.

Input from Keyboard or File

`:set-switch 'keyboard &optional (`⟨*hit*⟩` 'flop)`
`:set-specification-file` ⟨*file-name*⟩
`:set-observations-file` ⟨*file-name*⟩
`:set-data-file` ⟨*file-name*⟩

With the message `:set-switch 'keyboard` one can choose to read specification and observations from keyboards (standard input) when the messages of the following subsection are used. The three messages `:set-data-file`, `:set-specification-file` and `:set-observations-file` will name and rewind data-file (file with specification and file with observations). See “A Guide to CoCo” for details.

Read Data

```
:read-data
:read-specification
:read-factors
:read-names
:read-observations
:read-table
:read-list
```

These messages will read data from the data-file. See “A Guide to CoCo” for details. If some of the factor-names are longer than one character then all names in the specification should start with colon.

The methods are kept for comparability with the standalone version of CoCo, and because large datasets might be read faster from files. To reduce the number of commands, maybe only the method `:read-data` should been have retained.

Chapter 3

Models, Fitted Values and Tests in CoCo Objects

This chapter will describe how to enter models into the CoCo-object, handle the models in the model-list in the CoCo-object, return values computed under the models and perform tests between models in the CoCo-object. Also a message for computing measures of associations is described.

3.1 Models

First we will describe how to enter models into the CoCo-object, collaps the models, transform models from and to the dual representation, make association diagrams for models, move pointers to models in the model-list of the CoCo-object, and return models, edge lists, model numbers and characteristics of the models, test whether a model is a submodel of another model and test whether the model resulting from removing one edge from a model is decomposable, print models with adjacency matrix, dispose of models, print the model formula and a running intersection property ordering.

3.1.1 Read Model

`:read-model` ⟨*gc*⟩
`:read-n-interactions` ⟨*order*⟩ ⟨*set*⟩

The message

`(send` ⟨*CoCo-object*⟩ `:read-model` ⟨*a*⟩`)`

will add the hierarchical model ⟨*a*⟩ to the list of models in the CoCo object ⟨*CoCo-object*⟩.

The argument ⟨*a*⟩ has to be a single character string with the model written as models are written in CoCo: A list of generators, the maximal interaction terms, separated by comma, `,`, and terminated by period, `.`, e.g.:

(send ⟨*CoCo-object*⟩ :read-model "ACE,ADE,BC,F.")

or the form with `[` and `]` can be used:

(send ⟨*CoCo-object*⟩ :read-model "[[ACE][ADE][BC][F]]")

This is how generating classes is entered from XLISP-STAT to CoCo.

The *saturated model* can be abbreviated to `*`:

(send ⟨*CoCo-object*⟩ :read-model "*")

and the model with only *main effects* can be abbreviated to `.`:

(send ⟨*CoCo-object*⟩ :read-model ".")

The *uniform model* is entered by the empty set:

(send ⟨*CoCo-object*⟩ :read-model "[[]]")

If some of the factor names consist of more than one character, then the names should begin with (be separated by) `:`, so, e.g., models are entered by:

(send ⟨*CoCo-object*⟩ :read-model ":Age:Sex:Smoking.")
(send ⟨*CoCo-object*⟩ :read-model ":Age:Sex,:Sex:Smoking.")
(send ⟨*CoCo-object*⟩ :read-model "[[:Age:Sex][:Sex:Smoking]]")

Editing Models

:collaps-model ⟨*set*⟩
:normal-to-dual &optional (⟨*only*⟩ nil)
:dual-to-normal &optional (⟨*only*⟩ nil)

Messages for collapsing the **current** model onto the smallest model containing the subset ⟨*set*⟩ of the factors and for inserting the dual representation of the **current** model or the model for which the **current** model is the dual representation into the model list. See "A Guide to CoCo" for details.

Make Model or Graph for Model

:make-model &optional (⟨*model*⟩ 'current) &key (⟨*title*⟩ nil)
⟨*model*⟩ ⟶ { ⟨*gc*⟩ | 'current | 'base | 'last | ⟨*integer*⟩ }

:make-graph &key (⟨*model*⟩ 'current) (⟨*location*⟩ nil) (⟨*size*⟩ nil) (⟨*title*⟩ nil)
⟨*model*⟩ ⟶ { ⟨*gc*⟩ | 'current | 'base | 'last | ⟨*integer*⟩ }

The method `:make-model` will return a model object, see chapter 8. The returned model object is for the **current** model, if the optional argument ⟨*model*⟩

is not given to the message. The optional argument ⟨*model*⟩ can be given the value `'base` or `'last` to return the model object of the **base** model or the **last** model respectively, or an integer may be given as argument to get the model object of the model with that number, or finally, if a text-string with the generating class of a model is given as argument, then this model is read into the CoCo object and a model object of that model is returned. Analogously will `:make-graph` return an association diagram for the **current** model, see chapter 10. The argument ⟨*model*⟩ is here a keyword argument.

3.1.2 The Model-list

Moving Pointers in the Model-List

`:base`
`:current`
`:make-base` ⟨*no*⟩
  ⟨*no*⟩ ⟶ { ⟨*integer*⟩ | `'previous` | `'next` | `'current` | `'last` }
`:make-current` ⟨*no*⟩
  ⟨*no*⟩ ⟶ { ⟨*integer*⟩ | `'previous` | `'next` | `'base` | `'last` }

The message `:base` makes the **current** model the **base** model. `:current` makes the **last** model the **current** model. The messages `:make-current` ⟨*a*⟩ and `:make-base` ⟨*a*⟩ will make the model with number ⟨*a*⟩ the **current** or the **base** model respectively in CoCo, if the argument ⟨*a*⟩ is an integer.

Also the keywords `'previous`, `'next`, `'current`, `'base` and `'last` can be given as argument to the methods `:make-current` and `:make-base`. The message `:make-current 'last` is then the same as `:current` and the message `:make-base 'current` is the same as `:base`. The message `:make-current 'previous` will move the **current** pointer to the previous model, one model back in the model-list (independent of model numbers). Analogously with `:make-current 'next`. Because of the orientation of the model list the message `:make-current 'previous` is faster than `:make-current 'next`.

These four messages return the value `TRUE`, if the commands successful, else `nil`.

Returning the Number Model to Lisp

`:return-model-number` ⟨*model*⟩
  ⟨*model*⟩ ⟶ { `'current` | `'base` | `'last` }

The three messages `:return-model-number 'current`, `:return-model-number 'base` and `:return-model-number 'last` will return the model number of the **current** model, the **base** model and **last** model respectively, if the command succeeds, else `nil`.

Returning the Model to Lisp

`:return-model` ⟨*model*⟩ `&optional (`⟨*number*⟩ `nil)`
  ⟨*model*⟩ ⟶ { `'current` | `'base` | `'last` | `'number` ⟨*integer*⟩ }

`:return-model-set` ⟨*model*⟩ `&optional (`⟨*number*⟩ `nil)`
  ⟨*model*⟩ ⟶ { `'current` | `'base` | `'last` | `'number` ⟨*integer*⟩ }

`:return-edge-list` ⟨*model*⟩ `&key (`⟨*edges*⟩ `'in-model) (`⟨*fix*⟩ `'all-edges)`
  ⟨*edges*⟩ ⟶ { `'in-model` | `'not-in-model` | `'all-edges` }
  ⟨*fix*⟩ ⟶ { `'not-fix-edges` | `'fix-edges` | `'all-edges` }
  ⟨*model*⟩ ⟶ { `'current` | `'base` | `'last` | ⟨*integer*⟩ }

The message `:return-model 'current` returns a text string with the generating class for the **current** model, if the command succeeds, else `nil`. Analogously with `:return-model 'base`, `:return-model 'last` and `:return-model 'number` ⟨*number*⟩.

The message `:return-model-set` will return the factors in the model and `:return-edge-list` will return a list with the edges of the model. The returned list of edges is in the format used in the association diagram. With the arguments one can choose to return the edges in the model, the edges not in the model, or all edges in the saturated model with factors as the model. Fixing of edges can be ignored, or the returned edges restricted to edges not fixed or fixed edges.

Returning Characteristics of Model

`:is-graphical &optional (`⟨*model*⟩ `'current)`
  ⟨*model*⟩ ⟶ { ⟨*gc*⟩ | `'current` | `'base` | `'last` | ⟨*integer*⟩ }
`:is-decomposable &optional (`⟨*model*⟩ `'current)`
  ⟨*model*⟩ ⟶ { ⟨*gc*⟩ | `'current` | `'base` | `'last` | ⟨*integer*⟩ }
`:is-submodel-of &optional (`⟨*model-1*⟩ `'current) (`⟨*model-2*⟩ `'base)`
  ⟨*model-1*⟩`,` ⟨*model-2*⟩ ⟶ { ⟨*gc*⟩ | `'current` | `'base` | `'last`
   | ⟨*integer*⟩ }
`:is-in-one-clique` ⟨*edge*⟩ `&optional (`⟨*model*⟩ `'current)`
  ⟨*model*⟩ ⟶ { ⟨*gc*⟩ | `'current` | `'base` | `'last` | ⟨*integer*⟩ }

The two message `:is-graphical` will return the value `TRUE`, if the relevant model is *graphical*, else `nil`. The model is graphical, is the *cliques* of the *2-section graph* are the generating class of the model. In graph theory, the corresponding idea is that of a *conformal* graph. By default, the test is performed on the **current** model, but the argument ⟨*model*⟩ can be given the values `'base` or `'last` for returning the value for the **base** model or the **last** model, or an integer may be given as argument to get the value for the model with that number, or finally, a text-string with a model as when models are read into CoCo may be given as argument.

Analogously, the message `:is-decomposable` will return the value `TRUE`, if the relevant model is *decomposable*. A model is decomposable, if the model is graphical and every cycle of length $n \geq 4$ possesses a *chord*, i.e., two consecutive vertices that are neighbours. Others names for a decomposable graph are *triangulated graph*, *chordal graph* and *acyclic hypergraph*.

The message `:is-submodel-of` without arguments will test whether the **current** model is a submodel of the **base** model. If one argument is given, then it is tested whether the model given by the argument is a submodel of the **current** model, not of the **base** model.

The message `:is-in-one-clique` will test whether the edge ⟨*edge*⟩ is only in one clique in the relevant model. The argument ⟨*edge*⟩ has to be a text-string with two factor-names separated by a comma. If the model is decomposable and ⟨*edge*⟩ is only a subset of one clique, then the model resulting from removing the edge ⟨*edge*⟩ from the model is also decomposable.

3.1.3 Describing, Printing and Disposing of Models

`:print-model &optional (`⟨*model*⟩ `'current)` ⟨*a*⟩ ⟨*b*⟩
  ⟨*argument*⟩ ⟶ { `'current` | `'base` | `'last` | `'all`
  | `'number` ⟨*integer*⟩ | `'interval` ⟨*integer*⟩ ⟨*integer*⟩
  | `'list` ⟨*list of integer*⟩ }
`:describe-model &optional (`⟨*model*⟩ `'current)` ⟨*a*⟩ ⟨*b*⟩
  ⟨*argument*⟩ ⟶ { `'current` | `'base` | `'last` | `'all`
  | `'number` ⟨*integer*⟩ | `'interval` ⟨*integer*⟩ ⟨*integer*⟩
  | `'list` ⟨*list of integer*⟩ }
`:dispose-of-model &optional (`⟨*model*⟩ `'current)` ⟨*a*⟩ ⟨*b*⟩
  ⟨*argument*⟩ ⟶ { `'current` | `'base` | `'last` | `'all`
  | `'number` ⟨*integer*⟩ | `'interval` ⟨*integer*⟩ ⟨*integer*⟩
  | `'list` ⟨*list of integer*⟩ }

Messages for printing, describing and disposing of models. From XLISP-STAT, e.g., all models entered to the CoCo object can be printed by the message `:print-model 'all`. If the command is succeeded, the value `TRUE` is returned, else `nil`. See “A Guide to CoCo” for details.

The method `:dispose-of-model` should be used with care. Since the models of model objects and association diagrams are represented by there models in the CoCo object, the message `:dispose-of-model 'all` should not be used when in connection with association diagrams.

`:print-formula`
`:print-vertex-order`
`:dispose-of-formula`

These messages will print the vertex-order, i.e., a running intersection property ordering, print the model expression (with the non-decomposable atoms partitioned as described in Jiroušek (1991) after fitting the model by, e.g., `:test`) or dispose of the expression. See “A Guide to CoCo” for details.

Common Decompositions of models

`:print-common-decompositions`
`:decompose-models` ⟨*set*⟩

The message `:print-common-decompositions` will print common decompositions of the **current** model and the **base** model. `:decompose-models` will decompose the **current** model and the **base** model with respect to ⟨*set*⟩, a text-string with a set of factors, if both models are decomposable with respect to ⟨*set*⟩. The 4 resulting models are added to the model-list in the CoCo-object. See "A Guide to CoCo".

3.2 Description of Data and Fitted Values

This section will describe how to return or print fitted values under the models in the CoCo-object.

3.2.1 Returning Table-Values

`:return-vector` ⟨*type*⟩ ⟨*set*⟩ `&key (`⟨*permuted*⟩ `T) (`⟨*model*⟩ `'current)`
`(`⟨*random*⟩ `nil) (`⟨*complete*⟩ `nil)`
⟨*type*⟩ ⟶ { `'counts` | `'probabilities` | `'expected` | `'unadjusted`
| `'f-res` | `'r-f` | `'g-res` | `'r-g` | `'adjusted` | `'m-res`
| `'standardized` | `'deviance` | `'freeman-tukey` | `'sqrt` | `'power`
| `'index` | `'zero` }
⟨*model*⟩ ⟶ { `'current` | `'base` | `'last` | ⟨*integer*⟩ }

The message `:return-vector` ⟨*type*⟩ ⟨*set*⟩ will return a vector with the table values ⟨*type*⟩ from the ⟨*set*⟩-marginal table computed under the **current** model. ⟨*type*⟩ can be `'counts`, `'expected`, `'probabilities`, `'unadjusted`, `'f-res`, `'r-f`, `'g-res`, `'r-g`, `'adjusted`, `'deviance`, `'power`, `'standardized`, `'freeman-tukey`, `'sqrt`, `'index` or `'zero`. See "A Guide to CoCo". ⟨*set*⟩ is a text-string with a set of factors. By default, the values are computed under the **current** model. If the keyword argument ⟨*model*⟩ is `'base` (`'last`), then the values from the **base** (**last**) model are returned. If ⟨*model*⟩ is a number, then the values from model with number ⟨*model*⟩ are returned. E.g., the message `:return-vector 'adjusted "ABC" :model 'base` will return a vector with the adjusted residuals for the {`ABC`}-marginal table computed under the **base** model. The message `:return-vector 'expected "*" :model 7` will return a vector with the expected counts in the full table computed under model with number 7.

If the keyword argument ⟨*random*⟩ is set to `TRUE`, then the values are computed in a random table with sufficient marginals as in the **current** model (or the model given by the keyword argument ⟨*model*⟩).

If the keyword argument ⟨*complete*⟩ is set to `TRUE`, then invalid[1] values are returned for the cells zero by structure.

If the keyword argument ⟨*permuted*⟩ is set to `TRUE`, then the cells are ordered according the order of the names of the factors in ⟨*set*⟩, else the cells are ordered according the order of factors in the specification.

3.2.2 Returning a Matrix of Table-Values

`:return-matrix` ⟨*type-list*⟩ ⟨*set*⟩ `&key (`⟨*permuted*⟩ `T) (`⟨*model-list*⟩ `nil)`
`(`⟨*random-list*⟩ `nil) (`⟨*complete-list*⟩ `nil)`

⟨*type-list*⟩ ⟶ { `(list [` ⟨*type*⟩ `] )` }
⟨*type*⟩ ⟶ { `See :return-vector` }
⟨*model-list*⟩ ⟶ { `(list [` ⟨*model*⟩ `] )` }
⟨*model*⟩ ⟶ { `See :return-vector` }
⟨*random-list*⟩ ⟶ { `(list [` ⟨*random*⟩ `] )` }
⟨*random*⟩ ⟶ { `T` | `nil` }
⟨*complete-list*⟩ ⟶ { `(list [` ⟨*complete*⟩ `] )` }
⟨*complete*⟩ ⟶ { `T` | `nil` }

Since repeated calls to `:return-vector` with `:random T` will return vectors from different random tables of observed counts the method `:return-matrix` might be useful. This method is programmed in Lisp by first returning the seed for the random generator, and then, for each column in the matrix to return, first setting that seed and then returning the column by the method `:return-vector`.

The arguments except ⟨*permuted*⟩ all have to be lists of the same length.

3.2.3 Printing Table etc.

`:print-table` ⟨*type*⟩ ⟨*set*⟩ `&key (`⟨*permuted*⟩ `T) (`⟨*model*⟩ `'current)`
`(`⟨*random*⟩ `nil) (`⟨*log-trans*⟩ `nil) (`⟨*complete*⟩ `nil)`

⟨*model*⟩ ⟶ { `'current` | `'base` | `'last` | ⟨*integer*⟩ }

The ⟨*set*⟩-marginal table with values ⟨*type*⟩ is computed under the **current** model and printed with the message `:print-table` ⟨*type*⟩ ⟨*set*⟩. The argument ⟨*type*⟩ for this method and the following method `:describe-table` is as for the method `:return-vector`. Unless the keyword argument ⟨*permuted*⟩ is set to `nil`, the table is printed according to the order of the factors in ⟨*set*⟩. If the keyword argument ⟨*permuted*⟩ is set to `nil`, the factors are ordered according to the order given by the specification of the factors.

If the keyword argument ⟨*random*⟩ is set to `TRUE`, then the values are computed in a random table with sufficient marginals as in the **current** model (or the model given by the keyword argument ⟨*model*⟩).

---

[1] In the current version of Sparc the value `-NaN` is returned for invalid values. The function `(invalid-real` ⟨x⟩`)` can be used to determine whether a value is invalid. The function `(invalid-real` ⟨x⟩`)` is implemented as `(defun (x) (mapcar #'not (< x 2147483640)))`.

Printing of Row-, Column-, Total-percents or percents of any other marginals: The argument ⟨*set*⟩ is a text-string with a set of factors or, when ⟨*type*⟩ is `'counts`, it can also be a text-string with two sets of factors separated by `/`. When ⟨*type*⟩ is `'counts` and ⟨*set*⟩ is a text-string with two sets ⟨*set-A*⟩ and ⟨*set-B*⟩ of factors separated by `/` then the counts in the ⟨*set-A*⟩-marginal table divided by the counts in the corresponding cell in the ⟨*set-B*⟩-marginal table times 100 are printed. Then Row-, Column-, Total-percents or percents of any other marginals table are printed.

E.g.

```
(send ⟨CoCo-object⟩ :print-table 'observed "AB" :permuted nil
  :model 'current))
(send ⟨CoCo-object⟩ :print-table 'observed "AB/B" :model 'base))
(send ⟨CoCo-object⟩ :print-table 'observed "ABC/C" :model 'last))
(send ⟨CoCo-object⟩ :print-table 'observed "ABC/BC" :model 7
  :complete T))
```

To print Total-percents, let the ⟨*set-B*⟩ be the empty set:

```
(send ⟨CoCo-object⟩ :print-table 'observed "AB/." :model 'base))
```

If `ExcludeMissing` is `On` and ⟨*set*⟩ is not a subset of the factors with complete information, then an action similar to `:exclude-missing 'in` ⟨*set + Base-set + Current-set*⟩ is performed before printing the table. Analogously for the methods `:return-vector`, `:return-matrix` and the following four methods.

Consider using the message `:exclude-missing 'in` ⟨*set*⟩ before the message `:print-table`. See “A Guide to CoCo” for further details.

```
:describe-table ⟨type⟩ ⟨set⟩ &key (⟨probit⟩ nil) (⟨rankit⟩ nil)
  (⟨uniform⟩ nil) (⟨model⟩ 'current) (⟨random⟩ nil) (⟨log-trans⟩ nil)
  (⟨complete⟩ nil)
     ⟨model⟩ ⟶ { 'current | 'base | 'last | ⟨integer⟩ }
:plot ⟨x⟩ ⟨y⟩ ⟨set⟩ &key (⟨X-model⟩ 'current) (⟨X-random⟩ nil)
  (⟨X-log⟩ nil) (⟨Y-model⟩ 'current) (⟨Y-random⟩ nil) (⟨Y-log⟩ nil)
  (⟨complete⟩ nil)
     ⟨X-model⟩, ⟨Y-model⟩ ⟶ { 'current | 'base | 'last
       | ⟨integer⟩ }
:list-values ⟨set⟩
:case-list ⟨set⟩
```

See “A Guide to CoCo” for further details.

3.3 Tests

This section will describe how compute tests between models on the CoCo-object, how to return the result of a test in a list and how to edit models in the CoCo-object.

3.3.1 Computing the Deviance and χ^2

```
:set-switch 'adjusted-df &optional (⟨hit⟩ 'flop)
:set-power-lambda &optional (⟨lambda⟩ 0.666667)
```

The computed test statistics depend on values set by these methods. The power λ of the power divergence is set by `:set-power-lambda` and the adjusted number of degrees of freedom is only computed, if `AdjustedDF` is set `On` by `:set-switch`, `On` by default. See "A Guide to CoCo" for further details. Note also options for exact tests, see the next chapter.

```
:test
:find-log-l
:find-deviance
```

```
:factorize &optional (⟨code⟩ 'edges) (⟨set⟩ ";")
      ⟨code⟩ ⟶ { 'edges | 'interactions }
```

The message `:test` will print values for the test between the **current** and **base** model: the likelihood ratio test statistic -2Log(Q) (the deviance), Pearsons chi-square χ^2 and the power divergence $2nI^\lambda$ (Read & Cressie 1988), and for these statistics the p-values based on the unadjusted and the adjusted number of degrees of freedom (if `AdjustedDF` is on in CoCo), and, if `ExactTest` is on in CoCo, the exact p-values for the test statistics.

When two ordinal factors are tested conditionally independent Goodman and Kruskal's Gamma coefficient with p-value is also printed.

The method `:find-log-l` will print the logarithm of the maximum of the likelihood function for the **current** model. `:find-deviance` will print the deviance for the test between the **current** and **base** model and the p-value for this statistic based on the unadjusted and the adjusted number of degrees of freedom.

The message `:factorize` will factorize the test into a list of tests, where each pair of models either differs with an edge or an interaction term. The order of the edges and interaction terms can be controlled by the optional argument ⟨*set*⟩. See also "A Guide to CoCo" for further details.

```
:compute-test &optional (⟨model-1⟩ 'current) (⟨model-2⟩ 'base)
      ⟨model-1⟩, ⟨model-2⟩ ⟶ { ⟨gc⟩ | 'current | 'base | 'last
        | ⟨integer⟩ }
(print-test ⟨test⟩)
:compute-deviance
(print-deviance ⟨test⟩)
```

The message `:compute-test` to a CoCo object will return the unadjusted number of degrees of freedom, the adjustment of DF. (if `AdjustedDF` is on in CoCo), the likelihood ratio test statistic -2Log(Q), Pearsons chi-square χ^2 and the power divergence $2nI^\lambda$, and, if `ExactTest` is on in CoCo, the exact p-values for the test of the **current** model against the **base** model. When two ordinal factors are tested conditionally independent Goodman and Kruskal's Gamma coefficient with p-value is also returned. The values can be printed by the function `(print-test` ⟨*returned values from :compute-test*⟩`)`. See section 10.4 for how to extract the values from the list returned by the method `:compute-test`.

The method `:compute-deviance` will return values as printed by the method `:find-deviance`. For how to extract the values: See the definition of the function `(print-deviance` ⟨*returned values from :compute-deviance*⟩`)`.

3.3.2 The Test-List

```
:show-tests
:dispose-of-tests
:set-switch 'reuse-test &optional (⟨hit⟩ 'flop)
```

The methods will show or dispose of the list of computed tests in the CoCo-object. See "A Guide to CoCo" for further details.

3.3.3 Editing Models with Tests

```
:generate-decomposable &optional (⟨only⟩ nil)
:generate-graphical &optional (⟨only⟩ nil)
:drop-edges ⟨gc⟩ &optional (⟨only⟩ nil)
:add-edges ⟨gc⟩ &optional (⟨only⟩ nil)
:drop-interactions ⟨gc⟩ &optional (⟨only⟩ nil)
:add-interactions ⟨gc⟩ &optional (⟨only⟩ nil)
:meet-of-models &optional (⟨only⟩ nil)
:join-of-models &optional (⟨only⟩ nil)
```

The methods will edit the **current** model according to the name of the method, and, unless the argument ⟨*only*⟩ is set to `TRUE`, produce a test.

The message `:generate-graphical` will make the model of the *2-section graph* for the **current** model. The generated model is the model with the generating class equal to the cliques in the 2-section graph for the **current** model, i.e., the maximal interaction terms are the cliques in the 2-section graph for the **current** model. The **current** model stays **current**. The generated model is added to the model list. If **base** and **current** are the same models, the **current** model is tested against the generated model, else the generated model is tested against the **base** model (if not the keyword argument ⟨*only*⟩ is used).

With `:generate-decomposable` a *FILL-IN* is made (adding extra edges) which is minimal in the sense that no subset of the fill-in will make the graph decomposable. The minimal *FILL-IN* is added to the 2-section graph for the **current** model. The generated model is the model with the generating class equal to the cliques in that decomposable 2-section graph.

Note: The added *FILL-IN* is not unique (e.g., there are two ways to add an edge to a graph with only four vertices and consisting of a four-cycle in order that the graph becomes decomposable) and is not minimum: there may be another *FILL-IN* (not a subset of the selected) with fewer edges. Choose between three simple methods for finding the *FILL-IN* by the method `:set-algorithm`, see the online help in CoCo for which algorithm is selected. A solution to the problem of finding a *FILL-IN* giving a small state space by simulated anneling as described in Kjærulff (1992) is not implemented in CoCo (yet).

If **base** is equal to **current**, then the **current** model is tested against the generated model, else the generated model is tested against **base** model (if not the keyword argument ⟨*only*⟩ is used.)

With `:drop-edges` the edges ⟨*gc*⟩ in the 2-section graph for the **current** model are removed and the generated model is tested against the **base** model (if not the keyword argument ⟨*only*⟩ is used). Note that the generated model is graphical.

The method `:add-edges` adds the edges ⟨*gc*⟩ to the 2-section graph for the **current** model and tests the resulting graphical model against the **base** model. If **base** and **current** are the same models then the **current** model is tested against the generated model, else the generated model is tested against **base** model (if not the keyword argument ⟨*only*⟩ is used).

With the message `:drop-interactions` all interaction terms which contain a set in ⟨*gc*⟩ are set equal to 0. In CoCo this is computed as follows: The message `:drop-interactions` adds the sets ⟨*gc*⟩ to the dual representation for the **current** model. Supersets of other sets in the resulting set of sets are removed and normal representation for that dual representation is found. The generated model is tested against the **base** model.

The message `:add-interactions` adds the sets ⟨*gc*⟩ to the generating class for the **current** model. Subsets of other sets in the resulting set of sets are removed and the generated model is tested against the **base** model. If **base** and **current** are the same models then the **current** model is tested against the generated model, else the generated model is tested against **base** model.

`:meet-of-models` and `:join-of-models` is performed on the **current** and the **base** model. The meet is the largest model contained in both the **base** and the **current** model. The resulting model will contain the maximal generators that are a subset of at least one generator in both **base** and **current**. For each generator in **base** and for each generator in **current**, the intersection of the two generators is formed, and the resulting model is the generating class with the maximal of these intersections. The generation of the model is symmetric in **base** and **current**.

The join is the smallest model containing both the **base** and the **current** model. The resulting model will contain the maximal generators in both **base** and **current**. The generation of the model is symmetric in **base** and **current**.

See "A Guide to CoCo" for further details.

3.4 Measures of Associations

This message will compute lots of statistics for two factors independent etc. given other factors:

`:slice` ⟨*factor-name-a*⟩ ⟨*factor-name-b*⟩ `&optional (`⟨*set*⟩ `";")`

For each 'slice' (2 dimensional table of ⟨*factor-name-a*⟩ by ⟨*factor-name-b*⟩) given configurations of the factors ⟨*set*⟩ the following measures of associations, statistics, is computed and printed:

For each configurations of the factors ⟨*set*⟩ the 2 dimensional table of ⟨*factor-name-a*⟩ by ⟨*factor-name-b*⟩ is printed with margins, and, on each 'slice' (2 dimensional table), the following statistics is computed and printed:

Fisher's exact test, Pearson χ^2 test, G^2: likelihood ratio test, Continuity-adjusted χ^2, Yates corrected χ^2, McNemar's test of symmetry, κ: Kappa, Cramer's V, Phi and maximum of Phi, Contingency Coefficient C and maximum of Contingency Coefficient C, Cross-product ratio alpha and logarithm of Cross-product ratio alpha, Mantel-Haenszel χ^2, Yule's Q, Yule's Y, Cochran-Armitage Trend Test with Goodness of fit: linear trend, Pearson (product-moment) correlation coefficient, Spearman rank correlation coefficient, τ_b: Kendall's Tau b, τ_c: Stuart's Tau c, Somers' D, Goodman and Kruskal's τ, γ: Gamma, λ_{asym}: Optimal prediction lambda, λ: Symmetric optimal prediction lambda, $\lambda^\star$: Modified asymmetric optimal prediction lambda, Uncertainty coefficient U_{asym}, Symmetric uncertainty coefficient U.

Agresti (1990) and Appendix A.5 of the manual for the BMDP statistical computer package (Dixon 1983) are good sources for references and for formulas for computing the above measures of associations with standard errors.

Chapter 4

Controlling Computed Tests in Model Selection in CoCo Objects

The methods of this chapter will control the criterion for judging the suitability of models in model selection in the CoCo, i.e., the methods will set options for how to compute test statistics, select test statistics and set limits for when to accept or reject a model in the model selection by the methods `:backward` and`:forward` and in the EH-procedure. Also a method for restricting the model class of the selection procedures to the sub-class of decomposable models is described in this chapter.

Besides by the options set by methods in this chapter, the model selection may depend on choice of how to handle missing values, options for controlling the IPS-algorithm, output, print formats, timer and diary, etc.

The options set by methods in this chapter are common to the CoCo-object and all model- and graph-objects of that CoCo-object.

Options controlling the computed test statistics and options controlling exact tests will also effect the values computed by the methods `:test`, `:exact-test`, etc., the model-editing methods and the model selection by association diagrams, but options selecting the test statistics to classify the model according to and options from this chapter setting significance levels will only control the model selection by the methods `:backward` and `:forward` and the EH-procedure.

4.1 Decomposable Mode(ls)

`:set-switch 'decomposable-mode &optional (`⟨*hit*⟩ `'flop)`

This option will restrict both the stepwise forward selection, the backward elimination and the global EH search procedure to decomposable models. I.e., the search procedures will not try to visit non-decomposable models, but edges (or interactions) can still by removed by other methods so that a non-decomposable model is generated.

It is attractive to restrict model selection to the class of decomposable models for a variety of reasons. Firstly, computation efficiency: the ability to use explicit formulae for the maximum likelihood estimates can reduce computational time substantially. Secondly, exact conditional tests for nested decomposable models can be calculated.

A backward elimination by the method `:backward` with `DecomposableMode` set `On` (or the method `:drop-least-significant-edge` with the keyword argument ⟨*decomposable*⟩ set to `TRUE`) will give the model selection proposed in Wermuth (1976).

4.2 Computed Test-Statistics and Choosing Tests and Significance Level

This section will describe options for how to compute test statistics, select test statistics and set limits for when to accept or to reject a model in the model selection by the methods `:backward` and `:forward` and in the EH-procedure. Also a short description of the *Information Criterion* is given.

4.2.1 Computing Test-Statistics

`:set-power-lambda &optional (`⟨*lambda*⟩ `0.666667)`
`:set-switch 'adjusted-df &optional (`⟨*hit*⟩ `'flop)`
`:set-switch 'reuse-test &optional (`⟨*hit*⟩ `'flop)`

The computed test statistics in all the test and model search procedures depend on values set by these methods. The power λ of the power divergence is set by the method `:set-power-lambda`. If the message `:set-power-lambda 'null` is used or the argument $\langle\lambda\rangle$ is set to 0 or 1, then the power divergence $2nI^{\lambda}$ is not computed and not printed for computed tests.

The adjusted number of degrees of freedom is only computed, if `AdjustedDF` is set `On` by `:set-switch`, `On` by default.

If reuse of tests is on, set by the method `:set-switch 'reuse-test`, then changing options by the two methods `:set-switch 'adjusted-df` and `:set-power-lambda` will not result in changes in the used test statistics for already computed tests. See 3.3.2 and use `:show-tests` or `:dispose-of-tests`.

Since the values returned by the methods `:compute-test` depend on the options set by the methods `:set-switch 'adjusted-df` and `:set-power-lambda` then changing these two options will also effect the model selection by association diagrams, see chapter 11.

4.2.2 Selecting Test Statistic and Significance Level

`:set-ordinal` $\langle set \rangle$
`:set-test &optional` ($\langle code \rangle$ `'lr`)
    $\langle code \rangle \longrightarrow$ { `'lr` | `'pearson` | `'power` }
`:set-ic &optional` ($\langle code \rangle$ `'aic`) ($\langle kappa \rangle$ `2.0`)
    $\langle argument \rangle \longrightarrow$ { `'on` | `'off` | `'aic` | `'bic` | `'kappa` $\langle integer \rangle$ }

In the model selection models are accepted or rejected according to the limits set by the methods `:set-acceptance` and `:set-rejection`.

The test statistic to compute the p-value or IC-value from is selected according to the option set by the method `:set-test`, unless two ordinal variables are tested conditionally independent. Goodman and Kruskal's Gamma coefficient is used when two ordinal variables are tested conditionally independent, also when an information criterion is selected. Ordinal variables are declared by the method `:set-ordinal`, see also section 2.3.

If the Gamma coefficient or the information criterion is computed, then this value is used to determine whether to accept or reject models. Thus these two values cannot be computed and printed without using the value in the model selection[1]. The Gamma coefficient can be computed and printed without the using value in the model selection by association diagrams, see the next part of this guide.

If both the Gamma coefficient and a information criterion are computed for a model in a model selection, then the Gamma coefficient is used to decide whether to accept or to reject the model.

Selection by Information Criteria

The Information Criteria $IC^{\kappa}_{\mathcal{A}}$ (Sakamoto & Akaike 1978) for model $\mathcal{A}$ is computed as

$$\begin{aligned} IC^{\kappa}_{\mathcal{A}} &= D_{\mathcal{A}} + \kappa \cdot \dim(\mathcal{A}), \\ AIC_{\mathcal{A}} &= D_{\mathcal{A}} + 2 \cdot \dim(\mathcal{A}) \quad \text{and} \\ BIC_{\mathcal{A}} &= D_{\mathcal{A}} + \log(n) \cdot \dim(\mathcal{A}), \end{aligned}$$

where $D_{\mathcal{A}}$ is the deviance of $\mathcal{A}$ relative to the saturated model, the likelihood ratio test statistic for $\mathcal{A}$ tested under the saturated model.

The change of Information Criteria is then in backward elimination and forward selection computed by

$$\begin{aligned} \Delta(IC)^{\kappa}_{\mathcal{AB}} = IC^{\kappa}_{\mathcal{B}} - IC^{\kappa}_{\mathcal{A}} &= \Delta(D)_{\mathcal{AB}} - \kappa \cdot D.F._{\mathcal{AB}}, \\ \Delta(AIC)_{\mathcal{AB}} &= \Delta(D)_{\mathcal{AB}} - 2 \cdot D.F._{\mathcal{AB}} \quad \text{and} \\ \Delta(BIC)_{\mathcal{AB}} &= \Delta(D)_{\mathcal{AB}} - \log(n) \cdot D.F._{\mathcal{AB}}, \end{aligned}$$

[1] To lift this inconvenient restriction one could consider extending the `:set-test` method so the Gamma coefficient or the information criterion has to be selected by this method to be used to determine whether to accept or reject models.

where $\Delta(D)_{\mathcal{AB}} = D_{\mathcal{B}} - D_{\mathcal{A}}$ is the difference between the deviances for the two models considered and $D.F._{\mathcal{AB}}$ is the adjusted or unadjusted number of degrees of freedom for the test of $\mathcal{B}$ under $\mathcal{A}$. An adjusted number of degrees of freedom is used, if it is set on by the `:set-switch 'adjusted-df`-method. If $\mathcal{B}$ is a submodel of $\mathcal{A}$ then $\Delta(D)_{\mathcal{AB}}$ and $D.F._{\mathcal{AB}}$ is positive. In a model search the IC value is minimized.

The message `:set-ic 'kappa` ⟨*integer*⟩ changes the constant κ used in computation of the information criterion and sets the computation of this criterion on. The constant is to 2 if the message `:set-ic 'aic` is used, and is set to log(total number of cases) if the message `:set-ic 'bic` is used. When `ExcludeMissing` is `On`, the number of used cases may differ for the different parts of a partitioned test, and the value `BIC` can not be computed.

Note that the messages `:set-ic 'aic`, `:set-ic 'bic`, `:set-ic 'kappa` ⟨*integer*⟩ and `:set-ic 'on` besides choosing an information criterion also sets both the acceptance and the rejection limit to 0.

Asymptotic or Exact p-values

`:set-exact-test &optional (`⟨*code*⟩ `'flop)`

  ⟨*code*⟩ $\longrightarrow$ { `'on` | `'off` | `'flop` | `'all` | `'deviance` }

Asymptotic or exact p-values are used according to whether exact test are chosen by the method `:set-exact-test`. See the next section about exact tests.

Acceptance and Rejection of Models

`:set-acceptance &optional (`⟨*alfa-accepted*⟩ `0.05)`
`:set-rejection &optional (`⟨*alfa-rejected*⟩ `0.025)`
`:set-components &optional (`⟨*components-limit*⟩ `0.01)`
`:set-separators &optional (`⟨*separators-limit*⟩ `0.001)`

A hypothesis is accepted, if the p-value is greater than a given limit. Thus to be able to classify models by the same expressions regardless of whether p-values or an information criterion is used, minus the change in the information criterion is used to classify models.

If the p-value (or minus the IC-value) for a model in the stepwise edge or interaction selection or in the EH-procedure is greater than the value set by the method `:set-acceptance`, then the model is accepted (else the model is rejected in the EH-procedure). In backward elimination edges (or interaction-terms) with a p-value greater than this limit are eligible for removal. The recursive backward elimination continues only as long as there are edges (or interaction-terms) with a p-value greater than this acceptance limit. Only if the p-value for the edge to remove is greater than this acceptance limit a new model is inserted into the model list in the CoCo-object.

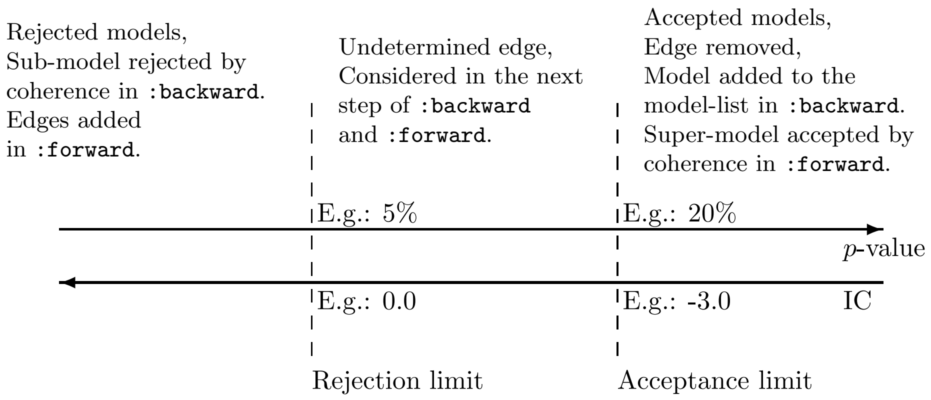


Figure 4.1: Significance levels for stepwise model selection and the EH-procedure.

Models with a p-value (or minus the IC-value) lower than or equal to the value set by the method `:set-rejection` are rejected, and sub-models can be rejected by *Coherence* in backward elimination.

Consider in each step of a forward selection all visited edges (or interaction-terms) not in the current model and with a p-value lower than or equal to the value set by `:set-rejection`, and consider the model with these edges added to the current model. This model will be the result of the forward selection step, and such a model is inserted in the model list for each step of the forward selection.

Models with a p-value (or minus the IC-value) greater than or equal to the value set by the method `:set-acceptance` are accepted, and models containing accepted models can be accepted by *Coherence* in forward selection.

When the messages `:set-ic 'on`, `:set-ic 'aic`, `:set-ic 'bic` and `:set-ic 'kappa` ⟨*integer*⟩ are used, then both the acceptance and the rejection limit are set to 0. If the acceptance limit is set to a value lower than the rejection limit, then the rejection limit is also set to the entered value. Analogously, if the rejection limit is set to a value greater than the acceptance limit, then the acceptance limit is set to the entered value.

Model control

Some further model checking is possible:

If the two models to be tested against each other have common decompositions, then the test is partitioned. The method `:set-switch 'partitioning` of the last section in this chapter will control this partitioning of tests. The deviance for the total test is then the sum of the deviances for each part of the test. If a limit ⟨*components-limit*⟩ is set by `:set-components`, then tests with a p-value lower than ⟨*components-limit*⟩ for any part of the test are rejected in the backward elimination and forward selection.

If two variables are accepted conditionally independent in a graph, then the two variables should by conditionally independent given any separator of the two variables in the graph. If a limit ⟨*separators-limit*⟩ is set by `:set-separators`, then models with a p-value lower than ⟨*separators-limit*⟩ for one separator are rejected in the stepwise model selection (when this model control is requested by the keyword `Separators` of the method for backward elimination of forward selection).

Since, in the model selection by association diagrams, the test statistic according to which to classify the model is selected by the method `:select-p-value` changing options from this section will not effect model selection by association diagrams, see chapter 11.

4.3 Exact Tests in Tests and Model Selection

`:set-exact-test &optional (`⟨*code*⟩` 'flop)`
    ⟨*code*⟩ ⟶ { `'on` | `'off` | `'flop` | `'all` | `'deviance` }
`:set-asymptotic &optional (`⟨*limit*⟩` 0.25)`
`:set-number-of-tables &optional (`⟨*number*⟩` 1000)`
    ⟨*number*⟩ ⟶ { ⟨*integer*⟩ | `'variating` | `'what` }
`:set-list-of-number-of-tables` ⟨*list-of-number-of-tables*⟩

`:set-switch 'exact-test-for-total-test &optional (`⟨*hit*⟩` 'flop)`
`:set-switch 'exact-test-for-parts &optional (`⟨*hit*⟩` 'flop)`
`:set-switch 'exact-test-for-unparted &optional (`⟨*hit*⟩` 'flop)`

`:set-seed &optional (`⟨*seed*⟩` 'random)`
    ⟨*seed*⟩ ⟶ { `'random` | ⟨*integer*⟩ | `'what` }
`:set-exact-epsilon &optional (`⟨*epsilon*⟩` 0.0000001)`
`:set-switch 'fast &optional (`⟨*hit*⟩` 'flop)`

If exact tests are computed in the model selection, these exact p-values are used. Computation of exact p-values is set on by the method `:set-exact-test`. The exact p-value of a test is only computed, if the asymptotic p-value of the test is smaller than the value set by `:set-asymptotic`, 1 by default. The number of tables to generate in the Monte Carlo simulation of the exact p-value is set by the method `:set-number-of-tables`. A variating number of tables is selected by the message `:set-number-of-tables 'variating`, and the number of tables to be generated is then set by the method `:set-list-of-number-of-tables`. See "A Guide to CoCo" for details.

Note that if the power divergence $2nI^{\lambda}$ or Pearson's chi-square χ^2 is selected by the method `:set-test`, and exact p-values are only computed for the likelihood ratio test-statistics because of the use of the message `:set-exact-test 'deviance`, then the asymptotic p-value will be used in the model selection procedures.

Please disregard the following options before you is familiar with exact tests in CoCo: The options `:set-switch 'exact-test-for-total-test`, `:set-switch 'exact-test-for-parts` and `:set-switch 'exact-test-for-unparted` will control computing of exact tests for the total of a test, that can be partitioned, the parts of a partitioned test and for tests not partitioned. The method `:set-switch 'partitioning` of the following section will control partitioning of tests, that is, partitioning of the two models by common decompositions.

The method `:set-seed` will set or return the seed for the random number generator used when generating the exact test. The message `:set-seed 'random` will set this seed to some random value depending on the process number of the XLISP-STAT session and the total used computing time. The value set by `:set-exact-epsilon` is to do with round of errors, and one of two algorithms to compute the exact p-value by is selected by `:set-switch 'fast`. See "A Guide to CoCo" for further details.

Since the values returned by `:compute-test` depends on options of the exact tests then changing options from this section will also effect model selection by association diagrams, see chapter 11.

4.4 Partitioning

`:set-switch 'partitioning &optional (`⟨*hit*⟩` 'flop)`
`:set-algorithm &optional (`⟨*code*⟩` 'a)`
    ⟨*code*⟩ ⟶ `{ 'a | 'b | 'c }`

The method `:set-switch 'partitioning` of the following section will control partitioning of tests, that is, partitioning of the two models by common decompositions. The tests by the methods `:test`, `:find-deviance`, `:compute-test` and `:compute-deviance`, the model editing methods and in model selection are then collapsed to the smallest tables on which the tests can be performed. When `ExcludeMissing` is `On`, the number of used cases may differ for the different parts of a partitioned test.

The method `:set-algorithm` will set an option to choose between different algorithms for factorization, computation of the Gamma coefficient, exact p-value, fill-in, etc. See "A Guide to CoCo" and the online help information of CoCo for further details.

Chapter 5

Stepwise Edge and Interaction Selection

This chapter is on stepwise model selection in CoCo-objects.

The incremental search in CoCo is performed by *forward inclusion* (Dempster 1972) and *backward elimination* (Wermuth 1976) of edges (or interaction-terms). In the backward elimination of edges in graphical models edges are sequentially eliminated from the independence graph. In the forward selection the edges are sequentially added to the independence graph.

Wermuth (1976) considers stepwise edge elimination on decomposable models. Two resent books Christensen (1990) and Whittaker (1990) each contain a chapter on model selection. In Whittaker (1990) with the title "Graphical Models" the model selection is of course treated in relation to graphical models, but also in Christensen (1990) an excellent discussion of model search on graphical model is given. Edwards (1993) considers some computational aspects of both the stepwise edge selection and the EH-procedure, see the next chapter.

In CoCo the backward elimination and the forward selection can be started from any initial model. The backward elimination procedure is not restricted to work on edges, but can also perform stepwise elimination of interaction-terms. Analogously with the forward selection. Steps of the backward elimination and the forward selection can be mixed together in any order, and between each step any set of edges or interaction-terms can be added or eliminated.

The backward elimination from the saturated model has the advantage that the tests are not performed under models later to be found rejected. In the forward selection from the uniform model the initial tests will be performed under models likely to be rejected. But in the forward selection the initial tests often will be collapsible to small tables, and much larger tables can be handled. The marginal associations tested in the first step of a forward selection from the uniform model might also be of some interest.

This chapter considers stepwise model selection in CoCo-objects. The result of this model selection is printed on standard output and models are added to the model list. In chapter 11 stepwise edge selection is considered on association diagrams. The results of the model selection on association diagrams can be new association diagrams.

For performing the same model selection the selection in CoCo-objects is the fastest, but addition to giving given the visual picture of the selection the model selection on association diagrams is more flexible: E.g., the user can program the selection criteria, and model selection on block-recursive models is possible. Test statistics can be computed and reported without being used.

In the model selection in CoCo-objects forward selection and backward elimination of interaction-terms is possible, and for each edge/interaction-term tested some model control is available.

The resulting models from each cycle of the backward elimination and the resulting model from the forward selection are added to the linked list of models in CoCo.

This list of models in the CoCo-object has been found to be very useful. Besides models generated by the incremental search procedures the model list also contain all models read into CoCo. The list of models helps keeping track of the search. The models in the list can be edited: edges or interaction-terms added or removed, the models tested against each other, fitted values in models tabulated, plotted, etc.

The methods `:backward` and `:forward` are together with the model-editing methods, the `:drop-edges` and `add-edges` methods, used in a highly user-controlled semi-automatic interactive model search. The procedures can be used to identify single (or a few) minimal acceptable models.

5.1 Selecting Test Statistic and Significance Level

The previous chapter will describe how to set the options for the computed test statistics, select the test statistic, how to set the level of significance and how to restrict the model class to decomposable models.

5.2 Fixing of Edges and Interactions

`:fix-edges` ⟨*edges*⟩
`:and-fix-edges` ⟨*edges*⟩

`:return-fix` ⟨*code*⟩
    ⟨*code*⟩ ⟶ { `'edges` | `'in` | `'out` }

The method `:fix-edges` will in the models considered in the stepwise model selection messages fix the edges and interaction-terms given by the argument of the `:fix-edges`-message. The argument to the method is a text string with a set of interaction terms, a generating class.

In a backward elimination of interactions, the interaction-terms a subset of a generator in the generating class set by `:fix-edges` are not eligible for removal. Also in a forward selection adding interaction terms containing a fixed generator to the model may not be considered.

In backward elimination of edges on graphical models, the edges (i.e., first-order interactions) given by `:fix-edges` are not tested for removal, and are never added in forward selection of edges. Thus in a forward selection of edges on graphical models an edge can be fixed without being containing a fixed generator.

Edges (and interactions) in the `FixEdgeList` can still be added or removed by the model-editing commands, i.e., by the `Drop-`, `Reduce-`, `Remove-` and `Add-`messages.

The `FixEdgeList` set by `:fix-edges` and `:and-fix-edges` is independent of the `FixIn` and `FixOut` used in the EH-procedure described in the following chapter[1].

Edges (interactions) can be added to the `FixEdgeList` by the method `:and-fix-edges`. Following use of the `:fix-edges` will clear and replace the `FixEdgeList`.

The method `:return-fix` will return a text-string with the fixing of edges or interactions for the stepwise model selection.

5.3 Backward Elimination

`:backward &key (`⟨*only*⟩ `nil) (`⟨*reversed*⟩ `nil) (`⟨*sorted*⟩ `nil)`
`(`⟨*short*⟩ `nil) (`⟨*headlong*⟩ `nil) (`⟨*recursive*⟩ `nil) (`⟨*coherent*⟩ `nil)`
`(`⟨*follow*⟩ `nil) (`⟨*least-significant*⟩ `T) (`⟨*separators*⟩ `nil) (`⟨*edges*⟩ `nil)`

The simplest form

`(send` ⟨*CoCo-object*⟩ `:backward )`

of the backward elimination message will visit all edges in the **current** model (if it is graphical), and for each visited edge produce a test. (See section 5.3.3 about choosing between backward elimination of edges and backward elimination of interaction-terms.) With the message

`(send` ⟨*CoCo-object*⟩ `:backward :recursive T)`

[1] One could consider removing the message `:fix-edges` and then also set the fix for stepwise backward elimination and forward selection with the methods `:fix-in` and `:fix-out` of the EH-procedure. This would also allow for different fixing for respectively backward elimination and forward selection. But for comparability with former versions of CoCo, separate fixes for the stepwise model selection and the global EH-procedure respectively have been kept.

all edges in the **current** model are visited and the least significant edge is removed. This step of the backward elimination is then repeated on the resulting model: All edges are visited in the model, and the least significant edge is removed. This is repeated until all edges are significant in the resulting model. When the keyword argument ⟨*recursive*⟩ is set to `TRUE`, the resulting accepted model from each step is added to the model list in CoCo.

5.3.1 Controlling the Output

By default a test is printed for each edge (or interaction-term) visited, and a report of rejected, eligible edges, the model resulting of the step of the recursive backward elimination, etc. is given between steps of the backward elimination.

Sorted

`(send` ⟨*CoCo-object*⟩ `:backward :sorted T :recursive T)`

Also a sorted list of the tests is printed for each step of the backward elimination. The list is sorted according to the selected p-value (statistics, if `IC` is selected). The p-values are ordered in increasing order (`IC` decreasing). The least significant edge (or interaction-term) is then the last edge in the sorted list.

Reversed

`(send` ⟨*CoCo-object*⟩ `:backward :reversed T :sorted T)`

If this keyword argument is used, the sorted list of tests is reversed.

Only

`(send` ⟨*CoCo-object*⟩ `:backward :only T :sorted T)`

Only the sorted list of tests is printed. This has the advantage of saving paper, if only the sorted list is wanted. But when running interactively, especially if exact p-values are computed and on large tables, there will be a lot of computation without any results shown, since the whole list of tests has to be computed for each step of the backward elimination before any output is printed. The keyword argument can of course be combined with the keyword argument ⟨*reversed*⟩.

`(send` ⟨*CoCo-object*⟩ `:backward :only T)`

If the keyword argument ⟨*only*⟩ is set to `TRUE` without the keyword argument ⟨*sorted*⟩ being set to `TRUE` then all printing from the backward elimination will be eliminated. The only result is then the models being inserted into the model list.

Short

`(send` ⟨*CoCo-object*⟩ `:backward :short T)`

By this keyword argument only a short report of the elimination is given: For each edge (or interaction-term) removed the corresponding test is printed.

If `Dump` is set `On` and the keyword argument `Short` is set to `TRUE` then for each completed step of the backward elimination, i.e., each removal of an edge or interaction, the `DumpFile` is rewound and a report of rejected, accepted, eligible edges and the model resulting of the step is printed on the `DumpFile`. In the following step of the backward elimination each tested edge and the selected test statistic for the test of the edge is printed on the `DumpFile`. This may be useful for monitoring a model selection on a large table and for recovering a terminated model selection.

5.3.2 Recursive Search

Recursive

If the keyword argument ⟨*recursive*⟩ is set to `TRUE` then the backward elimination is done recursively until no more edges (or interaction-terms) can be removed according to the selected significance level.

Coherent

`(send` ⟨*CoCo-object*⟩ `:backward :recursive T :coherent T)`

Once an edge (or interaction-term) in the backward elimination process is rejected, it is no more tested for removal.

Headlong

`(send` ⟨*CoCo-object*⟩ `:backward :headlong T :recursive T)`

or

`(send` ⟨*CoCo-object*⟩ `:backward :headlong T :recursive T`
`:coherent T)`

This keyword argument gives a recursive backward elimination strategy, where the first edge (or interaction) found to be non-significant is eliminated. In each step of this backward elimination, edges with p-value less than a given limit (e.g. 1% or 5%) are rejected, and, when the principle of weak rejection, *coherence*, is applied, they are not considered in sub-sequential steps. The first edge found in each step with p-value greater than a second limit (e.g. 20%) is removed. All eligible edges not visited in the current step are eligible in the next step together with the edges visited in the current step with a found p-value between the two limits. If not the keyword argument ⟨*coherent*⟩ is set to `TRUE`, then of course all edges of the model are eligible in the next step.

This gives a faster elimination than a backward elimination where all eligible edges in each step have to be visited in order to find the least significant edge.

If the p-value (or minus the IC-value) for a model in this stepwise edge or interaction selection is greater than the value set by `:set-acceptance`, then the model is accepted. I.e., in the backward elimination edges (or interaction-terms) are eligible for removal if the p-values of the edges are greater than this limit. The recursive backward elimination continues only as long as there are edges (or interaction-terms) with a p-value greater than this acceptance limit. Only if the p-value for the edge to be removed is greater than this acceptance limit a new model is inserted in the model list.

Edges with a p-value (or minus the IC-value) less than or equal to the value set by the method `:set-rejection` are rejected. Sub-models can be rejected by *Coherence* in the backward elimination.

Not to favour the removal of edges with, e.g., a low lexicographical order, the edges in this headlong backward elimination are visited in a random order. Hence several calls to the procedure might give different results, and a sample of models with all edges significant to a selected level of significance can be generated. The method `:set-seed` will set the seed for the used random number generator.

Follow

`(send` ⟨*CoCo-object*⟩ `:backward :recursive T :follow T)`

Tests are, if this keyword argument is set to `TRUE`, performed against the previously accepted model. In the first step of the `(send` ⟨*CoCo-object*⟩ `:backward :recursive T :follow T)` message the tests will be performed against the **current** model. In the next step, the tests are performed against the model accepted in the first step, etc. If the keyword argument ⟨*follow*⟩ not is set to `TRUE`, then all the tests will be performed against the **base** model.

Remove all non-significant edges or interactions in one step

`(send` ⟨*CoCo-object*⟩ `:backward :least-significant T)`

By default only the least significant edge (or interaction-term) is removed. With the message `(send` ⟨*CoCo-object*⟩ `:backward :least-significant nil)` all non significant edges are removed by one step of the backward elimination, i.e., all non-significant edges in the **current** model is removed from the model.

5.3.3 Graphical or Non-Graphical Models

Edges

`(send` ⟨*CoCo-object*⟩ `:backward :edges T)`

With the message `(send` ⟨*CoCo-object*⟩ `:backward :edges T)` the 2-section graph for the **current** model is created. In turn, each edge in the 2-section graph is then dropped, and the model with the generating class described by the cliques in the resulting graph is tested against the base model.

By `(send` ⟨*CoCo-object*⟩ `:backward :edges T)` all pairs of factors with an edge in the graph for the **current** model are tested conditionally independent given the remaining factors of the model.

Interactions

`(send` ⟨*CoCo-object*⟩ `:backward :edges nil)`

`(send` ⟨*CoCo-object*⟩ `:backward :edges nil)` will for each maximal interaction term in the **current** model remove the interaction-term, and then test the resulting model against the **base** model. This is computed by in turn adding the interaction-terms for the **current** model to the dual representation for the **current** model.

With `(send` ⟨*CoCo-object*⟩ `:backward :edges nil)` and a **current** model with only *n-th order effects* (`(send` ⟨*CoCo-object*⟩ `:read-n-interactions n "*;")` or `(send` ⟨*CoCo-object*⟩ `:read-n-interactions` ⟨*order*⟩ ⟨*set*⟩`))` all *n-th order partial associations* are tested.

For sparse contingency tables, Whittaker (1990) advocates the use of first-order log-linear models, i.e., hierarchical log-linear models with at most two-factor interactions. The model with all two-factor interactions can be read by the message `(send` ⟨*CoCo-object*⟩ `:read-n-interactions 1 "*;")` and then the backward elimination from this model can be performed by the message `(send` ⟨*CoCo-object*⟩ `:backward :edges nil :recursive T)`.

By default the message `:backward` is the same as the message `(send` ⟨*CoCo-object*⟩ `:backward :edges T)`, if the **current** model is graphical, else `(send` ⟨*CoCo-object*⟩ `:backward :edges nil)`.

5.3.4 Model Control

Separators

`(send` ⟨*CoCo-object*⟩ `:backward :separators T :edges T)`

By the message`(send` ⟨*CoCo-object*⟩ `:backward :separators T)` all tests for conditional independence are performed for each edge (or interaction-term) to remove: For each edge to remove, that is, for each pair of vertices connected in the model, the minimal *separators* between the two vertices are found, and the two vertices are tested conditionally independent given each separator.

If a limit ⟨*separators-limit*⟩ is set by `:set-separators`, then edges (or interaction-terms) with a p-value less than the limit ⟨*separators-limit*⟩ for one separator are rejected in a `(send` ⟨*CoCo-object*⟩ `:backward :separators T)` message.

Analogously, if a limit ⟨*components-limit*⟩ is set by `:set-components` and `Partitioning` is `On`, then edges (or interaction-terms) with a p-value less than ⟨*components-limit*⟩ for one part of the test are rejected in backward elimination.

5.4 Forward Selection

`:forward &key (`⟨*only*⟩ `nil) (`⟨*reversed*⟩ `nil) (`⟨*sorted*⟩ `nil)`
`(`⟨*short*⟩ `nil) (`⟨*headlong*⟩ `nil) (`⟨*recursive*⟩ `nil) (`⟨*coherent*⟩ `nil)`
`(`⟨*all-significant*⟩ `T) (`⟨*separators*⟩ `nil) (`⟨*edges*⟩ `nil)`

All edges not included in the **current** model are in turn added to the **current** model, and the **current** model is tested against the resulting model with the method `:forward`.

5.4.1 Controlling the Output

By default a test is printed for each edge (or interaction-term) visited and a report of rejected, eligible edges (or interaction-terms), the model resulting of each step of the forward selection, etc. is given between steps of the forward selection.

The keyword arguments ⟨*only*⟩, ⟨*reversed*⟩, ⟨*sorted*⟩ and ⟨*short*⟩ is as at the method `:backward` except that the p-values by default are ordered in decreasing order (`IC` increasing) in the sorted list.

5.4.2 Recursive Search

Also the keyword arguments ⟨*recursive*⟩, ⟨*coherent*⟩ and ⟨*headlong*⟩ are as at the method `:backward`.

A model, where rejected edges (or interaction-terms) are added to the **current** model, is added to the model list for each step of the `(send` ⟨*CoCo-object*⟩ `:forward :recursive T)` message.

Recursive

`(send` ⟨*CoCo-object*⟩ `:forward :recursive T)`

Forward selection is done recursively until no more edges have to be added according to the selected level of significance.

Coherent

`(send` ⟨*CoCo-object*⟩ `:forward :recursive T :coherent T)`

Once an edge (or interaction-term) is accepted in the forward elimination process, it is no more tested for adding.

Headlong

`(send` ⟨*CoCo-object*⟩ `:forward :headlong T :recursive T)`

or

`(send` ⟨*CoCo-object*⟩ `:forward :headlong T :recursive T :coherent T)`

In each step of this forward selection, edges (or interaction-terms) with p-value greater than a given limit (e.g. 10% or 20%) are not added, i.e., it is accepted that the edge can be omitted, and is not considered in sub-sequential steps when the principle of weak acceptance, coherence, is applied. The first edge found in each step with a p-value less than a second limit (e.g. 1%) is added to the model. All not visited eligible edges in the current step are eligible in the next step of the recursive forward selection together with the edges visited in the current step with a p-value found between the two limits. If not the keyword argument ⟨*coherent*⟩ is set to `TRUE`, then of course all edges not in the resulting model are eligible in the next step.

If the p-value (or minus the IC-value) for a model in this stepwise edge or interaction selection is greater than the value set by `:set-acceptance`, then the model is accepted, i.e., the edge or interaction-term is not added. By coherence these accepted edges are then not considered in sub-sequential steps of the forward selection.

Consider the edges with a p-value less than or equal to the value set by `:set-rejection` in a step of the forward selection. Then in each step of the forward selection these edges are added to the model of the current step, and the resulting model is used in the next step. The resulting model of each step is inserted in the model list. The recursive forward elimination continues only as long as there are found edges (or interaction-terms) with a p-value greater than this rejection limit. Only if the p-value for the edge to be added is smaller than this rejection limit a new model is inserted in the model list.

Not to favour the removal of edges with, e.g., a low lexicographical order, the edges in this headlong forward selection are visited in a random order. Hence several calls to the procedure might give different results, and a sample of models with all edges significant to a selected level of significance can be generated.

Add only the most significant edge/interaction

`(send` ⟨*CoCo-object*⟩ `:forward :all-significant T)`

By default, if a headlong forward selection is not performed, all significant edges (or interaction-terms) are added in each step of the forward selection. With the message `(send` ⟨*CoCo-object*⟩ `:forward :all-significant nil)` only the most significant edge (or interaction-term) is added in each step of the forward selection.

5.4.3 Graphical or Non-Graphical Models

This section is analogous to the section with the same name for the method `:backward`.

Edges

`(send` ⟨*CoCo-object*⟩ `:forward :edges T)`

With the message `(send` ⟨*CoCo-object*⟩ `:forward :edges T)` the 2-section graph for the **current** model is created. In turn, each edge not present in this 2-section graph is then added to the 2-section graph and the **current** model is then tested against the model with the generating class described by the cliques in the resulting graph.

By `(send` ⟨*CoCo-object*⟩ `:forward :edges T)` all pairs of factors without an edge in the graph for the **current** model are tested conditionally independent given the rest factors of the model.

With `(send` ⟨*CoCo-object*⟩ `:forward :edges T)` and a **current** model with only *main effects* (can be read by `(send` ⟨*CoCo-object*⟩ `:read-model ".;")`) all first order marginal associations are tested.

Interactions

`(send` ⟨*CoCo-object*⟩ `:forward :edges nil)`

`(send` ⟨*CoCo-object*⟩ `:forward :edges nil)` will for each minimal interaction-term (generator in the dual representation of the **current** model) not present in the **current** model add the interaction-term to the model, i.e., generate a model with the interaction added, and then test the **current** model against the resulting model.

By default the message `:forward` is the same as the message `(send` ⟨*CoCo-object*⟩ `:forward :edges T)`, if the **current** model is graphical, else it is the message `(send` ⟨*CoCo-object*⟩ `:forward :edges nil)`.

5.4.4 Model Control

Separators

This is analogous to the same keyword argument at the method `:backward`.

`(send` ⟨*CoCo-object*⟩ `:forward :recursive T :separators T)`

By the message `(send` ⟨*CoCo-object*⟩ `:forward :recursive T :separators T)` all tests for conditional independence are performed for each edge (or interaction-term) to enter: For each edge to add, that is, for each pair of vertices not connected in the model, the minimal separators between the two vertices are found, and the two vertices are tested conditionally independent given each separator.

If a limit ⟨*separators-limit*⟩ is set by `:set-separators` then edges (or interaction-terms) with a p-value less than the limit ⟨*separators-limit*⟩ for one separator are rejected in a `(send` ⟨*CoCo-object*⟩ `:forward :separators T)`-message. Thus these edges are to be added.

Analogously, if a limit ⟨*components-limit*⟩ is set by the `:set-separators` and `Partitioning` is `On`, then edges (or interaction-terms) with a p-value less than ⟨*components-limit*⟩ for one part of the test are rejected, i.e., the edges are added, in forward selection.

5.5 Asymmetry between Backward and Forward

The backward elimination and forward selection messages of course differ by removing and adding edges (or interaction-terms) respectively. In the backward elimination tests are performed against the previously accepted model, if the keyword argument ⟨*follow*⟩ is set to `TRUE` else, if the keyword argument ⟨*follow*⟩ not is set to `TRUE`, then all the tests will be performed against the **base** model. In the forward selection the model of the current step of the selection is tested against the model resulting from adding an edge (or interaction-term).

In the backward elimination only the least significant edge is removed in each step (by default), whereas in the forward selection all significant edges are added in each step (by default).

5.6 Examples

In relatively small tables we can do a backward elimination from the *saturated model* with exact tests. Since we can only compute the exact test for decomposable models, we set `DecomposableMode On`:

```
(send coco-object :read-model "*")
(send coco-object :set-exact-test 'all)
(send coco-object :set-switch 'decomposable-mode 'on)
(send coco-object :backward :recursive T)
```

In large models (more than 30 variables) we can do a backward elimination based on only the deviance. We choose selection by information criterion with the method `:set-ic` and then we set κ equal zero so the information criterion does not depend on the degrees of freedom:

```
(send coco-object :read-model "*")
(send coco-object :current)
(send coco-object :base)

(send coco-object :set-switch 'timer 'on)
(send coco-object :set-power-lambda 'NULL)
(send coco-object :set-switch 'decomposable-mode 'on)
(send coco-object :set-ic 'kappa 0)

(send coco-object :set-acceptance -5)
(send coco-object :set-rejection -2)
(send coco-object :status 'tests)
(send coco-object :backward :only T
                  :recursive T :headlong T :follow T :coherent T)
(send coco-object :current)
(send coco-object :print-model 'current)
(send coco-object :test)
(send coco-object :base)
```

or, we can do a forward selection based on the Bayesian information criterion from the model with only *main effects*:

```
(send coco-object :read-model ".")

(send coco-object :set-switch 'timer 'on)
(send coco-object :set-power-lambda 'NULL)
(send coco-object :set-switch 'decomposable-mode 'on)
(send coco-object :set-ic 'bic)

(send coco-object :set-rejection -5)
(send coco-object :forward)
(send coco-object :current)
(send coco-object :base)
(send coco-object :print-model 'current)
```

We have then created a model with all the edges that are significant marginally. The resulting model is not necessarily decomposable, so a fill in is added:

```
(send coco-object :is-decomposable)
(send coco-object :generate-decomposable)
(send coco-object :current)
(send coco-object :base)
(send coco-object :print-model 'current)
```

Now we try to remove edges that can be explained by other associations:

```
(send coco-object :set-acceptance -5)
(send coco-object :set-rejection -2)
(send coco-object :status 'tests)
(send coco-object :backward :only T
                  :recursive T :headlong T :follow T :coherent T)
(send coco-object :current)
(send coco-object :print-model 'current)
(send coco-object :test)
(send coco-object :base)
```

We then do a headlong recursive forward selection from the resulting model follow by adding of fill in:

```
(send coco-object :set-rejection -2)
(send coco-object :forward :only T :recursive T :headlong T)
(send coco-object :current)
(send coco-object :base)
(send coco-object :print-model 'current)

(send coco-object :is-decomposable)
(send coco-object :generate-decomposable)
(send coco-object :current)
(send coco-object :base)
(send coco-object :print-model 'current)
```

and a backward elimination with new limits for the information criterion:

```
(send coco-object :set-acceptance -2)
(send coco-object :set-rejection 0)
(send coco-object :status 'tests)
(send coco-object :backward :only T
                  :recursive T :headlong T :follow T :coherent T)
(send coco-object :current)
(send coco-object :print-model 'current)
(send coco-object :test)
(send coco-object :base)
```

Chapter 6

The EH-procedure

By the principles of *weak acceptance* and *weak rejection*, coherence, the search space of all hierarchical models on the contingency table is divided into two sets of models: the class of minimal acceptable models and the class of maximal rejected models. Hence, after using the 'Fast Procedure for Models Search', the *EH-procedure*, by Edwards & Havránek (1985) and Edwards & Havránek (1987) any model can be labeled as accepted or rejected.

By the EH-procedure the models (or sub-models of a specific base model, the model **SearchBase** read by the method `:read-base-model`) are classified into two regions $\mathcal{A}$ and $\mathcal{R}$: weakly accepted models and weakly rejected models. Weakly accepted models are models that include an accepted model, and weakly rejected models are models that are included in a rejected model. In turn, a boundary below the weakly accepted models and a boundary above the weakly rejected models, the R-dual $D_R(\mathcal{A})$ and the A-dual $D_A(\mathcal{R})$ respectively, are fitted.

In CoCo the global search can be started with the accepted class containing the saturated model and the class of rejected models containing the model with only main-effects, or any models can be entered into the two classes. The search stops when either all models in $D_A(\mathcal{R})\backslash\mathcal{A}$ are accepted or all models in $D_R(\mathcal{A})\backslash\mathcal{R}$ are rejected.

The search-module is only usable for small tables. The dimension of the table should not exceed 10 too much. There should not be too many choices between acceptable models. If there is no clear association between some factors (given others), i.e., there are many marginal associations but no clear conditional associations, then all models in $D_R(\mathcal{A})$ will be accepted for several fits of $D_R(\mathcal{A})$, and the duals will explode in size, i.e., if one can choose between too many acceptable models.

If one of the search-methods described in this chapter is used, then the search-module is initiated, i.e., initial classes of models and duals are found for the

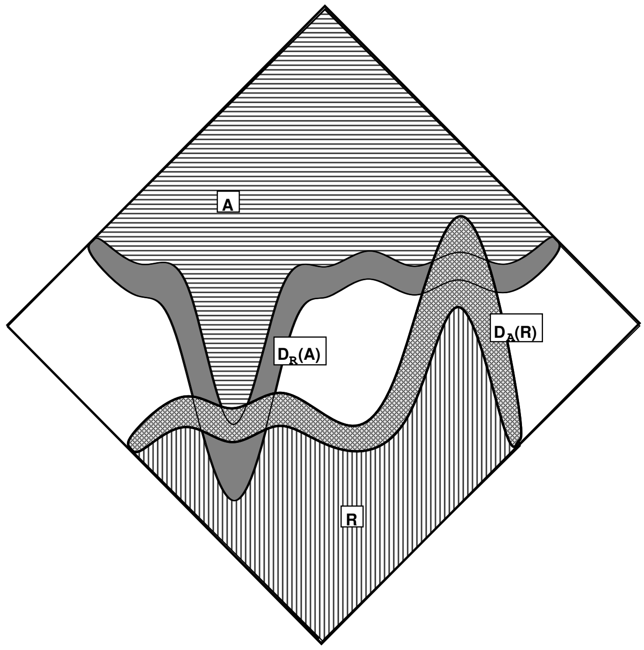


Figure 6.1: Weakly accepted and weakly rejected models and duals.

EH-procedure. By default, the *saturated model* is accepted, and the model with only main effects is rejected.

The message `:status 'search` will show the *accepted models*, the *rejected models*, the *A dual*, the *R dual*, the **SearchBase** (the base model used in the EH-procedure), the *FixIn*, and the *FixOut*. The method `:dispose-of-eh` will dispose of the *accepted models*, the *rejected models*, the *A dual* and the *R dual*.

A simple model search among first decomposable models, then among all graphical models and finally among hierarchical models can be performed by

```
(send ⟨CoCo-object⟩ :eh :sub-class 'decomposable)
(send ⟨CoCo-object⟩ :eh :sub-class 'graphical)
(send ⟨CoCo-object⟩ :eh :sub-class 'hierarchical)
```

The smallest dual in each step is fitted by these three messages. If instead, rough estimates of the dual sizes should be found in each step, and the dual with the smallest rough estimate is to be fitted in each step, then the messages

```
(send ⟨CoCo-object⟩ :eh :strategy 'rough :sub-class 'decomposable)
(send ⟨CoCo-object⟩ :eh :strategy 'rough :sub-class 'graphical)
(send ⟨CoCo-object⟩ :eh :strategy 'rough :sub-class 'hierarchical)
```

can be used.

6.1 Computed Tests and Selection Criteria

Chapter 4 will describe how to set the options for the computed test statistics, select the test statistic to classify the models according to, and how to set the level of significance.

6.2 Fix Edges/Interactions:

`:fix-in` $\langle gc \rangle$
`:fix-out` $\langle gc \rangle$
`:add-fix-in` $\langle gc \rangle$
`:add-fix-out` $\langle gc \rangle$
`:redo-fix-in`
`:redo-fix-out`

`:return-fix` $\langle code \rangle$
    $\langle code \rangle \longrightarrow \{$ `'edges` | `'in` | `'out` $\}$

The method `:return-fix` will return a text-string with the interaction-terms (or edges) fixed in the stepwise model selection or `FixIn` or `FixOut` in the EH-procedure depending on the argument.

6.2.1 FixIn

With the method `:fix-in` edges (generators) in the generating class are forced into all models in the EH-procedure. The message `:fix-in "ABC"` would force the terms <>, <A>, <B>, <C>, <AB>, <AC>, <BC> and <ABC> into all models.

6.2.2 FixOut

With the method `:fix-out` all terms containing a generator in the generating class are FixOut. The FixOut is the dual representation of the maximal model considered, see pp. 340 Edwards & Havránek (1985).

Note that only terms containing a FixOut-generator are FixOut. The message `:fix-out "ABC"` excludes the terms <ABC>, <ABCD>, <ABCE>, <ABCDE>, <ABCF>, <ABCDF>, <ABCEF>, <ABCDEF>, <ABCG>, etc. from models. But the terms <AB>, <AC> and <BC> are not excluded.

So to exclude edges in the graphical search only generators with cardinality 2 (edges) should be given to FixOut.

Main effects

`:set-main-effects` $\langle set \rangle$

The factors to be included in the search are set with `:set-main-effects`, default-value: `*`, that is, all factors read are used in the EH-procedure.

The EH-procedure can also be restricted to a subset of the declared factors by giving sets of cardinality one to the `:fix-out`-method. But CoCo works more efficiently, if the message `:set-read 'subset` ⟨*subset*⟩ is used when reading data instead of using `:fix-out`, `:set-main-effects` or `:set-read 'subset` ⟨*subset*⟩ methods in the EH-procedure.

6.2.3 Read Base Model

`:read-base-model` ⟨*gc*⟩

With the method `:read-base-model` a base model is read for the EH-procedure. Tests in the EH-procedure are then performed against the read model. It is not the same model as the **base** model used outside the EH-procedure, since `:read-base-model` effects `FixIn` and `FixOut`. Terms not in the base model are FixOut, i.e., FixOut is set to the dual representation of **SearchBase**. Since there is no distinction between FixOut due to the `FixOut`-message and FixOut due to `:read-base-model` and since a large **SearchBase** would give less FixOut due to **SearchBase**, previous FixOut is cancelled by the use of the method `:read-base-model`. Use `:redo-fix-out` to redo previous FixOut.

6.2.4 Interaction between FixIn, FixOut and SearchBase

Effect of FixIn on FixOut

FixIn terms are removed from FixOut, i.e., FixIn is added to the normal representation of FixOut, the maximal terms in the models.

Effect of FixIn on SearchBase

FixIn has no effect on **SearchBase**, but only edges in **SearchBase** are Fixed In. FixIn is reduced. It could have been chosen that the **SearchBase** with the FixIn method is extended with edges in FixIn but not in **SearchBase**. The extended **SearchBase** is printed by the use of the method `:fix-in`, but not used.

Effect of FixOut on FixIn

FixOut terms in FixIn are removed from FixIn, i.e., FixIn is restricted to normal representation of FixOut.

Effect of FixOut on SearchBase

FixOut has no effect on **SearchBase**, but terms not in the **SearchBase** are added to FixOut.

Effect of SearchBase on FixIn

Only terms in the **SearchBase** can be FixIn. FixIn thus is reduced by **SearchBase**.

Effect of SearchBase on FixOut

Terms not included in the base model are FixOut, i.e., FixOut is set to the dual representation of **SearchBase**. Since there is no distinction between FixOut due to the `:fix-out`-message and due to `:read-base-model` and since a large **SearchBase** would give less FixOut due to **SearchBase**, previous FixOut is cancelled. Use `:redo-fix-out` to redo previous FixOut.

6.2.5 Add FixIn and FixOut

A new `:fix-out`-message will clear the `FixOut`. The message `:add-fix-out` $\langle$*edges/gc*$\rangle$ will add $\langle$*edges/gc*$\rangle$ to the previous `FixOut`. The `FixIn` is effected as described in a previous section by the argument $\langle$*edges/gc*$\rangle$ in the `:add-fix-out`-message.

Similarly a new `:fix-in` $\langle$*edges/gc*$\rangle$-message will clear the `FixIn`. The method `:add-fix-in` will add $\langle$*edges/gc*$\rangle$ to the previous `FixIn`. The `FixOut` is affected as described in the previous section by the argument $\langle$*edges/gc*$\rangle$ in the `:add-fix-in`-method.

6.2.6 Redo FixIn and FixOut

The message `:redo-fix-out` will redo the combination of the last `:fix-out`-message and following `:add-fix-out`-messages. `FixIn` is then edited by the total specified `FixOut`.

Similarly will the message `:redo-fix-in` redo the combination of the last `:fix-in`-message and following `:add-fix-in`-messages. `FixOut` is then edited by the totally specified `FixIn`.

6.3 Selecting Model Class and Search Strategy

`:set-search` $\langle$*code*$\rangle$
    $\langle$*code*$\rangle$ $\longrightarrow$ { `'graphical` | `'hierarchical` | `'smallest`
     | `'alternating` | `'rough` }

Since the option set by this method can be given as an argument to the methods that uses the option, the method is actually not necessary. But by setting the option with this method, e.g., the calls of the directed search is shorter to type, and describing the method gives the possibility of describing the sub-classes, to which the EH-procedure is restricted:

6.3.1 Graphical Search

In the graphical search, for each step either all not classified models in the graphical A-dual are fitted or all not classified models in the graphical R-dual are fitted.

$D_A^g(\mathcal{R})$: The graphical A-dual $D_A^g(\mathcal{R})$ of the rejected models is the minimal models in the class of models that adds at least one edge to each rejected model.

$D_R^g(\mathcal{A})$: The graphical R-dual $D_R^g(\mathcal{A})$ of the accepted models is the maximal models in the class of models that omits at least one edge from each accepted model.

The `:set-search 'graphical` will set up the EH-procedure to a graphical search, and `:eh :sub-class 'graphical` will perform a model selection among graphical models, i.e., the graphical A-dual and R-dual are used.

6.3.2 Decomposable Mode

By a naive approach the selection can be restricted to decomposable models: non-decomposable models in the duals to fit are simply ignored. Non-decomposable models in the search are ignored with the message `:set-switch 'decomposable-mode 'on` before messages `:fit` and `:eh`. The search is then performed among only decomposable models. (This can also be performed by `:eh :sub-class 'decomposable`) After a search among only decomposable models, a search among graphical models can be performed with the result of the decomposable search as a starting-point. `DecomposableMode` is turned off again by typing `:set-switch 'decomposable-mode 'off`.

Since non-decomposable models in the graphical dual are just ignored in the decomposable search, there might still be graphical and decomposable models to fit, when both the set $D_A^g(\mathcal{R}) \setminus \mathcal{A}$ of not classified models in the graphical A-dual and the set $D_R^g(\mathcal{A}) \setminus \mathcal{R}$ of not classified models in the graphical R-dual contain no more decomposable models. So the decomposable search should be followed by a graphical search. One might consider adding edges to or removing edges from the non-decomposable models in the graphical duals in the models selection among decomposable models.

Since a lot of computing time in the search module is used to find duals, and not, as it might be expected, in computing the tests and fitting the models by the IPS-algorithm, the usefulness of the decomposable search is doubtful (unless exact tests are to be used).

If `ExactTest` is on, then `DecomposableMode` also should be on. If `ExactTest` is on and `DecomposableMode` is off, then asymptotic P-values will be used for the non-decomposable models.

Note that the messages

```
(send ⟨CoCo-object⟩ :eh :sub-class 'graphical)
(send ⟨CoCo-object⟩ :eh :sub-class 'hierarchical)
```

set `DecomposableMode` to off.

If no models are entered or found by EH-procedure, the search is started with the saturated model accepted and the model with only main effects rejected.

6.3.3 Hierarchical Search

In the hierarchical search we for each step either fit all not classified models in the hierarchical A-dual or all not classified models in the hierarchical R-dual.

$D_A(\mathcal{R})$: The hierarchical A-dual $D_A(\mathcal{R})$ of the rejected models is the minimal models in the class of models that adds at least one interaction term to each rejected model.

$D_R(\mathcal{A})$: The hierarchical R-dual $D_R(\mathcal{A})$ of the accepted models is the maximal models in the class of models that sets at least one interaction term in all accepted models to zero.

The message `:set-search 'hierarchical` will set the search-module up to a hierarchical search, and `:eh :sub-class 'hierarchical` will perform a model selection among hierarchical models, i.e., the hierarchical A-dual and R-dual are used.

If the hierarchical search follows a graphical search, the regions found in the graphical search are used as a starting point in the hierarchical search, else the search is started with the saturated model accepted and the model with only main effects rejected, or any models can be entered into the two classes.

6.4 Dispose of the Model Classes and Duals

```
:dispose-of-eh  &optional (⟨code⟩ 'all)
      ⟨code⟩ ⟶ { 'all | 'duals | 'a-dual | 'r-dual | 'classes
         | 'accepted | 'rejected }
```

The message `:dispose-of-eh` will dispose of the *accepted models*, the *rejected models*, the *A dual* and the *R dual*.

6.5 Read and Fit Models or Force Models into Classes

6.5.1 Fit Some Models

```
:fit 'models ⟨models⟩
```

With `:fit 'models` ⟨*models*⟩ you can enter a list of models that is classified into accepted and rejected models, and start the model-search from there. The argument ⟨*models*⟩ is a text-string with a set of models, a set of generating classes as a generating class is a set of sets, e.g.,

(send ⟨*CoCo-object*⟩ `:fit 'models "[ [[A][B][C]] [[AB][AC][BC]] ]")`

or the shorter form

(send ⟨*CoCo-object*⟩ `:fit 'models " A,B,C. AB,AC,BC; ")`

can be used.

6.5.2 Reading Accepted and Rejected Models

`:accept 'models` ⟨*models*⟩
`:reject 'models` ⟨*models*⟩

With the messages `:accept 'models` ⟨*models*⟩ and `:reject 'models` ⟨*models*⟩ the entered models are forced into the `accepted region` and the `rejected region` respectively. The argument ⟨*models*⟩ is a text-string with a set of models as at the method `:fit 'models` ⟨*models*⟩.

6.6 Copy Models Between the Models-List and the Search Classes

The following methods are used to copy models between the model list and the classes of models in the EH-procedure.

6.6.1 Models from List to Search Classes

`:fit` ⟨*model*⟩ `&optional` ⟨*a*⟩ ⟨*b*⟩
`:accept` ⟨*model*⟩ `&optional` ⟨*a*⟩ ⟨*b*⟩
`:reject` ⟨*model*⟩ `&optional` ⟨*a*⟩ ⟨*b*⟩
⟨*model*⟩ ⟶ { `'current` | `'base` | `'last` | `'all`
| `'number` ⟨*integer*⟩ | `'interval` ⟨*integer*⟩ ⟨*integer*⟩
| `'list` ⟨*list of integer*⟩ }

Models from the model list are fitted and put into a model class in the EH-procedure according to whether the models are accepted or rejected, or models from the model list are forced into a model class of the EH-procedure regardless of whether the model is accepted or rejected.

6.6.2 Models from Search Classes to Model List

:extract $\langle$*class*$\rangle$ &optional ($\langle$*sub-class*$\rangle$ nil)
$\langle$*class*$\rangle$ $\longrightarrow$ { 'accepted | 'rejected | 'a-dual | 'r-dual }
$\langle$*sub-class*$\rangle$ $\longrightarrow$ { 'decomposable | 'graphical | 'hierarchical | nil }

This method is used to copy a model class or a dual from the EH-procedure to the model list. This is, e.g., used in the following method, where the two classes of accepted and rejected models are extracted from the EH-procedure and then for each model in the two classes an association diagram is created:

`:plot-EH-search-result`

6.7 Find Duals

:find-dual $\langle$*dual*$\rangle$ &optional ($\langle$*sub-class*$\rangle$ nil)
$\langle$*dual*$\rangle$ $\longrightarrow$ { 'a-dual | 'r-dual | 'both-duals }
$\langle$*sub-class*$\rangle$ $\longrightarrow$ { 'decomposable | 'graphical | 'hierarchical | nil }

The messages `:find-dual 'a-dual`, `:find-dual 'r-dual` and `:find-dual 'both-duals` can be used, if you want to see the dual by `:status 'search` before fitting models.

If the keyword argument $\langle$*sub-class*$\rangle$ is given, then the class of models given by the keyword is used in the `:find-dual`-message, and the requested class is also selected in subsequential EH-messages needing a model class, else the class of models to use is selected by the options set by the methods `:set-search` and `:set-switch 'decomposable-mode` or other messages setting model class.

6.8 Directed Search

:fit $\langle$*dual*$\rangle$ &optional ($\langle$*sub-class*$\rangle$ nil)
$\langle$*dual*$\rangle$ $\longrightarrow$ { 'a-dual | 'r-dual | 'smallest-dual | 'largest-dual | 'both-duals }
$\langle$*sub-class*$\rangle$ $\longrightarrow$ { 'decomposable | 'graphical | 'hierarchical | nil }

This method is used to do a single step of the EH-procedure. The keyword argument $\langle$*sub-class*$\rangle$ is as for the `:find-dual`-message.

6.8.1 Fit Smallest Dual

With the message `:fit 'smallest-dual` both duals are found (if not already found), and all not classified models in the dual with the fewest not classified models are fitted and classified into accepted and rejected models.

6.8.2 Fit Largest Dual

With the message `:fit 'largest-dual` both duals are found (if not already found), and all not classified models in the largest dual are fitted and classified into accepted and rejected models.

6.8.3 Fit R-Dual

The message `:fit 'r-dual` finds (if not found) and fits the R dual. Repeated use of `:fit 'r-dual` gives a backward elimination. When there are no more models to fit, then the message `:fit 'r-dual` will use little computing time, so extra `:fit 'r-dual`-message will only produce output without using much computing time.

6.8.4 Fit A-Dual

`:fit 'a-dual` finds (if not found) and fits the A dual.

6.8.5 Fit Both Duals

`:fit 'both-duals` fits all not classified models in both duals as they are. Note that this need not give the same as `:fit 'a-dual`, followed by `:fit 'r-dual`.

6.9 Automatic Search

`:eh &optional (`⟨*strategy*⟩ `nil) (`⟨*sub-class*⟩ `nil)`
  ⟨*strategy*⟩ ⟶ { `'smallest` | `'alternating` | `'rough` | `nil` }
  ⟨*sub-class*⟩ ⟶ { `'decomposable` | `'graphical` | `'hierarchical`
   | `nil` }

This method is used to do a recursive search by the EH-procedure.

If the keyword argument ⟨*sub-class*⟩ is given, then the class of models given by this keyword argument is used in the EH-message, and the requested class is also selected in subsequential EH-messages needing a model class, else the class of models to use is selected by the options set by the methods `:set-search` and `:set-switch 'decomposable-mode` or other messages setting model class.

If the keyword argument ⟨*strategy*⟩ is given, then the strategy given by the keyword is used, and the strategy will also be used in subsequential `:eh`-messages, else the strategy set by the method `:set-search` or previous use of the `:eh`-method is used.

The three messages

`(send` ⟨*CoCo-object*⟩ `:eh :sub-class 'decomposable)`
`(send` ⟨*CoCo-object*⟩ `:eh :sub-class 'graphical)`
`(send` ⟨*CoCo-object*⟩ `:eh :sub-class 'hierarchical)`

will do a graphical search ignoring non-decomposable models, a graphical search and a hierarchical search respectively.

Below are described three strategies for selecting the dual to fit in each step of the EH-search. Which of the three strategies to use is set by the method `:set-search` or by a keyword argument of the `:eh`-method. For the time being the strategy `Smallest` is default.

Which of the three strategies is the fastest depends on the situation. Try them all three (they need not give the same result since e.g., weakly rejected models may be accepted). See the `ReportFile` for where CoCo uses computing time. Find some models (by e.g., `:backward`) and copy them to the EH-procedure by the methods `:fit`, `:accept` and `:reject`. before the automatic search is started you can do some directed search by the method `:fit`.

6.9.1 Smallest Automatic

With `:eh :strategy 'smallest` for each step (classification of all not already classified models in a dual into the accepted and the rejected regions) in the search, both duals are found, and the dual with the fewest not classified models is selected for the next step. This will often minimize the number of fitted models, but sometimes a lot of computing time is used for finding large duals that are not used.

6.9.2 Rough Automatic

With this strategy rough estimates of the dual sizes are found after each step, and the dual with the smallest rough estimate of dual size is selected for the next step. The rough estimate of dual size is the product of the number of edges (or interaction-terms) in the model (or the dual representation of the model) over all models in the class $\mathcal{A}$ or $\mathcal{R}$. Since it sometimes avoids finding large duals that are not used, this can be effective. By this strategy the complete search in Edwards & Havránek (1985) on a 6 dimensional table is done by CoCo in less than 700 milliseconds on a SPARCstation.

6.9.3 Alternating Automatic

The R-dual and the A-dual of the two model classes are in turn classified with the message `:eh :strategy 'alternating`. In a few examples, where alternating fitting the R-dual and the A-dual is optimal, this is the fastest. But often the strategy will start to select the largest dual, and then this strategy will be the slowest.

The automatic search stops when either all models in $D_A(\mathcal{R}) \setminus \mathcal{A}$ are accepted or all models in $D_R(\mathcal{A}) \setminus \mathcal{R}$ are rejected.

6.10 Force a Dual into a Model Class

`:reject` $\langle$*dual*$\rangle$ `&optional (`$\langle$*sub-class*$\rangle$ `nil)`
`:accept` $\langle$*dual*$\rangle$ `&optional (`$\langle$*sub-class*$\rangle$ `nil)`
    $\langle$*dual*$\rangle$ $\longrightarrow$ { `'a-dual` | `'r-dual` }
    $\langle$*sub-class*$\rangle$ $\longrightarrow$ { `'decomposable` | `'graphical` | `'hierarchical`
    | `nil` }

The message `:accept 'R-dual` adds the R-dual of the accepted models to the accepted models.

The message `:reject 'A-dual` adds the A-dual of the rejected models to the rejected models.

If we add the A-dual of the rejected models to the accepted models the search will be finished. So this method does not exist. The A-dual of the accepted models can be added to the accepted models by `:accept 'A-dual`. similarly the R-dual of the rejected models can be added to the rejected models by `:reject 'R-dual`.

Chapter 7

Options in CoCo Objects

This chapter will describe methods for the CoCo-object not dealt with so fare in this guide. The methods concern status of the options in the CoCo-objects, methods for naming the input- and output-files, methods for controlling additional output, print formats and the IPS- and EM-algorithm.

7.1 End and Restart

`(make-coco &key (`$\langle n \rangle$` 512) (`$\langle p \rangle$` 128) (`$\langle q \rangle$` 16) (`$\langle title \rangle$` nil))`

The function `(make-coco)` will return a *CoCo object*.

`:end`
`:resume`

`:isnew` $\langle$*coco-id*$\rangle$ `&key` $\langle$*title*$\rangle$
`:title &optional (`$\langle$*title*$\rangle$` nil set)`
`:current-coco`

`(quit)`

The method `:isnew` is for returning a CoCo-object, used by the function `(make-coco)`. The keyword argument `:title` will return or set the title of the CoCo-object. The method `:current-coco` will make the CoCo-object the current CoCo-object.

With `:resume` the CoCo-object is entered: Ordinary CoCo commands are then read from standard input until the command 'end' is given. The method `:end` will clear the CoCo-object: remove temporary files for the object, dispose data, models, tests, etc.

The function `(quit)` will terminate all CoCo-objects and end XLISP-STAT.

7.2 Status

`:status &optional (`⟨*code*⟩ `'all)`

  ⟨*code*⟩ ⟶ { `'all` | `'formats` | `'tests` | `'exact` | `'fix` | `'ips`
   | `'em` | `'specification` | `'observations` | `'limits` | `'files`
   | `'other` | `'search` }

This method will print status of the CoCo-object: setting of options, specified data, size of read data, etc.

7.3 Input files

`:coco-source` ⟨*file-name*⟩

With the message `:coco-source` ordinary CoCo commands are read from the file ⟨*file-name*⟩, that is, commands in the language of "A Guide to CoCo".

7.4 Output files

`:set-diary-file` ⟨*file-name*⟩
`:set-report-file` ⟨*file-name*⟩
`:set-log-file` ⟨*file-name*⟩
`:set-dump-file` ⟨*file-name*⟩
`:set-switch 'keep-diary &optional (`⟨*hit*⟩ `'flop)`
`:set-switch 'keep-report &optional (`⟨*hit*⟩ `'flop)`
`:set-switch 'keep-log &optional (`⟨*hit*⟩ `'flop)`
`:set-switch 'keep-dump &optional (`⟨*hit*⟩ `'flop)`

These messages will name files for additional output or prevent them from being removed when the CoCo-object is terminated. See "A Guide to CoCo" for details.

7.5 Diary, Timer etc.

`:set-switch 'diary &optional (`⟨*hit*⟩ `'flop)`
`:set-switch 'timer &optional (`⟨*hit*⟩ `'flop)`
`:set-switch 'echo &optional (`⟨*hit*⟩ `'flop)`
`:set-switch 'note &optional (`⟨*hit*⟩ `'flop)`
`:set-switch 'report &optional (`⟨*hit*⟩ `'flop)`
`:set-switch 'trace &optional (`⟨*hit*⟩ `'flop)`
`:set-switch 'debug &optional (`⟨*hit*⟩ `'flop)`
`:set-switch 'log &optional (`⟨*hit*⟩ `'flop)`
`:set-switch 'log-data &optional (`⟨*hit*⟩ `'flop)`
`:set-switch 'dump &optional (`⟨*hit*⟩ `'flop)`

The message `:set-switch` with these arguments will control additional output. The full specification of the method `:set-switch` is as follows:

`:set-switch` ⟨*switch*⟩ `&optional` (⟨*hit*⟩ `'flop`)
  ⟨*switch*⟩ ⟶ { `'Keyboard` | `'Diary` | `'Log` | `'Log-Data` | `'Dump`
   | `'Keep-Diary` | `'Keep-Report` | `'Keep-Log` | `'Keep-Dump`
   | `'Timer` | `'Echo` | `'Note` | `'Report` | `'Trace` | `'Debug`
   | `'Partitioning` | `'Graph-mode` | `'Decomposable-mode`
   | `'Graphical-search` | `'Pausing-of-output`
   | `'Short-test-output` | `'Reuse-tests` | `'Adjusted-df`
   | `'Exact-test` | `'Exact-only-log-l` | `'Fast` | `'Exact-test-total`
   | `'Exact-test-parts` | `'Exact-test-unparted` | `'Large` | `'Huge`
   | `'Sorted` }
  ⟨*hit*⟩ ⟶ { `'on` | `'flop` | `'off` | `'what` }

7.6 Print Formats

`:set-print-formats` ⟨*width*⟩ ⟨*decimals*⟩
`:set-table-formats` ⟨*width*⟩ ⟨*decimals-probabilities*⟩ ⟨ *-expected*⟩
 ⟨ *-residuals*⟩
`:set-test-formats` ⟨*statistic-width*⟩ ⟨*statistic-decimals*⟩ ⟨*probabilities-width*⟩
 ⟨*probabilities-decimals*⟩
`:set-page-formats` ⟨*line-length*⟩ ⟨*page-length*⟩
`:set-switch 'short-test-output &optional` (⟨*hit*⟩ `'flop`)
`:set-switch 'pause &optional` (⟨*hit*⟩ `'flop`)
`:set-paging-lenght` ⟨*length*⟩

Messages for setting print formats. See "A Guide to CoCo" for details. The arguments are (except for `:set-switch`) integers.

7.7 Controlling the IPS- and EM-algorithm

`:set-ips-stop-criterion &optional` (⟨*code*⟩ `'cell`)
  ⟨*code*⟩ ⟶ { `'cell` | `'sum` }
`:set-ips-epsilon &optional` (⟨*epsilon*⟩ `0.0000001`)
`:set-ips-max-iterations &optional` (⟨*max*⟩ `100`)

`:set-em-initial &optional` (⟨*code*⟩ `'uniform`)
  ⟨*code*⟩ ⟶ { `'uniform` | `'first` | `'last` | `'mean` | `'random`
   | `'input` }
`:set-em-epsilon &optional` (⟨*epsilon*⟩ `0.001`)
`:set-em-max-iterations &optional` (⟨*max*⟩ `100`)

Messages for controlling the IPS- and EM-algorithm. See "A Guide to CoCo" for details. The argument ⟨*epsilon*⟩ is a real and the argument ⟨*max*⟩ is an integer.

7.8 Dispose of Tables

`:dispose-of-tables`
`:dispose-of-q-table` ⟨*set*⟩
`:dispose-of-probabilities`
`:dispose-of-all-q-tables`

Messages for disposing of tables. See "A Guide to CoCo" for details.

7.9 Limitations and Precision

See "A Guide to CoCo" online help in CoCo and `:status 'limits` for details.

Chapter 8

CoCo Model Objects

CoCo model objects are interesting because they in the next chapter are specialized to association diagrams.

8.1 Making Model-Objects

`:make-model &optional (`⟨*model*⟩ `'current) &key (`⟨*title*⟩ `nil)`
    ⟨*model*⟩ $\longrightarrow$ { ⟨*gc*⟩ | `'current` | `'base` | `'last` | ⟨*integer*⟩ }

The message

`(send` ⟨*CoCo-object*⟩ `:make-model` ⟨*model*⟩ `:title` ⟨*title*⟩`)`

will return a *model object* for the model ⟨*model*⟩ in the CoCo object ⟨*coco-object*⟩. The optional argument ⟨*model*⟩ to the message `:make-model` can be a single character string with the model written as models are written in CoCo. Or the argument ⟨*model*⟩ can be `'current`, `'base`, `'last` or a number. If the argument ⟨*model*⟩ is a text-string, then the model is added to the list of models in the object ⟨*coco-object*⟩. The keyword argument ⟨*title*⟩ is an optional title for the model object.

8.2 Test a Model-Object against another Model-Object

The message

`:compute-test-against-model-object` ⟨*base*⟩

will return values for the test of the model of the object against the model of ⟨*base-model-object*⟩. The test is computed by using the method `:compute-test` after declaring the model of the object to be the **current** model and the model of ⟨*base-model-object*⟩ to be the **base** model, see also section 3.3. Before the method is ended, the **current** and the **base** model are of course returned to the original **current** and **base** models. The function `(print-test)` can be used to print the returned values.

8.3 Other Messages to the Model-Object

`:isnew` ⟨*model*⟩ `coco-object &key` ⟨*title*⟩

This message is for returning a CoCo-model-object. The method is used in the method `:make-model`.

8.3.1 Messages Specialized for the Model-Object

The methods `:return-model`, `:return-model-set`, `:return-model-number`, `:is-decomposable`, `:is-graphical`, `:is-in-one-clique`, `:is-submodel-of`, `:return-vector`, `:return-matrix`, `:print-table`, `:describe-table`, `:plot`, `:print-model`, `:describe-model`, `:dispose-of-model`, `:return-edge-list` and `:return-edge-list-list`, are for the CoCo-model-object specialized so that the default model is the model of the object. E.g., will the message `:return-vector` to a model object return a vector with the table values ⟨*type*⟩ from the ⟨*set*⟩-marginal table computed under, by default, the model for the *model object*, to which the message is send. If the argument ⟨*model*⟩ is set, then the values are of course computed under the relevant model. This is the case for all messages expecting a ⟨*model*⟩ argument.

For a CoCo model object one could also consider letting the methods `:collaps-model`, `:normal-to-dual`, `:dual-to-normal`, `:compute-deviance`, `:compute-test`, `:find-log-l`, `:find-deviance`, `:test`, `:decompose-models`, `:meet-of-models`, `:join-of-models`, `:factorize`, `:drop-edges`, `:add-edges`, `:drop-interactions`, `:add-interactions`, `:print-common-decompositions`, `:generate-graphical`, `:generate-decomposable`, `:backward` and `:forward` working on the model of the CoCo model object. This is only implemented for the two methods `:backward` and `:forward`.

8.4 Shared Values for Model-Objects

A message to a CoCo model object will not only effect the model object, but also the corresponding CoCo object and all CoCo model objects for that CoCo object, e.g., `:set-exact-test 'on` to a CoCo model object will set computation of exact test on for the object, the corresponding CoCo object and all CoCo model objects of that CoCo object.

		Sentence	
Victim	Murderer	Death	Other
Black	Black	11	2209
	White	0	111
White	Black	48	239
	White	72	2074

Table 8.1: *Death sentencing and race in Florida, 1973-79.*

8.5 Making a Graph-Object for a Model-Object

`:make-graph &key (`⟨*location*⟩` nil) (`⟨*size*⟩` nil) (`⟨*title*⟩` nil)`

The message `:make-graph` to a CoCo-model-object will make a plot with the association graph for the model of the object.

8.6 The Krippendorf 1986 Example

The following example tests the conditional independence between "Sentence" and "Murderer" given "Victim" without and with the assumption that the cell "Death sentence to white murdering black" is zero by structure (Krippendorf 1986). (Conditional quasi-independence might be a better word in the second case.) The difference between expected counts under the model with conditional independence found respectively with and without the assumption about zero by structure is computed.

First we start XLISP-STAT by `xlisp+coco`:

```
hardy:CoCo.1.2c.Beta.solaris> bin/xlisp+coco
XLISP-PLUS version 2.1g
Portions Copyright (c) 1988, by David Betz.
Modified by Thomas Almy and others.
XLISP-STAT 2.1 Release 3.39 (Beta).
Copyright (c) 1989-1994, by Luke Tierney.

; loading "mystatinit.lsp"

>
```

We then create a coco-object:

```
> (setf a (make-coco :title "Without structural zero"))
 CoCo started
```

```
 CoCo     -     A program for estimation, test and model search
 in very large ‘Co’mplete and ‘InCo’mplete ‘Co’ntingency tables.
 Version 1.3   Friday March 17 12:00:00 MET 1995
 Compiled with gcc, a C compiler for Sun4
 Copyright (c) 1991, by Jens Henrik Badsberg
 Licensed to ...

#<Object: 1347656, prototype = COCO-PROTO>
```

The table is declared:

```
> (send a :enter-names "smv" '(2 2 2))
 T
```

and the data read into the coco-object:

```
> (send a :enter-table '(11 2209 0 111 48 239 72 2074))
    8 cells with  4764 cases read.
 Finding all marginals.
T
```

The saturated model and a model with “Sentence” and “Murderer” conditionally independent given “Victim” is read:

```
> (setf Model-a-1
        (send a :make-model "*" :title "Saturated model"))
 #<Object: 4015336, prototype = COCO-MODEL-PROTO>

> (setf Model-a-2
        (send a :make-model "mv,vs"
              :title "Sentence cond. indp. of Murderer"))
 #<Object: 4010512, prototype = COCO-MODEL-PROTO>
```

The two models are read by the message `:make-model`, which will read the model and return an object of the model.

The two models are tested against each other:

```
> (print-test
   (send Model-a-2 :compute-test-against-model-object Model-a-1))
 Number of cases:          4764
                    Statistic  Probability     Adjusted
Deviance:              67.760      0.00000      0.00000
Power divergence:      84.729      0.00000      0.00000
Pearsons X^2:          97.055      0.00000      0.00000
DF.                                      2            2
NIL
```

A list with the observations is returned:

```
> (send a :return-vector 'observed "*")
 (11 2209 0 111 48 239 72 2074)
```

Then a CoCo-object in which the cell "Death sentence to white murdering black" is set to zero by structure is made:

```
> (setf b (make-coco :title "With structural zero"))
  CoCo started

 CoCo     -     A program for estimation, test and model search
 in very large 'Co'mplete and 'InCo'mplete 'Co'ntingency tables.
 Version 1.3   Friday March 17 12:00:00 MET 1995
 Compiled with gcc, a C compiler for Sun4
 Copyright (c) 1991, by Jens Henrik Badsberg
 Licensed to ...

#<Object: 3988216, prototype = COCO-PROTO>
```

The table is declared and the data is entered with the structurally zero cell. The value `-1` is entered in the cell to structurally zero:

```
> (send b :enter-names "smv" '(2 2 2))
 T

> (send b :enter-table '(11 2209 -1 111 48 239 72 2074))
    8 cells with  4764 cases read.
 Finding all marginals.
T
```

"Sentence" is tested conditionally quasi-independently of "Murderer" given "Victim" with the assumption that the cell "Death sentence to white murdering black" is zero by structure:

```
> (setf Model-b-1
        (send b :make-model "*"
              :title "Saturated model with struct. zero"))
 #<Object: 3977968, prototype = COCO-MODEL-PROTO>

> (setf Model-b-2
        (send b :make-model "mv,vs"
              :title "Sentence cond. quasi-indp. of Murderer"))
 #<Object: 3973144, prototype = COCO-MODEL-PROTO>

> (print-test
   (send Model-b-2 :compute-test-against-model-object Model-b-1))
```

```
 Number of cases:          4764
                     Statistic  Probability     Adjusted
Deviance:               66.684      0.00000      0.00000
Power divergence:       84.072      0.00000      0.00000
Pearsons X^2:           96.502      0.00000      0.00000
DF.                                       2            1
NIL
```

The difference between expected counts under the model with conditional independence found respectively with and without the assumption about zero by structure is computed:

```
> (- (send Model-a-2 :return-vector 'expected "*")
   (send Model-b-2 :return-vector 'expected "*"))
 (-0.523808 0.523926 0.52381 -0.523811 0 0 7.62939e-06 0)
```

The function `(quit)` is used to terminate all CoCo objects and end the XLISP-STAT session:

```
> (quit)
 CoCo resumed
 CoCo ended
 CoCo resumed
 CoCo ended
fisher:CoCo.guide>
```

Part III

CoCo Graph Objects

Chapter 9

Association Diagrams: A Graphical User Interface

The *Association Diagram Object* is an object for a graph window with the association graph of an interaction model. The model can be causal or not, and the variables can be discrete, ordinal, continuous etc.

The *Independence Graph*, association graph, is a simple undirected graph with as many vertices as the table has dimension. A *vertex* for each variable. Two vertices are *adjacent* in the independence graph for the model unless the two variables are conditionally independent given the other variables.

The association diagram is a very appealing working tool for model selection. The layout of the graph can be edited by dragging the vertices of the graph, points for variables, with the mouse, and the edges will then follow the vertices. New association diagrams can be created from a diagram by adding or dropping edges by the mouse. Tests of two variables conditionally independent can be performed by clicking the edge between the two variables by the mouse. Edges can be drawn with a width proportional to the significance of the edges are. In the chapters following this chapter methods for backward elimination and forward selection of edges in association diagrams are implemented and the association diagram is extended to causal models. In the backward elimination of edges the edges are redrawn as they are visited. If a test is computed for an edge, then the edge will be drawn with a width depending on the significance of the edge. If an edge is not fitted, then the edge will be drawn with a color depending on whether the edge is not fitted because the model is non-decomposable, the edge is fixed, the edge is rejected by coherence, etc.

		Mouse fixed	Mouse moved	
Shift	Control	Click point	Drag edge/vertex	Drag block
-	-	Drop edge	Add edge	Resize block
+	-	Label edge	Move vertex	Move block
-	+	Start idling	Rotate plot +	Stop idling
+	+	Remove edge label or block	Move vertex-, edge-, or block-label	

Table 9.1: Mouse events.

9.1 Mouse Interaction with the Association Diagram

This section is to give an overview of the interaction with the association diagram window by the mouse.

Since addition and deletion of edges are of primary interest in the analysis, these actions are performed by the mouse without using the "SHIFT" (extend modifier) and the "CONTROL" (option modifier) keys. To, e.g., edit the graph by moving a vertex, the "SHIFT" (extend modifier) has to be held down while moving the vertex with the mouse.

Edges are of primary interest when the layout of the graph, i.e., the positions of vertices, labels, blocks, etc., has been decided. Vertices are moved by dragging the vertices while the "SHIFT" (extend modifier) key is held down, and edge labels with p-values are added by clicking the edges while the "SHIFT" (extend modifier) key is held down without moving the mouse. The mouse is moved, if it is moved more than 4 pixels. Thus a computation of a test is started, a the vertex is not dragged more than 4 pixels, when one tries to move a vertex. To avoid this annoying time-consuming computation an option for an "edit mode" is available, see section 9.1.5.

9.1.1 Edges and Edge-labels

- `Drop edge:` Click the edge with the mouse. Keep the mouse fixed while pressing the button; if the mouse is moved more than 4 pixels, then an edge will be added.

- `Add edge:` Position the mouse at one of the vertices of the edge, press the mouse button and drag an edge to the other vertex of the edge, and release the mouse button.

- `Static:` In the "static" mode a new plot is created when edges are removed or added. If "static" is turned off by pressing the key `'s'` in the plot, then

		Mouse fixed	Mouse moved	
Shift	Control	Click point	Drag vertex	Drag block
-	-	Jump vertex	Move vertex	Resize block
+	-	Jump vertex	Move vertex	Move block
-	+	Jump vertex label	Move vertex label	
+	+	Jump label	Move vertex-, edge-, or block-label	

Table 9.2: Mouse events in “Edit mode”.

edges are added to or removed from the current plot. “static” mode is turned on by pressing the key `‘s’` again.

- `Label edge with a` p`-value:` Position the mouse at the edge, press a mouse button while “SHIFT” (extend modifier). Release. If the mouse is moved while the button is down, then a vertex will move. Or press `‘e’` at the edge.

- `Move edge label:` Position the mouse at the edge label, press a mouse button and both “SHIFT” (extend modifier) and “CONTROL” (option modifier), and drag the label. Release the mouse button. Or add another label to the edge, and type `‘r’` for “Redraw”.

- `Remove edge label:` Position the mouse at the edge, press a mouse button while both “SHIFT” (extend modifier) and “CONTROL” (option modifier). Release. If the mouse is moved while the button is down, then a vertex-, edge- or block-label will be moved. Or press `‘E’` at the edge.

- `Add and delete variables:` Variables are deleted by dragging vertices to the wastebasket; right to and below the right lower corner of the plot. Add a variable be selecting “Read model” from the graph menu and edit the generating class for the graph.

9.1.2 Vertices and Variable-Labels

- `Drag vertex:` Position the mouse at the vertex, press a mouse button and “SHIFT” (extend modifier), and drag the vertex. Release the mouse button.

- `Drag vertex-label:` Position the mouse at the vertex of the label, press a mouse button and both the “SHIFT” (extend modifier) and the “CONTROL” (option modifier) keys, and drag the label. Release the mouse button.

- `Set vertex-label:` Position the mouse at the vertex that should be labeled, and press '`l`'. Enter the label in the appearing dialog-window.
- `Plot grid:` Press '`g`' in the graph-window. Also cancelled by '`g`'.
- `Undo vertex move:` Press '`u`' in the graph-window.
- `Redo vertex move:` Press '`z`' in the graph-window.
- `Adjust vertex- and label-positions to grid:` Press the key '`y`' in the graph-window.

9.1.3 Blocks and Block-labels

- `Create blocks:` Select **Add block** or **Define blocks** from the graph menu.
- `Resize block:` Position the mouse at a corner of the block, press a mouse button, and drag the corner of the the block. The positions of the vertices of the graph are unchanged.
- `Drag block:` Position the mouse at a corner of the block, press a mouse button and "SHIFT" (extend modifier), and drag the block. The vertices of the graph falling in the block will follow the moved block.
- `Drag block-label:` Position the mouse at a block corner, press a mouse button and both "SHIFT" (extend modifier) and "CONTROL" (option modifier), and drag the label. Release the mouse button.
- `Delete block:` Position the mouse at a corner of the block, press a mouse button and both "SHIFT" (extend modifier) and "CONTROL" (option modifier), and move the mouse. Or press '`W`' at a block corner.
- `Adjust blocks to grid:` Press '`Y`' in the graph-window.

9.1.4 Rotation of the Graph

- `Rotate graph:` Position the mouse in the graph, press a mouse button and "CONTROL" (option modifier), and drag the graph. Release the mouse button.
- `Idling graph:` Idling (when added to the graph, see section 10.6) is started by pressing a mouse button and "CONTROL" (option modifier) while the mouse is fixed. Idling can be stopped by rotating the graph.

9.1.5 Edit mode

`:edit-graph &optional (⟨val⟩ nil set)`

In the "edit" mode computation of tests, removal and addition of edges and rotation of the graph by mouse events are disabled.

If the mouse is dragged without pressing the "SHIFT" (extend modifier) key, then instead of adding an edge a vertex will be dragged in this mode. If the mouse is clicked, an edge will not be dropped, but the closest vertex will jump to the position of the mouse. See also table 9.2. Note that only in edit mode it is possible to move a vertex less than 4 pixels without dragging the vertex for a tour longer than the 4 pixels.

9.1.6 Drag Graph mode

`:drag-graph &optional (⟨val⟩ nil set)`

If the slot `'drag-graph` is true, returned and set by the method `:drag-graph`, then the graph will be updated and redrawn continuously when dragging vertices, labels, etc. Else, which is faster, only a cursor will be dragged.

9.1.7 Static mode

`:static &optional (⟨val⟩ nil set)`

In the "static" mode a new plot is created when edges are removed or added. If "static" is turned off by the key-event `'s'` in the plot, then edges are added to or removed from the plot. "static" is turned on by pressing the key `'s'` again. "static" can also be returned, turned on or off by the method `:static`.

9.2 Controls

`:add-controls`

The method `:add-controls` adds controls for setting the options "Drag graph", "Static", controls for undoing and redoing point moves, controls for rotating the graph and controls for setting the options of the model search by forward selection and backward elimination of edges in the association diagram. See figure 9.1 for a diagram with controls added.

9.3 The Graph Menu

A subset of the following actions can be performed by selecting an item from the graph menu:

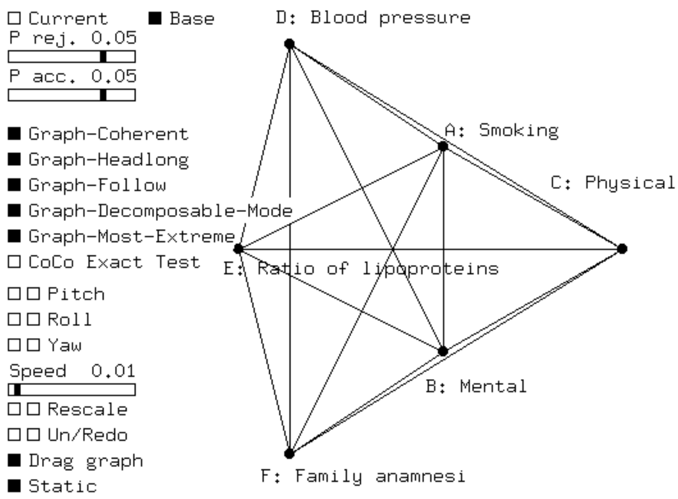


Figure 9.1: Graph with "controls".

Set new title: Set the title on the graph. A dialog window will ask for the title.

Save image in PostScript: Saves PostScript code for the graph in mono-chrome on a Unix file. A dialog window will ask for the file name, default: *"image.ps"*. Before dumping the graph on a PostScript file, the `:redraw` method of the graph should be redefined to redraw the graph, see section 10.8.4.

Save image in TeX: Dump TeX code for the graph on standard output or a Unix file. A dialog window will ask for a file name, default: *"image.tex"*, standard output by *"Cancel"*.

Name the graph: The graph is linked to an identifier with a name entered in an appearing dialog window.

Smooth graph, `'y'`: Adjusted the vertex positions and vertex label positions to a grid with width `5`. Can also be performed by the key event `'y'`. Type `'Y'` to adjust block and block labels positions.

Undo vertex move, `'u'`: Undo the last vertex or vertex label move.

Redo vertex move, `'z'`: Redo the last undone move.

Skip Undo vertex move, `'U'`: Skip a position, an item in the undo list.

Skip Redo vertex move, `'Z'`: Skip a position, an item in the redo list.

Drag graph: Switch dragging of graph.

Drop edge: Help information on dropping edges.

Set variable label, `'l'`: Help information on setting vertex label.

Label an edge with p-value, `'e'`: Help information on labelling edge with p-value.

Drop an edge label, `'E'`: Help information on removing edge label.

Add edge: Help information on adding edges.

Move vertex: Help information on dragging vertices.

Move label for vertex: Help information on dragging labels of vertices.

Add block: Add a block. The block-key is entered in the appearing dialog window.

Define blocks: Define blocks. The causal structure is entered in the appearing dialog window.

Fix edges: Set fixing of edges. Edges to be fixed are entered in the appearing dialog window.

Exact-Test: Switch Exact Tests for the CoCo object.

Graph-Coherent, `'x'`: Switch `:graph-coherent`.

Graph-Headlong, `'h'`: Switch `:graph-headlong`.

Graph-Random-order, `'o'`: Switch `:graph-random-order`.

Graph-Follow: local tests, `'f'`: Switch `:graph-follow`.

Graph-Decomposable-Mode, `'d'`: Switch `:graph-decomposable-mode`.

Graph-Most-Extreme, `'m'`: Switch `:graph-most-extreme`.

Set Graph P accepted: Remove p-value; Edges with a p-value greater than the value entered in the appearing window are removed in backward elimination and accepted by coherence in forward selection.

Set Graph P rejected: Add p-value; Edges with a p-value less than the value entered in the appearing window are rejected by coherence in backward elimination and are added in forward selection.

Label all edges, `'L'`: Visit all edges in the graph, and draw the eligible edges with a width "inverse proportional" to the p-value.

Drop least significant edge: Remove the least significant edge, the first non-significant edge or all non-significant edges.

- Recursive backward: Do a recursive removing of edges.

- Blockwise Recursive backward: Do block by block a recursive removing of edges for each block.

Add most significant edge: Add the most significant edge, the first significant edge or all significant edges.

- Recursive forward:	Do a recursive adding of edges.
- Blockwise recursive forward:	Do a recursive adding of edges block by block.
Read model:	Read a generating class in the appearing dialog window and make the graph of that model.
Graph for model number:	Read an integer in the appearing dialog window and make the graph for the model with that number.
Print the model, `'p'`:	Print the model for the graph.
Test graph against 'Base', `'t'`:	Test the model for the graph against the **base** model.
Print all models, `'a'`:	Print all models in the CoCo object.
Add Fill In:	Add a fill in to the graph.
Current, `'c'`:	Make the model for the graph the **current** model.
Base, `'b'`:	Make the model for the graph the **base** model.
Backward, `'B'`:	Do a "Backward" in the CoCo object.
Forward, `'F'`:	Do a "Forward" in the CoCo object.
Model Dynamic Spin Plot:	Make a "Model Dynamic Spin Plot" of three table values. The values to plot can be selected in appearing dialog windows.
Plot histogram:	Make a histogram. The value to use can be selected in the appearing dialog windows.
X-Y-plot of two cell-values:	Plot two table values against each other. The values to plot can be selected in appearing dialog windows.
Spin-plot of three cell-values:	Make a spin plot of three table values. The values to plot can be selected in appearing dialog windows.
Print table:	Print a table of values. The command in the appearing dialog window can be edited.
Global search:	Do a global search by the EH-procedure. The command in the appearing dialog window can be edited.
Plot search result:	Make an association diagram for each model in the resulting classes of a global search by the EH-procedure.
Status of CoCo-object:	Print the status of the CoCo object.
Names:	Return the name list.
Positions:	Return the position list.
Edges:	Return the edge list.
Static, `'s'`:	Switch static mode.
Grid, `'g'`:	Switch drawing of the grid.
Redraw, `'r'`:	Redraw the graph.

9.4 Key Events

The following actions can be performed by hitting keys in the association diagram:

- `‘s’`: Switch “Static mode”: A new graph is created when edges are dropped or added.
- `‘g’`: Switch the drawing of a grid.
- `‘i’`: Start or stop idling.
- `‘r’`: Redraw the graph.
- `‘l’`: Set variable-label for the vertex closest to the mouse. The label is entered in the appearing dialog window.
- `‘u’`: Undo the last vertex or vertex label move.
- `‘U’`: Skip a position in the undo list.
- `‘z’`: Redo the last undone move.
- `‘Z’`: Skip a position in the redo list.
- `‘y’`: Adjust vertex and vertex label positions to a grid with width `5`.
- `‘Y’`: Adjust block positions to a grid with width `5`.
- `‘W’`: Delete the block.
- `‘c’`: Make the model for the graph the **current** model.
- `‘b’`: Make the model for the graph the **base** model.
- `‘p’`: Print the model for the graph.
- `‘t’`: Test the model for the graph against the **base** model.
- `‘a’`: Print all models in the CoCo object.
- `‘e’`: Label edge with the p-value for the test of the model with the edge removed against the model for the graph.
- `‘-’`: Draw a dashed line from the edge label to the edge. The closest edge is selected.
- `‘=’`: Fix the position of the edge label. The closest edge is selected.
- `‘_’`: Draw a dashed line from the vertex label to the vertex. The closest vertex is selected.
- `‘+’`: Fix the position of the vertex label. The closest vertex is selected.
- `‘E’`: Remove edge label.
- `‘L’`: Draw all edges with a width inverse proportionally to the p-value for removal of the edge.
- `‘P’`: Do a block-wise backward elimination.
- `‘Q’`: Do a block-wise forward selection.

'`B`': Do a "Backward" in the CoCo object.

'`F`': Do a "Forward" in the CoCo object.

'`x`': Switch `:graph-coherent`.

'`h`': Switch `:graph-headlong`.

'`o`': Switch `:graph-random-order`.

'`f`': Switch `:graph-follow`.

'`d`': Switch `:graph-decomposable-mode`.

'`m`': Switch `:graph-most-extreme`.

'`H`': Model manager: Hide association diagram.

'`O`': Model manager: Open association diagram.

'`C`': Model manager: Close association diagram.

'`>`': Model manager: Update model manager.

'`<`': Model manager: Update screen.

Chapter 10

Messages to the Association Diagram

The *CoCo Graph Object* is a specialization of the *Association Diagram Object* and of the *CoCo Model Object*, i.e., the CoCo graph object inherits methods and slots from both the CoCo Model Object and the Association Diagram Object.

The methods for creation of graphs, for deletion and addition of edges and for tests are only implemented for the CoCo graph object. Unless otherwise stated, the methods in this chapter are implemented for the *Association Diagram Proto*. The Association Diagram Proto is a specialization of the prototype *Drag Graph Proto*, a general prototype with movable vertices and edges between some vertices such that the edges follows the vertices when the vertices are moved. Most of the methods for the Association Diagram Proto are inherited from Drag Graph Proto.

10.1 Creating a CoCo Graph Object

These methods and functions are only implemented for the CoCo graph object:

`coco-model-proto :make-graph &key (`⟨*location*⟩` nil) (`⟨*size*⟩` nil)`
  `(`⟨*title*⟩` nil)`
`coco-proto :make-graph &key (`⟨*model*⟩` nil) (`⟨*location*⟩` nil)`
  `(`⟨*size*⟩` nil) (`⟨*title*⟩` nil)`
`coco-graph-window-proto :return-child-coco-graph-window`
  `&key (`⟨*model*⟩` nil) (`⟨*location*⟩` nil) (`⟨*size*⟩` nil) (`⟨*title*⟩` nil)`
`coco-proto :plot-eh-search-result`
`coco-graph-window-proto :plot-eh-search-result`

The message `coco-model-proto :make-graph` to a CoCo model object will return the graph for the model object. The keyword arguments ⟨*location*⟩, ⟨*size*⟩

and ⟨*title*⟩ are for setting the location, size and title respectively of the graph. The message `coco-proto :make-graph` without the keyword argument ⟨*model*⟩ returns the graph for the **current** model. If the keyword argument ⟨*model*⟩ is `'base` (`'last`), then the graph for the **base** (**last**) model is returned. If ⟨*model*⟩ is a number, then the graph for the model with the number ⟨*model*⟩ is returned. If the argument ⟨*model*⟩ is a single character string with a model written as models are written in CoCo, then this model is read into the CoCo-object, and a graph for that model is returned.

The message `coco-proto :plot-eh-search-result` to a CoCo object will plot the result of a global search. One association diagram is made for each model in the class of minimal acceptable models and for each model in the class of maximal rejected models.

The method `:return-child-coco-graph-window` to a graph object will return an association diagram for the **current** model (unless the keyword argument ⟨*model*⟩ is given) with positions and labels etc. of the graph the message is send to. The keyword argument ⟨*model*⟩ can be a number, a text string, `'current`, `'base` or `'last`. That the returned graph has positions and labels of the graph to which the message is sent, means that the parent and child graph share the list containing the *positions* and the list containing the *names* (see section 10.8.1); If the position of a vertex in one graph is changed, then the position of the vertex is also changed in the parent and child graphs. The change in the parent and the child graph will first be seen when these other graphs are redrawn. Analogously with labels. The method is used when dropping edges, adding edges, etc.

The method `coco-graph-window-proto :plot-eh-search-result` will plot the result of an EH-search in association diagrams with positions and labels of the diagram to which the message is sent.

When new graphs are created by editing existing graphs, then the new graphs are not bound to variables. The graph, association diagram, resulting of the above messages can be bound to a variable by selecting **Name graph** from the graph menu. The default name of the graph-object is `last-graph-object`. Messages can then be sent to the graph from the Lisp-listener.

`:isnew &key` ⟨*location*⟩ ⟨*title*⟩

`:title &optional string`

`:size &optional width height`

`:location &optional (`⟨*left*⟩ `nil) (`⟨*top*⟩ `nil)`

`:close`

`:remove`

The messages `:title`, `:size`, `:location`, `:close` and `:remove` are some of the more useful messages inherited from the `graph-window-proto`. See Tierney (1990).

10.2 Editing the Layout: Moving Vertices, Labels and Blocks

10.2.1 Vertices and Vertex-labels

`:vertex-position` ⟨*variable-name*⟩ `&optional (`⟨*position*⟩ `nil set)`
`&key (`⟨*redraw*⟩ `nil)`
`:vertex-label` ⟨*variable-name*⟩ `&optional (`⟨*label*⟩ `nil set)`
`&key (`⟨*redraw*⟩ `nil)`
`:vertex-type` ⟨*variable-name*⟩ `&optional (`⟨*type*⟩ `nil set)`
`&key (`⟨*redraw*⟩ `nil)`
`:vertex-label-position` ⟨*variable-name*⟩ `&optional (`⟨*position*⟩ `nil set)`
`&key (`⟨*redraw*⟩ `nil)`
`:fix-vertex-label-position` ⟨*variable-name*⟩ `&optional (`⟨*val*⟩ `nil set)`
`&key (`⟨*redraw*⟩ `nil)`
`:vertex-label-arrow` ⟨*variable-name*⟩ `&optional (`⟨*val*⟩ `nil set)`
`&key (`⟨*redraw*⟩ `nil)`
`:vertex-index` ⟨*variable-name*⟩

Vertex-positions

The position of a vertex can be changed by positioning the mouse at the vertex, pressing first the "SHIFT" (extend modifier) key and then a mouse button, and then dragging the vertex. The vertex will first start to move when the mouse has been moved more than 4 pixels from where the button was clicked. The message `:vertex-position` will set or return the position of the vertex ⟨*variable-name*⟩. ⟨*variable-name*⟩ can be the name of the variable by a text string, e.g., `"A"` or `":Smoking"`, or it can be the index of a vertex. Because the index of a vertex is found from a text string by comparing text strings, the hole vertex name has to be given the association diagrams, and not as in the CoCo-object, a unique prefix of the name. The index of a vertex (or variable) is the number minus one of the vertex in the ordered list given by the specification of the table in the CoCo object. The argument ⟨*position*⟩ is a list of two or three reals between -50 and 50. The index of a vertex can be returned by `:vertex-index`.

After changing the position by `:vertex-position` the graph is only redrawn, if the keyword argument `:redraw T` is given.

Vertex-labels and label-positions

The position of the vertex-label is relative to the vertex. By default the vertex-label is placed 10 percent further away from the center of the window than the corresponding vertex. If the vertex label position is set, then the vector from the vertex to the vertex label will stay fixed while the vertex is moved. To move a vertex-label: Position the mouse at the vertex, press both the "SHIFT" (extend modifier) and "CONTROL" (option modifier) keys and a mouse button, and drag the label. Note that the selected vertex-label is determined by the vertex closets to the mouse and not by the label closest to the mouse. This helps avoiding mixing up labels. The position of the vertex-label relative to the vertex can be set by the message `:vertex-label-position`.

To set a vertex-label: Position the mouse at a vertex, and press `'l'`. Enter the label in the appearing dialog-window. Or use the message `:vertex-label` to set or return a variable-label. The optional argument ⟨*label*⟩ has to be a text-string.

The position of a vertex label can be fixed by `:fix-vertex-label-position` ⟨*variable-name*⟩ `T` or the key event `'+'`. If the position of a vertex label is fixed, then the position of the label will stay fixed when the vertex is moved, else the label will follow the vertex. If the value set or returned by `:vertex-label-arrow` is true, then a dashed line from the vertex to the label is drawn. The key event `'_'` at the vertex will switch the drawing of this line for the vertex.

See later sections about adjusting the vertex positions to a grid and about undoing vertex-moves.

Deleting variables

Variables can be deleted by dragging vertices to the wastebasket, right to and below the right lower corner of the plot. A new model with the variables removed is created and plotted in an association diagram. Add variables be selecting **Read model** from the graph menu and edit the generating class for the graph.

Ghost points: When a vertex is deleted from a graph the variable will be removed from the model and the drawing of the vertex omitted. But the vertex is still accessible in the graph by the mouse. Edges can be dragged from a *ghost point* of the vertex and the position of the vertex can be changed. The ghost point has the same position as the vertex of the ghost point before the vertex was deleted. The variable can be added to the graph again first by dragging an edge from the ghost of the vertex to another vertex (adds both the main-effects of the two variables of the edge and the interaction between the variable and the variable, to which the edge is dragged) and then delete the edge. If the deleted variable is moved in parent or offspring graphs, then the ghost point will also move.

10.2.2 Edges and Edge-labels

`:edge-label` ⟨*edge-key*⟩ `&optional (`⟨*val*⟩ `nil set) &key (`⟨*redraw*⟩ `nil)`
`:edge-label-position` ⟨*edge-key*⟩ `&optional (`⟨*position*⟩ `nil)`
  `&key (`⟨*redraw*⟩ `nil)`
`:edge-label-offset` ⟨*edge-key*⟩ `&optional (`⟨*position*⟩ `nil)`
  `&key (`⟨*redraw*⟩ `nil)`

`:fix-edge-label-position` ⟨*edge-key*⟩ `&optional (`⟨*val*⟩ `nil set)`
  `&key (`⟨*redraw*⟩ `nil)`
`:edge-label-arrow` ⟨*edge-key*⟩ `&optional (`⟨*val*⟩ `nil set)`
  `&key (`⟨*redraw*⟩ `nil)`
`:edge-width` ⟨*edge-key*⟩ `&optional (`⟨*val*⟩ `nil set) &key (`⟨*redraw*⟩ `nil)`
`:edge-test` ⟨*edge-key*⟩ `&optional (`⟨*val*⟩ `nil set) &key (`⟨*redraw*⟩ `nil)`
`:edge-status` ⟨*edge-key*⟩ `&optional (`⟨*val*⟩ `nil set) &key (`⟨*redraw*⟩ `nil)`
`:edge-index` ⟨*vertex-pair*⟩

`:set-edge-label-nth` ⟨*edge-index*⟩
`:drop-edge-label-nth` ⟨*edge-index*⟩

To label an edge with the p-value for the test of removal of the edge, i.e., test the two variables conditionally independent given the other variables in the graph: Position the mouse at the edge to label, press a mouse button while "SHIFT" (extend modifier). Release. If the mouse is moved while the button is down then a vertex will move. Or press `'e'` at the edge. When the graph is redrawn the edges are then drawn with a width "inverse proportional" to the p-value. See section 10.4 about selecting and formatting the p-value or statistic.

To move an edge label: Position the mouse at the edge label, press a mouse button while pressing both the "SHIFT" (extend modifier) and the "CONTROL" (option modifier) keys and drag the label. The edge-label can also be moved by adding another label to the edge, and type `'r'` for "Redraw". The previously set label of that edge is then removed.

The position of an edge label can be fixed by `:fix-edge-label-position` ⟨*edge-key*⟩ `T` or the key event `'='`. Else the edge label will follow the middle of the edge when the vertices of the edge are moved. A dashed line from the label to the middle of the edge can be set to be drawn by `:edge-label-arrow` ⟨*edge-key*⟩ `T`, the key event `'-'`.

To remove an edge label: Position the mouse at the edge, press a mouse button while pressing both the "SHIFT" (extend modifier) and the "CONTROL" (option modifier) keys. Then release the mouse button. If the mouse is moved while the button is down a vertex-, edge- or block-label will be moved. Or press `'E'` at the edge.

The edge label is set with the method `:set-edge-label-nth` and removed with the method `:drop-edge-label-nth`. The argument ⟨*edge-index*⟩ to these methods is the index of the edge. The index of an edge can be found by the method `:edge-index`, where the argument ⟨*vertex-pair*⟩ is a list with the names of the two vertices.

The method `:edge-width` will set or return the width of an edge. `:edge-test` will set or return, if computed, a list containing the result of testing the two variables of the edge conditionally independent given the other variables in the graph and the status of that test. The test is computed by `:compute-test`, and, if the test is possible, then the status of the test will be `nil`, else the test will be `nil` and the status will be `'non-decomposable 'not-submodel-of-base`, `'fix-edge`, `'coherence` or `'future-edge`. The status of the edge can be set or returned by the method `:edge-status`.

`:edge-label` will set or return, if the edge is labeled, a list with the offset of the label, a boolean for whether to fix the position of the edge label and a boolean for whether to draw a line from the edge to the edge label. The offset, set and returned by the method `:edge-label-offset`, of the label is the vector from the middle of the edge to the position of the label. The position of the label is set and returned by the method `:edge-label-position`.

10.2.3 Cleaning Edges

`:remove-tests`

The method `:remove-tests` will dispose of the computed tests, i.e., dispose of the labels and set the edge widths to one. The key-event `'r'` in the graph-window and the message `:redraw-graph` will then draw the graph with standard edges.

10.2.4 Blocks and Block-labels

`:stratum` ⟨*variable-name*⟩ `&optional (`⟨*block-key*⟩ `nil set)`
  `&key (`⟨*redraw*⟩ `nil)`
`:block-position` ⟨*block-key*⟩ `&optional (`⟨*position*⟩ `nil set)`
  `&key (`⟨*redraw*⟩ `nil)`
`:block-label` ⟨*block-key*⟩ `&optional (`⟨*label*⟩ `nil set)`
  `&key (`⟨*redraw*⟩ `nil)`
`:block-label-position` ⟨*block-key*⟩ `&optional (`⟨*position*⟩ `nil set)`
  `&key (`⟨*redraw*⟩ `nil)`
`:block-index` ⟨*block-key*⟩

See chapter 12 about block models.

10.2.5 Smoothing and Rescaling the Graph

`:grid &optional (`⟨*val*⟩ `nil set)`

`:adjust-vertices-to-grid &key (`⟨*vertex*⟩ `nil) (`⟨*delta*⟩ `1)`
  `(`⟨*redraw*⟩ `nil)`
`:adjust-blocks-to-grid &key (`⟨*delta*⟩ `1) (`⟨*redraw*⟩ `nil)`
`:rescale-vertex-positions &key (`⟨*vertex*⟩ `nil) (`⟨*scale*⟩ `1)`
  `(`⟨*redraw*⟩ `nil)`
`:rescale-block-positions &key (`⟨*scale*⟩ `1) (`⟨*redraw*⟩ `nil)`

The message `:adjust-vertices-to-grid :delta` ⟨X⟩ to an association diagram object will adjust the vertex- and vertex-label positions to a grid with the width ⟨X⟩ between the points. The key-event `'y'` in the graph-window will call this method with `:adjust-vertices-to-grid :delta 5`. If the keyword argument ⟨*vertex*⟩ is given then only the vertex with index ⟨*vertex*⟩ is adjusted, else all vertices are adjusted. Analogously will the message `:adjust-blocks-to-grid` adjust corners of blocks. The key-event `‘Y’` in the graph-window will adjust the block corners to a grid with a width of `5`.

The message `:grid` will turn on or off the drawing of a grid with a width of `10`. The key-event `‘g’` will switch `'grid`.

The `:rescale-vertex-positions :scale` ⟨*X*⟩ will transform the vertex-positions by multiplying the distance from the center of the window to the vertex by ⟨*X*⟩. Analogously with the message `:rescale-block-positions` for block-corners.

10.2.6 Undo Vertex Moves

`:push-vertex-undo` ⟨*number*⟩
`:push-block-points-undo` ⟨*n*⟩
`:undo-move`
`:skip-undo-move`
`:redo-move`
`:skip-redo-move`

The message `:undo-move` will undo the last move of a vertex or vertex-label. The key event `‘u’` in the graph will also undo the vertex move. A vertex move undone can be repeated by the message `:redo-move` or the key event `‘z’`.

When a vertex is moved, the old position of the vertex is pushed into the list `:slot-value 'undo-positions` by the method `:push-vertex-undo`. The message `:undo-move` will push the vertex position into the list `:slot-value 'redo-positions`, pop the first vertex position from the list `:slot-value 'undo-positions` and make the vertex position that position. Analogously for redo: The message `:redo-move` will push the vertex position into the list `:slot-value 'undo-positions`, pop the first vertex position from the list `:slot-value 'redo-positions` and make the vertex position that position.

Positions to undo or redo can be moved between the two lists `:slot-value 'undo-positions` and `:slot-value 'redo-positions` without changing the association diagram by the methods `:skip-undo-move` and `:skip-redo-move`.

Undo of move of blocks is not satisfactorily implemented. No undo of block, block-label and edge-label moves is possible. The moves of vertices when dragging a block are undone one by one.

10.2.7 Redrawing Exposed Graphs

`:redraw-graph &optional (`⟨*radius*⟩ `4)`
`:redraw`

The `:redraw` method will redraw the diagram by the method `:redraw-graph`:

```
(defmeth association-diagram-proto :redraw ()
  (send self :redraw-graph)
  )
```

The message `:redraw` is send to a graph, when the graph is opened and when graphs are moved on top of the graph, i.e., when the graph is exposed.

When the method `:redraw` is redrawing graphs, then in a model selection on association diagrams (see later chapters) with "Static" set to `TRUE` there will be used a lot of computing time to redraw graphs. Thus to get a better performance of the tool, the method `:redraw` can, e.g., of all graphs be redefined to not redraw the graphs by the expression

```
(defmeth association-diagram-proto :redraw ()
  )
```

When dumping the graph on a PostScript file, the `:redraw` method of the graph should redraw the graph by the method `:redraw-graph`.

10.3 Editing the Model: Adding and Removing Edges

`:graph-drop-edge-nth &optional` ⟨*p*⟩ `&key (`⟨*edges*⟩ `nil) (`⟨*point*⟩ `nil)`
`(`⟨*x-move*⟩ `0)`
`:graph-add-edge &optional` ⟨*p1*⟩ ⟨*p2*⟩ `&key (`⟨*edges*⟩ `nil set)`
`(`⟨*x-move*⟩ `0)`
`:add-fill-in`

The methods `:graph-drop-edge-nth`, `:graph-add-edge` and `:add-fill-in` are only implemented for the coco-graph-window-proto.

An edge can be removed from the graph by clicking the edge with the mouse. Keep the mouse fixed while pressing the button; if the mouse is moved more than 4 pixels then an edge will be added. If "Static" is not turned off, see later sections, then a new graph with the edge removed is created. When the mouse is clicked without moving, then, if the mouse is close enough to an edge, the method `:graph-drop-edge-nth` is called with the argument ⟨*p*⟩ the index of the closest edge. An edge is close enough, if the angle between the lines from the mouse to the two vertices determining the edge is greater than 135 degrees. The method `:graph-drop-edge-nth` can also be used to drop several edges or vertices by using the keyword arguments ⟨*edges*⟩ and ⟨*point*⟩.

To add an edge: Position the mouse at a vertex, press the mouse button and drag a line to another vertex. Then release the mouse button. The method `:graph-add-edge` is then called with ⟨*p1*⟩ and ⟨*p2*⟩ the indices of the vertices from and to which the line is dragged.

The method `:add-fill-in` adds a fill-in to the graph if the graph is non-decomposable.

10.4 Selecting the p-value and Formatting the Edge Label

These methods and functions are only implemented for the CoCo graph object:

`:select-p-value` ⟨*test*⟩ `&key` (⟨*print-test*⟩ `nil`)
`:p-to-width` ⟨*p-value*⟩
`:format-p-value` ⟨*p-value*⟩

The method `:select-p-value` extracts and computes from the list of values returned by the message `:compute-test` the p-value to be used in model selection on graphs (see the next section) and when setting edge-labels.

For a test with Goodman and Kruskal's Gamma coefficient available, the p-value for this statistic is extracted, else the p-value for *Pearson's chi-square* χ^2 is used by the following piece of code:

```
(defmeth coco-graph-window-proto :select-p-value
  (test &key~(print-test nil))
  (if print-test (print-test test))
  (let ((number-of-cases       (nth 0 (car test)))
        (df                    (nth 1 (car test)))
        (adj                   (if (> (nth 2 (car test)) 0)
                                   (nth 2 (car test)) 0))
        (number-of-tables      (nth 3 (car test)))
        (deviance              (nth 0 (cadr test)))
        (exact-p-deviance      (nth 1 (cadr test)))
        (square                (nth 2 (cadr test)))
        (exact-p-square        (nth 3 (cadr test)))
        (power                 (nth 4 (cadr test)))
        (exact-p-power         (nth 5 (cadr test)))
        (gamma                 (nth 6 (cadr test)))
        (gamma-s               (nth 7 (cadr test)))
        (gamma-s-1             (nth 8 (cadr test)))
        (exact-p-gamma-1-sided  (nth 9 (cadr test)))
        (exact-p-gamma-2-sided  (nth 10 (cadr test)))
        (df-real               (nth 11 (cadr test))))
    (if (and (> gamma -2) (< gamma 2))
        (if (and (> number-of-tables 0)
                 (> exact-p-gamma-2-sided -1))
            exact-p-gamma-2-sided
          (if (> gamma-s 0)
```

```
              (* 2 (- 1 (normal-cdf (/ (abs gamma)
                                       (sqrt gamma-s))))) 0))
      (if (and (> number-of-tables 0) (> exact-p-deviance -1))
          exact-p-deviance
        (- 1 (if (> (- df adj) 0)
                 (chisq-cdf deviance (- df adj)) 0)))))
  )
```

If exact p-values are available, then they are used. Edit the code for `:select-p-value` appropriately, if you want to use other statistics. Note that the result of `:select-p-value` is used in the methods `:p-to-width` and `:format-p-value` for setting edge widths and edge labels and in the models selection on the graph.

The method `:p-to-width` computes from the p-value returned by the method `:select-p-value` the edge width used when drawing the edges and arrows. The default method adds one unit to the edge width for approximately every halving of the p-value:

```
(defmeth association-diagram-proto :p-to-width (p)
  (length (which
    (< p (list 2 0.40 0.20 0.10 0.05 0.025 0.01 0.005
               0.0025 0.001 0.0005 0.00025 0.0001 0.00005
               0.000025 0.00001 0.000005 0.0000025 0.000001
               0.0000005 0.00000025 0.0000001 0.00000005))))
  )
```

The method can be edited appropriately. It should return positive integer values.

The method `:format-p-value` is used to formatting the p-value when setting edge-labels:

```
(defmeth association-diagram-proto :format-p-value (p)
  (format nil "P = ~7,5f" p)
  )
```

If you instead of labels $P = ...$ for the graph `my-graph` want labels like *p-value: = 0.00123*, then type

```
(defmeth my-graph :format-p-value (p)
  (format nil "p-value: = ~7,5f" p)
  )
```

in the Lisp-listener.

Item	Color
vertex	'blue
vertex-label	'cyan
new-edge	'yellow
not-fitted	'green
fitted	'blue
non-decomposable	'magenta
not-submodel-of-base	'red
coherence	'cyan
error-edge	'black
fix-edge	'yellow
grid	'red
rotate	'red
block	'green
future-edge	'blue
controls	'green
error	'black

Table 10.1: *Default colors.*

10.5 Colors

`:item-color` ⟨*item*⟩ `&optional (`⟨*value*⟩ `nil set)`
`:draw-color` ⟨*color*⟩
`:set-edge-color` ⟨*edge*⟩

The method `:item-color` returns or sets the color of an item. E.g., the message

```
(send graph-1 :item-color 'vertex-label 'red)
```

is used to set the color for vertex-labels to `'gold`.

The colors are shared by all graph objects. The message `:slot-value 'colors` returns the list `*default-colors*` common to all created graph objects. If you want different colors for different objects, then a message like

```
(send graph-1 :slot-value 'colors (copy-list *default-colors*))
```

will give `graph-1` a list of colors not shared by other graph objects. If the color on an item in `graph-1` then is changed, then the item color is only changed for `graph-1` and not for other graphs.

Available colors can be printed by `*default-colors*`. Some additional colors can be found in the file `mycolors.lsp` in the directory `$XCOCOLIB`. These colors can be loaded by

```
(load (concatenate 'string *xlisp-cocohome* "mycolors"))
```

The messages `:draw-color` and `:set-edge-color` are used when drawing the graph.

10.6 The 3-dimensional Plot: Rotation

`:transformation &optional (`⟨*val*⟩` nil set)`
`:do-hand-rotate` ⟨*x*⟩ ⟨*y*⟩ ⟨*m1*⟩ ⟨*m2*⟩

The graph is not 2-dimensional, but 3-dimensional. The graph can be rotated by positioning the mouse in the graph, press both a mouse button and the "CONTROL" (option modifier) key, and then drag the graph. The graph will then rotate around the center of the window in the same way as a sphere was dragged. The method `:do-hand-rotate` is used to drag the graph when rotating. The graph can also be rotated by control buttons, see the next section.

The stored positions of the vertices, labels, etc. are not changed, when rotating, but the coordinates are transformed by a rotation matrix when the diagram is drawn. The rotation matrix can be set or returned by the method `:transformation`. By the message `:transformation nil`, the rotation is cancelled. Use this message, if you accidentlly rotate the graph.

Some functions for computing the coordinates in or from the rotated system and for updating the rotation matrix:

`:project` ⟨*position*⟩
`:inverse-project` ⟨*position*⟩
`:canvas-to-sphere` ⟨*x*⟩ ⟨*y*⟩ `rad`
`:apply-transformation` ⟨*trans*⟩ `&optional (draw-box nil)`

The method `:project` will project the stored positions of a point onto the plotted coordinate system. The method `:inverse-project` will transform a 2-dimensional vector from the graph to the stored 3-dimensional system. This method is used when dragging vertices etc. The method `:apply-transformation` will multiply the existing transformation matrix by the matrix ⟨*trans*⟩.

When the graph is rotated by pressing both a mouse button and the "CONTROL" (option modifier) key and dragging the mouse then the message `:do-hand-rotate` is sent. This method will use the method `:canvas-to-sphere` to find the change of the transformation matrix. See also Tierney (1990).

Idling the graph

`:add-idle`
`:angle &optional (`⟨*val*⟩` nil set)`
`:do-idle`

`:idle-on (&optional 'on)`

The message `:add-idle` to a graph will add methods for idling of the graph, i.e., continuous rotation of the graph. Idling can be started by pressing a mouse button and "CONTROL" (option modifier) while the mouse is fixed. Idling can be stopped by rotating the graph. Idling can also be started and stopped by the message `:idle-on`. The angle set or returned by the method `:angle` controls the angle the graph is rotated for each step in the idling. If the graph has been rotated, the idling will start by first doing 50 step by the last change of rotation, and then do a random idling, else a random idling is performed.

Some functions for computing the rotation-matrix and doing the idling:

`:tour-step`
`:tour-on &rest` ⟨*args*⟩

`(sphere-rand` ⟨*n*⟩`)`

The function `(sphere-rand)` generates random points on the ⟨*n*⟩-dimensional unit sphere.

Touring of the graph is performed by the method `:tour-step` and set on or of by `:tour-on` (same as `:idle-on`). Methods from Tierney (1990) are modified to the association diagrams.

10.7 Saving Images

The methods of this chapter are used to save the picture of diagrams for, e.g., inclusion of the picture in a document. For saving the positions, labels, etc. of a diagram, use the method `:save`, see section 10.8.2.

10.7.1 PostScript: Bit-map dumps

`:save-image` ⟨*file-name*⟩

The message `:save-image` to a graph will on Unix-systems save the association digram in the PostScript file ⟨*file-name*⟩ by a mono-chrome bit-map dump. The message can be selected from the graph menu. This guide contains several figures created in this way, e.g., 1.1 and 9.1.

Before dumping the graph on a PostScript file, the `:redraw` method of the graph should be redefined to redraw the graph:

```
(defmeth the-graph :redraw () (send self :redraw-graph))
```

or redefining the method for all graphs by the expression:

```
(defmeth association-diagram-proto :redraw ()
  (send self :redraw-graph)
  )
```

10.7.2 TeX: Picture-code

`:dump-tex &key (`⟨*file-name*⟩ `nil) (`⟨*radius*⟩ `4) (`⟨*unadjusted*⟩ `nil)`
`(`⟨*box*⟩ `nil) (`⟨*centered*⟩ `nil) (`⟨*char-width*⟩ `5) (`⟨*char-height*⟩ `10)`
`(`⟨*extra-width*⟩ `2) (`⟨*extra-height*⟩ `1) (`⟨*size*⟩ `"(360,360)(0,0)")`

The message `:dump-tex` to a graph will write TeX code for the association diagram[1]. The method will generate TeX code for drawing vertices, edges, arrows, vertex-labels, edge-labels and blocks in the *picture* environment. If the keyword argument ⟨*file-name*⟩ is given, then the TeX code is written on the file ⟨*file-name*⟩, else the TeX code is dumped on standard output. The keyword argument ⟨*size*⟩ is passed as a string to the `\begin{picture}` line.

If the keyword argument ⟨*box*⟩ is set to `nil`, the default, then the variables are drawn as points and the vertex-labels at positions determined by the positions of the labels in the graph, see figure 10.1 *a)* and *b)*. The keyword argument ⟨*radius*⟩ controls the radius of vertices. Arrows and edges are in each end shortened by the length ⟨*radius*⟩.

The code written for discrete variables is `\Dot{`⟨*radius*⟩`}`, the code written for ordinal variables is `\Ordinal{`⟨*radius*⟩`}` and the code written for continuous variables is `\Circle{`⟨*radius*⟩`}`, which by default is set to respectively `\circle*{`⟨*radius*⟩`}`, `\circle*{`⟨*radius*⟩`}` and `\circle{`⟨*radius*⟩`}`.

If ⟨*box*⟩ is set to `TRUE`, then the variables are drawn as boxes with the labels in the boxes, see the figures 14.1 and 10.1 *c)* and *d)*. If ⟨*centered*⟩ is set to `TRUE`, then the terminating points of the edges will be centered on the boarders of the boxes, see 10.1 *d)*, else the edges will have direction to the centers of the vertices as in figure 14.1 and 10.1 *c)*.

Discrete variables are drawn by `\framebox`, ordinal variables by `\dashbox`, and continuous variables are drawn by `\oval`.

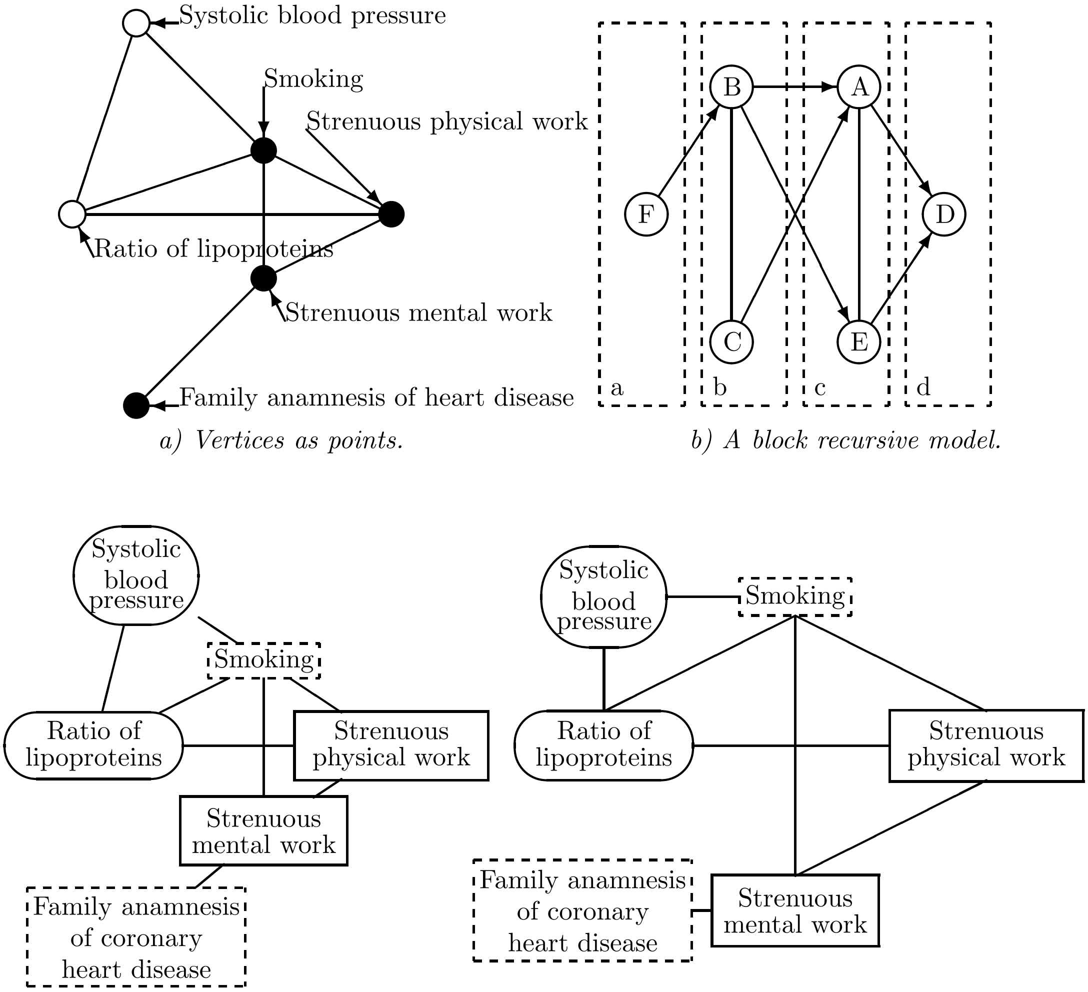
The number of lines for a label is one plus the number of times the character \ is found in the label. The label length is set after the longest line in the label: The longest substring in the label, where the label is split into in substrings by the character \.

Let *label-height* denote the number of occurrences of the character \ in the variable label plus one times ⟨*char-height*⟩, and let *label-width* denote the maximal number of characters between two characters \ (or ends of the label) in the variable label times ⟨*char-width*⟩. Then the height of the box of the label is *label-height* plus ⟨*extra-height*⟩, and the width of the box is *label-width* plus ⟨*extra-height*⟩.

The label is positioned half the *label-height* and half the *label-width* from the center of the box, the vertex position.

The individual label sizes can only be controlled by the characters in the label (and by the parameters common to all the labels in the graph, the parameters

[1] Later an option ⟨*postscript*⟩ for getting PostScript code from this message may be available.



a) Vertices as points. *b) A block recursive model.*

c) Vertices as boxes. *d) Centered lines.*

Figure 10.1: Figures created with `:dump-tex`.

given by the keyword arguments ⟨*char-width*⟩, ⟨*char-height*⟩, ⟨*extra-width*⟩ and ⟨*extra-height*⟩). If you find the box for a label too small, then insert some extra line separators, \, and some extra characters, e.g., blanks. If the box is too large, then omit some parts of the label in the graph label, and edit the label to the correct label in the dumped TeX code. In the dumped TeX code the \ in the labels should be edited to \\ to divide the label on more lines. By changing the argument `c` in `\shortstack[c]` to `l` or `r` the lines of a label can be aligned to the left or right respectively.

The `\framebox` drawn by TeX seems a little too big, a line width. This could be edited in the generated code. Edges and arrows terminating at corners of ovals, `\oval`, are too short, since the terminating point of an edge is found as the crossing of the edge and a rectangular box. Edges are only terminating at corners when they not are centered, i.e., when ⟨*centered*⟩ not is set to `TRUE`.

TeX is only able to handle lines and arrows with a slope, a fraction of two positive integers less than 7 (5 for arrows). With the keyword argument ⟨*unadjusted*⟩ set to `nil`, the default, the slopes of lines and arrows are adjusted to the closest legal slopes. The line (arrow) is rotated a little around the center of the line (arrow). The length of the line (arrow) is unchanged.

If the keyword argument ⟨*unadjusted*⟩ set to `TRUE`, then the slopes of lines and arrows are unmodified. You then either have to adjust the vertex positions to a grid before dumping the TeX code (by, e.g., the messages `:adjust-vertices-to-grid :delta` ⟨*15*⟩ and `:adjust-blocks-to-grid :delta` ⟨*15*⟩) so that you get legal slopes, or use EmTeX, or you have to get a hold on the `pspic.sty`[2] style file by Kresten Krab Thorup, which will enable you to draw lines and arrows of any slope. Adjusting to a course grid will only help you, if the termination points of edges are not centered on the boarders of the boxes, i.e., if ⟨*centered*⟩ is not set to `TRUE`.

Since the length of non-vertically edges and arrows also is measured by their horizontal component when the style file `pspic.sty` is used, then the length of almost vertically edges and arrows might be due to some rounding off errors.

The names of the vertices are inserted as comments at the code of the vertices and at the code of the edges to facilitate editing of the generated code.

The TeX code of figure 10.1 is made by the message `:dump-tex` to association diagrams for the table of the introduction. The Lisp code for creating this figure can be found in the file `*xlisp-cocohome*/TestTeX.lsp`, where the variable `*xlisp-cocohome*` is bound and can be echoed in Xlisp+CoCo.

[2] The style file `pspic.sty` by Kresten Krab Thorup, *krab@iesd.auc.dk*, can be obtained by anonymous *ftp* over internet from *ftp.iesd.auc.dk*. It is also found in the directory with the source code or executable of CoCo.

10.7.3 Changing the Position and Scaling of the Saved Diagram

Scaling and Position of the TeX Code

`:x-pos-to-tex` $\langle x \rangle$
`:y-pos-to-tex` $\langle y \rangle$

The methods `:x-pos-to-tex` $\langle x \rangle$ and `:y-pos-to-tex` $\langle y \rangle$ are used to transform the stored coordinates (and rotated) of the plot into the coordinates of the TeX-figure. The default methods will produce a Tex figure of size 160×160 units:

```
(defmeth association-diagram-proto :x-pos-to-tex (x)
  (* 2 (+ 50 x))
  )

(defmeth association-diagram-proto :y-pos-to-tex (y)
  (* 2 (- 50 y))
  )
```

The interval `[ -50, 50 ]` for the horizontal axis is mapped onto the interval `[ 0, 200 ]`, and the interval `[ -50, 50 ]` for the vertical axis is mapped onto the interval `[ 200, 0 ]`.

In the graph the second axis is oriented downwards, and in the Tex figure upwards. The upper left corner has in the stored graph the position `( -50, -50)`. This point has in the TeX figure the coordinates `( 0, 200)`. The coordinates of the lower right corner are in the graph `( -50, 50)` and in the TeX figure `( 0, 0)`

The methods `:x-pos-to-tex` and `:y-pos-to-tex` are edited as follows for the graphs of figure 10.1:

```
(defmeth graph-a :x-pos-to-tex (x) (* (/ 16 10) (+ 50 x)))
(defmeth graph-a :y-pos-to-tex (y) (+ (* (/ 16 10) (- 50 y)) 200))
(send graph-a :dump-tex :file-name "A.tex" :radius 5)

(defmeth graph-b :x-pos-to-tex (x) (+ (* (/ 16 10) (+ 50 x)) 200))
(defmeth graph-b :y-pos-to-tex (y) (+ (* (/ 16 10) (- 50 y)) 200))
(send graph-b :dump-tex :file-name "B.tex" :radius 8)

(defmeth graph-c :x-pos-to-tex (x) (* (/ 16 10) (+ 50 x)))
(defmeth graph-c :y-pos-to-tex (y) (* (/ 16 10) (- 50 y)))
(send graph-c :dump-tex :file-name "C.tex" :box T :char-height 12)

(defmeth graph-d :x-pos-to-tex (x) (+ (* (/ 16 10) (+ 50 x)) 200))
(defmeth graph-d :y-pos-to-tex (y) (* (/ 16 10) (- 50 y)))
```

```
(send graph-d :dump-tex :file-name "D.tex"
      :box T :char-height 12 :extra-height 2 :centered T)
```

E.g., the TeX code of `graph-d` is a figure of the size 160×160 units and moved 200 units to the right.

Methods for Generating the TeX Code

The rest of this section (and the rest of the chapter) is some implementation notes, and can be skipped.

`(gcd` ⟨*a*⟩ ⟨*b*⟩`)`
`(int-round` ⟨*a*⟩`)`

`:to-x-tex` ⟨*x*⟩
`:to-y-tex` ⟨*y*⟩
`:to-tex` ⟨*x*⟩
`:tex-draw-color` ⟨*f*⟩ ⟨*color*⟩
`:tex-line-type` ⟨*f*⟩ ⟨*type*⟩
`:tex-line-width` ⟨*f*⟩ ⟨*w*⟩
`:tex-draw-vertex` ⟨*f*⟩ ⟨*type*⟩ ⟨*x*⟩ ⟨*radius*⟩ ⟨*width*⟩
`:tex-draw-string` ⟨*f*⟩ ⟨*s*⟩ ⟨*x*⟩
`:tex-return-length` ⟨*x*⟩ ⟨*y*⟩ ⟨*a*⟩ ⟨*b*⟩ `&optional (d 0)`
`:tex-return-direction` ⟨*x*⟩ ⟨*y*⟩ ⟨*a*⟩ ⟨*b*⟩
`:tex-draw-vector` ⟨*f*⟩ ⟨*x*⟩ ⟨*y*⟩ ⟨*a*⟩ ⟨*b*⟩ ⟨*radius*⟩ ⟨*unadjusted*⟩
`:tex-draw-line` ⟨*f*⟩ ⟨*x*⟩ ⟨*y*⟩ ⟨*a*⟩ ⟨*b*⟩ ⟨*radius*⟩ ⟨*unadjusted*⟩
`:tex-draw-dashed-line` ⟨*f*⟩ ⟨*x*⟩ ⟨*y*⟩ ⟨*a*⟩ ⟨*b*⟩ ⟨*radius*⟩ ⟨*unadjusted*⟩
`:tex-draw-box` ⟨*f*⟩ ⟨*x*⟩ ⟨*y*⟩
`:tex-draw-vertex-in-box` ⟨*f*⟩ ⟨*type*⟩ ⟨*x*⟩ ⟨*extra-width*⟩ ⟨*extra-height*⟩ ⟨*label-size*⟩ ⟨*s*⟩
`:tex-draw-vertex-and-label` ⟨*f*⟩ ⟨*position*⟩ ⟨*name*⟩ ⟨*use-variable*⟩ ⟨*radius*⟩ ⟨*unadjusted*⟩ ⟨*box*⟩ ⟨*extra-width*⟩ ⟨*extra-height*⟩ ⟨*label-size*⟩
`:tex-draw-vertices` ⟨*f*⟩ ⟨*radius*⟩ ⟨*unadjusted*⟩ ⟨*box*⟩ ⟨*extra-width*⟩ ⟨*extra-height*⟩ ⟨*label-sizes*⟩
`:tex-draw-edge-label` ⟨*f*⟩ ⟨*edge*⟩ ⟨*unadjusted*⟩
`:tex-set-edge-color` ⟨*f*⟩ ⟨*edge*⟩
`:tex-cut-edge` ⟨*point*⟩ ⟨*box*⟩ ⟨*centered*⟩
`:tex-draw-thick-edge` ⟨*f*⟩ ⟨*a*⟩ ⟨*b*⟩ ⟨*width*⟩ ⟨*radius*⟩ ⟨*unadjusted*⟩ ⟨*box*⟩ ⟨*centered*⟩ ⟨*dx*⟩ ⟨*dy*⟩
`:tex-draw-arrow` ⟨*f*⟩ ⟨*a*⟩ ⟨*b*⟩ ⟨*width*⟩ ⟨*radius*⟩ ⟨*unadjusted*⟩ ⟨*box*⟩ ⟨*centered*⟩ ⟨*dx*⟩ ⟨*dy*⟩
`:tex-draw-edge` ⟨*f*⟩ ⟨*edge*⟩ ⟨*radius*⟩ ⟨*unadjusted*⟩ ⟨*box*⟩ ⟨*centered*⟩ ⟨*extra-width*⟩ ⟨*extra-height*⟩ ⟨*label-sizes*⟩
`:tex-draw-edges` ⟨*f*⟩ ⟨*radius*⟩ ⟨*unadjusted*⟩ ⟨*box*⟩ ⟨*centered*⟩ ⟨*extra-width*⟩ ⟨*extra-height*⟩ ⟨*label-sizes*⟩

`:tex-draw-rectangle-line` ⟨*f*⟩ ⟨*a*⟩ ⟨*b*⟩ ⟨*unadjusted*⟩
`:tex-draw-rectangle-box` ⟨*f*⟩ ⟨*a*⟩ ⟨*b*⟩
`:tex-draw-rectangle` ⟨*f*⟩ ⟨*a*⟩ ⟨*b*⟩ ⟨*c*⟩ ⟨*x*⟩ ⟨*y*⟩ ⟨*z*⟩ ⟨*unadjusted*⟩
`:tex-draw-block` ⟨*f*⟩ ⟨*block*⟩ ⟨*unadjusted*⟩
`:tex-draw-blocks` ⟨*f*⟩ ⟨*unadjusted*⟩

These methods are used for generating the TeX code of the graph. The structure of `:dump-tex` is very much that of `:redraw-graph`:

The method `:dump-tex` will write the *picture* code for the vertices, edges and blocks, all with labels, if any, by the methods `:tex-draw-vertices`, `:tex-draw-edges` and `:tex-draw-blocks`.

The method `:tex-draw-vertices` will write the code for each vertex with the label by the method `:tex-draw-vertex-and-label`, which will produce the code for the vertex by the method `:tex-draw-vertex` unless the vertices are to be drawn with boxes. Then the code for the vertices with labels is made by `:tex-draw-vertex-in-box`.

The method `:tex-draw-edges` will write the code for each edge by the method `:tex-draw-edge` and for each edge label by `:tex-draw-edge-label`. The code for the edge is made by the method `:tex-draw-arrow`, if the two vertices of the edge do not belong to the same stratum, else `:tex-draw-thick-edge` is used. The method `:tex-cut-edge` is used to determine the crossing point between a line from origin to the point ⟨*point*⟩ and a rectangle of size ⟨*box*⟩ around origin, i.e., the point `( 0, 0)`.

TeX code for setting the line type and the line width is written by the methods `:tex-line-type` and `:tex-line-width`.

`:tex-draw-blocks` will make code of each block by `:tex-draw-block`. The method `:tex-draw-block` will use the method `:tex-draw-rectangle` to make code for the block and write the block label by `:format-block-label`.

If the graph is not rotated then the lines in the block is written by the method `:tex-draw-rectangle-box`, else the box is written by the method `:tex-draw-rectangle-line`.

The method `:tex-draw-string` will write code for putting a string at a position.

The three similar methods `:tex-draw-line`, `:tex-draw-dashed-line` and `:tex-draw-vector` will write code for respectively a line, a dashed line and a vector between two points. The two points are transformed into the direction and length used in TeX by the methods `:tex-return-length` and `:tex-return-direction`. Code for a rectangle is written by `:tex-draw-box`.

TeX code for setting the line type and the line width is written by the methods `:tex-line-type` and `:tex-line-width`. These methods could be modified to set some slots that the code generated by the methods `:tex-draw-line` and `:tex-draw-vector` might depend on.

When colors are set then the method `:tex-draw-color` and is called, which will do nothing.

10.8 Implementation notes

Except for the methods of the first two subsections of this section the functions and methods of this section are of no interest to the ordinary user. The methods of the second subsection are used when reading a saved diagram.

10.8.1 The Name-, Position-, Edge- and Block-Lists

Default Lists

`(return-coco-graph-window` ⟨*identification*⟩ ⟨*model-number*⟩ ⟨*names*⟩
⟨*edges*⟩ ⟨*positions*⟩ ⟨*blocks*⟩ ⟨*use-variable*⟩ `&key (`⟨*location*⟩ `nil)`
`(`⟨*size*⟩ `nil) (`⟨*title*⟩ `nil))`
`(return-default-positions` ⟨*n*⟩`)`
`(return-default-names` ⟨*n*⟩`)`
`(return-default-edge-list` ⟨*n*⟩ ⟨*p*⟩`)`

The function `(return-coco-graph-window)` will return a CoCo graph object. This function is used by the three methods `coco-graph-window-proto :return-child-coco-graph-window`, `coco-model-proto :make-graph` and `coco-proto :make-graph`. The function is normally not used by the user.

The argument ⟨*identification*⟩ is the identification key of the CoCo object. ⟨*model-number*⟩ is the number of the relevant model in the CoCo object. Note that the only description of the model in the association diagram is the ⟨*model-number*⟩ on the model in the CoCo object. Be careful with `:dispose-of-model`, since when the model for an association diagram is disposed of in the CoCo object, then the association diagram is corrupted.

The argument ⟨*names*⟩ is a list containing the names, labels, types of variables, strata, etc. of the vertices. A default name-list can be returned by the function `(return-default-names)`. The argument ⟨*use-variable*⟩ is a list of items, each item indicating whether the variable is used in the graph or not.

The argument ⟨*edges*⟩ is the list of edges in the graph, a list of lists containing pairs of indices of variables. Furthermore, each list in the edge-list might contain the result of the test of removal of the edge, edge width etc. The function `(return-default-edge-list)` will return a random list of edges for a graph with ⟨*n*⟩ vertices and where each edge is in the graph with the probability ⟨*p*⟩ percent.

The argument ⟨*positions*⟩ is a list of the positions of the vertices in the graph and the positions of the vertex-labels relative to the vertices. The function `(return-default-positions)` will for a graph with ⟨*n*⟩ vertices return positions, where the vertices are placed in a regular polygon.

For causal models the argument ⟨*blocks*⟩ is a list of positions of blocks.

When necessary, new items have been added to the elements of the lists ⟨*names*⟩, ⟨*edges*⟩, ⟨*positions*⟩ and ⟨*blocks*⟩. The format of these lists can in future versions of this tool be changed. Maybe it would have been a more clean programming, if the lists was split up, e.g., the ⟨*names*⟩-list were split in to a list of variable-names, a list of variable-labels, a list of variable-types etc. But the current implementation is probably more efficient.

Setting and Returning the Lists

The format of lists set and returned by the methods of this section can be changed in later versions of this tool.

```
:names &optional (⟨val⟩ nil)
:use-variables &optional (⟨val⟩ nil)
:positions &optional (⟨val⟩ nil)
:edges &optional (⟨val⟩ nil set)
:blocks &optional (⟨val⟩ nil set)
```

The method `:names` sets or returns a list containing the names, labels, type of variables, etc. for the vertices. A default list of names can be returned by the function `(return-default-names)`.

The first item of each element in the list is the variable name. The second item is the label of the vertex, a text-string. The third item of each sublist in the name list is the type of the variable: For discrete variables with code `'1'` a *dot* is drawn: •, For ordinal variables with code `'2'` a *painted box* is drawn: ■, For continuous variables with code `'0'` a *circle* is drawn: ∘. The fourth item is for causal models the stratum of the variable. For edges drawn between two variables not in same stratum an arrow is drawn from the variable in the lowest stratum to the variable in the highest stratum. Explanatory and response variables to variables are also found using this stratum indicator.

The method `:use-variables` refers to a slot containing a list of items, each item indicating whether the variable is used in the graph or not.

The method `:positions` sets the list of positions of the vertices in the graph.

The function `(return-default-positions)` will for a graph with ⟨*n*⟩ vertices return positions, where the vertices are placed in a regular polygon.

The elements in the list of positions have four items: a list of three reals for the vertex position, a list of three reals for the vertex label position, a boolean for whether to fix the vertex label position and a boolean for whether to draw a line from the vertex to the vertex label.

The method `:edges` gives the list of edges in the graph, a list of pairs of indices for variables. The function `(return-default-edge-list)` will return a random list of edges for a graph with ⟨*n*⟩ vertices and where each edge is in the graph with probability ⟨*p*⟩ percent.

Each element in the edge-list contains: index of first vertex, index of second vertex, edge width (integer between 0 and 20), a boolean for whether to draw the line dashed or solid, if computed, the result of the test of the removal of the edge, else `nil`, and a list with three items for the edge label: edge label position relative to the center of the edge, a boolean for fixing the position of the edge label and a boolean for drawing a line from the edge to the edge label.

`:blocks` is a list of positions of the blocks. The first item of each list in the list is the position of one block: The positions of two diagonal corners. Then each list contains the block-key, i.e., the stratum number, the block label and the block label offset relative to the first of the two block points.

10.8.2 Saving the Association Diagram

`:save`

The message `:save` will on standard output echo Lisp code for setting positions, labels etc. The code will read the relevant model into the CoCo object `COCO-OBJECT`, and set positions, labels etc.

The CoCo object `COCO-OBJECT` must have been created with the appropriate data, when the code is used. The format of the lists ⟨*names*⟩, ⟨*edges*⟩, ⟨*positions*⟩ and ⟨*blocks*⟩, arguments to `:names`, `:edges`, `:positions` and `:blocks`, can be changed in later versions of this tool. The name `COCO-OBJECT` can be edited to another identifier. `-NaN` should be replaced be a valid value before reading the output as a source file.

10.8.3 Mouse Events

`:do-click` ⟨*x*⟩ ⟨*y*⟩ ⟨*m1*⟩ ⟨*m2*⟩
`:do-motion` ⟨*x*⟩ ⟨*y*⟩
`:do-key` ⟨*c*⟩ ⟨*m1*⟩ ⟨*m2*⟩

When the mouse is moved in the graph the method `:do-motion` is called continuously. The slots `'x` and `'y` are updated for each call. When a mouse-button is hit, then the method `:do-click` is called, and the vertex, edge or block closest to the point determined the arguments ⟨*x*⟩ and ⟨*y*⟩ is selected. An action is then selected according to whether the "SHIFT" (extend modifier) and the "CONTROL" (option modifier) keys are pressed, and the action is performed continuously while the mouse-button is held down and the mouse moved.

If you move the mouse too fast, then there will not be time enough to update the slots `'x` and `'y` appropriately, and an unexpected vertex might be selected.

The method `:do-key` is called when a key is hit in the association diagram.

Moving Vertices and Labels

`:drag-point` ⟨*x*⟩ ⟨*y*⟩ ⟨*m1*⟩ ⟨*m2*⟩ ⟨*point*⟩
`:drag-edge-label` ⟨*x*⟩ ⟨*y*⟩ ⟨*m1*⟩ ⟨*m2*⟩ ⟨*point*⟩
`:drag-block-point` ⟨*x*⟩ ⟨*y*⟩ ⟨*m1*⟩ ⟨*m2*⟩ ⟨*point*⟩
`:drag-points` ⟨*n*⟩ ⟨*new-pos*⟩

`:drag-line-from-point` ⟨*x*⟩ ⟨*y*⟩ ⟨*point*⟩

When a vertex or vertex-label is dragged, then the method `:drag-point` is called continuously. The two messages `:drag-edge-label` and `:drag-block-point` drag edge labels and block points. The message `:drag-points` is used to drag the vertices in a selected block. The message `:drag-line-from-point` drags the dashed line from a vertex when adding an edge.

10.8.4 Drawing Vertices, Edges, and Blocks

`:draw-vertex` ⟨*type*⟩ ⟨*x*⟩ ⟨*y*⟩ ⟨*z*⟩ ⟨*radius*⟩ ⟨*width*⟩
`:draw-vertex-and-label` ⟨*position*⟩ ⟨*name*⟩ ⟨*use-variable*⟩ ⟨*radius*⟩
`:draw-vertices` ⟨*radius*⟩
`:draw-edge-label` ⟨*edge*⟩
`:draw-edge-label-nth` ⟨*p*⟩
`:draw-thick-edge` ⟨*x*⟩ ⟨*y*⟩ ⟨*z*⟩ ⟨*a*⟩ ⟨*b*⟩ ⟨*c*⟩ ⟨*width*⟩ ⟨*radius*⟩
`:draw-arrow` ⟨*x*⟩ ⟨*y*⟩ ⟨*z*⟩ ⟨*a*⟩ ⟨*b*⟩ ⟨*c*⟩ ⟨*width*⟩ ⟨*fitted*⟩ ⟨*radius*⟩
`:draw-edge` ⟨*edge*⟩ `&optional (`⟨*radius*⟩ `4)`
`:draw-edges` ⟨*radius*⟩
`:draw-rectangle-line` ⟨*a*⟩ ⟨*b*⟩
`:draw-rectangle` ⟨*a*⟩ ⟨*b*⟩ ⟨*c*⟩ ⟨*x*⟩ ⟨*y*⟩ ⟨*z*⟩
`:format-block-label` ⟨*block*⟩
`:draw-block` ⟨*block*⟩
`:draw-blocks`
`:draw-shadows`
`:draw-grid`
`:redraw-graph &optional (`⟨*radius*⟩ `4)`
`:redraw`

Methods for drawing the graph.

The method `:redraw-graph` will draw the vertices, edges and blocks of the graph by the methods `:draw-vertices`, `:draw-edges` and `:draw-blocks`. A grid might be drawn by the method `:draw-grid`, shadows by `:draw-shadows` and overlays by `:redraw-overlays`.

The method `:draw-vertices` will draw each vertex with labels by the method `:draw-vertex-and-label`, which will draw the vertex by the method `:draw-vertex` and then draw the label.

The method `:draw-edges` will draw each edge by the method `:draw-edge` and each edge label by the method `:draw-edge-label`. The edge is drawn by the method `:draw-arrow`, if the two vertices of the edge do not belong to the same stratum, else, if the width of the edge is set, then the edge is drawn by the method `:draw-thick-edge`, else the method `:draw-line` is used. The method `:draw-edge-label-nth` is used by the method `:set-edge-label-nth`.

The method `:draw-blocks` will draw each block by `:draw-block`. The method `:draw-block` will use the method `:draw-rectangle` to draw the block and set the block label by the method `:format-block-label`. The lines in the block are drawn by `:draw-rectangle-line`.

Lower level functions and methods for Edges

Some lower level functions for handling lists of edges etc.:

`:edge-in-list` ⟨*p1*⟩ ⟨*p2*⟩ ⟨*edges*⟩
`:position-to-name` ⟨*p1*⟩ ⟨*p2*⟩
`:edge-list-to-string` ⟨*edges*⟩
`:point-to-string` ⟨*vertex-index*⟩

`(to-string` ⟨*a*⟩ ⟨*sep*⟩`)`
`(split-string` ⟨*str*⟩`)`
`(split-block-string` ⟨*block*⟩`)`
`(split-name-string` ⟨*names*⟩`)`
`(string-to-block-list` ⟨*str*⟩`)`

`(copy-list-list` ⟨*list*⟩`)`
`(list-to-comma-string` ⟨*list*⟩`)`

The method `:edge-in-list` will return `TRUE`, if the edge determined by the vertices with indices ⟨*p1*⟩ and ⟨*p2*⟩ is in the edge list ⟨*edges*⟩. The method `:position-to-name` will return a text-string with the names of the vertices with indices ⟨*p1*⟩ and ⟨*p2*⟩. The method `:edge-list-to-string` will use this method to return a text-string with the edge list ⟨*edges*⟩. The name of the vertex with index ⟨*vertex-index*⟩ can be returned by the method `:point-to-string`.

The function `(split-string)` will split the text string in parts by the characters `:`, `,`, \, `[`, `]`, $<$ and $>$, and then return a list of lists of the characters in each part. The function is used in the function `split-name-string`: If the string contains some of the splitting-characters, then the sting is divided into substring by the spilling characters, else a list of strings each with a character of the string is returned.

The function `(to-string)` will concatenate a list ⟨*a*⟩ of strings or characters to a string with the separator ⟨*sep*⟩ inserted between elements.

The function `(string-to-block-list)` will split the text string in parts by the characters \, $<$ and $>$, and then return a list with a list of lists of the characters in each part and `TRUE`, if the parts are separated by the character $>$, else `nil`. The function is used in the method `:define-blocks`.

The function `(copy-list-list)` will copy a list of lists. The function `(list-to-comma-string)` will concatenate the list to strings, where a comma is put between each item.

10.8.5 Controls

`:add-controls`

As described in section 9.2 the method `:add-controls` adds controls for setting the options "Drag graph", "Static", controls for undoing and redoing point moves, controls for rotating the graph and controls for setting the options of the model search by forward selection and backward elimination of edges in the association diagram.

`:redraw-overlays`
`:overlay-click` $\langle x \rangle$ $\langle y \rangle$ $\langle m1 \rangle$ $\langle m2 \rangle$
`:add-overlay` $\langle ov \rangle$
`:delete-overlay` $\langle ov \rangle$

The individual overlays are added to or deleted from the association diagram by the methods `:add-overlay` and `:delete-overlay`. The message `:redraw-graph` to a graph will redraw the overlays of the graph by the method `:redraw-overlays`. When the mouse is clicked in the graph, then the method `:overlay-click` is used to see if an overlay is clicked. See about "Plot Overlays" in Tierney (1990).

10.8.6 Pixels and Coordinate Systems

`:x &optional (`$\langle val \rangle$` nil set)`
`:y &optional (`$\langle val \rangle$` nil set)`
`:in-wastebasket`

`:to-x-pixel` $\langle x \rangle$
`:to-y-pixel` $\langle y \rangle$
`:from-x-pixel` $\langle x \rangle$
`:from-y-pixel` $\langle y \rangle$

`:resize`

`(rescale-procent` $\langle p \rangle$ $\langle x \rangle$`)`
`(2-3-list` $\langle position \rangle$`)`

`(add-coco-graph-menu` $\langle x \rangle$`)`

The methods `:x` and `:y` will return the last *mouse position* in the graph, the position in the *canvas coordinate system*. `:in-wastebasket` returns `TRUE`, if the mouse was dragged to the *wastebasket*, i.e., right to and below the right lower corner of the plot.

The positions of vertices, labels and block are represented in a coordinate system, the *stored coordinate system*, independently of rotation of the graph and independent of the size of the canvas. The coordinates of points etc. after rotation are in the *scaled coordinate system* or the *rotated coordinate system*. When these coordinates are transformed into the coordinates of the canvas they are in the *canvas coordinate system*. The methods `:to-x-pixel` and `:to-y-pixel` transforms the scaled coordinates of a point (after rotation) to the position of the point on the plotted graph, the canvas. The methods `:from-x-pixel` and `:from-y-pixel` are the inverse of that transformation. From the canvas coordinates to the scaled coordinate system.

The method `:resize` is called when the graph is resized.

The function `(rescale-procent)` will simply multiply $\langle p \rangle$ with $\langle x \rangle$. The function `(2-3-list)` returns `TRUE`, if the argument is a list of two or three integers.

The function `(add-coco-graph-menu)` adds the default menu to the graph.

10.8.7 Closest Vertex, Edge and Block

`(edge-distance` $\langle x \rangle$ $\langle y \rangle$ $\langle a \rangle$ $\langle b \rangle$ $\langle c \rangle$ $\langle d \rangle$`)`
`(point-potential` $\langle x \rangle$ $\langle y \rangle$ $\langle \mathit{point} \rangle$`)`

The *point potential* is minus the reciprocal of the distance from the mouse to the point.

The *edge potential* is minus the reciprocal of the product of the distance from the mouse to the center of the edge and of the area of the triangle spanned by the mouse and the edge (the distance from the edge times half the length of the edge).

These functions are in the following methods used to find the point, the block, the edge or the edge-label closest to the mouse:

`:return-closest-vertex-with-potential` $\langle x \rangle$ $\langle y \rangle$
`:return-closest-vertex-in-canvas` $\langle x \rangle$ $\langle y \rangle$
`:edge-potential` $\langle x \rangle$ $\langle y \rangle$ $\langle \mathit{edge} \rangle$ $\langle \mathit{positions} \rangle$
`:return-closest-edge` $\langle x \rangle$ $\langle y \rangle$
`:return-closest-edge-in-canvas` $\langle x \rangle$ $\langle y \rangle$
`:return-closest-edge-label-with-potential` $\langle x \rangle$ $\langle y \rangle$
`:visit-block-points` $\langle u \rangle$ $\langle v \rangle$ $\langle a \rangle$ $\langle b \rangle$ $\langle c \rangle$ $\langle x \rangle$ $\langle y \rangle$ $\langle z \rangle$
`:return-closest-block-point-with-potential` $\langle x \rangle$ $\langle y \rangle$
`:return-closest-block-point-in-canvas` $\langle x \rangle$ $\langle y \rangle$

The method `:return-closest-vertex-with-potential` will return a list with the potential and index of the vertex closest to point $\langle x \rangle$ and $\langle y \rangle$ in the scaled coordinate system. The method `:return-closest-vertex-in-canvas` with the arguments $\langle x \rangle$ and $\langle y \rangle$ the coordinates in the canvas coordinate system will return the index of the closest vertex.

The method `:return-closest-edge` will by use of `:edge-potential` return the index of the edge with the smallest potential. The arguments $\langle x \rangle$ and $\langle y \rangle$ to the method `:return-closest-edge` have to be in the scaled coordinate system.

The method `:return-closest-edge-in-canvas` will do the same, but with the arguments $\langle x \rangle$ and $\langle y \rangle$, the coordinates in the canvas coordinate system.

`:return-closest-edge-label-with-potential` will return the potential and index of the edge label closest to point $\langle x \rangle$ and $\langle y \rangle$ in the scaled coordinate system.

The method `:visit-block-points` will for a block with the two diagonal corners ($\langle a \rangle$, $\langle b \rangle$, $\langle c \rangle$) and ($\langle x \rangle$, $\langle y \rangle$, $\langle z \rangle$) return the index of the corner closest to the point ($\langle u \rangle$, $\langle v \rangle$).

`:return-closest-block-point-with-potential` will return the potential of the block corner closest to the point $\langle x \rangle$ and $\langle y \rangle$ in the scaled coordinate system, the index of the block and the index of the corner. The method `:return-closest-block-point-in-canvas` with the arguments $\langle x \rangle$ and $\langle y \rangle$ the coordinates in the canvas coordinate system will return a list with the index of the closest block and the index of the closest corner of that block.

Method to determine whether the mouse has been moved:

`(close-point` $\langle x \rangle$ $\langle y \rangle$ $\langle u \rangle$ $\langle v \rangle$`)`
`:close-click` $\langle x \rangle$ $\langle y \rangle$

This function and method will return `TRUE` if the mouse has not been moved more than 4 pixels after clicking the mouse.

The following method finds the 3-dimensional vector in the not rotated coordinate system determining the move of the mouse, i.e., the coordinates in the stored system of the vector from the projection of the point $\langle p \rangle$ onto the scaled coordinate system to the point ($\langle x \rangle$, $\langle y \rangle$) in the scaled system:

`:find-move` $\langle x \rangle$ $\langle y \rangle$ $\langle p \rangle$

Block-Points

`(in-block` $\langle \mathit{position} \rangle$ $\langle \mathit{block} \rangle$`)`
`(return-block-point` $\langle n \rangle$ $\langle \mathit{block} \rangle$`)`
`(to-block-points` $\langle n \rangle$ $\langle p \rangle$`)`

The function `(in-block)` returns `TRUE`, if the point $\langle \mathit{position} \rangle$ is in the 3-dimensional rectangle spanned by the two diagonal points in the list $\langle \mathit{block} \rangle$. `(return-block-point)` will return the $\langle n \rangle$-the (0 to 7) of the 8 corners of the block $\langle \mathit{block} \rangle$. The function `(to-block-points)` will find the change of the two diagonal block points corresponding to that the $\langle n \rangle$-th block-point has been dragged, moved by the three-dimensional vector $\langle p \rangle$.

10.8.8 Arrows

`:update-arrows &optional (`$\langle x \rangle$` nil) &key (`$\langle \mathit{redraw} \rangle$` nil)`

Updates the stratum of variables when vertices or blocks have been moved.

Chapter 11

Stepwise Edge Selection and Elimination in Diagrams

As at the messages `:backward` and `:forward` to CoCo-objects the messages `:drop-least-significant-edge` and `:add-most-significant-edge` to graphs can together with the model-editing actions, the `:graph-drop-edge-nth-` and `:graph-add-edge-`method, be used in a highly user-controlled interactive model search on association diagrams.

In the backward elimination all pairs of vertices adjacent in a specific model are visited by the messages `:drop-least-significant-edge` and each edge is drawn with a width depending on a selected test statistic. The non-significant edge with the largest p-value (or smallest value of the information criterion) is then removed. This stepwise edge elimination process can be performed recursively until no more edges can be removed, i.e., all edges in the resulting model are significant with respect to the selected level of significance. Or, at each step of the backward elimination a set of edges can be eliminated based a graph with the edges drawn with a width depending on a statistic.

In the forward selection in turn, each edge not present in the independence graph is added and a test is performed. Edges found to be significant can be added to the graph.

In the "static" mode a new plot is created, when edges are removed or added. If "static" mode is off, then edges are added to or removed from the current plot, and the plot becomes an association diagram for another model in the CoCo-object. "Static" is turned off and on by pressing the key `'s'` in the plot. "static" can also be returned, turned on or off by the method `:static`.

Edges are drawn with a color depending of whether the test for the edge has been computed, the edge is fixed, the edge is rejected by coherence, etc. When an edge is added, the edge is drawn with `'yellow`. An edge not tested is `'green`. When the test for the two variables conditionally independent has been computed, then the edge is `'blue`. If the removal of the edge will result in a non-decomposable model, then, when `DecomposableMode` is `On`, the edge is drawn with `'magenta`. If the model resulting of dropping the edge is not a sub-model of the **base** model, then the edge is `'red`, if not local tests are used. If an edge has been rejected by coherence, then the edge is `'cyan`. Fixed edges (and edges between explanatory variables to the relevant block in the block-wise model selection) are `'yellow`. Edges to and between response variables of the relevant block is `'blue`. If a test is not computed because of some other error, then the edges are `'black`. See the section 10.5 about changing the used colors.

This section is only implemented for the `coco-graph-window-proto`. The necessary modifications of the methods to make them work for other models, e.g., a `mixed-association-model-dialog-proto`, which is not implemented, concerns the computation of tests (for adding and dropping edges) and the extraction of p-values from the computed tests.

11.1 Primitives

The following methods for adding, removing and testing edges and for selecting the test statistic to perform the model selection according to are essential for the model selection.

11.1.1 Dropping and Adding Edges

`:graph-drop-edge-nth &optional` ⟨*p*⟩ `&key (`⟨*edges*⟩ `nil) (`⟨*point*⟩ `nil)`
`(`⟨*x-move*⟩ `0)`
`:graph-add-edge &optional` ⟨*p1*⟩ ⟨*p2*⟩ `&key (`⟨*edges*⟩ `nil set)`
`(`⟨*x-move*⟩ `0)`
`:add-fill-in`

These methods will create graphs with respectively edges dropped, edges added or a fill in added, see section 10.3. They are only implemented for the prototype `coco-graph-window-proto`.

11.1.2 Testing Edges

`:test-edge-nth` ⟨*p*⟩ `&key (`⟨*block-model*⟩ `nil) (`⟨*follow*⟩ `T)`
`(`⟨*decomposable-mode*⟩ `nil)`
`:edge-index` ⟨*vertex-pair*⟩

The method `:test-edge-nth` returns the test for the removal of the ⟨*p*⟩-th edge in the edge list: A model with the edge removed from the graph is created. If the keyword argument ⟨*decomposable-mode*⟩ is not `TRUE`, or the resulting model is decomposable, a test is performed. If the keyword argument ⟨*follow*⟩ is set to `TRUE`, then the resulting model is tested against the model of the graph, else the resulting model is tested against the **base** model. The result of the test is besides returned also inserted in the edge list.

The method `:edge-index` can be used to find the argument ⟨*p*⟩ to the method `:test-edge-nth`. The argument to the method `:edge-index` is a list with a pair of indices of vertices. Or the argument is a text string with the names of the vertices for the edge.

The method `:test-edge-nth` is used when labeling edges in a graph and in the backward elimination.

In block-recursive models, see the next chapter, the test is performed in the model ⟨*block-model*⟩, if this argument is supplied, else the appropriate model is created.

`:test-add-edge` ⟨*vertex-pair*⟩ `&key (`⟨*block-model*⟩ `nil) (`⟨*follow*⟩ `T)`
`(`⟨*decomposable-mode*⟩ `nil)`

The method `:test-add-edge` tests whether to add the edge ⟨*vertex-pair*⟩ to the graph. The argument is a list with a pair of indices for vertices. Or the argument is a text string with the names of the vertices for the edge. A model with the edge added to the graph is made, and if the keyword argument ⟨*decomposable-mode*⟩ is not `TRUE`, or the resulting model is decomposable, a test is returned. If the keyword argument ⟨*follow*⟩ is set to `TRUE`, then the model of the graph is tested against the model resulting of adding the edge, else the resulting model is tested against the **base** model.

In block-recursive models, see the next chapter, the test is performed in the model ⟨*block-model*⟩, if this argument is supplied, else the appropriate model is created.

11.1.3 Selecting Statistic and Setting Edge Labels

`:select-p-value` ⟨*test*⟩ `&key (`⟨*print-test*⟩ `nil)`
`:p-to-width` ⟨*p-value*⟩
`:format-p-value` ⟨*p-value*⟩

When setting labels on edges in association diagrams, the label is formatted by `:format-p-value`. The method `:p-to-width` will transform a selected test statistic into the width of the edges. The argument to these methods is selected by the method `:select-p-value` from the returned values of the message `:compute-test`.

Edge labels can currently only be set for the `coco-graph-window-proto` since the methods `:test-edge-nth` and `:select-p-value` for computing the test and selecting the p-value from the result of a test is only implemented for the `coco-graph-window-proto`.

The method `:select-p-value` will with the values returned from the message `:compute-test` as argument selected or calculated the test statistic used when labeling edges and in model selection on association diagrams.

The returned value can depend on the deviance, Pearson's chi-square χ^2 or the power divergence $2nI^\lambda$ (Read & Cressie 1988), and for ordinal variables the Goodman and Kruskal's gamma coefficient can be applied. The number of degrees of freedom can be adjusted for non-estimable parameters or simulated exact tests can be applied. E.g., the Akaike or Bayesian information criterion (Schwarz 1978) can be returned.

See section 10.4 about how to edit the method `:select-p-value`. The following code for the method `:select-p-value` will compute the Bayesian information criterion from the computed test statistics[1]. See also section 4.2.2 about the information criterion.

```
(defmeth last-graph-object :select-p-value
  (test &key~(print-test nil))
  (if print-test (print-test test))
  (let ((number-of-cases  (nth 0 (car test)))
        (df               (nth 1 (car test)))
        (adj              (nth 2 (car test)))
        (deviance         (nth 0 (cadr test))))
    (print (list number-of-cases deviance df adj))
    (if (or (invalid-integer number-of-cases)
            (= number-of-cases 0))
        0
      (- (- deviance (* (log number-of-cases) (- df adj))))))
  )

(defmeth  last-graph-object :format-p-value (p)
  (format nil "BIC = ~7,5f" p)
  )

(defmeth  last-graph-object :p-to-width (p)
  (1+ (length
       (which
        (< p
         (list -320 -160 -80 -40 -20 -10 -5 0 5 10 20 40 80)))))
  )
```

The Bayesian information criterion, BIC, is not always computed for the total of a partitioned test, when `ExcludeMissing` is `On`, since the number of cases may differ for the different parts of the test.

If local tests are used, i.e., the $\langle$*follow*$\rangle$ is set to `TRUE`, in a backward elimination, then the used IC-value is the change of the IC-value from the previously accepted model to the model considered. And the removal of edges is continued as long as ΔIC is greater than the value set by `:graph-p-accepted`.

[1] `(invalid-integer x)` is implemented as `(defun (x) (> x 2147483645))`.

The following is not implemented for IC, see **Exercise 1**: If global tests are used, i.e., the ⟨*follow*⟩ is set to `nil`, in the backward elimination, then the change in IC is the difference between the IC value for the **base** model and the IC value for model considered. The edge-elimination should then continue as long as the removal of an edge results in a smaller value of IC. The IC value of the current graph in `:drop-least-significant-edge` should thus be given as the keyword argument ⟨*p-accepted*⟩ in call of `:label-all-edges`.

11.2 Controlling the Edge Elimination and Selection

In association diagrams the backward elimination and the forward selection of edges is performed by the two methods `:drop-least-significant-edge` and `:add-most-significant-edge`.

The behavior of the two methods `:drop-least-significant-edge` and `:add-most-significant-edge` is primarily affected by the choice of the graph, of the **base** model and of the keyword arguments ⟨*block-model*⟩, ⟨*p-accepted*⟩, ⟨*p-rejected*⟩, ⟨*recursive*⟩, ⟨*coherent*⟩, ⟨*headlong*⟩, ⟨*random-order*⟩ and ⟨*follow*⟩. See the following sections.

Next the model selection is affected by the option set by the method `:fix-edges` and the argument ⟨*decomposable-mode*⟩ (or the slot set by the method `:graph-decomposable-mode`, when the method is called by the graph menu) which causes some models not to be considered.

The selection is also affected by options set in the CoCo-object, see the following section.

In the recursive backward elimination, the process is repeated as long as eligible edges have a p-value (`IC`-value) larger than the limit ⟨*p-accepted*⟩ (set by `:graph-p-accepted`). Edges with a p-value less than the value ⟨*p-rejected*⟩ (set by `:graph-p-rejected`) are rejected, and submodels can be rejected by coherence.

In the forward selection the edges with a p-value less than the value ⟨*p-rejected*⟩ are added. A model with a p-value larger than the limit ⟨*p-accepted*⟩ is accepted, i.e., the edge is not added, and models containing such a model can be accepted by coherence.

The p-value or statistic used to classify the models into accepted or rejected is selected by the method `:select-p-value`.

11.2.1 Options Set in the CoCo Object

The options set by methods in this section have effect for all graphs and all models of the relevant CoCo object. A short resume of the description of the options that both effect the model selection in CoCo-objects and the selection by association diagrams is in this and the next section given. See also chapter 4.

```
:set-switch 'partitioning &optional (⟨hit⟩ 'flop)
:set-algorithm &optional (⟨code⟩ 'a)
      ⟨code⟩ ⟶ { 'a | 'b | 'c }

:set-switch 'adjusted-df &optional (⟨hit⟩ 'flop)
:set-power-lambda &optional (⟨lambda⟩ 0.666667)
:set-switch 'reuse-test &optional (⟨hit⟩ 'flop)

:set-ordinal ⟨set⟩

:set-exact-test &optional (⟨code⟩ 'flop)
      ⟨code⟩ ⟶ { 'on | 'off | 'flop | 'all | 'deviance }
:set-asymptotic &optional (⟨limit⟩ 0.25)
:set-number-of-tables &optional (⟨number⟩ 1000)
      ⟨number⟩ ⟶ { ⟨integer⟩ | 'variating | 'what }
:set-list-of-number-of-tables ⟨list-of-number-of-tables⟩

:set-switch 'exact-test-for-total-test &optional (⟨hit⟩ 'flop)
:set-switch 'exact-test-for-parts &optional (⟨hit⟩ 'flop)
:set-switch 'exact-test-for-unparted &optional (⟨hit⟩ 'flop)

:set-seed &optional (⟨seed⟩ 'random)
      ⟨seed⟩ ⟶ { 'random | ⟨integer⟩ | 'what }
:set-exact-epsilon &optional (⟨epsilon⟩ 0.0000001)
:set-switch 'fast &optional (⟨hit⟩ 'flop)
```

The computed test-statistics might depend on `:set-switch 'adjusted-df` and on the values set by `:set-power-lambda`. Only for ordinal variables, declared by `:set-ordinal`, Goodman and Kruskal's gamma coefficient is computed.

Exact tests might be selected by the method `:set-exact-test`. If exact tests are not available or not computed for the selected statistics, then the asymptotic p-value is used by the default `:select-p-value` method.

If `ExactTest` is selected, then the methods of section 4.3 should be noted. See also the section about exact tests in the chapter 6 "Tests" of "A Guide to CoCo".

`Reuse` of test is controlled by `:set-switch 'reuse-test`. Finally the convergence-criterion for the IPS- and EM-algorithm, choice of how to handle missing values and the choice of data structure will have effect on the computing time.

11.2.2 Fix Edges

The options set by methods in this section have effect for all graphs and all models for the relevant CoCo object. See also section 5.2.

```
:fix-edges ⟨edges⟩
:and-fix-edges ⟨edges⟩
```

`:return-fix` ⟨*code*⟩
  ⟨*code*⟩ ⟶ { `'edges` | `'in` | `'out` }

`:return-edge-list &optional (`⟨*model*⟩ `nil) &key (`⟨*edges*⟩ `'in-model)`
`(`⟨*fix*⟩ `'all-edges)`
  ⟨*edges*⟩ ⟶ { `'in-model` | `'not-in-model` | `'all-edges` }
  ⟨*fix*⟩ ⟶ { `'not-fix-edges` | `'fix-edges` | `'all-edges` }
  ⟨*model*⟩ ⟶ { `nil` | `'current` | `'base` | `'last` | ⟨*integer*⟩ }

The edges (i.e., first-order interactions) given by `:fix-edges` are not tested for removal in the backward elimination and ever added in the forward selection. Edges in the `FixEdgeList` can still be removed or added by the model-editing messages, the `:drop-edges`, `add-edges`, `:graph-drop-edge-nth` and `:graph-add-edge` messages and by interaction with the association diagram by the mouse. Edges can be added to the `FixEdgeList` by `:and-fix-edges`. The following use of the `:fix-edges` will clear and replace the `FixEdgeList`. The `FixEdgeList` set by `:fix-edges` and `:and-fix-edges` is independent of the `FixIn` and `FixOut` used in the EH-procedure.

The `FixEdgeList` can be returned in a text-string by `:return-fix`.

The method `:return-edge-list` can return edges not fixed, all fixed edges ignoring which edges are in the graph, the edges not in the relevant graph and not fixed, etc. in the format used in the association diagram.

11.2.3 Setting Graph Options

These methods sets options that are given as arguments to the backward elimination and the forward selection when calling the methods by the graph menu:

`:graph-p-rejected &optional (`⟨*val*⟩ `nil set)`
`:graph-p-accepted &optional (`⟨*val*⟩ `nil set)`
`:graph-coherent &optional (`⟨*val*⟩ `nil set)`
`:graph-headlong &optional (`⟨*val*⟩ `nil set)`
`:graph-random-order &optional (`⟨*val*⟩ `nil set)`
`:graph-decomposable-mode &optional (`⟨*val*⟩ `nil set)`
`:graph-follow &optional (`⟨*val*⟩ `nil set)`
`:graph-most-extreme &optional (`⟨*val*⟩ `nil set)`

The options can also be set by selecting items from the “Menu” or “Overlays”. The options are inherited by child graphs.

`:graph-coherent`: The principle of coherence can be applied: if a model is rejected, then all its sub-models are also rejected. In the backward elimination this means that if the elimination of an edge is rejected at one step, the edge is not eligible for elimination at subsequent steps. Once an edge in the backward elimination process is rejected, it is no more tested for removal.

Analogously for forward selection: Once in a forward selection it has been accepted that an edges should not be added to the graph, it is not in subsequential steps of the forward selection tested whether the edge should be added.

`:graph-headlong`: Headlong backward elimination is a recursive backward elimination strategy, where the first edge found to be non-significant is eliminated in each step. `:graph-p-rejected`: In each step of this backward elimination, edges with a p-value less than a given limit (e.g. 1% or 5%) is rejected, and is, when the principle of weak rejection is applied, not considered in sub-sequential steps. `:graph-p-accepted`: The first edge found in each step with p-value greater than a second limit (e.g. 20%) is removed. Visited edges in the current step with a found p-value between the two limits and all not visited eligible edges in the current step are eligible in the next step. This gives a faster elimination than a backward elimination, where all eligible edges in each step have to be visited in order to find the least significant edge.

Analogously for forward selection.

`:graph-random-order`: Not to favour the removal of edges with, e.g., a low lexicographical order, the edges are in the headlong model selection visited in a random order. Hence several calls to the procedure might give different results, and a sample of models with all edges significant to a selected level of significance can be generated.

`:graph-decomposable-mode`: The incremental search is done either among graphical models or the search can be restricted to decomposable models. Different edge elimination orderings might result in the same model, and since a sequence of decomposable models differing with one edge always exists between any two nested decomposable models then restricting to decomposable models will not prevent the backward elimination from finding the "true" model, if the "true model is decomposable. And since when restricting to decomposable models, the IPS-algorithm is not used, restricting to decomposable models might speed up the model search. If exact tests are applied, then restricting to decomposable models is also useful since the exact tests are only available in CoCo for decomposable models.

`:graph-follow`: In the backward elimination the tests can either be made locally, i.e., nested tests: the model is tested against the previously accepted model, or the model can be tested against a fixed base model. When the tests are performed against the previously accepted model, then in the first step of the backward elimination the tests will be performed against the model for the graph. In the next step, the tests are performed against the model accepted in the first step, etc.

In forward selection the model of the graph is tested against the model resulting of adding the edge, or the resulting model is tested against the **base** model.

If `:graph-follow` is `TRUE`, then when setting edge labels by the key event `'e'` the resulting model is tested against the current graph, else the resulting model is tested against the **base** model.

`:graph-most-extreme`: If this option is set to `TRUE`, then only the least significant edge is removed in the backward elimination, else all non significant edges are removed. The first non significant edge is removed in the headlong backward elimination regardless of this option. Analogously for forward selection.

11.2.4 Rejected and Accepted Edges

`:rejected-edges &optional (`⟨*val*⟩` nil set)`
`:accepted-edges &optional (`⟨*val*⟩` nil set)`

The edges in the lists set and returned by these methods are rejected (accepted) by coherence in backward elimination (forward selection). Consider setting the lists to `nil` after changing significance levels by `:graph-p-rejected` and `:graph-p-accepted`. Or you might get confused, if, e.g., the level of significance is decreased and some models then should be accepted, but they are still rejected (by coherence).

Child graphs will share these lists with the parent graph.

11.2.5 Current and Base

`:make-graph-current-model`
`:make-graph-base-model`

These methods make the model for the graph the **current** or **base** model respectively in the CoCo object. The methods can be called by the key events `'c'` and `'b'` respectively in graph windows for the graph, or by selecting appropriate items from the graph menu, or by controls, see section 9.2.

These methods are edited and used in the "Model Dynamic Spin Plot", see section 13.1.3.

If `:graph-follow` is not `TRUE`, then all models are tested against the **base** model.

11.3 Backward Elimination

Visiting All Edges in the Graph

`:label-all-edges &key (`⟨*block-model*⟩` nil) (`⟨*print-test-for-edge*⟩` nil)`
`(`⟨*p-accepted*⟩` 0.10) (`⟨*p-rejected*⟩` 0.05) (`⟨*coherent*⟩` nil)`
`(`⟨*headlong*⟩` nil) (`⟨*random-order*⟩` nil) (`⟨*follow*⟩` T)`
`(`⟨*decomposable-mode*⟩` nil) (`⟨*least-significant*⟩` T) (`⟨*make-graph*⟩` T)`

`:label-all-edges` will for each first-order interaction in the model for the graph remove the first-order interaction and higher order interactions containing the first-order interaction. I.e., the first-order interactions are in turn added to the dual representation for the model for the graph.

If the keyword argument ⟨*follow*⟩ is set to `TRUE`, then the model resulting of dropping an edge is tested against the previously accepted model, the model for the current graph, else the model is tested against the **base** model.

If the keyword argument ⟨*least-significant*⟩ is set to `TRUE`, then the index of the least significant edge is returned, else a list of all non significant edges is returned.

If the keyword argument ⟨*headlong*⟩ is set to `TRUE`, then the first found non-significant edge, i.e., edge with a p-value greater than the limit given as keyword argument ⟨*p-accepted*⟩, is returned, and no further edges are visited.

If the keyword argument ⟨*random-order*⟩ is set to `TRUE`, then the edges in the graph are visited in a random order.

If the keyword argument ⟨*coherent*⟩ is set to `TRUE`, then the rejected edges, i.e., edges in the list set and returned by the method `:rejected-edges`, are not visited, and, if an edge is rejected, i.e., the p-value is less than the limit given as the keyword argument ⟨*p-rejected*⟩, then the edge is added to the list `:rejected-edges` and is thus not visited in sub-sequential steps.

Edges in the list `:fix-edges` are not visited.

If the keyword argument ⟨*decomposable-mode*⟩ is set to `TRUE`, then edges resulting in a non-decomposable model are not visited.

If the keyword argument ⟨*print-test-for-edge*⟩ is set to `TRUE`, then the test for each edge is printed by the function `(print-test` ⟨*test*⟩`)` in the Lisp listener.

For block-models: If ⟨*block-model*⟩ is given to `:label-all-edges` then only edges in the model ⟨*block-model*⟩ is labeled, else all edges are labeled and the test are performed in appropriate models.

If ⟨*make-graph*⟩ is set to `TRUE`, then an association diagram with the returned edges removed from the current graph is made.

Recursive Drop of the Least Significant Edge

`:drop-least-significant-edge &key (`⟨*block-model*⟩ `nil)`
`(`⟨*p-accepted*⟩ `0.10) (`⟨*p-rejected*⟩ `0.05) (`⟨*recursive*⟩ `nil)`
`(`⟨*coherent*⟩ `nil) (`⟨*headlong*⟩ `nil) (`⟨*random-order*⟩ `nil) (`⟨*follow*⟩ `T)`
`(`⟨*decomposable-mode*⟩ `nil) (`⟨*least-significant*⟩ `T) (`⟨*x-move*⟩ `0)`

The method `:drop-least-significant-edge` will by use of `:label-all-edges` visit edges in the graph, and create a new graph with the least significant edge, all non significant edges or the first found non-significant edge dropped.

Backward elimination is done recursively until no more edges can be removed according to the selected significance level, if the keyword argument ⟨*recursive*⟩ is set to `TRUE`.

If ⟨*recursive*⟩ is `TRUE`, the accepted model for each step is added to the model list in the CoCo-object.

For block-models: If the keyword argument ⟨*block-model*⟩ is given to the message `:drop-least-significant-edge` then only edges in the model ⟨*block-model*⟩ is visited, else all edges are visited and the tests are performed in appropriate models.

11.4 Forward Selection

Visiting All Other Edges, All Edges not in the Graph

`:label-all-other-edges &key` (⟨*block-model*⟩ `nil`)
  (⟨*print-test-for-edge*⟩ `nil`) (⟨*p-accepted*⟩ `0.10`) (⟨*p-rejected*⟩ `0.05`)
  (⟨*coherent*⟩ `nil`) (⟨*headlong*⟩ `nil`) (⟨*random-order*⟩ `nil`) (⟨*follow*⟩ `T`)
  (⟨*decomposable-mode*⟩ `nil`) (⟨*most-significant*⟩ `T`) (⟨*make-graph*⟩ `T`)

The message `:label-all-other-edges` will for each first-order interaction not present in the model of the graph, generate a model with the first-order interaction added.

If the keyword argument ⟨*follow*⟩ is set to `TRUE`, then the model of the graph is tested against the model resulting of adding the edge, else the resulting model is tested against the **base** model.

If the keyword argument ⟨*most-significant*⟩ is set to `TRUE`, then a list with the vertex indices of the most significant edge is returned, else a list of all significant edges is returned.

If the keyword argument ⟨*headlong*⟩ is set to `TRUE`, then the first significant edge found, i.e., edge with a p-value less than the limit set by the keyword argument ⟨*p-rejected*⟩, is returned, and no further edges are visited.

If the keyword argument ⟨*random-order*⟩ is set to `TRUE`, then the edges are visited in a random order.

If the keyword argument ⟨*coherent*⟩ is set to `TRUE`, then the accepted edges, i.e., edges in the list set and returned by the method `:accepted-edges`, are not visited, and, if an edge is accepted, i.e., edges with a p-value greater than the limit given as the keyword argument ⟨*p-accepted*⟩, then the edge is added to the list `:accepted-edges` and is thus not visited in sub-sequential steps.

Edges in the list `:fix-edges` are not visited.

If the keyword argument ⟨*decomposable-mode*⟩ is set to `TRUE`, then edges resulting in a non-decomposable model are not visited.

If the keyword argument ⟨*print-test-for-edge*⟩ is set to `TRUE`, then the test for each edge is echoed in the Lisp listener.

For block-models: If ⟨*block-model*⟩ is given to `:label-all-edges` then only edges in the model ⟨*block-model*⟩ are labeled, else all edges are labeled and the tests are performed in appropriate models.

If ⟨*make-graph*⟩ is set to `TRUE`, then an association diagram with the returned edges added to the current graph is made.

Recursively Adding Edges

`:add-most-significant-edge &key` (⟨*block-model*⟩ `nil`)
(⟨*p-accepted*⟩ `0.10`) (⟨*p-rejected*⟩ `0.05`) (⟨*recursive*⟩ `nil`)
(⟨*coherent*⟩ `nil`) (⟨*headlong*⟩ `nil`) (⟨*random-order*⟩ `nil`) (⟨*follow*⟩ `T`)
(⟨*decomposable-mode*⟩ `nil`) (⟨*most-significant*⟩ `T`) (⟨*x-move*⟩ `0`)

The message `:add-most-significant-edge` will to a CoCo graph, by use of the method `:label-all-other-edges`, visit edges not in the graph, and create a new graph with the most significant edge, all significant edges or the first found significant edge added.

The forward selection is done recursively until no more edges can be added according to the selected significance level, if the keyword argument ⟨*recursive*⟩ is set to `TRUE`.

If ⟨*recursive*⟩ is set to `TRUE`, the generated model for each step of the forward selection is added to the model list in the CoCo-object.

For block-models: If ⟨*block-model*⟩ is given to `:add-most-significant-edge` then only edges in the model ⟨*block-model*⟩ are visited, else all edges are visited and the tests are performed in appropriate models.

11.5 Adding a Fill In

`:add-fill-in`

Where as the resulting graph from a backward elimination with decomposable mode on and only one edge removed in each step is decomposable, the resulting model from a forward selection, where in one step all significant edges are added, is not necessarily decomposable. The method `:add-fill-in` will for a non decomposable model create a new graph with a fill in added.

11.6 Exercises

Exercise 1 Modify the method `:drop-least-significant-edge` so global tests can be used in backward elimination by the *IC* criteria, i.e., so the method is also correct when ⟨*follow*⟩ is set to `nil`.

Consider inserting a keyword argument in this method and in methods using the method for controlling global tests together with the *IC* criteria.

Exercise 2 Modify the method `:label-all-edges` so that all edges rejected by the criteria of Holm are returned (Holm 1979). See also Hommel & Bernhard (1993) about *multiple hypotheses testing*.

Let $(\alpha_1, \ldots, \alpha_n)$ denote $(\alpha/n, \alpha/(n-1), \ldots, \alpha/2, \alpha/1)$, where α is the level of significance and n is the number of tests performed. The ordered p-values $p_{(i)}$ are then compared with α_i. The stepwise rule of Holm (1979) is that the hypotheses H_j is rejected, if $p_{(j)} \leq \alpha_j$ and $p_{(i)} \leq \alpha_i$ for all $i < j$.

You have to find my code for the method `:label-all-edges` (in the file `cocotest.lsp`). All tests should be collected, e.g., by inserting the line

```
(setf all-edges (cons i all-edges))
```

just below the line

```
(if (< p-accepted p)
    (setf accepted-edges (cons i accepted-edges)))
```

The list `all-edges` to hold all performed tests should then have been defined by changing the line

```
(let ((accepted-edges nil)
```

to

```
(let ((all-edges nil)
      (accepted-edges nil)
```

Before returning from the method, the list `all-edges` should be sorted according to the tests in the list, and some edges collected from the list.

Consider inserting a keyword argument in this method and in methods using the method for this criteria.

Exercise 3 Consider the tests for elimination of a specific edge in a backward elimination from the saturated model. As the models get progressively simpler, so the power of the test increases: in other words, the initial tests have relative low power. Thus the risk of Type II error (false elimination of edges) is highest in the initial steps.

A variant of the backward elimination procedure to counteract this is suggested in Edwards (1993): At each step examine the edge elimination tests with the largest power (least degrees of freedom) and remove the least significant edge among these. In other words, instead of deciding which edge to remove from the p-values alone, sort edges by degrees of freedom (ascending) and then within these by p-values (descending).

Modify the method `:label-all-edges` to do so.

Chapter 12

Model Selection in Causal Models

Block recursive models, *Chain graph models*, are models with both symmetric and asymmetric associations between variables. The symmetric associations have been dealt with so far in this guide.

At the asymmetric association, directed association, between two variables, the second variable might depend on the first variable, but the first variable is independent of the second variable. The first variable is then *explanatory* to the the second variable, the *response* variable. In the graph we then draw an arrow from the first variable to the second variable. The explanatory variables are also called the *exogenous* variables and the response variables the *endogenous* variables. The endogenous variables have an undirected graph associated with them. Each endogenous factor is the end point for one or more direct edges. The directed edges can originate at either exogenous or endogenous variables.

In *recursive* models response variables of some explanatory variables cannot be explanatory variables to the same explanatory variables, i.e., there cannot in the graph be any oriented cycles, cycles where one goes along undirected edges and in the direction of arrows.

Conditional independence statements for recursive causal models are examined in Wermuth & Lauritzen (1983). The so-called block recursive graphical models or graphical chain models are further treated in Lauritzen & Wermuth (1989), Lauritzen (1989), Wermuth & Lauritzen (1990), Lauritzen, Dawid, Larsen & Leimer (1990). The chain graph Markov property is investigated in details in Frydenberg (1989) and Frydenberg (1990). Model selection methods for creating a diagnostic system by causal models on discrete variables are discussed in Lauritzen, Thiesson & Spiegelhalter (1992). Also the text books Lauritzen (1993), Whittaker (1990) and Christensen (1990) contain chapters on block recursive models.

In the following the fact will be used stating that in block-recursive models the test of the two variables conditionally independent can be performed given explanatory variables to the two variables and ignoring response variables.

12.1 Creating and Deleting Blocks

A block recursive model is in this tool a specialization of an undirected graph. Thus all the methods of an undirected graph are also available for a graph with a causal model. But some of the methods will have the result as if the method was applied on the associated undirected graph, i.e., the graph achieved by removing the arrows from the edges of the directed graph. See the last section of this chapter about methods not implemented correctly for block recursive models.

`:add-block` $\langle n \rangle$ `&key (`$\langle a \rangle$ `(list -45 -45 -50))`
  `(`$\langle b \rangle$ `(list -40 -40 50)) (`$\langle label \rangle$ `nil)`
`:define-blocks` $\langle block\text{-}list \rangle$ `&optional (`$\langle labels \rangle$ `nil)`
`:delete-block` $\langle n \rangle$

The two methods `:add-block` and `:define-blocks` are used to define block-recursive models.

Which block a variable belongs to is determined by the position of the vertex.

The message `:add-block` adds a block with *block-key* $\langle n \rangle$ to the block-list `'blocks`. The keyword arguments $\langle a \rangle$ and $\langle b \rangle$ are the positions in the association diagram of two diagonal corners of the block. The default position is a small block in the upper left corner of the association diagram. The keyword argument $\langle label \rangle$ is a label for the block, the *block-label*. The block-keys for the blocks define the causal structure. A variable is in a block, if the position of the variable in the association diagram is in the rectangle determined by the two diagonal corners of the block. Variables in blocks with lower keys are explanatory to variables with higher keys. The position of a block in the list of blocks is called the index of the block, the *block-index*. Blocks can be refereed by their block-key, their index and by their label. The order of the blocks is used in the block wise model selection. See the next section about how to edit the block, so that variables come to belong to the corresponding stratum.

The message `:define-blocks` declares the causal structure and overwrites existing blocks for a graph with a model. The causal structure is given by a text-string of variable-names. Blocks are separated by the character $<$. Variables preceding a $<$ are explanatory to variables following the $<$. Consider, e.g., the following command:

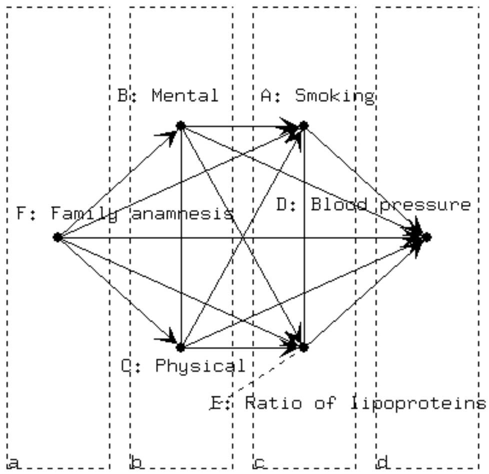


Figure 12.1: A bit-map dump of a causal graph created by `:define-blocks`.

```
(send ⟨graph-object⟩ :define-blocks "F<BC<AE<D"
  (list "a" "b" "c" "d"))
```

The five variables `'F'`, `'B'`, `'C'`, `'A'` and `'E'` are explanatory to the variable `'D'`, the three variables `'F'`, `'B'`, `'C'` are explanatory to the three variables `'A'`, `'E'` and `'D'`, and the variable `'F'` is explanatory to the five variables `'B'`, `'C'`, `'A'`, `'E'` and `'D'`. The 4 strata are given the labels `'a'`, `'b'`, `'c'`, and `'d'` respectively. The block-keys are integers increasing by one and starting with `1` for the first strata. The variable `'F'` is put in stratum `1` with block index `0` and block label `"a"`, the variables `'B'` and `'C'` are put in stratum `2` with block index `1` and block label `"b"`, etc. The variables and blocks are given a default position. Explanatory variables to the left and response variables to the right in the graph. If the block-labels are `nil`, then the block-key, the stratum number, is typed in the graph relative to one of the two diagonal corners of the block, else the block label is typed.

The methods `:add-block` and `:define-blocks` can be performed by selecting "Add block" or "Define blocks" respectively from the graph menu.

The message `:delete-block` will delete the block with block key $\langle n \rangle$ (a number) or block label $\langle n \rangle$ (a text-string).

Nanny-mode

The same block-structure as that of figure 12.1 is declared by

```
(send ⟨graph-object⟩ :define-blocks "D>AE>BC>F"
  (list "d" "c" "b" "a"))
```

but response variables are then placed to the left in the plot and explanatory to the right. Block keys and block indices are independent of whether the "nanny-mode" has been used or not. The variable `'F'` is put in stratum `1` with block index `0` and block label `"a"`, etc.

12.2 Editing Blocks

The block can be resized by the mouse: Position the mouse at a corner of the block, press a mouse button, and drag the corner of the block; Or the created block can be moved: Position the mouse at a corner of the block, press a mouse button and "SHIFT" (extend modifier), and drag the block. When a block is resized the positions of the vertices are unchanged. If some vertices then fall outside the block, the variables will no longer be in the stratum, and if other vertices are now in the block, then the other variables will be put in the stratum. When dragging a block the vertices in the block will follow the block.

If a vertex is in more than one block, then the variable will belong to the stratum of the last block (block with highest block index) of the block in which the vertex is.

```
:stratum ⟨variable-name⟩ &optional (⟨block-key⟩ nil set)
  &key (⟨redraw⟩ nil)
:block-position ⟨block-key⟩ &optional (⟨position⟩ nil set)
  &key (⟨redraw⟩ nil)
:block-label ⟨block-key⟩ &optional (⟨label⟩ nil set)
  &key (⟨redraw⟩ nil)
:block-label-position ⟨block-key⟩ &optional (⟨position⟩ nil set)
  &key (⟨redraw⟩ nil)
:block-index ⟨block-key⟩
```

The message `:stratum` will place the vertex ⟨*variable-name*⟩ in the block ⟨*block-key*⟩. The argument ⟨*variable-name*⟩ is identifying the vertex, either by a single character given the name of the variable, or by an integer giving the index of the vertex. ⟨*block-key*⟩ is either a number given the key of the stratum, the label of the stratum, or a list containing the index of the stratum. The position of the vertex is then changed to the center of the block. The index of the block with block-key ⟨*block-key*⟩ or label ⟨*block-key*⟩ can be returned by `:block-index`.

The stratum for vertex ⟨*variable-name*⟩ is returned by `:stratum`.

The message `:block-position` sets or returns the positions of two diagonal corners in the block ⟨*block-key*⟩. The argument ⟨*block-key*⟩ is an identification of the block (the stratum) as in the message `:stratum`. The argument ⟨*position*⟩ is a list of two lists of two or three integers given the positions of the two diagonal corners of the block ⟨*block-key*⟩.

The message `:block-label-position` sets or returns the position of the block label relative to a specific corner. The argument ⟨*position*⟩ is a list of two or three integers given the position of the label relative to one of the two diagonal corners in the block ⟨*block-key*⟩.

The message `:block-label` sets or returns the label of the block ⟨*block-key*⟩.

To delete a block: Press `'W'` at a block corner.

When vertices or blocks are moved, so that vertices change blocks, then the edge labels, *p*-values and tests, are removed.

See also section 10.2.

`:adjust-blocks-to-grid &key (`⟨*delta*⟩` 1) (`⟨*redraw*⟩` nil)`
`:rescale-block-positions &key (`⟨*scale*⟩` 1) (`⟨*redraw*⟩` nil)`

The method `:adjust-blocks-to-grid` will adjust the block corners to a grid with the distance ⟨*scale*⟩ between points. `:rescale-block-positions` reposition the block corners so that the distance from the center of the graph to the corners is multiplied by ⟨*scale*⟩.

Lower level functions

The following functions are of no interest to the ordinary user:

`(in-block `⟨*position*⟩` `⟨*block*⟩`)`
`(return-block-point `⟨*n*⟩` `⟨*block*⟩`)`
`(to-block-points `⟨*n*⟩` `⟨*p*⟩`)`

The function `(in-block)` will return `TRUE`, if the point ⟨*position*⟩ (list of three reals) is in the rectangle determined by ⟨*block*⟩, a list with the two lists of three reals. The function `(return-block-point)` will return the position of the ⟨*n*⟩-the (0 to 7) corner of the block with the two diagonal corners ⟨*block*⟩. The function `(to-block-points)` will find the change of the two diagonal block points corresponding to that the ⟨*n*⟩-the block-point has been dragged, moved by the three-dimensional vector ⟨*p*⟩.

See also section 10.8.

`:update-arrows &optional (`⟨*x*⟩` nil) &key (`⟨*redraw*⟩` nil)`

The method `:update-arrows` will after having moved vertices between blocks or changed the number of blocks, after addition or deletion of blocks, update the stratum numbers of the variables and dispose of edge labels.

12.3 Tests

`:position-to-max-block `⟨*p1*⟩` `⟨*p2*⟩
`:return-history-and-future `⟨*block-key*⟩` &key (`⟨*redraw*⟩` nil)`

```
:return-history-model-nr ⟨block-key⟩ &key (⟨redraw⟩ nil)
  (⟨current-model⟩ nil) (⟨fix-history⟩ nil)
```

```
:test-edge-nth ⟨p⟩ &key (⟨block-model⟩ nil) (⟨follow⟩ T)
  (⟨decomposable-mode⟩ nil)
:test-add-edge ⟨vertex-pair⟩ &key (⟨block-model⟩ nil) (⟨follow⟩ T)
  (⟨decomposable-mode⟩ nil)
```

The method `:position-to-max-block` will for two vertex indices return the largest block key among the blocks, to which the two vertices belong.

In tests of the two variables conditionally independent (by the methods `:test-edge-nth` and `:test-add-edge`) the tests are performed given explanatory variables to the two variables and ignoring response variables.

Explanatory variables (the history) to the two variables will be variables in blocks with a block key smaller than the block key ⟨*block-key*⟩ returned by the method `:position-to-max-block`.

Response variables (the future) to the two variables are variables in blocks with a block key greater than the block key ⟨*block-key*⟩.

The method `:return-history-and-future` will then with the block key ⟨*block-key*⟩ as argument return a list with three lists of vertex names: vertices in blocks prior the block ⟨*block-key*⟩, vertices in the block and vertices past the block.

This method is used in the following method: If the keyword argument ⟨*current-model*⟩ is `nil`, then the method `:return-history-model-nr` with the argument ⟨*block-key*⟩ will return the model number of the model[1] achieved by restricting the model of the graph to vertices in the block ⟨*block-key*⟩ and blocks with block key less than ⟨*block-key*⟩. All interactions among variables in blocks with block key less than ⟨*block-key*⟩ are added to the model.

If the keyword argument ⟨*current-model*⟩ is `TRUE`, then the model number of the model achieved by restricting the **current** model to vertices in the block ⟨*block-key*⟩ and blocks with block keys less than ⟨*block-key*⟩ is returned. Also here all interactions among explanatory variables are added.

If the keyword argument ⟨*block-model*⟩ is not given to the two methods `:test-edge-nth` and `:test-add-edge` in block-recursive models, then the method `:return-history-model-nr` will be used to find the model in which to perform the test. If the keyword argument ⟨*block-model*⟩ is given, then the tests are performed under the model ⟨*block-model*⟩.

The keyword argument ⟨*follow*⟩ should in block-models be `TRUE`, or you would get the result of a test between two undirected models.

[1] If `*my-trace*` is set to a value greater than `0`, then a plot with the returned model is made.

12.4 Stepwise Edge Elimination and Selection

`:block-backward &key` (⟨*block*⟩ `nil`) (⟨*p-accepted*⟩ `0.10`)
(⟨*p-rejected*⟩ `0.05`) (⟨*recursive*⟩ `nil`) (⟨*coherent*⟩ `nil`) (⟨*headlong*⟩ `nil`)
(⟨*random-order*⟩ `nil`) (⟨*follow*⟩ `T`) (⟨*decomposable-mode*⟩ `nil`)
(⟨*least-significant*⟩ `T`) (⟨*x-move*⟩ `0`)

`:block-forward &key` (⟨*block*⟩ `nil`) (⟨*p-accepted*⟩ `0.10`)
(⟨*p-rejected*⟩ `0.05`) (⟨*recursive*⟩ `nil`) (⟨*coherent*⟩ `nil`) (⟨*headlong*⟩ `nil`)
(⟨*random-order*⟩ `nil`) (⟨*follow*⟩ `T`) (⟨*decomposable-mode*⟩ `nil`)
(⟨*most-significant*⟩ `T`) (⟨*x-move*⟩ `0`)

These methods will perform backward elimination and forward selection in block-recursive models by the methods `:drop-least-significant-edge` and `:add-most-significant-edge`.

If ⟨*block*⟩ is a number, then a model selection among edges in and arrows to the block ⟨*block*⟩ is performed. If ⟨*block*⟩ is `nil`, then term by term a model selection for each block in the list of blocks for the graph is performed. Each model selection is done among edges in and to the relevant block. Response variables are ignored and explanatory variables "completed". Since CoCo is using collapsibility of tests, this "completion" of the explanatory variables will not give more complex models than just moralizing. If ⟨*block*⟩ is a list of blocks, by, e.g., `(reverse (send ... :blocks))`, then term by term a model selection for each block in the list ⟨*block*⟩ of blocks is performed.

Since the block indices do not depend on whether the "nanny-mode" has been used when declaring the blocks, the models search is independent of "nanny-mode".

Note that the search can only be used to select edges in the graph, not to determine the causal structure.

A result of a headlong backward elimination on a causal model on the data from the introduction is seen in figure 10.1*b)*.

12.5 Search Strategies

In large models, models with more than 10 variables, the search in association diagrams will take considerable time. Start with a search on the undirected models in the CoCo-object.

12.6 Decomposable Models

Although the graph has no 4-cycles and cycles of greater length the moral graph might not be decomposable, and a directed graph with cycles of a length greater than 3 might have a decomposable moral graph.

12.7 Returning Fitted Cell Values and Tests for the Undirected Models

The fitted table values are found under the corresponding undirected graph. Arrows are just replaced with undirected edges, no moralizing, etc. Thus the values returned by `:return-vector` are "wrong".

Tests and fitted values computed by `:return-vector`, `:return-matrix`, `:is-decomposable`, `:print-table`, `:describe-table`, `:plot`, `:list-values`, `:case-list`, `:compute-test`, `:compute-deviance`, `:test`, `:find-deviance`, `:find-log-l`, `:generate-graphical`, `:generate-decomposable`, `:factorize`, `:meet-of-models`, `:join-of-models`, `:drop-edges`, `:drop-interactions`, `:add-edges`, `:add-interactions`, `:add-fill-in` and the EH-procedure are computed under the undirected graph.

See the following exercises for how to update these methods to causal models.

The tests performed by the methods `:test-edge-nth`, `:test-add-edge`, `:label-all-edges`, `:drop-least-significant-edge`, `:block-backward`, `:label-all-other-edges`, `:add-most-significant-edge`, `:block-forward` and by the key-event `'e'`, labeling an edge in the graph, are performed in the directed graph.

12.8 Exercises

Exercise 1 Write a method that computes the vector of the maximum likelihood estimates of the cell probabilities in the block recursive model, see, e.g., Theorem 5.42 of Lauritzen (1993). The methods `:return-history-and-future`, `:return-history-model-nr` and `:return-vector` should be useful.

Exercise 2 Write, using the result of **Exercise 1**, a method that computes the Deviance, Pearsons chi-square χ^2 and the Power Divergence $2nI^\lambda$ of the test of any two nested block recursive models against each others.

Exercise 3 Write, using the fact that the deviance and the degrees of freedom is additive over blocks, a method that computes the Deviance and the degrees of freedom for the test of any two nested block recursive models against each others.

Exercise 4 Write a method which, for a block recursive model, returns the *parents* of a set, e.g., by first writing a function that returns the *boundary* of a set, and then in the method to write take intersections with sets returned by the methods `:return-history-and-future`.

Exercise 5 Write a method which, for a block recursive model, returns the *moral graph*.

Exercise 6 Write a method which, for a given set and a block recursive model returns the block-recursive model collapsed to that set.

Exercise 7 Use the result of **Exercise 6** to solve **Exercise 1**.

Exercise 8 Write a method that performs a block-wise model search by the EH-procedure, using the method `:eh`.

Chapter 13

Examples on Extensions

In this chapter some extensions of the CoCo-graph-object are considered.

13.1 Model Dynamic Spin Plots

A model dynamic spin plot is a spin plot, where the values in the plot are re-evaluated when associations diagrams for the relevant CoCo object are made the **current** or the **base** model, or when a corresponding associations diagram is updated.

The object of the model dynamic spin plot is created with some expressions for computing the values in the plot. Each expression can contain calls of the method `:return-vector`, where the keyword argument ⟨*model*⟩ is used to select between returning values from the model of an association diagram, the **current**, the **base**, the **last** model or a model with a specific number. When some other model is named **current**, **base** or **last** or the association diagram of the model dynamic spin plot is updated then the values returned by these expressions will change.

When a forward selection or a backward elimination of edges is performed on association diagrams, the spin plots are updated for each new association diagram created. The **current** model follows the last created association diagram, and the **base** model is the second last created association diagram.

In the dynamic mode, that is, when "Static" is set to `nil`, the model dynamic CoCo spin plots for the association diagram will also change, when edges are added to or removed from the association diagram.

Since this is a very interesting example on multiple inherence, storing expressions in Lisp, since showing the implementation enables you to implement, e.g., a "Model Dynamic CoCoHistogram", and since it is possible to implement the tool in only about 80 lines, the code for the implementation is given in this guide. (Some minor modifications necessary for updating model dynamic spin plots, when new graphs are created, are omitted from this description. These modifications are in the methods `:graph-drop-edge-nth`, `:graph-add-edge`, `:label-all-edges` and `:label-all-other-edges`.)

13.1.1 Initial Consideration

Assume that we have implemented a "Model Dynamic CoCo Spin Proto" with the method `:change-models` for re-evaluating plotted values in the spin-plot. The method `:change-models` uses some expressions of the plot to compute the values in the plot, and then replaces the points in the plot by the new values, and redrawes the plot. This method should be called when we make another association diagram the **current** model. Thus we change the method `:make-graph-current-model` from

```
(defmeth coco-graph-window-proto :make-graph-current-model ()
  (send self :make-current (slot-value 'model-number)))))
```

to

```
(defmeth coco-graph-window-proto :make-graph-current-model
         (&key (redraw-plots nil set))
  (let ((b (send self :make-current (slot-value 'model-number))))
    (if (and b redraw-plots)
        (progn
          (dolist (i (send dynamic-coco-spin-proto :slot-value
                                 'instances-dynamic-coco-spin))
                  (send i :change-models))))
    b))
```

All instances of the "Model Dynamic Spin Proto" are then sent the message `:change-models`, when some CoCo-graph-objects receive the message `:make-graph-current-model`, e.g., by the key event '`c`'.

If the expressions for computing the three vectors are stored in the slots `'a`, `'b` and `'c`, then the method `:change-models` can be implemented as:

```
(defmeth dynamic-coco-spin-proto :change-models ()
  (setf *this-graph* self)
  (let* ((x (eval (slot-value 'a)))
         (y (eval (slot-value 'b)))
         (z (eval (slot-value 'c))))
    (send self :point-coordinate 0 1 x)
    (send self :point-coordinate 1 1 y)
    (send self :point-coordinate 2 1 z)
    (send self :adjust-to-data))
  )
```

The slots `'a`, `'b` and `'c` are expressions for computing the vectors of values in the spin plot. The vectors are computing by using `(eval)` on the expressions. When the expressions are evaluated the identifier `*this-graph*` is bound to the model dynamic CoCo spin plot, that receives the message `:change-models`[1]. The model dynamic spin plot holds the identification of the model of the association diagram, from which the model dynamic spin plot was created. The message `:return-vector` without the argument ⟨*model*⟩ sent to the object `*this-graph*` will then return a vector of values as if it was sent to the association diagram, from which the model dynamic spin plot was created. Methods on the object `*this-graph*` can then be used in the expressions.

The slots `'a`, `'b` and `'c` could for the CoCo-graph object `last-graph-object` be defined by, eg.:

```
(let ((a '(send *this-graph* :return-vector 'adjusted "*"
                :model 'current))
      (b '(send *this-graph* :return-vector 'adjusted "*"
                :model 'base))
      (c '(send *this-graph* :return-vector 'observed "*"
                :model nil)))
    (send last-graph-object
          :return-dynamic-coco-spin-plot a b c)
    )
```

13.1.2 Implementation

`coco-graph-window-proto :return-dynamic-coco-spin-plot` ⟨*expressions*⟩
  `&key (`⟨*location*⟩ `nil) (`⟨*title*⟩ `nil)`

`dynamic-coco-spin-proto :isnew` ⟨*dim*⟩ `&key` ⟨*location*⟩ ⟨*title*⟩
`dynamic-coco-spin-proto :expressions &optional (`⟨*val*⟩ `nil set)`
`dynamic-coco-spin-proto :nth-expression` ⟨*n*⟩
  `&optional (`⟨*val*⟩ `nil set)`
`dynamic-coco-spin-proto :values`
`dynamic-coco-spin-proto :nth-value` ⟨*n*⟩
`dynamic-coco-spin-proto :change-models`

We are then ready to give the implementation of the "Model Dynamic CoCo Spin Proto". The `dynamic-coco-spin-proto` must inherit methods from both the `spin-proto` and the `coco-graph-window-proto`. The spin plot is made from `spin-proto`. The prototype `coco-graph-window-proto` will give us the method `:return-vector`. We must add the slot `'expressions` for storing the list of the expressions for computing the vectors of values.

[1]For some reasons messages cannot be sent to `self` in the `(eval)` function.

```
(defproto dynamic-coco-spin-proto
  '(expressions) '(instances-dynamic-coco-spin)
  (list spin-proto coco-graph-window-proto))

(send dynamic-coco-spin-proto :documentation
      'proto "CoCo Spin based on spin proto type")

(defmeth dynamic-coco-spin-proto :isnew (dim &key location title)
  (let ((plot (apply #'call-method spin-proto :isnew dim
                     :location location :title (list title))))
    (send dynamic-coco-spin-proto :slot-value
          'instances-dynamic-coco-spin
          (cons self (slot-value 'instances-dynamic-coco-spin)))
    plot)
  )
```

In the method `:isnew` we keep track of all instances of the "Model Dynamic CoCo Spin Proto" so that we can send the message `:change-models` to all those instances, when one of the two the pointers **current** or **base** is moved, and when association diagrams are updated.

Then the method `:return-dynamic-coco-spin-plot` for returning a model dynamic CoCo spin plot from a CoCo graph window is implemented as follows. The two slots `'identification` and `'model-number` identify the CoCo object and the model of the CoCo graph object. The argument ⟨*expressions*⟩ gives the expressions for the vectors of the values in the plot. Since the dimension of a spin-plot just has to be greater than or equal to three, the "Model dynamic CoCo spin proto" is generalized so that we need not to give exactly three expressions, but a list of any length of expressions:

```
(defmeth coco-graph-window-proto :return-dynamic-coco-spin-plot
  (expressions &key (location nil) (title nil))
  (setf *this-graph* self)
  (let ((plot (send dynamic-coco-spin-proto
                    :new (length expressions)
                    :location location :title title)))
    (send plot :slot-value 'identification
          (slot-value 'identification))
    (send plot :slot-value 'model-number
          (slot-value 'model-number))
    (send plot :slot-value 'expressions expressions)
    (send plot :add-points (mapcar #'eval expressions))
    (send plot :adjust-to-data)
    plot)
  )
```

The method `:return-dynamic-coco-spin-plot` is called on an association diagram with a list of expressions as argument. The list should hold at least three expressions, and the expressions should return vectors of the same length. If more than three expressions are given, then the variables to display can be selected by the method `:content-variables`, see Tierney (1990).

The list of expressions for computing the vectors of values can, after the plot has been created, be returned or set by the following method:

```
(defmeth dynamic-coco-spin-proto :expressions
  (&optional (val nil set))
  (if set
      (progn (setf (slot-value 'a) val)
             (send self :change-models)))
  (slot-value 'expressions)
  )
```

and a single expression (the $\langle n \rangle$-th) can be set or returned by:

```
(defmeth dynamic-coco-spin-proto :nth-expression
  (n &optional (val nil set))
  (if set
      (progn (setf (slot-value 'a) val)
             (send self :change-models)))
  (nth n (slot-value 'expressions))
  )
```

The values of the vectors can be returned by the following method:

```
(defmeth dynamic-coco-spin-proto :values ()
  (mapcar #'eval (slot-value 'expressions))
  )
```

or the $\langle n \rangle$-th vector returned by:

```
(defmeth dynamic-coco-spin-proto :nth-value (n)
  (eval (nth n (slot-value 'expressions)))
  )
```

If only one of the vectors is wanted, there are no need to evaluate all the vectors.

The method `:change-models` is modified to allow for the list of expressions to be off any length:

```
(defmeth dynamic-coco-spin-proto :change-models ()
  (setf *this-graph* self)
  (let* ((points (mapcar #'eval (slot-value 'expressions)))
         (l (mapcar #'(lambda (x)
                        (length
                         (which (mapcar #'not (invalid-real-list x)))))
                    points)))
    (if (= (min l) (max l) (send self :num-points))
        (progn
          (if (send self :linked)
              (mapcar #'(lambda (i l x)
                          (send self :point-coordinate
                                i (iseq l) x))
                      (iseq (length points)) l points)
            (progn
              (send self :clear-points)
              (send self :add-points points)))
          (send self :adjust-to-data))))
  )
```

The method is complicated a little by invalid values and linked views. The lengths of the vectors of valid values[2] has to be equal and equal to the length of the vectors of the original plot. The call of `:add-points` ought to be faster than the several calls of `:point-coordinate`, but `:clear-points` and `:add-points` cannot be used when the plot is linked.

The methods `:make-graph-current-model` also have to redraw 'controls':

```
(defmeth coco-graph-window-proto :make-graph-current-model
  (&key (redraw-plots nil))
  (let ((c (send self :make-current (slot-value 'model-number))))
    (if (and c redraw-plots)
        (progn
          (dolist (i (send current-control-proto :slot-value
                           'instances-current-control))
                  (send i :redraw))
          (dolist (i (send dynamic-coco-spin-proto :slot-value
                           'instances-dynamic-coco-spin))
                  (if (= (send i :slot-value 'identification)
                         (send self :slot-value 'identification))
                      (send i :change-models)))))
    c)
  )
```

[2]The function `(invalid-real-list x)` is implemented as `(defun (x) (mapcar #'not (< x 2147483640)))`.

Analogously with `:make-graph-base-model`.

13.1.3 Creating a Model Dynamic Spin Plot

The following will create two "Model Dynamic Spin Plots" for the graph of the introduction:

```
(let ((a '(send *this-graph* :return-vector 'adjusted "*"
                :model 'current))
      (b '(send *this-graph* :return-vector 'adjusted "*"
                :model 'base))
      (c '(send *this-graph* :return-vector 'observed "*"
                :model nil)))
  (def graph-1-spin (send graph-1 :return-dynamic-coco-spin-plot
                           (list a b c)))
  (send graph-1-spin :location 700 500)
  )

(let ((a '(send *this-graph* :return-vector 'unadjusted "*"
                :model nil))
      (b '(send *this-graph* :return-vector 'adjusted "*"
                :model 'base))
      (c '(send *this-graph* :return-vector 'adjusted "*"
                :model 'current))
      (d '(send *this-graph* :return-vector 'observed "*"
                :model nil)))
  (def graph-2-spin (send graph-1 :return-dynamic-coco-spin-plot
                           (list a b c d)))
  (send graph-2-spin :location 500 500)
  )
```

13.2 Bootstrapping and Jackknifing

In this section we will implement a method for resampling from the case list, and do a backward elimination on each sample.

So far in this guide the edges have in the association diagram been drawn with a width growing with the reciprocal of the p-value or growing with BIC, the Bayesian information criterion. In this section we will let the width of the edges be proportional to the number of times the edges are found in the final model of a headlong backward elimination on a random subset of the cases.

The following method will return a list of the lists of significant edges in each headlong backward elimination:

```
(defmeth coco-proto :resampling-backward
  (case-list &optional (proportion 0.90) (times 100)
             &key (replacement nil) (only nil) (reversed nil)
             (sorted nil) (short T) (headlong T) (recursive T)
             (coherent T) (follow T) (least-significant T)
             (separators nil) (edges nil))
  (let* ((edge-lists nil)
         (case-list (if (listp case-list) case-list
                        (coerce case-list 'list)))
         (number-of-cases (round (* proportion
                                    (length case-list)))))
    (dotimes (i times edge-lists)
             (send self :enter-list
                   (element-seq (sample case-list number-of-cases
                                        replacement)))
             (send self :backward
                   :only only :reversed reversed
                   :sorted sorted :short short
                   :headlong headlong :recursive recursive
                   :coherent coherent :follow follow
                   :least-significant least-significant
                   :separators separators :edges edges)
             (setf edge-lists
                   (concatenate
                    'list edge-lists
                    (list (send self :return-edge-list-list
                                'last)))))
    (send self :enter-list (element-seq case-list))
    edge-lists)
  )
```

Half the code is about passing arguments to the `:backward` method. In the loop three actions are performed: A random sample from the case list is drawn with or without replacement depending on the keyword argument ⟨*replacement*⟩, `TRUE` or `nil`. The optional argument ⟨*times*⟩ is the number of samples, the backward elimination is performed on. The size of the sample is `number-of-cases`, which is computed from the optional argument ⟨*proportion*⟩. The argument ⟨*case-list*⟩ has to hold the case list, either as a list of lists, where each list is a case, or as a 2-dimensional matrix. The sampling is done by the function `(sample)` buildt into XLISP-STAT. Then the method `:backward` is used to do the backward elimination. Keyword arguments are passed to the `:backward` message. Note that the default value of ⟨*short*⟩ is set `TRUE` to reduce the output. Finally for each cycle in the loop the list of edges in the model resulting of the backward elimination is added to a list of edge-lists. A lower level version `:return-edge-list-list` of the method `:return-edge-list` is used. If the model has n edges, then `:return-edge-list-list` will return a list of length $2n$ with integers. Each pair of integers in the list is the indices of the vertices of an edge in the graph.

After the simulation, the cases in the CoCo-object are reset to the argument ⟨*case-list*⟩. Note, that because that we are entering observations, tests are disposed of in the CoCo-object.

The method returns the list of edge lists. But we want to see the simulation happening on a graph: Edges getting thicker as they are found in more backward eliminations, and other edges not found that often getting thinner. To do so, we need a function for accumulating the edge-lists, and a method for setting edge-widths:

The following function will transform the list of edge lists to an adjacency matrix: A list with half the adjacency matrix minus the diagonal is returned. Each integer in the list will denote the number of times a corresponding edge is found in the list of edge lists. If the indices of the two vertices of an edge is a and b, $a < b$, then the index of the edge in the returned list is $a + \frac{1}{2}b(b-1)$, computed by `(+ (/ (* b (1- b)) 2) a)`:

```
(defun edge-lists-to-matrix (edge-lists &optional (n nil))
  (let* ((n (if n n (1- (max edge-lists))))
         (result (repeat 0 (/ (* n (1- n)) 2))))
    (dolist (i edge-lists result)
            (dolist (j (split-list i 2) result)
                    (let ((index (+ (/ (* (cadr j)
                                          (1- (cadr j))) 2)
                                    (car j))))
                      (setf (nth index result)
                            (1+ (nth index result)))))))
  )
```

Using the results of `:edge-lists-to-matrix` we can set the widths of the edges in the association diagram:

```
(defmeth coco-graph-window-proto :set-edge-widths (edge-matrix)
  (send self :add-slot 'resampling-backward edge-matrix)
  (dolist (j (send self :edges) nil)
          (let ((index (+ (/ (* (cadr j) (1- (cadr j))) 2)
                          (car j))))
            (send self :edge-width (list (car j) (cadr j))
                  (round (/ (* 20 (nth index edge-matrix))
                            (max edge-matrix))))))
  (send self :start-buffering)
  (send self :redraw)
  (send self :buffer-to-screen)
  )
```

We then implement a simulation method for the association diagram, where we in addition to returning the list of edge lists set the edge widths by `:set-edge-widths` after each cycle in the loop, i.e., after each backward elimination:

```
(defmeth coco-graph-window-proto :resampling-backward
  (case-list &optional (proportion 0.90) (times 100)
             &key (replacement nil) (only nil) (reversed nil)
             (sorted nil) (short T) (headlong T) (recursive T)
             (coherent T) (follow T) (least-significant T)
             (separators nil) (edges nil))
  (let* ((edge-lists nil)
         (case-list (if (listp case-list) case-list
                        (coerce case-list 'list)))
         (n-of-variables (length (send self :return-names)))
         (number-of-cases (round (* proportion
                                    (length case-list)))))
    (dotimes (i times edge-lists)
             (send self :enter-list
                   (element-seq
                    (sample case-list number-of-cases
                            replacement)))
             (send self :backward :only only :reversed reversed
                   :sorted sorted :short short
                   :headlong headlong :recursive recursive
                   :coherent coherent :follow follow
                   :least-significant least-significant
                   :separators separators :edges edges)
             (setf edge-lists
                   (concatenate
                    'list edge-lists
                    (list (send self :return-edge-list-list
                                'last))))
             (send self :set-edge-widths
                   (edge-lists-to-matrix
                    edge-lists n-of-variables)))
    (send self :enter-list (element-seq case-list))
    edge-lists)
  )
```

The following is an example on the use of this method. First we set some options:

```
(send reinis-coco-object :set-rejection 0.001)
(send reinis-coco-object :set-acceptance 0.01)
```

```
(send reinis-coco-object :set-switch 'decomposable-mode 'on)
```

From the object ⟨*reinis-coco-object*⟩ of the introduction we return the case list by the method `:return-case-list` , which is implemented in next section:

```
(def case-list (send reinis-coco-object :return-case-list))
```

We then do 100 "Headlong Backward Eliminations", each on a random sample of 50 percent of the cases:

```
(setf edges
      (send graph-1 :resampling-backward case-list 0.50 100
            :replacement nil
            :only nil :reversed nil :sorted nil :short T
            :headlong T :recursive T :coherent T :follow T
            :least-significant T :separators nil :edges T))
```

The graph `graph-1` is the graph of the same name from the introduction.

13.2.1 Cases only available in a table

In the code above we did the sampling on the list of cases. But what if we only have the table of counts? The following will transform the table of counts to a case list:

The following function will return a list of cases, where each possible case occurs once, and the cases are sorted in the standard order. The argument ⟨x⟩ is a list with the number of levels at each factor. A case for each cell in the table:

```
(defun return-table-indices (x)
  (let ((pred 1)
        (post (prod x))
        (result nil))
    (dolist (i x result)
            (setf post (/ post i))
            (setf result
                  (concatenate
                   'list result
                   (list (repeat (repeat (iseq i) post)
                                 (repeat pred (* i post))))))
            (setf pred (* pred i)))
    (coerce (transpose
             (make-array (list (length x) pred)
                         :initial-contents result)) 'list))
  )
```

The following method will then, from a CoCo-object with a table of counts entered, return the case list:

```
(defmeth coco-proto :return-case-list ()
  (split-list
   (1+ (repeat
        (return-table-indices (send self :return-level-list))
        (round (send self :return-vector 'observed "*"))))
   (length (send self :return-level-list)))
  )
```

(For small tables with many cases, often the case when we only have the table of counts, it is probably more efficient to do the simulation on the table of counts. This is left as an exercise.)

13.3 Model Manager

This was first considered as something to suggest for the future, then as an exercise. Then it was implemented in two afternoons and 500 lines of XLISP-STAT code. The source code of the model manager is 1739 words, 15730 characters, but the TeX code of this very short description is 1569 words, 13643 characters. Object oriented programming is very efficient:

An *Model Manager* is an *Overview Graph*, where each point (vertex) in the graph is a model, a CoCo-graph-object. The overview graph contains a point for each instance of the CoCo-graph-object. When edges are removed and when edges are added to association diagrams, and thus new association diagrams are created, a point is added to the overview graph, and an arrow from the larger to the smaller model is drawn. When an edge is dragged between two points, two models, a test is performed. The p-value (or another statistic) of the test is set as a label of the edge.

Association diagrams can be opened and closed by key events on the corresponding points in the overview graph. The symbol of a point in the overview graph depends of the association diagram being open, closed or deleted. The position of points in the overview graph is linked to the positions of the association diagrams on the screen.

13.3.1 Creating the Model Manager

`(return-manager &optional (`⟨*identification*⟩ `nil) &key (`⟨*location*⟩ `nil)`
`(`⟨*size*⟩ `nil) (`⟨*title*⟩ `nil))`

`manager-proto :isnew &key` ⟨*location*⟩ `(`⟨*title*⟩ `nil)`
`(return-manager-positions` ⟨*graphs*⟩`)`
`(return-manager-names` ⟨*graphs*⟩`)`

`(return-graphs)`
`coco-graph-window-proto :isnew &key` ⟨*location*⟩ ⟨*title*⟩

`coco-proto :return-manager &key` ⟨*location*⟩ ⟨*size*⟩ ⟨*title*⟩
`manager-proto :return-graphs`

The *model manager* with all the CoCo graphs is created by the function `(return-manager)` with no arguments:

```
(setf manager (return-manager))
```

This function will use the method `:isnew` of `manager-proto` to return a graph window. The prototype `manager-proto` is a specialization of the prototype `association-diagram-proto`.

The function `(return-graphs)` is used to return a list with all graphs, i.e., a list with all instances of the prototype `coco-graph-window-proto`[3]. The `:isnew` method for `coco-graph-window-proto` is modified so that model managers are updated when new instances of the `coco-graph-window-proto` are created. Thus, a new point is added to the model manager when a new association diagram is created. The two functions `(return-manager-positions)` and `(return-manager-names)` are used to return the list of the locations and the list of titles of all graphs, i.e., all instances of `coco-graph-window-proto`. These two lists are returned in the format of the position list and the name list used in the `association-diagram-proto`.

The stored coordinates of the points in the model manager are the locations of the association diagrams on the window. Thus the methods `:to-x-pixel`, `:to-y-pixel`, `:from-x-pixel` and `:from-y-pixel` are modified for the model manager.

`manager-proto :x-pos-to-tex` ⟨x⟩
`manager-proto :y-pos-to-tex` ⟨y⟩
`manager-proto :to-x-pixel` ⟨x⟩
`manager-proto :to-y-pixel` ⟨y⟩
`manager-proto :from-x-pixel` ⟨x⟩
`manager-proto :from-y-pixel` ⟨y⟩

Different Model Managers

The message `:return-manager` to a CoCo-object, a CoCo-model-object or a CoCo-graph-object will return a model manager with only associations diagrams of that same CoCo object. Thus, if more than one CoCo object are active at the same time, the association diagrams of the different CoCo objects can be divided on more model managers.

[3] If you want all instances of the prototype `association-diagram-proto` to occur in the model manager, then the function `(return-graphs)` and the `:isnew` method for the `association-diagram-proto` should be edited. The relevant code is found as comments in the source file of the model manager.

The function `(return-manager)` with the optional argument ⟨*identification*⟩ is used to create this model manager. The method `:return-graphs` will only return a list with graphs with the same identification of the object to which the message is sent, if the object to which the message is sent, has a slot-value `'identification` different from `nil`.

13.3.2 Show, Hide and Close

`manager-proto :vertex-index` ⟨*variable-name*⟩
`manager-proto :vertex-type` ⟨*variable-name*⟩ `&optional (`⟨*type*⟩ `nil set)`
`&key (`⟨*redraw*⟩ `nil)`
`manager-proto :hide-vertex` ⟨*vertex-index*⟩
`association-diagram-proto :hide-window`
`manager-proto :show-vertex` ⟨*vertex-index*⟩
`association-diagram-proto :show-window`
`manager-proto :close-vertex` ⟨*vertex-index*⟩
`association-diagram-proto :close`

Association diagrams can be shown (opened) by clicking the corresponding point in the overview graph and the association diagrams can be shown, hidden and closed (deleted) by the key events `'O'`, `'H'` and `'C'` respectively on the corresponding points in the overview graph.

The symbol of a point in the overview graph depends of whether the corresponding association diagram is shown (circle), hidden (dot) or closed (triangle). The symbol of a point is determined by the value set and returned by the method `:vertex-type`.

The messages `:show-vertex`, `:hide-vertex` and `:close-vertex` are used to show, hide or/and close the association diagrams.

When graphs are shown, hidden or closed by the operating system the messages `:show-vertex`, `:hide-vertex` and `:close-vertex` are not called, and thus the symbol of the graph will not be updated. This could be fixed, if the operating system would send the messages `:show-window`, `:hide-window` and `:close`.

`manager-proto :click-vertex` ⟨*vertex-index*⟩

`manager-proto :graph-drop-edge-nth &optional` ⟨*p*⟩ `&key (`⟨*edges*⟩ `nil)`
`(`⟨*point*⟩ `nil) (`⟨*x-move*⟩ `0)`

When a vertex in the model manager is clicked then the method `:click-vertex` is called, and this method will then call the method `:show-vertex`.

Since dropping of edges is not relevant in the model manager, the method `:graph-drop-edge-nth` for a model manager will call `:click-vertex`.

13.3.3 Location

`manager-proto :drag-point` ⟨*x*⟩ ⟨*y*⟩ ⟨*m1*⟩ ⟨*m2*⟩ ⟨*point*⟩

`manager-proto :vertex-position` ⟨*variable-name*⟩
  `&optional (`⟨*position*⟩ `nil set) &key (`⟨*redraw*⟩ `nil)`
`association-diagram-proto :location`

`manager-proto :vertex-label` ⟨*variable-name*⟩
  `&optional (`⟨*label*⟩ `nil set) &key (`⟨*redraw*⟩ `nil)`
`association-diagram-proto :title`

The positions of the points in the overview graph are linked to the positions of the association diagrams on the screen. When a vertex is dragged in the model manager, then the method `:drag-point` is called, and this method is for the model manager modified to besides moving the vertex also send the message `:location` to the association diagram of the point, and the location of the association diagram on the screen will then change.

The method `:vertex-position` is for the model manager modified to also set the location of the corresponding association diagram. Analogously will the method `:vertex-label` set the title of the corresponding association diagram. New titles are in Open Windows not put on the title bar.

As with show, hide and close by the operating system there is a problem when the location of a graph window is changed by the operating system. No messages are sent, and the position of the corresponding point in the model manager is not updated. It would be nice, if the method `:location` was called, when a graph window was dragged on the screen.

`association-diagram-proto :add-graph-to-managers`
`association-diagram-proto :update-managers`
`association-diagram-proto :update-screen`

The message `:update-manager` will update the model manager so that the positions of vertices are in agreement with the location of association diagrams on the screen. The key event '`>`' in the model manager will send this message. The message `:update-screen` (key event '`<`') should be unnecessary, but will update the screen to fit the model manager.

13.3.4 current and base

`manager-proto :make-graph-current-model &key (`⟨*redraw-plots*⟩ `nil)`
  `(`⟨*key-event*⟩ `nil)`
`manager-proto :make-graph-base-model &key (`⟨*redraw-plots*⟩ `nil)`
  `(`⟨*key-event*⟩ `nil)`

The two methods `:make-graph-current-model` and `:make-graph-base-model`, which are called at the key events '`c`' and '`b`' respectively in the model manager, will make the association diagram of the vertex closest to the mouse the **current** or the **base** model.

13.3.5 Submodels and Tests

Sub- and Supermodels

`association-diagram-proto :add-manager-edge` ⟨*from*⟩ ⟨*to*⟩

`coco-graph-window-proto :graph-drop-edge-nth &optional` ⟨*p*⟩
  `&key (`⟨*edges*⟩ `nil) (`⟨*point*⟩ `nil) (`⟨*x-move*⟩ `0)`
`coco-graph-window-proto :graph-add-edge &optional` ⟨*p1*⟩ ⟨*p2*⟩
  `&key (`⟨*edges*⟩ `nil set) (`⟨*x-move*⟩ `0)`
`coco-graph-window-proto :add-fill-in`

When the message `:graph-drop-edge-nth` is sent to a CoCo-graph window, a object of prototype `coco-graph-window-proto`, then, besides creating the new association diagram, the message `:add-manager-edge` is sent to the model manager and in the model manager an arrow from the large model to the smaller model is drawn. Analogously with the messages `:graph-add-edge` and `:add-fill-in`.

Drawing Edges

`manager-proto :draw-edge` ⟨*edge*⟩ `&optional (`⟨*radius*⟩ `4)`

The method `:draw-edge` is for the model manager modified to draw an arrow from the larger model to the smaller. The order of the indices of the vertices of the edge determine which model is the larger and which is the smaller.

Tests

`manager-proto :test-two-objects` ⟨*a*⟩ ⟨*b*⟩ ⟨*p*⟩
  `&key (`⟨*decomposable-mode*⟩ `nil)`
`manager-proto :test-edge-nth` ⟨*p*⟩ `&key (`⟨*decomposable-mode*⟩ `nil)`
`manager-proto :graph-add-edge` ⟨*p1*⟩ ⟨*p2*⟩

At the key event `'e'` at an arrow between two vertices in the model manager the method `:test-edge-nth` is called. This method is for the model manager modified to use the method `:test-two-objects` to perform the test between the models of the two corresponding association diagrams. The test is computed by the method `:compute-test-against-model-object`, which for the time being only for undirected models will return the correct test. By the methods `:select-p-value` and `:format-p-value` of the model manager the result of the test is put as a label at the edge between the to vertices.

If an edge is dragged between two vertices in the model manager, the method `:graph-add-edge` is called, and if the edge is dragged to a submodel of the model, from which the edge is dragged, then an edge (arrow) is added to the model manager, and the edge is labeled with the result of the test between the two models.

Selecting Test Statistic

`manager-proto :select-p-value` ⟨*test*⟩ `&key (`⟨*print-test*⟩ `nil)`
`manager-proto :format-p-value` ⟨*p*⟩
`manager-proto :p-to-width` ⟨*p-value*⟩

The messages `:select-p-value`, `:format-p-value` and `:p-to-width` will by default call the corresponding methods of the `coco-graph-window-proto`. These methods can of course be replaced by other methods.

13.3.6 Inherited Methods

`manager-proto :do-key` ⟨*c*⟩ ⟨*m1*⟩ ⟨*m2*⟩

Also `:do-key` will by default call the corresponding methods of the prototype `coco-graph-window-proto`. Some of the key events will not result in legal messages for the model manager.

Other examples of tools for managing the data analysis are *Statistical Analysis Maps* (Oldford & Peters 1988b) and *Auditing of Data Analyses* (Becker & Chambers 1988).

Chapter 14

Discussion

14.1 The Prototypes of the Association Diagrams

This section is to give an overview of the prototypes defined in this guide. See figure 14.1.

A *CoCo object* is an object with two slots: The slot `'identification` is a reference to a structure holding the global variables for a CoCo-session. `'title` is an optional title on the object. The available methods of CoCo objects can be listed by the message `(send coco-proto :own-methods)`. Analogously can the methods owned by other prototypes be listed by the `:own-methods` method.

The *CoCo model object* is a specialization of the *CoCo object*. A *CoCo model object* is an object with three slots: `'identification` is a reference to a structure holding global variables for a CoCo-session. The slot `'model-number` is the number of a model in the CoCo-session. The slot `'title` is the title of the model object. The methods owned by the CoCo object is inherited by the CoCo model object. Some of the methods are modified for the CoCo model object. The methods modified are some of those working at a single model.

The prototype `drag-graph-proto` is general prototype with movable vertices and edges between some vertices such that the edges follows the vertices when the vertices are moved. This prototype have many applications other than the association diagram. E.g., I have used the `drag-graph-proto` to make a tool for demonstrate interpolating polynomials and splines.

The *association diagram object* is a specialization of the *drag graph object* (and of the *independence object*). (The *independence object* is just a plain object. The graph window object has precedence over the independence object.)

The *CoCo graph object* is a specialization of the *association diagram object* and of the *CoCo model object*, i.e., the CoCo graph object inherits methods

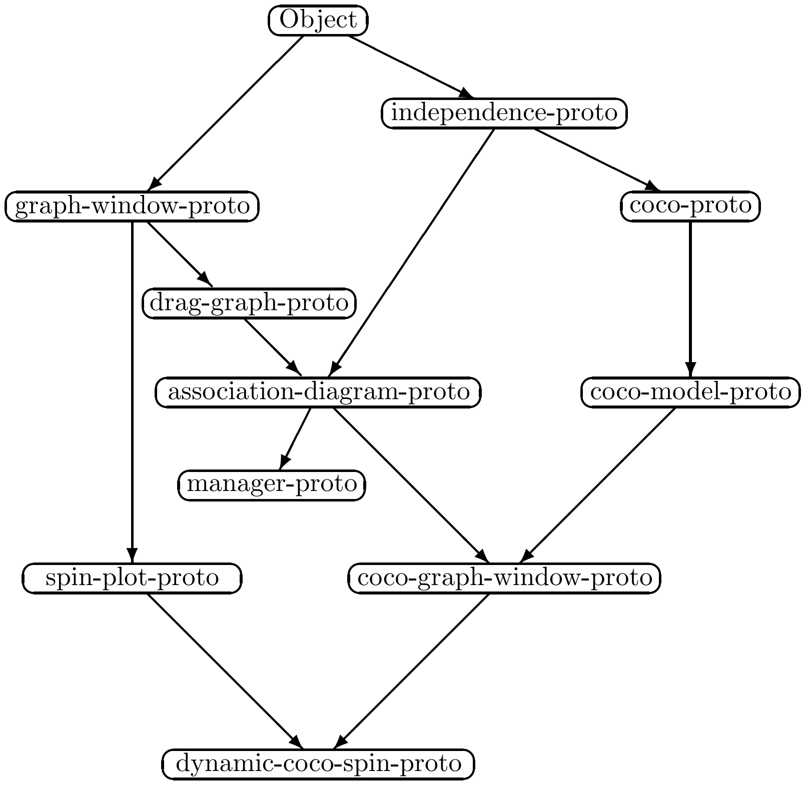


Figure 14.1: Inheritance graph for the Association Diagram Objects.

and slots from both the CoCo model object and the association diagram object. The CoCo model object has higher precedence than the association diagram object, i.e., if a method exists for both the `coco-model-proto` and the `association-diagram-proto`, then the method of the `coco-model-proto` is used.

The *model dynamic CoCo spin plot* is a specialization of both the *spin plot* and of the *CoCo graph object*. The spin object has higher precedence than the CoCo graph object.

The *model manager* is a specialization of the *association diagram object*.

14.1.1 Values Shared among Model-Objects

The independence prototype, the association diagram prototype, the CoCo graph prototype and the model dynamic CoCo spin prototype each has a slot shared among all the objects of the same prototype holding the instances of the prototype.

But besides that values are shared between objects by these slots, messages to a CoCo model object will not only effect the model object, but also the corresponding CoCo object and all CoCo model objects for that CoCo object, e.g., the message `:set-exact-test 'on` to a CoCo model object will set computation of exact test on for the object, the corresponding CoCo object and all CoCo model objects for that CoCo object.

That an option only is to be changed on one model object or graph object of an CoCo object to be changed on all model- and graph objects of that CoCo object is very convenient. One have only to give a single command to change an options for many object. But it can also be confusing to new users. Especially since some options have been added as slots to the graph objects, and thus is owned by the individual graphs and not shared by others, and since some options can be given as arguments to methods.

14.2 Why XLISP-STAT?

In Tierney (1990) of course a lot of reasons are given for using Lisp as a platform for a computer environment for statistical computing. Some of them are contained in the following:

- `Interactive:` The days of computer interaction limited to punctuation cards and batch jobs are long gone. In a computer environment for statistical computation it must be possible to write expressions such as, e.g., the mean of the elements of a vector divided by the square rote of the variance of the same elements, or a matrix product involving the inverse of a matrix. The values of these expressions should appear immediately on the screen without the tiresome task of, e.g., calling a compiler. Thus the computer environment for statistical computation must contain a *high-level* **interactive** *programming language.*

- `Incremental:` One may need to evaluate the same expressions over and over again, but on different sets of data, and thus it should be possible to write functions in the computer environment, i.e., the language should be incremental.

- `Object oriented:` Experience has shown that object oriented programming is very useful for graphics programming in general and statistical graphics programming in particular (Stuetzle 1987, McDonald & Pedersen 1988, Hitz & Hudec 1994). Object-oriented methods have also many other applications within a statistical system (Tierney 1990). They can be used for developing flexible data structures (McDonald 1986, Stuetzle 1987) and for representing statistical models (Oldford & Peters 1988a).

Instead of using an object oriented language one could have used a functional programming language. This approach is for generalized linear models tried in Gilchrist & Scallan (1988). Functional languages are in some sense the opposite of object-oriented languages. In a functional language there is no state, i.e., no objects holding variables, so there can be no confusion about which values are owned be which objects, and thus debugging is easier. But graphics programming would be a challenge in a functional language, since there is no way of storing the characteristics of the graph windows in a functional language.

- `Extendable:` A very important feature of the computer environment for this current project is that the language is *extendable*, meaning that functions and procedures written in some other languages, e.g., C, PASCAL and FORTRAN, can be loaded into the computer environment after compilation, and then used as builtin functions in the computer environment.

- `''Statistical functions'':` Vectorized algebra, matrices and operations on these, probability functions, graphics, etc. These are some of the features, important for statisticians, added to Xlisp by Luke Tierney to get XLISP-STAT.

Several other languages have been suggested and used as the basis for statistical environments. A language that has received considerable attention is APL. APL has many useful features especially for statisticians, including a wide range of functions for handling matrices and arrays. But it does not have the ability to easily handle high-level data, such as functions or expressions, nor does it lead itself readily to support the object-oriented programming style that is so important for graphical programming. Adding these features to APL appears to be considerably harder than adding matrices and functions to Lisp (Tierney 1990).

In stead of adapting an existing language for statistical computation one could develop a new high-level language from scratch. This approach has been taken in developing the S system (Becker, Chambers & Wilks 1988, Chambers & Hastie 1991).

The S system is also extendable. CoCo can be loaded into this system, and the most basic interface functions have been made. But S-Plus does not support graphics programming as well as XLISP-STAT, and I found the programming in XLISP-STAT more elegant than the programming in S-Plus. Thus the work of writing interface functions between CoCo and S-Plus has been terminated and no graphical interface is made.

14.3 Further Developments

The current tool may be considered an ad-hoc development of CoCo.

To handle mixed interaction models another approach should be adopted. The tool should be built around the hierarchy of models. For each class of models an estimation function should be implemented. The CoCo object with the data should be removed. Functions for fitting models, returning fitted values from models, testing models against each other and search routines should be extracted from the code of CoCo. Necessary data to the resulting functions should be given as arguments to the functions. This will be at the cost of some computational speed, e.g., because the code for reusing already computed tests thus cannot be implemented as efficiently as in CoCo.

It should be possible to put variables (objects with information whether they are discrete, ordinal, continuous, which variables the variable is a response variable to etc.) into a model, and then the model object should be specialized appropriately to whether the model is a general mixed interaction model, a linear regression, a log linear model on a contingency table, a block recursive model, etc. These variable objects could also know where to position themselves in the association diagram, etc. Hierarchy of models is discussed in Anglin & Oldford (1993) and Tierney (1991). The notion of data frames of Chambers & Hastie (1991), i.e., a collection of named vectors of the same length, may also be useful here.

- `Speed:` The current implementation of the association diagrams has been made to investigate the usefulness of these diagrams, and the diagrams are implemented so new features can easily be added, but no attempts to make fast code have been made.

  The performance of the current implementation is fine for small graphs, i.e., graphs with less than 10 variables. Editing the layout of a graph with 40 variables is bearable. The code for the drawing of the graphs should be optimized.

- `Variables as objects:` Variables should be objects (Oldford & Peters 1986, Oldford 1988), and when the variables are put into a model, the model object should be created appropriately as a pure discrete model (a CoCo model), a linear regression, a mixed interaction model, a block recursive model, etc. as discussed above.

- `Graphical interface to variables:` It should be possible to click the vertex for opening a dialog window for setting vertex position, vertex label, position of vertex label, variable type, stratum, list marginal table, list values, etc.

- `Model formulas:` Besides building the models by graphical interaction it should also be possible to specify the models by commands. An extension of the GLIM-language (Wilkinson & Rogers 1979) would be nice. This language is also used in Chambers & Hastie (1991). A language for mixed interaction models is considered in Edwards (1989) and Edwards (1990).

- `Mixed models:` The tool should be extended to handle mixed models: The symbols should depend on the type of variable: Continuous (circles) and discrete (dots). The association diagram is ready for this change. Discrete nominal variables are drawn with dots and ordinals with squares.

  The problem is the estimation in the mixed causal models and the hierarchy of models. Algorithms for estimation in mixed undirected models are implemented in MIM (Edwards 1989).

- `Hypergraphs:` The tool should be extended to handle hierarchical non-graphical models. It should also be possible to draw graphs as hypergraphs.

- `Methods without graphs:` It should be possible to declare a CoCo model object as a block recursive model, and then perform a model search among block recursive models on the model without the graph. This will also enable a model search on undirected models by a criteria programmed by the end user.

Part IV

Appendix

Appendix A

Installing CoCo

A.1 Availability

The source code in C, executable for Sun 4 (Sparc) and executable of the standalone version of CoCo for PC's and Macintosh for this version of CoCo is available free of charge for non-commercial use.

The source code may only be read and edited for the purpose of porting CoCo to other machines. No new features may be added to CoCo and no parts of the program may be included in other systems or new interface-procedures to New S, S-Plus, XLISP-STAT or any other extendable system may be made without the written permission from the author.

The source code in C and executable for Sun 4 (Sparc), both with Lisp-code for the graphics, and executable of the standalone version of CoCo for PC's and Macintosh can be obtained by anonymous *ftp* over internet from *ftp.iesd.auc.dk*, or by *WWW* from *http://www.iesd.auc.dk/pub/packages/CoCo*. Or CoCo is available on disks (3.5" or 5.25" High density). You should, however, by prepared to bear the costs of copying, e.g., by supplying a disk or tape and a stamped mailing envelope. This guide and the guide to CoCo is also available in TeX and postscript code from the ftp site or printed from Aalborg University.

If you have problems with CoCo or find bugs, please contact the author. If possible, send on disk or by Email the `LogFile` for the session which caused the problem. I will do my best to solve your problems and provide updates, if necessary.

If you have comments on CoCo, suggestions for improvements or new facilities that you feel would make the software more useful to you, please feel free to contact me.

Jens Henrik Badsberg
Department of Mathematics and Computer Science
Institute for Electronic Systems, Aalborg University

Fredrik Bajers Vej 7
DK-9220 Aalborg, DENMARK
Fax: +45 98 15 81 29
Phone: +45 98 15 85 22 ext. 5074
Email: coco@iesd.auc.dk

A.2 Installation

The following section is the installation guide for both the standalone version of CoCo and of the interface functions for the graphics. The graphics will only run on Unix systems with XLISP-STAT. XLISP-STAT must on your system be compiled so it is able to do dynamic loading.

A.2.1 DOS: On PC

Read the latest `READ.ME` and `README.DOS` file.

Copy the contents of the distribution disk into a directory, the *home directory* for CoCo. Add the home directory of CoCo to your path.

The files `COCO.EXE`, `COCO.OVR` and `COCO.TAB` must be in the home directory for CoCo. The two files `COCO.HLP` and `COCO.DAT` should also be there. If the `HelpFile` `COCO.HLP` not is in the directory, there will be no online help information available in CoCo. The default `DataFile` `COCO.DAT` can be replaced by another data-file.

By `copy/b coco.exe + coco.ovr` the overlay file is attached to the end of the `.EXE` file `COCO.EXE`. You can then remove `COCO.OVR`.

Type `COCO` to start CoCo.

Test examples are found in the directory `EXAMPLES`. Run the test examples with `runall` or `runsome`. `runall` will take approximately 24 hours and use three Mbytes of diskspace to diaries. Datasets and CoCo-source-files are found in the directory `DATASETS`.

CoCo runs on IBM/XT/ATs (and compatible) with or without coprocessor 8087/80287/80387 and with or without expanded memory (EMS). If the numeric coprocessor is not present, it is emulated. If EMS is present, the overlay file is loaded into the expanded memory when sufficient space is available. See the sections “DOS: Overlays” in the chapter “Miscellaneous Options for ...” in “A Guide to CoCo” about how to control the size of the overlay buffer.

See the `README.DOS` file for `Run-Time Errors`.

A.2.2 Macintosh

Read the disk with CoCo for Macintosh or uncompress the “sit.Hqx” file. Click the folder “CoCo” to start CoCo.

A.2.3 Unix: On Workstations

Read the latest `READ.ME` and `README.Unix` file.

Copy the contents of the disk or tape into a directory, the build directory of CoCo. Or obtain relevant files by anonymous *ftp* over internet from *ftp.iesd.auc.dk* or by *WWW* from *http://www.iesd.auc.dk/pub/packages/CoCo* and uncompress. Change to the directory.

1) Set the variables

```
BINDIR
COCOHOME
SCOCOHOME
XCOCOHOME
```

to appropriate values in the Makefile. `$BINDIR` (e.g., /home/local/bin) is where the script for starting CoCo goes. The value of `$COCOHOME` (e.g., /home/local/lib/coco) is the *home directory* of CoCo. `$SCOCOHOME` (e.g., /home/local/lib/coco or /home/local/lib/Splus/local) is where the functions to use CoCo in `S-Plus` go. The `XLISP-STAT`-interface-functions are copied to `$XCOCOHOME` (e.g., /home/local/lib/coco/lsp).

2a) Move the executable `coco.sparc` or `coco.sun3` to `coco` by, e.g., the command `mv coco.sparc coco`, unless you have uncompressed a tar-file with only one file `coco`.

or

2b) Compile CoCo by 'make coco'.

Installation of CoCo is then done by:

3) Type 'make installcoco'. This will copy necessary files (`coco`, `COCO.TAB`, `COCO.HLP` and `COCO.DAT`) to the *home directory* `COCOHOME` for CoCo, edit the script CoCo (insert the environment variable `COCOHOME`) and place the result in `BINDIR`.

You only need to do step 4 and 5 or 6 if you want to use CoCo in S-Plus or XLISP-STAT:

4a) Move the object files `scoco.sparc` or `scoco.sun3` to `scoco.o` by, e.g., the command `mv scoco.sparc scoco.o`, unless you have uncompressed a tar-file with only one executable file `scoco.o`.

or use

4b) 'make scoco.o' to make the object file `scoco.o`.

To install S+CoCo, do

5a) Edit also the environment variable `S`, if necessary.

5b) Type 'make installscoco'. This will edit the script `S+coco` (insert the environment variables `S`, `SCOCOHOME` and `COCOHOME`) and then place the result in `BINDIR`, copy `scoco.o` to `COCOHOME`, make the interface-functions to S-Plus by running S-Plus on `cocoapi.S` and copy the resulting `S-functions` to `SCOCOHOME/.Functions`.

The CoCo-S-functions `SCOCOHOME/.Functions` could be moved to some directory in the default S-Plus search list. The command `attach(paste(getenv( "SCOCOHOME" ), "/.Functions", sep = "" ))` is then unnecessary to get access to the CoCo-S-functions.

To install Xlisp+CoCo, do

6a) Set the environment variable `XCOCO` to `scoco.o`, if you are using XLISP-STAT version 2.1 Release 2, or to `libscoco.so`, if you are using XLISP-STAT 2.1 Release 3.39 (or later). Edit also the environment variable `XLISPSTAT`, if necessary.

6b) Type 'make installxcoco'. This will edit the script `xlisp+coco` (insert the two environment variables `XLISPSTAT`, `XCOCOHOME` and `COCOHOME`) and place the result in `BINDIR`, write the file `loadcoco.lsp` for loading necessary interface files, copy `scoco.o` (or `libscoco.so`) to `COCOHOME`, and copy the files `cocoapix.lsp`, `cocometh.lsp`, `cocograph.lsp`, etc. to `XCOCOHOME`. *Do not do step "manually", since other actions might have been added since the writing of this.*

XLISP-STAT must be compiled with `FOREIGN_FLAG = -DFOREIGNCALL` to make dynamic loading working. See the 'Makefile' for XLISP-STAT.

If you are running `Solaris`, then you should use version 2.1 Release 3.39 (Beta) or later of XLISP-STAT.

After successfully having installed Xlisp+CoCo you should compile the Lisp-file for Xlisp+CoCo. Change to the directory `XCOCOHOME` and type 'make'. This will compile the Lisp-files for Xlisp+CoCo and create a saved workspace for Xlisp+CoCo. By the saved workspace the upstart of Xlisp+CoCo will be much faster, and by the byte-compilation the program will run faster.

`XLISP-STAT` by Luke Tierney can be obtained by anonymous *ftp* over internet from *umnstat.stat.umn.edu* (128.101.51.1) or mail a message containing the line

```
send index from xlispstat
```

to *statlib@templer.stat.cmu.edu* for how to find out to obtain XLISP-STAT from the `statlib` archive. XLISP-STAT can be obtained free of charge.

`fep` is a very useful tool, when running CoCo and XLISP-STAT. In the manual page to `fep`, the general purpose front end processor by K. Utashiro, Software Research Associates, Inc., Japan, the Email address utashiro@sra.junet is found. `fep` can be obtained free of charge.

`S-Plus` by Statistical Sciences, Inc, P.O. Box 85625, Seattle, WA 98145-1625, U.S.A. is a commercial product. It adds features to `New S` by Becker, Chambers and Wilks, see Becker et al. (1988) for New S. Electronic mail, Europe: sales@statsci.co.uk, mktg@statsci.co.uk, support@statsci.co.uk; USA: sales@statsci.com.

A.3 Reporting Errors

Report errors by sending the `LogFile` (DOS: COCO???.LOG, Macintosh: CoCo.log.no and Unix: /tmp/CoCo.log.004.X(pid-number)) and data-files to the author. Please note whether the error appears again, if you run CoCo with the `LogFile` as input. On DOS: rename the `LogFile` before starting CoCo again. Note machine-type, operating system, version of operating system and version of CoCo.

A.4 Warranty

NO WARRANTY

I PROVIDE ABSOLUTELY NO WARRANTY. EXCEPT WHEN OTHERWISE STATED IN WRITING, I AND/OR OTHER PARTIES PROVIDE CoCo "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU, SHOULD THE CoCo PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COSTS OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

IN NO EVENT I AND/OR ANY OTHER PARTY WHO MAY MODIFY AND REDISTRIBUTE CoCo, MAY BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY LOST PROFITS, LOST MONIES, OR OTHER SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY THIRD PARTIES OR A FAILURE OF THE PROGRAM TO

OPERATE WITH PROGRAMS NOT DISTRIBUTED BY ME) THE PROGRAM, EVEN IF YOU HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES, OR FOR ANY CLAIM BY ANY OTHER PARTY.

Appendix B

Quick Reference Card

B.1 coco-proto

B.1.1 End and Restart

:end
:resume

(make-coco &key ($\langle n \rangle$ 65535) ($\langle p \rangle$ 65535) ($\langle q \rangle$ 1023) ($\langle title \rangle$ nil))

:isnew $\langle$*coco-id*[1]$\rangle$ &key $\langle title \rangle$
:title &optional ($\langle title \rangle$ nil set)
:current-coco

(quit)

B.1.2 Help and Quick-Reference-Card

These CoCo-commands concerning the parser in CoCo are not useful in Lisp.

B.1.3 Abbreviations

These CoCo-commands concerning the parser in CoCo are not useful in Lisp.

B.1.4 Status

:status &optional ($\langle code \rangle$ 'all)

[1] Arguments not specified in this section are either booleans or text-strings, e.g., ⟨*file-name*⟩, ⟨*set*⟩, ⟨*gc*⟩, ⟨*models*⟩, ⟨*factor-name*⟩, ⟨*name*⟩, ⟨*names*⟩, etc. or numbers, e.g., ⟨*a*⟩, ⟨*b*⟩, ⟨*no*⟩, ⟨*epsilon*⟩, ⟨*max*⟩, ⟨*width*⟩, ⟨*decimals*⟩, etc. and for a few messages, lists of integers (or reals): ⟨*cell*⟩, ⟨*list*⟩, ⟨*table*⟩, ⟨*cutpoints*⟩, etc.

$\langle code \rangle \longrightarrow \{$ `'all` | `'formats` | `'tests` | `'exact` | `'fix` | `'ips`
| `'em` | `'specification` | `'observations` | `'limits` | `'files`
| `'other` | `'search` $\}$

B.1.5 Input from Keyboard or File

`:set-switch 'keyboard &optional (`$\langle hit \rangle$` 'flop)`

B.1.6 Input Files

`:set-specification-file` $\langle \textit{file-name} \rangle$
`:set-observations-file` $\langle \textit{file-name} \rangle$
`:set-data-file` $\langle \textit{file-name} \rangle$
`:coco-source` $\langle \textit{file-name} \rangle$

B.1.7 Output Files

`:set-diary-file` $\langle \textit{file-name} \rangle$
`:set-report-file` $\langle \textit{file-name} \rangle$
`:set-log-file` $\langle \textit{file-name} \rangle$
`:set-dump-file` $\langle \textit{file-name} \rangle$
`:set-switch 'keep-diary &optional (`$\langle hit \rangle$` 'flop)`
`:set-switch 'keep-report &optional (`$\langle hit \rangle$` 'flop)`
`:set-switch 'keep-log &optional (`$\langle hit \rangle$` 'flop)`
`:set-switch 'keep-dump &optional (`$\langle hit \rangle$` 'flop)`

B.1.8 Diary, Timer etc.

`:set-switch 'diary &optional (`$\langle hit \rangle$` 'flop)`
`:set-switch 'timer &optional (`$\langle hit \rangle$` 'flop)`
`:set-switch 'echo &optional (`$\langle hit \rangle$` 'flop)`
`:set-switch 'note &optional (`$\langle hit \rangle$` 'flop)`
`:set-switch 'report &optional (`$\langle hit \rangle$` 'flop)`
`:set-switch 'trace &optional (`$\langle hit \rangle$` 'flop)`
`:set-switch 'debug &optional (`$\langle hit \rangle$` 'flop)`
`:set-switch 'log &optional (`$\langle hit \rangle$` 'flop)`
`:set-switch 'log-data &optional (`$\langle hit \rangle$` 'flop)`
`:set-switch 'dump &optional (`$\langle hit \rangle$` 'flop)`

`:set-switch` $\langle switch \rangle$ `&optional (`$\langle hit \rangle$` 'flop)`
$\langle switch \rangle \longrightarrow \{$ `'Keyboard` | `'Diary` | `'Log` | `'Log-Data` | `'Dump`
| `'Keep-Diary` | `'Keep-Report` | `'Keep-Log` | `'Keep-Dump`
| `'Timer` | `'Echo` | `'Note` | `'Report` | `'Trace` | `'Debug`
| `'Partitioning` | `'Graph-mode` | `'Decomposable-mode`
| `'Graphical-search` | `'Pausing-of-output`

| 'Short-test-output | 'Reuse-tests | 'Adjusted-df
| 'Exact-test | 'Exact-only-log-l | 'Fast | 'Exact-test-total
| 'Exact-test-parts | 'Exact-test-unparted | 'Large | 'Huge
| 'Sorted }
⟨*hit*⟩ ⟶ { 'on | 'flop | 'off | 'what }

B.1.9 Print Formats

:set-print-formats ⟨*width*⟩ ⟨*decimals*⟩
:set-table-formats ⟨*width*⟩ ⟨*decimals-probabilities*⟩ ⟨ *-expected*⟩
⟨ *-residuals*⟩
:set-test-formats ⟨*statistic-width*⟩ ⟨*statistic-decimals*⟩ ⟨*probabilities-width*⟩
⟨*probabilities-decimals*⟩
:set-page-formats ⟨*line-length*⟩ ⟨*page-length*⟩
:set-switch 'short-test-output &optional (⟨*hit*⟩ 'flop)
:set-switch 'pause &optional (⟨*hit*⟩ 'flop)
:set-paging-lenght ⟨*length*⟩

B.1.10 Controlling the IPS- and EM-algorithm

:set-ips-stop-criterion &optional (⟨*code*⟩ 'cell)
⟨*code*⟩ ⟶ { 'cell | 'sum }
:set-ips-epsilon &optional (⟨*epsilon*⟩ 0.0000001)
:set-ips-max-iterations &optional (⟨*max*⟩ 100)

:set-em-initial &optional (⟨*code*⟩ 'uniform)
⟨*code*⟩ ⟶ { 'uniform | 'first | 'last | 'mean | 'random
| 'input }
:set-em-epsilon &optional (⟨*epsilon*⟩ 0.001)
:set-em-max-iterations &optional (⟨*max*⟩ 100)

B.1.11 Partitioning and Factories

:set-switch 'partitioning &optional (⟨*hit*⟩ 'flop)
:set-algorithm &optional (⟨*code*⟩ 'a)
⟨*code*⟩ ⟶ { 'a | 'b | 'c }

B.1.12 Do only Graph Stuffs, only for Debugging

:set-switch 'graph-mode &optional (⟨*hit*⟩ 'flop)

B.1.13 Visit only Decomposable Models in Stepwise Model Selection and in the Global EH Search

:set-switch 'decomposable-mode &optional (⟨*hit*⟩ 'flop)

B.1.14 Computed Test-Statistics and Choosing Tests and Significance Level for Model Selection

`:set-switch 'adjusted-df &optional (`⟨*hit*⟩ `'flop)`
`:set-power-lambda &optional (`⟨*lambda*⟩ `0.666667)`
`:set-switch 'reuse-test &optional (`⟨*hit*⟩ `'flop)`

`:set-test &optional (`⟨*code*⟩ `'lr)`
  ⟨*code*⟩ ⟶ { `'lr` | `'pearson` | `'power` }
`:set-ic &optional (`⟨*code*⟩ `'aic) (`⟨*kappa*⟩ `2.0)`
  ⟨*argument*⟩ ⟶ { `'on` | `'off` | `'aic` | `'bic` | `'kappa` ⟨*integer*⟩ }
`:set-acceptance &optional (`⟨*alfa-accepted*⟩ `0.05)`
`:set-rejection &optional (`⟨*alfa-rejected*⟩ `0.025)`
`:set-components &optional (`⟨*components-limit*⟩ `0.01)`
`:set-separators &optional (`⟨*separators-limit*⟩ `0.001)`

B.1.15 Exact Tests in Tests and Model Selection

`:set-exact-test &optional (`⟨*code*⟩ `'flop)`
  ⟨*code*⟩ ⟶ { `'on` | `'off` | `'flop` | `'all` | `'deviance` }
`:set-asymptotic &optional (`⟨*limit*⟩ `0.25)`
`:set-number-of-tables &optional (`⟨*number*⟩ `1000)`
  ⟨*number*⟩ ⟶ { ⟨*integer*⟩ | `'variating` | `'what` }
`:set-list-of-number-of-tables` ⟨*list-of-number-of-tables*⟩
`:set-switch 'exact-test-for-total-test &optional (`⟨*hit*⟩ `'flop)`
`:set-switch 'exact-test-for-parts &optional (`⟨*hit*⟩ `'flop)`
`:set-switch 'exact-test-for-unparted &optional (`⟨*hit*⟩ `'flop)`
`:set-seed &optional (`⟨*seed*⟩ `'random)`
  ⟨*seed*⟩ ⟶ { `'random` | ⟨*integer*⟩ | `'what` }
`:set-exact-epsilon &optional (`⟨*epsilon*⟩ `0.0000001)`
`:set-switch 'fast &optional (`⟨*hit*⟩ `'flop)`

B.1.16 Read Data without Selection etc. from Data-File

`:read-data`

B.1.17 Read Data: Specification

`:read-specification`
`:read-factors`
`:read-names`

`:set-read` ⟨*code*⟩ `&optional` ⟨*subset*⟩
  ⟨*argument*⟩ ⟶ { `'all` | `'subset` ⟨*string*⟩ }
`:set-datastructure &optional (`⟨*code*⟩ `'all)`

⟨*code*⟩ ⟶ { `'all` | `'necessary` | `'file` }
`:set-switch 'sorted &optional (`⟨*hit*⟩ `'flop)`

Declare Subset of Variables to be Ordinal

`:set-ordinal` ⟨*set*⟩

Enter Specification of Variables from Lisp

`:enter-names` ⟨*list of names*⟩ ⟨*list of number of levels*⟩ `&optional`
`(`⟨*list of number of levels to be marked as missing*⟩ `nil)`

`:return-name-list &key (`⟨*full*⟩ `nil)`
`:return-level-list &key (`⟨*full*⟩ `nil)`
`:return-missing-list &key (`⟨*full*⟩ `nil)`
`:return-names &key (`⟨*full*⟩ `nil)`

B.1.18 Read Data: Selection of Cases form Case-List

`:select-cases` ⟨*set*⟩ `&optional` ⟨*cell*⟩
`:or-select-cases` ⟨*set*⟩ `&optional` ⟨*cell*⟩
`:reject-cases` ⟨*set*⟩ `&optional` ⟨*cell*⟩
`:or-reject-cases` ⟨*set*⟩ `&optional` ⟨*cell*⟩

Redeclaring a Factor for Cutpoints after Entering of Specification

`:redefine-factor` ⟨*name*⟩ ⟨*levels*⟩ ⟨*missing levels*⟩
`:cutpoints` ⟨*name*⟩ ⟨*cutpoints*⟩

B.1.19 Skip Cases with Missing Values During Reading Observations

`:skip-missing`

B.1.20 Read Data: Observations

`:read-observations`
`:read-table`
`:read-list`

Enter Cases from Lisp as Table or List

`:enter-table` ⟨*counts: list of integers*⟩
`:enter-list` ⟨*cases: list of integers*⟩ `&key (`⟨*accumulated*⟩ `nil)`
`(`⟨*ncol*⟩ `nil) (`⟨*select-case-fun*⟩ `nil set) (`⟨*columns*⟩ `nil subset)`

⟨*select-case-fun*⟩ ⟶ { `function on a case (list of integers): returning the value 'True' for cases to be selected.` }

`:enter-names-and-list` ⟨*case-list*⟩ ⟨*names*⟩ ⟨*levels*⟩
`&key (`⟨*missing levels*⟩ `nil) (`⟨*select-case-fun*⟩ `nil set)`
`(`⟨*columns*⟩ `nil subset)`

Replace Observation with a Random Table with Margins as in the current Model

`:substitute`

B.1.21 Read Data: Q-Tables: Incomplete Table = Structural Zeros and Initial Values for the IPS-Algorithm

`:enter-q-table` ⟨*set*⟩ ⟨*table: list of integers*⟩
`:enter-q-list` ⟨*set*⟩ ⟨*cells: list of integers*⟩

Remove Cases in Cells Zero by Structure

`:clean-data`

B.1.22 Select Use Only Cases with Complete Observations after Reading Observations

`:exclude-missing &optional (`⟨*code*⟩ `'flop) (`⟨*set*⟩ `";")`
⟨*argument*⟩ ⟶ { `'on` | `'off` | `'in` ⟨*string*⟩ }

B.1.23 Request the EM-algorithm

`:em-on`

B.1.24 Data Description

`:return-vector` ⟨*type*⟩ ⟨*set*⟩ `&key (`⟨*permuted*⟩ `T) (`⟨*model*⟩ `'current)`
`(`⟨*random*⟩ `nil) (`⟨*complete*⟩ `nil)`
⟨*type*⟩ ⟶ { `'counts` | `'probabilities` | `'expected` | `'unadjusted` | `'f-res` | `'r-f` | `'g-res` | `'r-g` | `'adjusted` | `'m-res` | `'standardized` | `'deviance` | `'freeman-tukey` | `'sqrt` | `'power` | `'index` | `'zero` }
⟨*model*⟩ ⟶ { `'current` | `'base` | `'last` | ⟨*integer*⟩ }
`:return-matrix` ⟨*type-list*⟩ ⟨*set*⟩ `&key (`⟨*permuted*⟩ `T) (`⟨*model-list*⟩ `nil)`
`(`⟨*random-list*⟩ `nil) (`⟨*complete-list*⟩ `nil)`

⟨*type-list*⟩ ⟶ { `(list [` ⟨*type*⟩ `] )` }
⟨*type*⟩ ⟶ { `See :return-vector` }
⟨*model-list*⟩ ⟶ { `(list [` ⟨*model*⟩ `] )` }
⟨*model*⟩ ⟶ { `See :return-vector, etc.` }
`:print-table` ⟨*type*⟩ ⟨*set*⟩ `&key (`⟨*permuted*⟩ `T) (`⟨*model*⟩ `'current)`
`(`⟨*random*⟩ `nil) (`⟨*log-trans*⟩ `nil) (`⟨*complete*⟩ `nil)`
⟨*model*⟩ ⟶ { `'current` | `'base` | `'last` | ⟨*integer*⟩ }
`:describe-table` ⟨*type*⟩ ⟨*set*⟩ `&key (`⟨*probit*⟩ `nil) (`⟨*rankit*⟩ `nil)`
`(`⟨*uniform*⟩ `nil) (`⟨*model*⟩ `'current) (`⟨*random*⟩ `nil) (`⟨*log-trans*⟩ `nil)`
`(`⟨*complete*⟩ `nil)`
⟨*model*⟩ ⟶ { `'current` | `'base` | `'last` | ⟨*integer*⟩ }
`:plot` ⟨*x*⟩ ⟨*y*⟩ ⟨*set*⟩ `&key (`⟨*X-model*⟩ `'current) (`⟨*X-random*⟩ `nil)`
`(`⟨*X-log*⟩ `nil) (`⟨*Y-model*⟩ `'current) (`⟨*Y-random*⟩ `nil) (`⟨*Y-log*⟩ `nil)`
`(`⟨*complete*⟩ `nil)`
⟨*X-model*⟩, ⟨*Y-model*⟩ ⟶ { `'current` | `'base` | `'last`
| ⟨*integer*⟩ }
:print-sparse-table ⟨*set*⟩
`:list-values` ⟨*set*⟩
`:case-list` ⟨*set*⟩

B.1.25 Read Model

`:read-model` ⟨*gc*⟩
`:read-n-interactions` ⟨*order*⟩ ⟨*set*⟩

Make Model or Graph for Model

`:make-model &optional (`⟨*model*⟩ `'current) &key (`⟨*title*⟩ `nil)`
⟨*model*⟩ ⟶ { ⟨*gc*⟩ | `'current` | `'base` | `'last` | ⟨*integer*⟩ }
`:make-graph &key (`⟨*model*⟩ `'current) (`⟨*location*⟩ `nil) (`⟨*size*⟩ `nil)`
`(`⟨*title*⟩ `nil)`
⟨*model*⟩ ⟶ { ⟨*gc*⟩ | `'current` | `'base` | `'last` | ⟨*integer*⟩ }

B.1.26 Edit Model

`:collaps-model` ⟨*set*⟩
`:normal-to-dual &optional (`⟨*only*⟩ `nil)`
`:dual-to-normal &optional (`⟨*only*⟩ `nil)`

B.1.27 Moving Pointers in the Model-List

`:base`
`:current`
`:make-base` ⟨*no*⟩

⟨*no*⟩ ⟶ { ⟨*integer*⟩ | `'previous` | `'next` | `'current` | `'last` }
`:make-current` ⟨*no*⟩
⟨*no*⟩ ⟶ { ⟨*integer*⟩ | `'previous` | `'next` | `'base` | `'last` }

Return Number Model to Lisp

`:return-model-number` ⟨*model*⟩
⟨*model*⟩ ⟶ { `'current` | `'base` | `'last` }

B.1.28 Return, Describe, Print and Dispose of Models

`:print-formula`
`:print-vertex-order`
`:dispose-of-formula`

`:print-model &optional` (⟨*model*⟩ `'current`) ⟨*a*⟩ ⟨*b*⟩
⟨*argument*⟩ ⟶ { `'current` | `'base` | `'last` | `'all`
| `'number` ⟨*integer*⟩ | `'interval` ⟨*integer*⟩ ⟨*integer*⟩
| `'list` ⟨*list of integer*⟩ }
`:describe-model &optional` (⟨*model*⟩ `'current`) ⟨*a*⟩ ⟨*b*⟩
⟨*argument*⟩ ⟶ { `'current` | `'base` | `'last` | `'all`
| `'number` ⟨*integer*⟩ | `'interval` ⟨*integer*⟩ ⟨*integer*⟩
| `'list` ⟨*list of integer*⟩ }
`:dispose-of-model &optional` (⟨*model*⟩ `'current`) ⟨*a*⟩ ⟨*b*⟩
⟨*argument*⟩ ⟶ { `'current` | `'base` | `'last` | `'all`
| `'number` ⟨*integer*⟩ | `'interval` ⟨*integer*⟩ ⟨*integer*⟩
| `'list` ⟨*list of integer*⟩ }

Return Characteristics of Model

`:is-graphical &optional` (⟨*model*⟩ `'current`)
⟨*model*⟩ ⟶ { ⟨*gc*⟩ | `'current` | `'base` | `'last` | ⟨*integer*⟩ }
`:is-decomposable &optional` (⟨*model*⟩ `'current`)
⟨*model*⟩ ⟶ { ⟨*gc*⟩ | `'current` | `'base` | `'last` | ⟨*integer*⟩ }
`:is-submodel-of &optional` (⟨*model-1*⟩ `'current`) (⟨*model-2*⟩ `'base`)
⟨*model-1*⟩, ⟨*model-2*⟩ ⟶ { ⟨*gc*⟩ | `'current` | `'base` | `'last`
| ⟨*integer*⟩ }
`:is-in-one-clique` ⟨*edge*⟩ `&optional` (⟨*model*⟩ `'current`)
⟨*model*⟩ ⟶ { ⟨*gc*⟩ | `'current` | `'base` | `'last` | ⟨*integer*⟩ }

Return Model to Lisp

`:return-model-set` ⟨*model*⟩ `&optional` (⟨*number*⟩ `nil`)
⟨*model*⟩ ⟶ { `'current` | `'base` | `'last` | `'number` ⟨*integer*⟩ }

:return-model $\langle \textit{model} \rangle$ &optional ($\langle \textit{number} \rangle$ nil)
    $\langle \textit{model} \rangle \longrightarrow \{$ 'current | 'base | 'last | 'number $\langle \textit{integer} \rangle$ $\}$

B.1.29 Common Decompositions of Models

:print-common-decompositions
:decompose-models $\langle \textit{set} \rangle$

B.1.30 Tests

:test
:find-log-l
:find-deviance

:factorize &optional ($\langle \textit{code} \rangle$ 'edges) ($\langle \textit{set} \rangle$ ";")
    $\langle \textit{code} \rangle \longrightarrow \{$ 'edges | 'interactions $\}$

:compute-test &optional ($\langle \textit{model-1} \rangle$ 'current) ($\langle \textit{model-2} \rangle$ 'base)
    $\langle \textit{model-1} \rangle$, $\langle \textit{model-2} \rangle \longrightarrow \{$ $\langle \textit{gc} \rangle$ | 'current | 'base | 'last
    | $\langle \textit{integer} \rangle$ $\}$
(print-test $\langle \textit{test} \rangle$)
:compute-deviance
(print-deviance $\langle \textit{test} \rangle$)

Compute Lots of Statistics for $\langle a \rangle$ and $\langle b \rangle$ Independent etc. Given the Factors $\langle \textit{set} \rangle$

:slice $\langle \textit{factor-name-a} \rangle$ $\langle \textit{factor-name-b} \rangle$ &optional ($\langle \textit{set} \rangle$ ";")

B.1.31 Test-List

:show-tests
:dispose-of-tests

B.1.32 Dispose of Tables

:dispose-of-tables
:dispose-of-q-table $\langle \textit{set} \rangle$
:dispose-of-probabilities
:dispose-of-all-q-tables

B.1.33 Editing Models with Tests

`:generate-decomposable &optional (`$\langle$*only*$\rangle$` nil)`
`:generate-graphical &optional (`$\langle$*only*$\rangle$` nil)`
`:drop-edges `$\langle$*gc*$\rangle$` &optional (`$\langle$*only*$\rangle$` nil)`
`:add-edges `$\langle$*gc*$\rangle$` &optional (`$\langle$*only*$\rangle$` nil)`
`:drop-interactions `$\langle$*gc*$\rangle$` &optional (`$\langle$*only*$\rangle$` nil)`
`:add-interactions `$\langle$*gc*$\rangle$` &optional (`$\langle$*only*$\rangle$` nil)`
`:meet-of-models &optional (`$\langle$*only*$\rangle$` nil)`
`:join-of-models &optional (`$\langle$*only*$\rangle$` nil)`

B.1.34 Stepwise Edge or Interaction Selection and Elimination

`:fix-edges `$\langle$*edges*$\rangle$
`:and-fix-edges `$\langle$*edges*$\rangle$

`:return-fix `$\langle$*code*$\rangle$
$\langle$*code*$\rangle$ $\longrightarrow$ { `'edges` | `'in` | `'out` }

`:backward &key (`$\langle$*only*$\rangle$` nil) (`$\langle$*reversed*$\rangle$` nil) (`$\langle$*sorted*$\rangle$` nil)`
`(`$\langle$*short*$\rangle$` nil) (`$\langle$*headlong*$\rangle$` nil) (`$\langle$*recursive*$\rangle$` nil) (`$\langle$*coherent*$\rangle$` nil)`
`(`$\langle$*follow*$\rangle$` nil) (`$\langle$*least-significant*$\rangle$` T) (`$\langle$*separators*$\rangle$` nil) (`$\langle$*edges*$\rangle$` nil)`
`:forward &key (`$\langle$*only*$\rangle$` nil) (`$\langle$*reversed*$\rangle$` nil) (`$\langle$*sorted*$\rangle$` nil)`
`(`$\langle$*short*$\rangle$` nil) (`$\langle$*headlong*$\rangle$` nil) (`$\langle$*recursive*$\rangle$` nil) (`$\langle$*coherent*$\rangle$` nil)`
`(`$\langle$*all-significant*$\rangle$` T) (`$\langle$*separators*$\rangle$` nil) (`$\langle$*edges*$\rangle$` nil)`

B.1.35 Global Search: The EH-procedure

`:set-main-effects `$\langle$*set*$\rangle$
`:read-base-model `$\langle$*gc*$\rangle$

B.1.36 EH: Fix Edges/Interactions

`:fix-in `$\langle$*gc*$\rangle$
`:fix-out `$\langle$*gc*$\rangle$
`:add-fix-in `$\langle$*gc*$\rangle$
`:add-fix-out `$\langle$*gc*$\rangle$
`:redo-fix-in`
`:redo-fix-out`

B.1.37 EH: Model Class and Search strategy

`:set-search `$\langle$*code*$\rangle$
$\langle$*code*$\rangle$ $\longrightarrow$ { `'graphical` | `'hierarchical` | `'smallest`
| `'alternating` | `'rough` }

B.1.38 EH: Dispose of Model-Classes and Duals

`:dispose-of-eh  &optional (`⟨*code*⟩ `'all)`
    ⟨*code*⟩ ⟶ { `'all` | `'duals` | `'a-dual` | `'r-dual` | `'classes`
     | `'accepted` | `'rejected` }

B.1.39 EH: Read and Fit Models or Force Models into Classes

`:fit 'models` ⟨*models*⟩
`:accept 'models` ⟨*models*⟩
`:reject 'models` ⟨*models*⟩

B.1.40 EH: Copy Models between the Model-List and the Search Classes

Models from List to Search Classes

`:fit` ⟨*model*⟩ `&optional` ⟨*a*⟩ ⟨*b*⟩
`:accept` ⟨*model*⟩ `&optional` ⟨*a*⟩ ⟨*b*⟩
`:reject` ⟨*model*⟩ `&optional` ⟨*a*⟩ ⟨*b*⟩
    ⟨*model*⟩ ⟶ { `'current` | `'base` | `'last` | `'all`
     | `'number` ⟨*integer*⟩ | `'interval` ⟨*integer*⟩ ⟨*integer*⟩
     | `'list` ⟨*list of integer*⟩ }

Models from Search Classes to Model List

`:extract` ⟨*class*⟩ `&optional (`⟨*sub-class*⟩ `nil)`
    ⟨*class*⟩ ⟶ { `'accepted` | `'rejected` | `'a-dual` | `'r-dual` }
    ⟨*sub-class*⟩ ⟶ { `'decomposable` | `'graphical` | `'hierarchical`
     | `nil` }

`:plot-EH-search-result`

B.1.41 EH: Find Duals

`:find-dual` ⟨*dual*⟩ `&optional (`⟨*sub-class*⟩ `nil)`
    ⟨*dual*⟩ ⟶ { `'a-dual` | `'r-dual` | `'both-duals` }
    ⟨*sub-class*⟩ ⟶ { `'decomposable` | `'graphical` | `'hierarchical`
     | `nil` }

B.1.42 EH: Directed Search

`:fit` ⟨*dual*⟩ `&optional (`⟨*sub-class*⟩ `nil)`
    ⟨*dual*⟩ ⟶ { `'a-dual` | `'r-dual` | `'smallest-dual`
     | `'largest-dual` | `'both-duals` }

⟨*sub-class*⟩ ⟶ { 'decomposable | 'graphical | 'hierarchical
| nil }

B.1.43 EH: Automatic Search

:eh &optional (⟨*strategy*⟩ nil) (⟨*sub-class*⟩ nil)
⟨*strategy*⟩ ⟶ { 'smallest | 'alternating | 'rough | nil }
⟨*sub-class*⟩ ⟶ { 'decomposable | 'graphical | 'hierarchical
| nil }

B.1.44 EH: Force a Dual into a Model Class

:reject ⟨*dual*⟩ &optional (⟨*sub-class*⟩ nil)
:accept ⟨*dual*⟩ &optional (⟨*sub-class*⟩ nil)
⟨*dual*⟩ ⟶ { 'a-dual | 'r-dual }
⟨*sub-class*⟩ ⟶ { 'decomposable | 'graphical | 'hierarchical
| nil }

B.2 coco-model-proto

:isnew ⟨*model*⟩ coco-object &key ⟨*title*⟩

:compute-test-against-model-object ⟨*base*⟩

:return-model &optional (⟨*model*⟩ nil) (⟨*number*⟩ nil)
⟨*model*⟩ ⟶ { nil | 'current | 'base | 'last
| 'number ⟨*integer*⟩ }

:return-model-set &optional (⟨*model*⟩ nil) (⟨*number*⟩ nil)
⟨*model*⟩ ⟶ { nil | 'current | 'base | 'last
| 'number ⟨*integer*⟩ }

:return-edge-list &optional (⟨*model*⟩ nil) &key (⟨*edges*⟩ 'in-model)
(⟨*fix*⟩ 'all-edges)
⟨*edges*⟩ ⟶ { 'in-model | 'not-in-model | 'all-edges }
⟨*fix*⟩ ⟶ { 'not-fix-edges | 'fix-edges | 'all-edges }
⟨*model*⟩ ⟶ { nil | 'current | 'base | 'last | ⟨*integer*⟩ }

:return-model-number &optional (⟨*model*⟩ nil)
⟨*model*⟩ ⟶ { nil | 'current | 'base | 'last }

:return-vector ⟨*type*⟩ ⟨*set*⟩ &key (⟨*permuted*⟩ T) (⟨*model*⟩ nil)
(⟨*random*⟩ nil) (⟨*complete*⟩ nil)
⟨*type*⟩ ⟶ { 'counts | 'probabilities | 'expected | 'unadjusted
| 'f-res | 'r-f | 'g-res | 'r-g | 'adjusted | 'm-res
| 'standardized | 'deviance | 'freeman-tukey | 'sqrt | 'power
| 'index | 'zero }

```
      ⟨model⟩ ⟶ { nil | 'current | 'base | 'last | ⟨integer⟩ }
:return-matrix ⟨type-list⟩ ⟨set⟩ &key (⟨permuted⟩ T) (⟨model-list⟩ nil)
   (⟨random-list⟩ nil) (⟨complete-list⟩ nil)
      ⟨type-list⟩ ⟶ { (list [ ⟨type⟩ ] )  }
      ⟨type⟩ ⟶ { See :return-vector }
      ⟨model-list⟩ ⟶ { (list [ ⟨model⟩ ] )  }
      ⟨model⟩ ⟶ { See :return-vector, etc. }
:print-table ⟨type⟩ ⟨set⟩ &key (⟨permuted⟩ T) (⟨model⟩ nil)
   (⟨random⟩ nil) (⟨log-trans⟩ nil) (⟨complete⟩ nil)
      ⟨model⟩ ⟶ { nil | 'current | 'base | 'last | ⟨integer⟩ }
:describe-table ⟨type⟩ ⟨set⟩ &key (⟨probit⟩ nil) (⟨rankit⟩ nil)
   (⟨uniform⟩ nil) (⟨model⟩ nil) (⟨random⟩ nil) (⟨log-trans⟩ nil)
   (⟨complete⟩ nil)
      ⟨model⟩ ⟶ { nil | 'current | 'base | 'last | ⟨integer⟩ }
:plot ⟨x⟩ ⟨y⟩ ⟨set⟩ &key (⟨X-model⟩ nil) (⟨X-random⟩ nil)
   (⟨X-log⟩ nil) (⟨Y-model⟩ nil) (⟨Y-random⟩ nil) (⟨Y-log⟩ nil)
   (⟨complete⟩ nil)
      ⟨X-model⟩, ⟨Y-model⟩ ⟶ { nil | 'current | 'base | 'last
       | ⟨integer⟩ }

:print-model &optional (⟨model⟩ nil) ⟨a⟩ ⟨b⟩
      ⟨argument⟩ ⟶ { nil | 'current | 'base | 'last | 'all
       | 'number ⟨integer⟩ | 'interval ⟨integer⟩ ⟨integer⟩
       | 'list ⟨list of integer⟩ }
:describe-model &optional (⟨model⟩ nil) ⟨a⟩ ⟨b⟩
      ⟨argument⟩ ⟶ { nil | 'current | 'base | 'last | 'all
       | 'number ⟨integer⟩ | 'interval ⟨integer⟩ ⟨integer⟩
       | 'list ⟨list of integer⟩ }
:dispose-of-model &optional (⟨model⟩ nil) ⟨a⟩ ⟨b⟩
      ⟨argument⟩ ⟶ { nil | 'current | 'base | 'last | 'all
       | 'number ⟨integer⟩ | 'interval ⟨integer⟩ ⟨integer⟩
       | 'list ⟨list of integer⟩ }

:is-graphical &optional (⟨model⟩ nil)
      ⟨model⟩ ⟶ { nil | ⟨gc⟩ | 'current | 'base | 'last | ⟨integer⟩ }
:is-decomposable &optional (⟨model⟩ nil)
      ⟨model⟩ ⟶ { nil | ⟨gc⟩ | 'current | 'base | 'last | ⟨integer⟩ }
:is-submodel-of &optional (⟨model-1⟩ nil) (⟨model-2⟩ nil)
      ⟨model-1⟩, ⟨model-2⟩ ⟶ { nil | ⟨gc⟩ | 'current | 'base | 'last
       | ⟨integer⟩ }
:is-in-one-clique ⟨edge⟩ &optional (⟨model⟩ nil)
      ⟨model⟩ ⟶ { nil | ⟨gc⟩ | 'current | 'base | 'last | ⟨integer⟩ }

:make-graph &key (⟨location⟩ nil) (⟨size⟩ nil) (⟨title⟩ nil)
```

B.3 drag-graph-proto

B.3.1 New Object and Save

:isnew &key $\langle \textit{location} \rangle$ $\langle \textit{title} \rangle$
:save

B.3.2 Pixels and Graph Options

:location &optional ($\langle \textit{left} \rangle$ nil) ($\langle \textit{top} \rangle$ nil)
:x &optional ($\langle \textit{val} \rangle$ nil set)
:y &optional ($\langle \textit{val} \rangle$ nil set)
:in-wastebasket

:to-x-pixel $\langle x \rangle$
:to-y-pixel $\langle y \rangle$
:from-x-pixel $\langle x \rangle$
:from-y-pixel $\langle y \rangle$

:resize

(rescale-procent $\langle p \rangle$ $\langle x \rangle$)
(2-3-list $\langle \textit{position} \rangle$)

(add-coco-graph-menu $\langle x \rangle$)

:rejected-edges &optional ($\langle \textit{val} \rangle$ nil set)
:accepted-edges &optional ($\langle \textit{val} \rangle$ nil set)

:static &optional ($\langle \textit{val} \rangle$ nil set)
:grid &optional ($\langle \textit{val} \rangle$ nil set)
:drag-graph &optional ($\langle \textit{val} \rangle$ nil set)
:edit-graph &optional ($\langle \textit{val} \rangle$ nil set)

B.3.3 Colors

:item-color $\langle \textit{item} \rangle$ &optional ($\langle \textit{value} \rangle$ nil set)
:draw-color $\langle \textit{color} \rangle$
:set-edge-color $\langle \textit{edge} \rangle$

B.3.4 Closest Vertex-Edge

(edge-distance $\langle x \rangle$ $\langle y \rangle$ $\langle a \rangle$ $\langle b \rangle$ $\langle c \rangle$ $\langle d \rangle$)
(point-potential $\langle x \rangle$ $\langle y \rangle$ $\langle \textit{point} \rangle$)

:return-closest-vertex-with-potential $\langle x \rangle$ $\langle y \rangle$
:return-closest-vertex-in-canvas $\langle x \rangle$ $\langle y \rangle$
:edge-potential $\langle x \rangle$ $\langle y \rangle$ $\langle \textit{edge} \rangle$ $\langle \textit{positions} \rangle$

:return-closest-edge ⟨*x*⟩ ⟨*y*⟩
:return-closest-edge-in-canvas ⟨*x*⟩ ⟨*y*⟩
:return-closest-edge-label-with-potential ⟨*x*⟩ ⟨*y*⟩
:visit-block-points ⟨*u*⟩ ⟨*v*⟩ ⟨*a*⟩ ⟨*b*⟩ ⟨*c*⟩ ⟨*x*⟩ ⟨*y*⟩ ⟨*z*⟩
:return-closest-block-point-with-potential ⟨*x*⟩ ⟨*y*⟩
:return-closest-block-point-in-canvas ⟨*x*⟩ ⟨*y*⟩

(close-point ⟨*x*⟩ ⟨*y*⟩ ⟨*u*⟩ ⟨*v*⟩)
:close-click ⟨*x*⟩ ⟨*y*⟩

:find-move ⟨*x*⟩ ⟨*y*⟩ ⟨*p*⟩

B.3.5 Block-Points

(in-block ⟨*position*⟩ ⟨*block*⟩)
(return-block-point ⟨*n*⟩ ⟨*block*⟩)
(to-block-points ⟨*n*⟩ ⟨*p*⟩)

B.3.6 Arrows

:update-arrows &optional (⟨*x*⟩ nil) &key (⟨*redraw*⟩ nil)

B.3.7 Rotate

:transformation &optional (⟨*val*⟩ nil set)
:do-hand-rotate ⟨*x*⟩ ⟨*y*⟩ ⟨*m1*⟩ ⟨*m2*⟩

(sphere-rand ⟨*n*⟩)

:project ⟨*position*⟩
:inverse-project ⟨*position*⟩
:canvas-to-sphere ⟨*x*⟩ ⟨*y*⟩ rad
:apply-transformation ⟨*trans*⟩ &optional (draw-box nil)
:add-idle
:angle &optional (⟨*val*⟩ nil set)
:do-idle
:idle-on (&optional 'on)
:tour-step
:tour-on &rest ⟨*args*⟩

B.3.8 Name-, Position-, Edge- and Block-List

:names &optional (⟨*val*⟩ nil)
:use-variables &optional (⟨*val*⟩ nil)
:positions &optional (⟨*val*⟩ nil)
:edges &optional (⟨*val*⟩ nil set)
:blocks &optional (⟨*val*⟩ nil set)

B.3.9 Positions, Labels and Blocks

`:vertex-position` ⟨*variable-name*⟩ `&optional (`⟨*position*⟩ `nil set) &key (`⟨*redraw*⟩ `nil)`

`:vertex-label` ⟨*variable-name*⟩ `&optional (`⟨*label*⟩ `nil set) &key (`⟨*redraw*⟩ `nil)`

`:vertex-type` ⟨*variable-name*⟩ `&optional (`⟨*type*⟩ `nil set) &key (`⟨*redraw*⟩ `nil)`

`:vertex-label-position` ⟨*variable-name*⟩ `&optional (`⟨*position*⟩ `nil set) &key (`⟨*redraw*⟩ `nil)`

`:fix-vertex-label-position` ⟨*variable-name*⟩ `&optional (`⟨*val*⟩ `nil set) &key (`⟨*redraw*⟩ `nil)`

`:vertex-label-arrow` ⟨*variable-name*⟩ `&optional (`⟨*val*⟩ `nil set) &key (`⟨*redraw*⟩ `nil)`

`:vertex-index` ⟨*variable-name*⟩

`:edge-label` ⟨*edge-key*⟩ `&optional (`⟨*val*⟩ `nil set) &key (`⟨*redraw*⟩ `nil)`

`:edge-label-position` ⟨*edge-key*⟩ `&optional (`⟨*position*⟩ `nil) &key (`⟨*redraw*⟩ `nil)`

`:edge-label-offset` ⟨*edge-key*⟩ `&optional (`⟨*position*⟩ `nil) &key (`⟨*redraw*⟩ `nil)`

`:fix-edge-label-position` ⟨*edge-key*⟩ `&optional (`⟨*val*⟩ `nil set) &key (`⟨*redraw*⟩ `nil)`

`:edge-label-arrow` ⟨*edge-key*⟩ `&optional (`⟨*val*⟩ `nil set) &key (`⟨*redraw*⟩ `nil)`

`:edge-width` ⟨*edge-key*⟩ `&optional (`⟨*val*⟩ `nil set) &key (`⟨*redraw*⟩ `nil)`

`:edge-test` ⟨*edge-key*⟩ `&optional (`⟨*val*⟩ `nil set) &key (`⟨*redraw*⟩ `nil)`

`:edge-status` ⟨*edge-key*⟩ `&optional (`⟨*val*⟩ `nil set) &key (`⟨*redraw*⟩ `nil)`

`:edge-index` ⟨*vertex-pair*⟩

`:stratum` ⟨*variable-name*⟩ `&optional (`⟨*block-key*⟩ `nil set) &key (`⟨*redraw*⟩ `nil)`

`:block-position` ⟨*block-key*⟩ `&optional (`⟨*position*⟩ `nil set) &key (`⟨*redraw*⟩ `nil)`

`:block-label` ⟨*block-key*⟩ `&optional (`⟨*label*⟩ `nil set) &key (`⟨*redraw*⟩ `nil)`

`:block-label-position` ⟨*block-key*⟩ `&optional (`⟨*position*⟩ `nil set) &key (`⟨*redraw*⟩ `nil)`

`:block-index` ⟨*block-key*⟩

`:adjust-vertices-to-grid &key (`⟨*vertex*⟩ `nil) (`⟨*delta*⟩ `1) (`⟨*redraw*⟩ `nil)`

`:adjust-blocks-to-grid &key (`⟨*delta*⟩ `1) (`⟨*redraw*⟩ `nil)`

`:rescale-vertex-positions &key (`⟨*vertex*⟩ `nil) (`⟨*scale*⟩ `1) (`⟨*redraw*⟩ `nil)`

`:rescale-block-positions &key (`⟨*scale*⟩` 1) (`⟨*redraw*⟩` nil)`

`:add-block `⟨*n*⟩` &key (`⟨*a*⟩` (list -45 -45 -50))`
`  (`⟨*b*⟩` (list -40 -40 50)) (`⟨*label*⟩` nil)`
`:define-blocks `⟨*block-list*⟩` &optional (`⟨*labels*⟩` nil)`
`:delete-block `⟨*n*⟩

B.3.10 Drawing Points, Point-Labels, Edges and Edge-Labels

`:draw-vertex `⟨*type*⟩ ⟨*x*⟩ ⟨*y*⟩ ⟨*z*⟩ ⟨*radius*⟩ ⟨*width*⟩
`:draw-vertex-and-label `⟨*position*⟩ ⟨*name*⟩ ⟨*use-variable*⟩ ⟨*radius*⟩
`:draw-vertices `⟨*radius*⟩
`:draw-edge-label `⟨*edge*⟩
`:draw-edge-label-nth `⟨*p*⟩
`:draw-thick-edge `⟨*x*⟩ ⟨*y*⟩ ⟨*z*⟩ ⟨*a*⟩ ⟨*b*⟩ ⟨*c*⟩ ⟨*width*⟩ ⟨*radius*⟩
`:draw-arrow `⟨*x*⟩ ⟨*y*⟩ ⟨*z*⟩ ⟨*a*⟩ ⟨*b*⟩ ⟨*c*⟩ ⟨*width*⟩ ⟨*fitted*⟩ ⟨*radius*⟩
`:draw-edge `⟨*edge*⟩` &optional (`⟨*radius*⟩` 4)`
`:draw-edges `⟨*radius*⟩
`:draw-rectangle-line `⟨*a*⟩ ⟨*b*⟩
`:draw-rectangle `⟨*a*⟩ ⟨*b*⟩ ⟨*c*⟩ ⟨*x*⟩ ⟨*y*⟩ ⟨*z*⟩
`:format-block-label `⟨*block*⟩
`:draw-block `⟨*block*⟩
`:draw-blocks`
`:draw-shadows`
`:draw-grid`
`:redraw-graph &optional (`⟨*radius*⟩` 4)`
`:redraw`

`:set-edge-label-nth `⟨*edge-index*⟩
`:drop-edge-label-nth `⟨*edge-index*⟩

B.3.11 Undo

`:push-vertex-undo `⟨*number*⟩
`:push-block-points-undo `⟨*n*⟩
`:undo-move`
`:skip-undo-move`
`:redo-move`
`:skip-redo-move`

B.3.12 Moving Points and Labels

`:drag-point` ⟨*x*⟩ ⟨*y*⟩ ⟨*m1*⟩ ⟨*m2*⟩ ⟨*point*⟩
`:drag-edge-label` ⟨*x*⟩ ⟨*y*⟩ ⟨*m1*⟩ ⟨*m2*⟩ ⟨*point*⟩
`:drag-block-point` ⟨*x*⟩ ⟨*y*⟩ ⟨*m1*⟩ ⟨*m2*⟩ ⟨*point*⟩
`:drag-points` ⟨*n*⟩ ⟨*new-pos*⟩
`:drag-line-from-point` ⟨*x*⟩ ⟨*y*⟩ ⟨*point*⟩

B.3.13 Mouse Interaction

`:do-click` ⟨*x*⟩ ⟨*y*⟩ ⟨*m1*⟩ ⟨*m2*⟩
`:do-motion` ⟨*x*⟩ ⟨*y*⟩
`:do-key` ⟨*c*⟩ ⟨*m1*⟩ ⟨*m2*⟩

B.3.14 Dump TeX Code

`(gcd` ⟨*a*⟩ ⟨*b*⟩`)`
`(int-round` ⟨*a*⟩`)`

`:dump-tex &key (`⟨*file-name*⟩ `nil) (`⟨*radius*⟩ `4) (`⟨*unadjusted*⟩ `nil)`
`(`⟨*box*⟩ `nil) (`⟨*centered*⟩ `nil) (`⟨*char-width*⟩ `5) (`⟨*char-height*⟩ `10)`
`(`⟨*extra-width*⟩ `2) (`⟨*extra-height*⟩ `1) (`⟨*size*⟩ `"(360,360)(0,0)")`

`:x-pos-to-tex` ⟨*x*⟩
`:y-pos-to-tex` ⟨*y*⟩

`:to-x-tex` ⟨*x*⟩
`:to-y-tex` ⟨*y*⟩
`:to-tex` ⟨*x*⟩
`:tex-draw-color` ⟨*f*⟩ ⟨*color*⟩
`:tex-line-type` ⟨*f*⟩ ⟨*type*⟩
`:tex-line-width` ⟨*f*⟩ ⟨*w*⟩
`:tex-draw-vertex` ⟨*f*⟩ ⟨*type*⟩ ⟨*x*⟩ ⟨*radius*⟩ ⟨*width*⟩
`:tex-draw-string` ⟨*f*⟩ ⟨*s*⟩ ⟨*x*⟩
`:tex-return-length` ⟨*x*⟩ ⟨*y*⟩ ⟨*a*⟩ ⟨*b*⟩ `&optional (d 0)`
`:tex-return-direction` ⟨*x*⟩ ⟨*y*⟩ ⟨*a*⟩ ⟨*b*⟩
`:tex-draw-vector` ⟨*f*⟩ ⟨*x*⟩ ⟨*y*⟩ ⟨*a*⟩ ⟨*b*⟩ ⟨*radius*⟩ ⟨*unadjusted*⟩
`:tex-draw-line` ⟨*f*⟩ ⟨*x*⟩ ⟨*y*⟩ ⟨*a*⟩ ⟨*b*⟩ ⟨*radius*⟩ ⟨*unadjusted*⟩
`:tex-draw-dashed-line` ⟨*f*⟩ ⟨*x*⟩ ⟨*y*⟩ ⟨*a*⟩ ⟨*b*⟩ ⟨*radius*⟩ ⟨*unadjusted*⟩
`:tex-draw-box` ⟨*f*⟩ ⟨*x*⟩ ⟨*y*⟩
`:tex-draw-vertex-in-box` ⟨*f*⟩ ⟨*type*⟩ ⟨*x*⟩ ⟨*extra-width*⟩ ⟨*extra-height*⟩
⟨*label-size*⟩ ⟨*s*⟩
`:tex-draw-vertex-and-label` ⟨*f*⟩ ⟨*position*⟩ ⟨*name*⟩ ⟨*use-variable*⟩ ⟨*radius*⟩
⟨*unadjusted*⟩ ⟨*box*⟩ ⟨*extra-width*⟩ ⟨*extra-height*⟩ ⟨*label-size*⟩
`:tex-draw-vertices` ⟨*f*⟩ ⟨*radius*⟩ ⟨*unadjusted*⟩ ⟨*box*⟩ ⟨*extra-width*⟩
⟨*extra-height*⟩ ⟨*label-sizes*⟩

`:tex-draw-edge-label` ⟨*f*⟩ ⟨*edge*⟩ ⟨*unadjusted*⟩
`:tex-set-edge-color` ⟨*f*⟩ ⟨*edge*⟩
`:tex-cut-edge` ⟨*point*⟩ ⟨*box*⟩ ⟨*centered*⟩
`:tex-draw-thick-edge` ⟨*f*⟩ ⟨*a*⟩ ⟨*b*⟩ ⟨*width*⟩ ⟨*radius*⟩ ⟨*unadjusted*⟩ ⟨*box*⟩ ⟨*centered*⟩ ⟨*dx*⟩ ⟨*dy*⟩
`:tex-draw-arrow` ⟨*f*⟩ ⟨*a*⟩ ⟨*b*⟩ ⟨*width*⟩ ⟨*radius*⟩ ⟨*unadjusted*⟩ ⟨*box*⟩ ⟨*centered*⟩ ⟨*dx*⟩ ⟨*dy*⟩
`:tex-draw-edge` ⟨*f*⟩ ⟨*edge*⟩ ⟨*radius*⟩ ⟨*unadjusted*⟩ ⟨*box*⟩ ⟨*centered*⟩ ⟨*extra-width*⟩ ⟨*extra-height*⟩ ⟨*label-sizes*⟩
`:tex-draw-edges` ⟨*f*⟩ ⟨*radius*⟩ ⟨*unadjusted*⟩ ⟨*box*⟩ ⟨*centered*⟩ ⟨*extra-width*⟩ ⟨*extra-height*⟩ ⟨*label-sizes*⟩
`:tex-draw-rectangle-line` ⟨*f*⟩ ⟨*a*⟩ ⟨*b*⟩ ⟨*unadjusted*⟩
`:tex-draw-rectangle-box` ⟨*f*⟩ ⟨*a*⟩ ⟨*b*⟩
`:tex-draw-rectangle` ⟨*f*⟩ ⟨*a*⟩ ⟨*b*⟩ ⟨*c*⟩ ⟨*x*⟩ ⟨*y*⟩ ⟨*z*⟩ ⟨*unadjusted*⟩
`:tex-draw-block` ⟨*f*⟩ ⟨*block*⟩ ⟨*unadjusted*⟩
`:tex-draw-blocks` ⟨*f*⟩ ⟨*unadjusted*⟩

B.4 association-diagram-proto

B.5 coco-graph-window-proto

B.5.1 Creating CoCo Graphs

`coco-model-proto :make-graph &key (`⟨*location*⟩ `nil) (`⟨*size*⟩ `nil) (`⟨*title*⟩ `nil)`
`coco-proto :make-graph &key (`⟨*model*⟩ `nil) (`⟨*location*⟩ `nil) (`⟨*size*⟩ `nil) (`⟨*title*⟩ `nil)`
`coco-graph-window-proto :return-child-coco-graph-window &key (`⟨*model*⟩ `nil) (`⟨*location*⟩ `nil) (`⟨*size*⟩ `nil) (`⟨*title*⟩ `nil)`
`coco-proto :plot-eh-search-result`
`coco-graph-window-proto :plot-eh-search-result`

`(return-coco-graph-window` ⟨*identification*⟩ ⟨*model-number*⟩ ⟨*names*⟩ ⟨*edges*⟩ ⟨*positions*⟩ ⟨*blocks*⟩ ⟨*use-variable*⟩ `&key (`⟨*location*⟩ `nil) (`⟨*size*⟩ `nil) (`⟨*title*⟩ `nil))`
`(return-default-positions` ⟨*n*⟩`)`
`(return-default-names` ⟨*n*⟩`)`
`(return-default-edge-list` ⟨*n*⟩ ⟨*p*⟩`)`

`(add-coco-graph-menu` ⟨*x*⟩`)`
`(coco-resume &key (`⟨*coco-id*⟩ `*current-coco*))`
`(coco-end &key (`⟨*coco-id*⟩ `*current-coco*))`

B.5.2 P-values

`:select-p-value` ⟨*test*⟩ `&key` (⟨*print-test*⟩ `nil`)
`:p-to-width` ⟨*p-value*⟩
`:format-p-value` ⟨*p-value*⟩

B.5.3 Options for Graph Edge Elimination

`:graph-p-rejected &optional` (⟨*val*⟩ `nil set`)
`:graph-p-accepted &optional` (⟨*val*⟩ `nil set`)
`:graph-coherent &optional` (⟨*val*⟩ `nil set`)
`:graph-headlong &optional` (⟨*val*⟩ `nil set`)
`:graph-random-order &optional` (⟨*val*⟩ `nil set`)
`:graph-decomposable-mode &optional` (⟨*val*⟩ `nil set`)
`:graph-follow &optional` (⟨*val*⟩ `nil set`)
`:graph-most-extreme &optional` (⟨*val*⟩ `nil set`)
:set-exact-test &optional (⟨*val*⟩ nil set)

B.5.4 Current and Base

`:make-graph-current-model`
`:make-graph-base-model`

B.5.5 Drop and Add Edge

`:edge-in-list` ⟨*p1*⟩ ⟨*p2*⟩ ⟨*edges*⟩
`:position-to-name` ⟨*p1*⟩ ⟨*p2*⟩
`:edge-list-to-string` ⟨*edges*⟩
`:point-to-string` ⟨*vertex-index*⟩

`(to-string` ⟨*a*⟩ ⟨*sep*⟩`)`
`(split-string` ⟨*str*⟩`)`
`(split-block-string` ⟨*block*⟩`)`
`(split-name-string` ⟨*names*⟩`)`
`(string-to-block-list` ⟨*str*⟩`)`

`(copy-list-list` ⟨*list*⟩`)`
`(list-to-comma-string` ⟨*list*⟩`)`

`:graph-drop-edge-nth &optional` ⟨*p*⟩ `&key` (⟨*edges*⟩ `nil`) (⟨*point*⟩ `nil`)
  (⟨*x-move*⟩ `0`)
`:graph-add-edge &optional` ⟨*p1*⟩ ⟨*p2*⟩ `&key` (⟨*edges*⟩ `nil set`)
  (⟨*x-move*⟩ `0`)
`:add-fill-in`

`:remove-tests`

B.5.6 Tests

`:position-to-max-block` ⟨*p1*⟩ ⟨*p2*⟩
`:return-history-and-future` ⟨*block-key*⟩ `&key (`⟨*redraw*⟩ `nil)`
`:return-history-model-nr` ⟨*block-key*⟩ `&key (`⟨*redraw*⟩ `nil)`
`(`⟨*current-model*⟩ `nil) (`⟨*fix-history*⟩ `nil)`

`:test-edge-nth` ⟨*p*⟩ `&key (`⟨*block-model*⟩ `nil) (`⟨*follow*⟩ `T)`
`(`⟨*decomposable-mode*⟩ `nil)`
`:test-add-edge` ⟨*vertex-pair*⟩ `&key (`⟨*block-model*⟩ `nil) (`⟨*follow*⟩ `T)`
`(`⟨*decomposable-mode*⟩ `nil)`

B.5.7 Edge Elimination and Selection

`:label-all-edges &key (`⟨*block-model*⟩ `nil) (`⟨*print-test-for-edge*⟩ `nil)`
`(`⟨*p-accepted*⟩ `0.10) (`⟨*p-rejected*⟩ `0.05) (`⟨*coherent*⟩ `nil)`
`(`⟨*headlong*⟩ `nil) (`⟨*random-order*⟩ `nil) (`⟨*follow*⟩ `T)`
`(`⟨*decomposable-mode*⟩ `nil) (`⟨*least-significant*⟩ `T) (`⟨*make-graph*⟩ `T)`
`:drop-least-significant-edge &key (`⟨*block-model*⟩ `nil)`
`(`⟨*p-accepted*⟩ `0.10) (`⟨*p-rejected*⟩ `0.05) (`⟨*recursive*⟩ `nil)`
`(`⟨*coherent*⟩ `nil) (`⟨*headlong*⟩ `nil) (`⟨*random-order*⟩ `nil) (`⟨*follow*⟩ `T)`
`(`⟨*decomposable-mode*⟩ `nil) (`⟨*least-significant*⟩ `T) (`⟨*x-move*⟩ `0)`
`:block-backward &key (`⟨*block*⟩ `nil) (`⟨*p-accepted*⟩ `0.10)`
`(`⟨*p-rejected*⟩ `0.05) (`⟨*recursive*⟩ `nil) (`⟨*coherent*⟩ `nil) (`⟨*headlong*⟩ `nil)`
`(`⟨*random-order*⟩ `nil) (`⟨*follow*⟩ `T) (`⟨*decomposable-mode*⟩ `nil)`
`(`⟨*least-significant*⟩ `T) (`⟨*x-move*⟩ `0)`

`:label-all-other-edges &key (`⟨*block-model*⟩ `nil)`
`(`⟨*print-test-for-edge*⟩ `nil) (`⟨*p-accepted*⟩ `0.10) (`⟨*p-rejected*⟩ `0.05)`
`(`⟨*coherent*⟩ `nil) (`⟨*headlong*⟩ `nil) (`⟨*random-order*⟩ `nil) (`⟨*follow*⟩ `T)`
`(`⟨*decomposable-mode*⟩ `nil) (`⟨*most-significant*⟩ `T) (`⟨*make-graph*⟩ `T)`
`:add-most-significant-edge &key (`⟨*block-model*⟩ `nil)`
`(`⟨*p-accepted*⟩ `0.10) (`⟨*p-rejected*⟩ `0.05) (`⟨*recursive*⟩ `nil)`
`(`⟨*coherent*⟩ `nil) (`⟨*headlong*⟩ `nil) (`⟨*random-order*⟩ `nil) (`⟨*follow*⟩ `T)`
`(`⟨*decomposable-mode*⟩ `nil) (`⟨*most-significant*⟩ `T) (`⟨*x-move*⟩ `0)`
`:block-forward &key (`⟨*block*⟩ `nil) (`⟨*p-accepted*⟩ `0.10)`
`(`⟨*p-rejected*⟩ `0.05) (`⟨*recursive*⟩ `nil) (`⟨*coherent*⟩ `nil) (`⟨*headlong*⟩ `nil)`
`(`⟨*random-order*⟩ `nil) (`⟨*follow*⟩ `T) (`⟨*decomposable-mode*⟩ `nil)`
`(`⟨*most-significant*⟩ `T) (`⟨*x-move*⟩ `0)`

B.5.8 Model Dynamic Spin Plot

`coco-graph-window-proto :return-dynamic-coco-spin-plot` ⟨*expressions*⟩
`&key (`⟨*location*⟩ `nil) (`⟨*title*⟩ `nil)`

B.5.9 Plot Global Search Result

`coco-proto :plot-eh-search-result`
`coco-graph-window-proto :plot-eh-search-result`

B.5.10 Overlay Controls

`:add-controls`

`:redraw-overlays`
`:overlay-click` ⟨*x*⟩ ⟨*y*⟩ ⟨*m1*⟩ ⟨*m2*⟩
`:add-overlay` ⟨*ov*⟩
`:delete-overlay` ⟨*ov*⟩

B.6 dynamic-coco-spin-proto

`coco-graph-window-proto :return-dynamic-coco-spin-plot` ⟨*expressions*⟩
`&key (`⟨*location*⟩ `nil) (`⟨*title*⟩ `nil)`

`dynamic-coco-spin-proto :isnew` ⟨*dim*⟩ `&key` ⟨*location*⟩ ⟨*title*⟩
`dynamic-coco-spin-proto :expressions &optional (`⟨*val*⟩ `nil set)`
`dynamic-coco-spin-proto :nth-expression` ⟨*n*⟩
`&optional (`⟨*val*⟩ `nil set)`
`dynamic-coco-spin-proto :values`
`dynamic-coco-spin-proto :nth-value` ⟨*n*⟩
`dynamic-coco-spin-proto :change-models`

B.7 manager-proto

`(return-manager &optional (`⟨*identification*⟩ `nil) &key (`⟨*location*⟩ `nil)`
`(`⟨*size*⟩ `nil) (`⟨*title*⟩ `nil))`

`(return-graphs)`
`association-diagram-proto :isnew &key` ⟨*location*⟩ ⟨*title*⟩

`(return-graphs)`
`coco-graph-window-proto :isnew &key` ⟨*location*⟩ ⟨*title*⟩

`manager-proto :isnew &key` ⟨*location*⟩ `(`⟨*title*⟩ `nil)`
`(return-manager-positions` ⟨*graphs*⟩`)`
`(return-manager-names` ⟨*graphs*⟩`)`

Different Model Managers

`coco-proto :return-manager &key` ⟨*location*⟩ ⟨*size*⟩ ⟨*title*⟩
`manager-proto :return-graphs`

`manager-proto :x-pos-to-tex` ⟨*x*⟩
`manager-proto :y-pos-to-tex` ⟨*y*⟩
`manager-proto :to-x-pixel` ⟨*x*⟩
`manager-proto :to-y-pixel` ⟨*y*⟩
`manager-proto :from-x-pixel` ⟨*x*⟩
`manager-proto :from-y-pixel` ⟨*y*⟩

Show, Hide and Close

`manager-proto :vertex-index` ⟨*variable-name*⟩
`manager-proto :vertex-type` ⟨*variable-name*⟩ `&optional (`⟨*type*⟩ `nil set)`
  `&key (`⟨*redraw*⟩ `nil)`
`manager-proto :hide-vertex` ⟨*vertex-index*⟩
`association-diagram-proto :hide-window`
`manager-proto :show-vertex` ⟨*vertex-index*⟩
`association-diagram-proto :show-window`
`manager-proto :close-vertex` ⟨*vertex-index*⟩
`association-diagram-proto :close`

`manager-proto :click-vertex` ⟨*vertex-index*⟩

`manager-proto :graph-drop-edge-nth &optional` ⟨*p*⟩ `&key (`⟨*edges*⟩ `nil)`
  `(`⟨*point*⟩ `nil) (`⟨*x-move*⟩ `0)`

Location and Title

`manager-proto :drag-point` ⟨*x*⟩ ⟨*y*⟩ ⟨*m1*⟩ ⟨*m2*⟩ ⟨*point*⟩

`manager-proto :vertex-position` ⟨*variable-name*⟩
  `&optional (`⟨*position*⟩ `nil set) &key (`⟨*redraw*⟩ `nil)`
`association-diagram-proto :location`
`manager-proto :vertex-label` ⟨*variable-name*⟩
  `&optional (`⟨*label*⟩ `nil set) &key (`⟨*redraw*⟩ `nil)`
`association-diagram-proto :title`

Coherence

`association-diagram-proto :add-graph-to-managers`
`association-diagram-proto :update-managers`
`association-diagram-proto :update-screen`

current and base

manager-proto :make-graph-current-model &key (⟨*redraw-plots*⟩ nil) (⟨*key-event*⟩ nil)
manager-proto :make-graph-base-model &key (⟨*redraw-plots*⟩ nil) (⟨*key-event*⟩ nil)

Sub- and Supermodels

association-diagram-proto :add-manager-edge ⟨*from*⟩ ⟨*to*⟩

Drawing Edges

manager-proto :draw-edge ⟨*edge*⟩ &optional (⟨*radius*⟩ 4)

Tests

manager-proto :test-two-objects ⟨*a*⟩ ⟨*b*⟩ ⟨*p*⟩ &key (⟨*decomposable-mode*⟩ nil)
manager-proto :test-edge-nth ⟨*p*⟩ &key (⟨*decomposable-mode*⟩ nil)
manager-proto :graph-add-edge ⟨*p1*⟩ ⟨*p2*⟩

Selecting Test Statistic

manager-proto :select-p-value ⟨*test*⟩ &key (⟨*print-test*⟩ nil)
manager-proto :format-p-value ⟨*p*⟩
manager-proto :p-to-width ⟨*p-value*⟩

Inherited Methods

manager-proto :do-key ⟨*c*⟩ ⟨*m1*⟩ ⟨*m2*⟩
manager-proto :in-wastebasket

Bibliography

Agresti, A. (1990). *Categorical data analysis*, Wiley, Chichester.

Anglin, D. G. & Oldford, R. W. (1993). Modelling response models in software, *Preliminary Papers of the Fourth International Workshop on Artificial Intelligence and Statistics*, pp. 483–494.

Asmussen, S. & Edwards, D. (1983). Collapsibility and response variables in contingency tables, *Biometrika* **70**: 567–578.

Badsberg, J. H. (1986). *Kontingenstabeller – implementation og kompleksitet af algoritmer for estimation og test i store kontingenstabeller*, Masters thesis, Aalborg University.

Becker, R. A., Chambers, J. M. & Wilks, A. R. (1988). *The New S Language, A Programming Environment for Data Analysis and Graphics*, Wadsworth & Brooks/Cole Advanced Books & Software, Pacific Grove, California.

Becker, R. A. & Chambers, J. (1988). Auditing of data analyses, *SIAM J. SCI. Stat. Comput* **9**(4): 747–760. AT&T Bell Laboratories.

Chambers, J. M. & Hastie, T. J. (1991). *Statistical models in S*, Wadsworth & Brooks/Cole Advanced Books & Software, Pacific Grove, California.

Christensen, R. (1990). *Log-Linear Models*, Springer-Verlag.

Darroch, J. N., Lauritzen, S. L. & Speed, T. P. (1980). Markov fields and log-linear interaction models for contingency tables, *Annals of Statistics* **8**: 522–539.

Dawid, A. P. & Skene, A. M. (1979). Maximum likelihood estimation of observer error-rates using the em algorithm, *Journal of the Royal Statistical Society, Series C* **28**(1): 20–28.

Dempster, A. P. (1972). Covariance selection, *Biometrika* **28**: 157–175.

Dixon, W. J. (1983). *BMDP Statistical Software*, University of California Press, 1985 printing.

Edwards, D. (1989). A guide to MIM version 1.7, *Research Report 12*, Statistical Research Unit, University of Copenhagen. A Guide to MIM version 2.0, 1991.

Edwards, D. (1990). Hierarchical interaction models (with discussion), *Journal of the Royal Statistical Society, Series B* **52**: 3–20 and 51–72.

Edwards, D. (1993). Some computational aspects of graphical model selection, *in* J. Antoch (ed.), *Computational Aspects of Model Choice*, Springer-Verlag, pp. 187–210.

Edwards, D. & Havránek, T. (1985). A fast procedure for model search in multidimensional contingency tables, *Biometrika* **72**: 339–351.

Edwards, D. & Havránek, T. (1987). A fast model selection procedure for large families of models, *Journal of the American Statistical Association* **82**: 205–211.

Frydenberg, M. (1989). The chain graph Markov property, *Scandinavian Journal of Statistics* **17**: 333–353.

Frydenberg, M. (1990). Marginalization and collapsibility in graphical interaction models, *Annals of Statistics* **18**: 790–805.

Frydenberg, M. & Lauritzen, S. L. (1989). Decomposition of maximum likelihood in mixed interaction models, *Biometrika* **76**: 539–555.

Fuchs, C. (1982). Maximum likelihood estimation and model selection in contingency tables with missing data, *Journal of the American Statistical Association* **77**(378): 270–278.

Gilchrist, R. & Scallan, A. (1988). Funigirls: A prototype functional programming language for the analysis of generalized linear models, *in* D. Edwards & N. E. Raun (eds), *COMPSTAT*, Vol. 1, Physica Verlag: Heidelberg, pp. 207–212. COMPSTAT 1988, Copenhagen.

Goodman, L. A. (1971). Partitioning of chi-square, analysis of marginal contingency tables, and estimation of expected frequencies in multidimensional contingency tables, *Journal of the American Statistical Association* **66**: 339–344.

Haberman, S. J. (1974). *Log-linear Models For Frequency Data*, Univ. of Chicago Press, Chicago, Illinois.

Hitz, M. & Hudec, M. (1994). Applying the object oriented paradigm to statistical computing, *in* R. Dutter & W. Grossmann (eds), *Computational Statistics*, Physica Verlag: Heidelberg, pp. 389–394. COMPSTAT 1994, Vienna.

Holm, S. (1979). A simple sequentially rejective multiple test procedure, *Scandinavian Journal of Statistics* **6**: 65–70.

Hommel, G. & Bernhard, G. (1993). Multiple hypotheses testing, *in* J. Antoch (ed.), *Computational Aspects of Model Choice*, Springer-Verlag, pp. 187–210.

Huber, P. J. (1994). “Huge” data sets, *in* R. Dutter & W. Grossmann (eds), *Computational Statistics*, Physica Verlag: Heidelberg, pp. 3–13. COMPSTAT 1994, Vienna.

Jiroušek, R. (1991). Solution of the marginal problem and decomposable distributions, *Kybernetike* **27**(5): 403–412.

Kjærulff, U. (1992). Optimal decomposition of probabilistic networks by simulated anneling, *Statistics and Computing* **2**: 7–17.

Kreiner, S. (1989). User’s guide to DIGRAM— a program for discrete graphical modelling, *Technical Report 89–10*, Statistical Research Unit, University of Copenhagen.

Krippendorf, K. (1986). Quantitative applications in social sciences, *Information Theory: Structural Models for Qualitative Data*.

Lauritzen, S. L. (1982). *Lectures on Contingency Tables*, (3rd edition 1989), Aalborg: University of Aalborg Press, Denmark.

Lauritzen, S. L. (1989). Mixed graphical association models (with discussion), *Scandinavian Journal of Statistics* **16**: 273–306.

Lauritzen, S. L. (1993). Graphical association models, DRAFT, *Technical Report IR 93–2001*, Department of Mathematics and Computer Science, Aalborg University, Denmark.

Lauritzen, S. L., Dawid, A. P., Larsen, B. N. & Leimer, H.-G. (1990). Independence properties of directed Markov fields, *Networks* **20**: 491–505.

Lauritzen, S. L. & Wermuth, N. (1989). Graphical models for associations between variables, some of which are qualitative and some quantitative, *Annals of Statistics* **17**: 31–57.

Lauritzen, S. L., Thiesson, B. & Spiegelhalter, D. J. (1992). Diagnostic systems created by model selection methods - a case study, *Technical report*, The university of Aalborg, Institute for Electronic Systems, Department of Mathematics and Computer Science.

Leimer, H.-G. (1993). Optimal decomposition by clique separators, *Discrete Mathematics* **113**: 90–123.

McDonald, J. A. (1986). Antelope: data analysis with object oriented programming and constraints, *Proceeding of 1985 Joint Statistical Meetings, Statistical Computing Section*, pp. 1–10.

McDonald, J. A. & Pedersen, J. (1988). Computing environments for data analysis III: Programming environments, *SIAM J. Sci. Stat. Comput.* **9**(2): 380–400.

Oldford, R. W. (1988). Object-oriented software representations for statistical data, *J. of Econometrics* **38**(3): 227–246.

Oldford, R. W. & Peters, S. C. (1986). Object-oriented data representations for statistical data, *COMPSTAT*, Physica Verlag: Heidelberg. COMPSTAT 1986.

Oldford, R. W. & Peters, S. C. (1988a). DINDE: Towards more sophisticated software environments for statistics, *SIAM Journal on Computing* **9**(1): 191–211.

Oldford, R. W. & Peters, S. C. (1988b). Statistical analysis maps, *Technical Research Report STAT-88-21*, University of Waterloo, Canada, Dept. of Stat. & Act. science.

Read, T. R. C. & Cressie, N. A. C. (1988). *Goodness-of fit Statistics for Discrete Multivariate Data*, Springer-Verlag, New York.

Reiniš, Z., Pokorný, J., Bašiká, V., Tišerová, J., Goričan, K., Horáková, D., Stuchliková, E., Havránek, T. & Hrabovský, F. (1981). Prognostický význam rizikového profilu v prevenci ischemické choroby srdce, *Bratis. lek. Listy.* **76**: 137–150. (Prognostic significance of the risk profile in the prevention of coronary heart disease).

Rose, D. J., Tarjan, R. E. & Lueker, G. S. (1976). Algorithmic aspects of vertex elimination on graphs, *SIAM Journal on Computing* **5**: 266–283.

Sakamoto, Y. & Akaike, H. (1978). Analysis of cross classified data by aic, *Ann. Inst. Statist. Math.* **30**: 185–197.

Schwarz, G. (1978). Estimating the dimension of a model, *Annals of Statistics* **6, 2**: 461–464.

Stuetzle, D. G. (1987). Plot windows, *Journal of the American Statistical Association* **82**: 466–475.

Sundberg, R. (1975). Some results about decomposable (or markov-type) models for multidimensional contingency tables: Distribution of marginals and partitioning of tests, *Scandinavian Journal of Statistics* **2**: 771–779.

Tarjan, R. E. (1985). Decomposition by clique separators, *Discrete Mathematics* **55**: 221–232.

Tarjan, R. E. & Yannakakis, M. (1984). Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs, *SIAM Journal on Computing* **13**: 566–579.

Tierney, L. (1990). *LISP-STAT — An Object-Oriented Environment for Statistical Computing and Dynamic Graphics*, Wiley, New York.

Tierney, L. (1991). Generalized linear models in lisp stat, *Technical Report 557*, School of Statistics, University of Minnesota.

Wedelin, D. (1993). Efficient algorithms for probabilistic inference, combinatorial optimization and the discovery of causal structure from data, *Ph.d. thesis*, Department of Computer Sciences, Chalmers University of Technology, University of Göteborg, Göteborg, Sweden.

Wermuth, N. (1976). Model search among multiplicative models, *Biometrics* **32**: 253–264.

Wermuth, N. & Lauritzen, S. L. (1983). Graphical and recursive models for contingency tables, *Biometrika* **70**: 537–552.

Wermuth, N. & Lauritzen, S. L. (1990). On substantive research hypotheses, conditional independence graphs and graphical chain models (with discussion), *Journal of the Royal Statistical Society, Series B* **52**: 21–72.

Whittaker, J. (1990). *Graphical Models in Applied Multivariate Statistics*, Wiley, Chichester.

Wilkinson, G. N. & Rogers, C. E. (1979). Symbolic description of factorial models for analysis of variance, *Journal of the Royal Statistical Society, Series C* **22**(3): 392–399.

Index