DSC 2003 Working Papers
(Draft Versions)
http://www.ci.tuwien.ac.at/Conferences/DSC-2003/

Draft: More complex graph computations for undirected graphical models by the CoCo bundle for R

Jens Henrik Badsberg
Danish Institute of Agricultural Sciences
JensHenrik.Badsberg@agrsci.dk

Introduction

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 contingency tables, but the program has been enhanced to handle also incomplete tables and latest also graphical models with continuous variables.

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 (1972). See Darroch et al. (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.

CoCo is a program designed to perform estimation and tests in large contingency

tables. By using graph-theoretical results the hierarchical mixed interaction models are decomposed. The iterative algorithm is not used on the full table, but only on the non-decomposable irreducible components. Furthermore, the optimized version of the IPS-algorithm of Jiroušek (1991) is used on these non-decomposable discrete atoms.

Besides incomplete tables also tables with incomplete observations can be handled in CoCo, e.g. by the EM algorithm. Exact tests between any two nested decomposable discrete models can be computed.

The CoCo bundle for R is available from `http://www.jbs.agrsci.dk/Biometry/Software-Datasets/CoCo/CoCo.1.5/`. The package for handling models with both discrete and continuous variables are currently called CoCoCg. This module will be included in the next version of the CoCo bundle made available later this year. Some of the function names used in the example section this version of the paper might be changed in later versions of the CoCo bundle.

1 Notation

Graphs

We shall consider *simple undirected graphs* $G = (V(G), E(G))$ with *vertices* $V(G) \subseteq \Delta \cup \Gamma$ and *edges* $E(G)$. A graph is simple if it does not contain multiple edges and loops, that is, no identical edges and no edges with identical vertices. We say that two vertices in a graph are *adjacent* or *neighbours*, if there is an edge between them. If v is adjacent to w, that is, $\{v, w\} \in E(G)$, we will also write $v \sim w$.

We shall consider graphs with two types of vertices. $\Delta \subseteq V(G)$ is the set of vertices, we shall call *marked* for *discrete* variables, and $\Gamma = V(G) \setminus \Delta$ contains the *unmarked* vertices for *continuous* variables.

A *path* in G between vertices $v, w \in V(G)$ is a sequence $v_{(0)}, v_{(1)}, \cdots, v_{(n)} \in V(G)$ with $(v, w) = (v_{(0)}, v_{(n)})$ and $\{v_{(i-1)}, v_{(i)}\} \in E(G)$ for $i = 1, 2, \cdots, n$. A *cycle* or *n-cycle* in G is a sequence $v_{(1)}, v_{(2)}, \cdots, v_{(n)} \in V(G)$ with $\{v_{(n)}, v_{(1)}\} \in E(G)$ and $\{v_{(i)}, v_{(i+1)}\} \in E(G)$ for $i = 1, 2, \cdots, n-1$. A *chord* of a cycle (or path) $v_{(1)}, v_{(2)}, \cdots, v_{(n)} \in V(G)$ is two vertices $v_{(i)}$ and $v_{(j)}$ with $\{v_{(i)}, v_{(j)}\} \in E(G)$, $i < j$ and $i + 1 \neq j$. Two vertex sets $a \subseteq V(G)$ and $b \subseteq V(G)$ are *separated* by $c \subseteq V(G)$, if every path from a vertex in a to a vertex in b contains a vertex from c.

We define the *boundary* of a subset a of $\Delta \cup \Gamma$, written ∂a, as those vertices that are not in a but are adjacent to some vertex in a. A set a is called *complete*, if all possible edges between the vertices of a are present in the graph. If a is complete and a is not a subset of another complete subset of the graph, then a is called a *clique*. A vertex v is *perfect*, if the set ∂v of neighbours of v is complete. The subgraph *induced* by a subset $A \subseteq V(G)$ of the vertices of a graph $G = (V(G), E(G))$ is the graph $G_A = (A, E_A)$, where $E_A \subseteq E(G)$ are the edges $\{v, w\} \in E(G)$ with both $v \in A$ and $w \in A$. The *connected components* of a graph are a partitioning of the graph into subgraphs such that two vertices are in the same connected component, if and only if there is a path between the vertices. If a graph only contains one connected component, then the graph is *connected*. If a graph is connected and

contains no cycles then the graph is a *tree*.

Connected components, shortests paths (paths without chords) and cutsets (subsets of vertices separating two given subsets of the graph) can be found by respectively Breadth-first or Depth-first search, Dijkstra algorithm and Ford-Fulkersons algorithm for the maximum flow problem.

Definition 1 Two subsets a and b of G form a decomposition of a graph $G = (V(G),\ E(G))$, if $a \cup b = V(G)$, $a \setminus b \neq \emptyset$, $b \setminus a \neq \emptyset$, $a \setminus b$ and $b \setminus a$ are separated by $a \cap b$ in the graph, and $a \cap b$ is a complete subset.

In words, for the 2-section graph of a graphical model, two subgraphs of the 2-section graph form a *decomposition* of the graph with respect to a subset c of the vertices of the graph, if the graph is the union of two subgraphs and the intersection c between the two subgraphs is complete. The set c is a *clique separator*. The graph is *decomposed* into the two subgraphs. The subgraphs may be further decomposed into subgraphs.

A clique separator C is called *admissible* for $G = (V(G), E(G))$ if there are at least two different connected components G_A and G_B of G with $\partial A = \partial B = C$, that is, each vertex of C is adjacent to at least one vertex of both A and B. $C \subseteq V(G)$ is a *minimal separator* of $v, w \in V(G)$, if C, but no proper subset of C, separates $\{v\}$ and $\{w\}$ in G. C is a *relative minimal separator* for G, if there are vertices $v, w \in V(G)$ such that C is a minimal separator for v and w. It is obvious that a separator is a relative minimal separator if and only if the separator is an admissible separator.

The decomposition formed by a and b is *strong*, if at least one of the three conditions $a \cap b \subseteq \Delta$, $a \setminus b \subseteq \Gamma$, $b \setminus a \subseteq \Gamma$ holds.

Definition 2 If a graph and its subgraphs can be decomposed recursively until all the subgraphs are complete, then the graph is decomposable.

Note that a graph may be decomposed without being decomposable. We say that the graph is *reducible*, if it can be decomposed, that is, its vertex set contains a clique separator, otherwise the graph is said to be *irreducible*, a *prime* or a *non-separable atom*. A subgraph G_A of a graph G is an *irreducible component* or a *maximal prime subgraph* of G, if G_A is irreducible and G_B is reducible for all B with $A \subset B \subseteq V(G)$.

A graph is *triangulated*, if it contains no cycles of length greater than 3 without a chord. It is a well-known fact Lauritzen et al. (1984) that the decomposable graphs are the triangulated graphs, the *chordal graphs* or *rigid circuit graphs*.

Vertex Elimination Orderings and the Fill-In Graph

An *elimination ordering* π of the vertices V of a graph $G = (V, E)$ with $|V| = n$ is a bijection $\pi : V \leftrightarrow \{1, 2, \cdots, n\}$. Often we will write the ordering π as $\{v_1, v_2, \cdots, v_n\}$ with the index i of the vertex v_i the order of the vertex, $\pi(v_i) = i$. If $\pi(v) < \pi(w)$, that is, the order of v is less than the order of w, then we say that v is less than w, and write $v < w$.

The *fill-in* F_π caused by the ordering π is the set of edges defined as follows:

$$F_\pi = \{\{v,w\}|\ v \neq w,\ \text{there is no edge between } v \text{ and } w,\ \text{and there is a path } v = v_{(1)}, v_{(2)}, \cdots, v_{(k)} = w \text{ such that } \pi(v_{(i)}) < \min\{\pi(v), \pi(w)\} \text{ for } i = 2, 3, \cdots, k-1\}. \quad (1)$$

An elimination ordering π is *perfect*, if the fill-in caused by the ordering is empty, *minimum*, if $|F_\pi|$ is minimum over all possible orderings, and *minimal*, if there is no ordering σ such that $F_\sigma \subset F_\pi$. It is a well known fact, see for example Rose et al. (1976), that a graph has a perfect vertex ordering if and only if the graph is decomposable, and that an ordering $\{v_1, v_2, \cdots, v_n\}$ of the vertices $v_{(i)}$, $i = 1, 2, \cdots, n$, in a graph is perfect if $\partial v_i \cap \{v_i, v_{i+1}, \cdots, v_n\}$ is complete for each vertex v_i, $i = 1, 2, \cdots, n$, in the graph, that is, if and only if v_i is perfect in the subgraph induced by $\{v_i, v_{i+1}, \cdots, v_n\}$. The *fill-in graph* of a graph $G = (V(G), E(G))$ for an ordering π is the graph $F = (V(G), E(G) \cup F_\pi)$ with the fill-in F_π added. Since the fill-in graph for the ordering π of the fill-in graph $F = (V(G), E(G) \cup F_\pi)$ is empty, the fill-in graph is decomposable. If v is adjacent to w in the fill-in graph F for an ordering π, that is, $\{v, w\} \in E(G) \cup F_\pi$, we write $v \sim_\pi w$. The boundary $\partial_\pi v$ are the vertices adjacent to v in the fill-in graph.

The set $\partial v_i \cap \{v_{i+1}, v_{i+2}, \cdots, v_n\}$ of vertices following v_i and adjacent to v_i is called the *monotone adjacency set*. By $C(v_i)$ we will denote the *monotone adjacency set in the fill-in graph*:

$$C(v_i) = \partial_\pi v_i \cap \{v_{i+1}, v_{i+2}, \cdots, v_n\}. \quad (2)$$

Maximum Cardinality Search

The algorithm Maximum Cardinality Search of Tarjan and Yannakakis (1984) finds an ordering of the vertices in a graph in $O(e + n)$ time.

In Maximum Cardinality Search the vertices are numbered from n to 1 in decreasing order as follows. Give an arbitrary vertex the ordering n. As the next vertex to give a number, select the vertex adjacent to the highest number of previously numbered vertices, breaking ties arbitrarily.

The ordering Max-Card found is a perfect vertex elimination ordering, if the graph is decomposable. With the ordering found, decomposability can be checked in time $O(e+n)$ by the algorithm Test for Zero Fill-In of the same paper. The ordering Max-Card can be used to find a closed form expression for maximum likelihood estimates in decomposable log-linear models, Badsberg (1996b).

Minimal Vertex Elimination Orderings

In Tarjan (1985) an algorithm for finding the clique separators of a graph in a total time of $O(ne + n^2)$ is presented. Leimer (1993) presents a version of this algorithm that is optimal in the sense that the separators are minimal and that the graph is only decomposed into the irreducible components. These algorithms and the algorithm of Badsberg (1996b) can by used to reduce the computations needed to find the maximum likelihood estimates of non-decomposable log-linear models.

In the above decomposition algorithms the vertices of the graph has to be visited according to a minimal vertex elimination ordering. The ordering produced by the algorithm Maximum Cardinality Search of the previous section will not do, since this ordering is not necessary minimal for a non-decomposable graph.

The minimal ordering of the vertices can be found by the algorithm Lex M of Rose et al. (1976) in $O(ne)$. In Lex M the vertices are numbered from n to 1 in decreasing order using a modified lexicographical search, where *lexicographical search* is defined as follows. For each unnumbered vertex v, maintain a label, a list of the numbers of the numbered vertices adjacent to v, with the numbers in each list arranged in decreasing order. For the next vertex to number, select the vertex whose label is lexicographically the greatest, breaking ties arbitrarily. Although somewhat complicated, lexicographical search can be implemented to run in $O(e+n)$ time Rose et al. (1976), and the modified version of lexicographical search used in Lex M to find the minimal ordering runs in $O(ne)$ time.

To find the minimal ordering by Lex M a lexicographic ordering scheme which is a special type of Breadth-first search is used. The vertices of the graph are numbered from n to 1. During the search, each vertex v has an associated *label* consisting of a set of numbers selected from $\{1, 2, \cdots, n\}$, ordered in *decreasing* order. Given two labels $L_1 = [p_1, p_2, \cdots, p_k]$ and $L_2 = [q_1, q_2, \cdots, q_l]$, we define $L_1 < L_2$ if, for some j, $p_i = q_i$ for $i = 1, 2, \cdots, j-1$ and $p_j < q_j$, or if $p_i = q_i$ for $i = 1, 2, \cdots, k$ and $k < l$. $L_1 = L_2$ if $k = l$ and $p_i = q_i$ for $1 \leq i \leq l$.

First the empty label is assigned to all the vertices. Then the vertices are ordered in decreasing order as follows. The vertex with the largest label is picked, breaking ties arbitrary. (A random vertex is numbered first with the order n.) After assigning the order i to the vertex v the label of each unnumbered vertex w such that there is a chain $[v = v_1, v_2, \cdots, v_{k+1} = w]$ with v_j unnumbered and $label(v_j) < label(w)$ for $j = 2, 3, \cdots, k$ the number i is added to the label $label(w)$ of w. After updating the labels of unnumbered vertices, the unnumbered vertex with the largest label is selected as the next vertex to order, labels are updated, etc. until the last vertex is assigned the order 1.

RIP orderings

If a system $\zeta = \{R_1, \cdots, R_J\}$ of J, $J \geq 1$, subsets $R_j \subseteq \Delta$ are ordered in a sequence $(R_1, \cdots, R_J)$ such that

$$\forall\ j = 2, \cdots, J \quad \exists\ k, 1 \leq k < j : \quad (R_j \cap \bigcup_{l=1}^{j-1} R_l) \subseteq R_k, \tag{3}$$

then the sequence $(R_1, \cdots, R_J)$ fulfills the *running intersection property*.
In particular, if R_j, $j = 1, \cdots, J$, are the cliques of a graph, then the sequence $(R_1, \cdots, R_J)$ fulfills the running intersection property, if the cliques in the sequence are ordered such that for each clique the intersection between the clique and the union of previous cliques is a subset of a previous clique. A graph is decomposable, if and only if such an ordering of the cliques exists.

Junction trees

Let $\mathcal{M}$ be a finite collection of subsets of a set Δ, e.g., a generating class. The *junction graph* $J(\mathcal{M}) = (\mathcal{M}, E(J))$ for $\mathcal{M}$ is a graph with vertices $V(J) = \mathcal{M}$ the elements of $\mathcal{M}$ and edges $E(J) = \{\{c_i, c_j\} \subseteq \mathcal{M} |\ \mathrm{c_i} \cap \mathrm{c_j} \neq \emptyset\}$. In other words, there is an edge between two nodes of the junction graph, if the intersection of the two vertex sets of the two nodes is not empty.

Any spanning tree for $J(\mathcal{M})$ will be called a *junction tree for* $J(\mathcal{M})$, if for any pair $c_i, c_j \in \mathcal{M}$ all vertices on the path between c_i and c_j contain $c_i \cap c_j$, see also Jensen (1988). If $J(\mathcal{M})$ is not connected, then the graph $J(\mathcal{M})$ does not have a junction tree, but each connected component of $J(\mathcal{M})$ may have. A connected component of a graph, that is the generating class consisting of the cliques of the connected component, has a junction tree, if and only if the connected component is decomposable, Jensen (1988).

A junction tree can be constructed from a sequence $(R_1, \cdots, R_J)$ fulfilling the running intersection property: the tree is given vertices $(R_1, \cdots, R_J)$, and for $j = 2, \cdots, J$ the vertex R_j is connected to one of the vertices R_k fulfilling (3). Also a sequence fulfilling the running intersection property can be read off a junction tree: Pick any vertex as root, and then traverse the tree either Breadth-first or Depth-first to visit all vertices of the junction tree and let R_i be the elements of the i-th visited node of the tree.

The Index

In decomposable log-linear models for contingency tables, the problem of finding a closed form expression of the maximum likelihood estimates is a matter of computing the *index* or the *adjusted replication number* for subsets of the graph associated with the model Darroch et al. (1980).

To define the index we have to define the pieces of the graph relative to some subset of the vertices. Let $G = (V(G), E(G))$ be a connected graph and $d \subseteq E(G)$ be a complete subset. The pieces of G relative to d are defined as follows. Remove d from G and form the subgraph $G_{V(G)\setminus d}$ with vertices $V(G) \setminus d$ and edges which are those in $E(G)$ that do not involve vertices in d. $G_{V(G)\setminus d}$ now has one or more connected components A_t, $t \in T$, say. Let G_t be the subgraph of G obtained be rejoining d to the subgraph A_t, that is, G_t has the vertex set $A_t \cup d$ and edges which are those in $E(G)$ that only involve vertices in $A_t \cup d$. G_t, for all $t \in T$, are *the pieces of* G *relative to* d.

Then for each complete subset d of $E(G)$ the *index* $\nu(d)$ is defined as follows:

$$\nu(d) = 1 - \text{the number of pieces of } G \text{ relative to } d \text{ in which } d \text{ is not a clique.} \quad (4)$$

If a graph G with index ν_G is decomposed into the two subgraphs A and B, and these are both connected graphs with indices ν_A and ν_B and with vertex sets a and b respectively, then the indices ν_A, ν_B and ν_G will satisfy

$$\nu_G(d) = \begin{cases} \nu_A(d) + \nu_B(d) & \text{for } d \neq a \cap b, \\ \nu_A(d) + \nu_B(d) - 1 & \text{for } d = a \cap b. \end{cases} \quad (5)$$

This is Lemma 8 of Lauritzen et al. (1984).

In this paper we will for every (complete) subset d of the vertices of a decomposable graph compute the index defined by the sum of the indices of the connected components of the graph. If d is not a subset of the vertices of a connected component, then $\nu(d) = 0$. For $d = \emptyset$ we have $\nu(d) = \nu(\emptyset) = 1 - |T|$, where $|T|$ is the number of connected components of the graph.

Hypergraphs

Restating, a *generating class* is a set of subsets of a finite set such that no element in the generating class is a subset of another element.

The generating class $\mathcal{H}$ of a model may be viewed as the edges of a *generating class hypergraph*, in this paper called a *hypergraph*: a graph $\mathcal{H} = (\mathrm{V}(\mathcal{H}), \mathcal{H})$ with vertices, nodes, $V(\mathcal{H}) \subseteq \Delta \cup \Gamma$ the variables of the model and edges $\mathcal{H}$ the maximal permissible interaction terms between variables.

The edges $\mathcal{H}$ are a generating class and are not only subsets of cardinality 2 of the variables, but subsets of any size of the set of vertices.

The *2-section graph* of a hypergraph $\mathcal{H} = (\mathrm{V}(\mathcal{H}), \mathcal{H})$ is a simple undirected graph $G^{\mathcal{H}} = (V(G^{\mathcal{H}}), E(G^{\mathcal{H}}))$ with vertices $V(G^{\mathcal{H}}) = V(\mathcal{H})$ and edges $E(G^{\mathcal{H}}) = \{\{v, w\} | \exists c \in \mathcal{H} : \{\mathrm{v}, \mathrm{w}\} \subseteq \mathrm{c}\}$. To distinguish the edges of the hypergraph from the edges of the 2-section graph, we call the edges of the hypergraph, that is, the elements of the generating class, *generators*.

A hypergraph is *conformal* if the cliques of its' 2-section graph are the edges of the hypergraph. Thus a model $\mathcal{H}$ is graphical, if the hypergraph with generators, that is, edges, $\mathcal{H}$ is conformal. In Badsberg (1996b) an algorithm for testing this can be found.

Definition 3 Two subsets a and b form a decomposition of $V(\mathcal{H})$ relative to a hypergraph $\mathcal{H} = (\mathrm{V}(\mathcal{H}), \mathcal{H})$, if $a \cup b = V(\mathcal{H})$, $a \setminus b \neq \emptyset$, $b \setminus a \neq \emptyset$, a and b are separated by $a \cap b$ in the 2-section graph of $\mathcal{H}$, and $a \cap b \subseteq c$ for some generator of $\mathcal{H}$.

If a hypergraph is conformal and further the 2-section graph of the hypergraph is decomposable, then the hypergraph is *acyclic*. For a hierarchical model to be decomposable, the hypergraph of the model has to be acyclic, that is, the hypergraph has to be conformal and the 2-section graph of the model has to be decomposable. (A hypergraph need not be a generating class hypergraph to be an acyclic hypergraph, that is, an acyclic hypergraph may contain edges that are subsets of other edges.)

For hypergraphs the terms *reducible*, *irreducible (prime)* and *irreducible component (maximal prime subgraph)* are defined analogously to graphs.

On contingency tables a *decomposable model* is a log-linear model associated with a decomposable graph or an acyclic hypergraph, a *graphical model* is associated with a graph or a conformal hypergraph, and each (generating class) hypergraph corresponds to a *hierarchical model*.

Dual representation

The dual representation of a model is the minimal interaction terms set to zero, see Edwards and Havránek (1987).

Collapsible

Definition 4 A hierarchical log-linear model $\mathcal{M}$ is collapsible onto a, if one of the two following equivalent properties hold:

i) for all $p = p(i) \in \mathcal{M}$, we have that $p(i_a) \in \mathcal{M}_{\mathrm{a}}$,

ii) for all $i_a \in \mathcal{I}_{\mathrm{a}}$, $\hat{p}(i_a) = \hat{p}_a(i_a)$.

For proof of equivalence, see Asmussen and Edwards (1983). Note that $\hat{p}(i_a)$ is the marginalized maximum likelihood estimate whereas $\hat{p}_a(i_a)$ is the maximum likelihood estimate in the restricted model $\mathcal{M}_{\mathrm{a}}$. See (*e.g.*) Lauritzen (1996) for notation for cells and probabilities.

Theorem 2.3 of Asmussen and Edwards (1983) states that a hierarchical model $\mathcal{M}$ is collapsible onto a, if and only if the boundary of every connected component of a^c is contained in a generator of $\mathcal{M}$.

An algorithm for determining the smallest set containing a given set such that a given hierarchical log-linear model is collapsible onto the set can be found in Badsberg (1996a).

Mixed models

For mixed models, models with both discrete and continuous variables, the following queries can be returned from CoCoCg: *MIM-model*, *homogeneous*, *mean-linear*, *D-collapsible*, *Q-equivalent*. See Edwards (1990), Edwards (2000) or Lauritzen (1996) for definition of the three first terms and Edwards (2000) for the two last terms. These characteristics can be returned in CoCoCg as flags for each irreducible component of a mixed model when returning junction trees.

Model edition

In CoCo and CoCoCg the most fundamental *model editing* actions are, Badsberg (2001):

Generate graphical: A graphical model is generated for a hierarchical model by finding the cliques of the 2-section graph of the model. *Generate decomposable*: *A chordal cover* is made by adding a fill-in to the 2-section graph of a model. *Adding* and *dropping of edges*: These operations are performed on the 2-section graph of a model, and the new models are formed by finding the cliques of the resulting graphs. *Add interaction*: Add interaction trivially adds generators to the generating class of a model, and cleans up the resulting set of sets by removing subsets of other generators to form a generating class. *Drop interaction*: The action to do is to to add generators to the dual representation of the model. Supersets of others sets in the resulting set of sets are removed and the normal representation for the

resulting dual representation is found. *Meet* (or intersection) of two models is the largest model contained in both models. *Join* (or union) of two models is the smallest model containing both models, and forms the same generating class as adding interactions does.

In model selection (*e.g.*) it is interesting to know whether a model is decomposable after dropping an edge. Since finding the cliques from an edge list can be hard this query should not be answered by first removing the edge from the 2-section graph of the model, then find the cliques of the resulting model, and finally determine whether this model is decomposable. It is much easier to answer the query by investigating whether the edge to drop is the subset of exactly one clique of the initial model. Similarly for other queries. Thus queries about the class of a model can be asked with one of the model editing operations as additional argument. A important query taking two models as arguments is the query testing whether the two models are nested.

2 Examples

Returning sets and generating classes for components of the graph

The following will on a final model of Edwards and Havránek (1987) show examples on returning components as neighbours, connected components, prime components, separators, shortest paths and cut sets:

```
>   library(CoCo)
Loading required package: methods
Loading required package: CoCoCore
Loading 3/4 of the CoCo-bundle. Copyright, Jens Henrik Badsberg, 2002.

 o CoCoCore:    The single interface function for communication bet-
                ween R (Splus) and CoCo, and 134 auxiliary functions.
 o CoCoObjects: About 24 functions for creating CoCo- and CoCo-model-
                objects, and for recovering these objects.
 o CoCoOldData: About 45 deprecated functions for reading CoCo-data
                from files and for setting and returning options.
 o CoCoRaw:     The 71 (103) functions of interest to you.

For documentation see, Badsberg, J.H.: A guide to CoCo, JSS, 2001, and
Badsberg, J.H.: Xlisp+CoCo, Aalborg, 1996.  The names of the functions
of R+CoCo are similar to those of Xlisp+CoCo; with '-' replaced by '.'
or for a few functions omitted and the following letter capitalized.
Some functions have been replaced by arguments to others, e.g. all the
set-.. commands replaced by optionsCoCo(), and the functions collected
in editModel(). A very few commands have been renamed.
The manual pages will give you the form of arguments of the functions.
```

```
The four calls ''data(Reinis); read.model("*", coco.id = Reinis);
backward(recursive=T); eh()'' will give you a small example.
Please quit by '.quit()' to remove temporary files.

Loading required package: CoCoObjects
Loading required package: CoCoRaw
>   data(Reinis)
>   read.model("ACE,ADE,BC,F;", coco.id = Reinis)
Recovering CoCo-object: ' Reinis '.

 CoCo     -     A program for estimation, test and model search
 in very large `Co'mplete and `InCo'mplete `Co'ntingency tables.
 1.5.R2.138           Fri Dec 20 09:00:00 CET 2002
 Compiled with cc, a C compiler for ...
 Copyright (c) 1991, by Jens Henrik Badsberg
 Licensed to ...

Setting slot-value  .specification  of ' Reinis '.
Setting slot-value  .observations  of ' Reinis '.

[1] "ACE,ADE,BC,F;"
>   return.sets(type = "neighbours", set = "AC")
$string
[1] "[[BDE]]"

>   return.sets(model = "current",
+               type = "connected.components")
$string
[1] "[[F][ABCDE]]"

>   return.sets(model = "current",
+               type = "connected.component", set = "C")
$string
[1] "[[ABCDE]]"

>   return.sets(model = "current", type = "prime.components")
$string
[1] "[[F][BC][ADE][ACE]]"

>   return.sets(model = "current", type = "separators")
$string
[1] "[[C][AE]]"

>   return.sets(model = "current", type = "shortests.paths",
+               u = "D", v = "B")
$string
```

```
[1] "[[AC][CE]]"

>   return.sets(model = "current", type = "cut.sets",
+               u = "B", v = "D")
$string
[1] "[[C][AE]]"

>   return.sets(model = "current", type = "cut.sets",
+               set.a = "B", set.b = "AD")
$string
[1] "[[C]]"
```

Returning vertex orders orders

Vertex orders are returned in a matrix with the columns ordered as the order of the vertices in the specification of the data, or with the columns ordered as the found vertex order. The paths found in the previous section gives the vertices of the paths from "D" to "B", but not the order in which the vertices should be visited. This order can also be found by the function `return.vertex.order`.

```
>   read.model("[[F][BC][AD][DE][AC][CE]];")
[1] "[[F][BC][AD][DE][AC][CE]];"
>
>   # Lex-M order:
>   return.vertex.order(model = "current", invers.order = FALSE,
+                       default.order = FALSE,  max.card = FALSE)
         A B C D E F
Order    6 2 5 4 3 1
Complete 1 1 1 1 0 1
>
>   # Lex-M order, inverse:
>   return.vertex.order(model = "current", invers.order = TRUE,
+                       default.order = FALSE,  max.card = FALSE)
         [,1] [,2] [,3] [,4] [,5] [,6]
Index    "5"  "1"  "4"  "3"  "2"  "0"
Complete "1"  "1"  "0"  "1"  "1"  "1"
Name     "F"  "B"  "E"  "D"  "C"  "A"
>
>   # Max-card order:
>   return.vertex.order(model = "current", invers.order = FALSE,
+                       default.order = FALSE,  max.card = TRUE)
         A B C D E F
Order    6 4 5 3 2 1
Complete 1 1 1 1 0 1
>
>   # Lex-M order, start by ordering "F":
```

```
>   return.vertex.order(model = "current", invers.order = FALSE,
+                       default.order = FALSE,  max.card = FALSE,
+                       marked = "F")
         A B C D E F
Order    5 1 4 3 2 6
Complete 1 1 1 1 0 1
>
>   # Return a vertex order for the path from "B" to "D"
>   return.vertex.order(model = "current", sub.path = TRUE,
+                       path.order = TRUE,
+                       marked = paste("AEC"), u = "B", v = "D")
      [,1] [,2] [,3] [,4]
Index "1"  "2"  "4"  "3"
Name  "B"  "C"  "E"  "D"
```

Queries to models and sets returning a boolean

The queries about the class of a model, whether a model is a submodel of an other model and the test of whether a set is a complete separator of the model returns a boolean.

```
>   read.model("[[F][BC][AD][DE][AC][CE]];")
[1] "[[F][BC][AD][DE][AC][CE]];"
>
>   property.model(model = "current", "graphical")
[1] TRUE
>   property.model(model = "current", "decomposable")
[1] FALSE
>   property.model(model = "current", "connected")
[1] FALSE
>   property.model(model = "current", "tree")
[1] TRUE
>
>   property.set(model = "current", query = "separator", set = "AE;")
[1] FALSE
>
>   read.model("[[F][BC][ADE][ACE]];")
[1] "[[F][BC][ADE][ACE]];"
>
>   property.model(model = "current", "graphical")
[1] TRUE
>   property.model(model = "current", "decomposable")
[1] TRUE
>   property.model(model = "current", "decomposable",
+                  prior.action = "drop.edges", modification = "BC;")
[1] TRUE
```

```
>   property.model(model = "current", "decomposable",
+                  prior.action = "drop.edges", modification = "AE;")
[1] FALSE
```

Editing models

The model editing functions will return a boolean, true, if the model resulting of applying the modification is different from the model of the argument. The resulting model can be returned by `return.model`.

```
>   read.model("ACE,ADE,BC,F;")
[1] "ACE,ADE,BC,F;"
>
>   editModel(action = "normal.to.dual") ;
[1] TRUE
>   printModel("last")
 Model no.   2 [[BE][BD][AB][CD][AF][BF][CF][DF][EF]]
[1] TRUE
>
>   editModel(action = "dual.to.normal") ;
[1] TRUE
>   printModel("last")
 Model no.   3 [[BDE][CDE][ABE][ABD][ACD]]
[1] TRUE
>
>   editModel(action = "collaps.model", modification = "CD;") ;
[1] TRUE
>   printModel("last")
 Model no.   4 [[ACE][ADE]]
[1] TRUE
>
>   editModel(action = "marginal.model", modification = "CD;") ;
[1] TRUE
>   printModel("last")
 Model no.   5 [[CD]]
[1] TRUE
>
>   editModel(action = "drop.interactions", modification = "ACE;")
[1] TRUE
>   current()
[1] TRUE
>   printModel("last")
 Model no.   6 [[F][ADE][CE][AC][BC]]
[1] TRUE
>
>   editModel(action = "generate.graphical") ;
```

```
[1] TRUE
>   printModel("last")
 Model no.   7 [[ACE][ADE][BC][F]]
[1] TRUE
>
>   editModel(action = "drop.edges", modification = "AE;")
[1] TRUE
>   printModel("last")
 Model no.   8 [[F][AD][DE][CE][AC][BC]]
[1] TRUE
>
>   editModel(action = "generate.decomposable") ;
[1] TRUE
>   printModel("last")
 Model no.   9 [[ACE][ADE][BC][F]]
[1] TRUE
>
>   read.model("ACE,ADE,BC,F;")
[1] "ACE,ADE,BC,F;"
>
>   editModel(action = "drop.edges",  modification = "AE") ;
[1] TRUE
>   printModel("last")
 Model no.  11 [[F][BC][AD][DE][AC][CE]]
[1] TRUE
>
>   editModel(action = "add.edges",   modification = "CD") ;
[1] TRUE
>   printModel("last")
 Model no.  12 [[ACDE][F][BC]]
[1] TRUE
>
>   editModel(action = "drop.interactions", modification = "AE") ;
[1] TRUE
>   printModel("last")
 Model no.  13 [[F][DE][AD][CE][AC][BC]]
[1] TRUE
>
>  # editModel(action = "add.interactions", modification = "ACDE") ;
>  # printModel("last")
>
>   base()
[1] TRUE
>   read.model(order = 1, set = "*") ;
  14: [[EF][DF][DE][CF][CE][CD][BF][BE][BD][BC][AF][AE][AD][AC][AB]]
 Model is not graphical
```

```
 Cliques:[[ABCDEF]]
 2-Section Graph is decomposable
[1] TRUE
>
>   editModel(action = "meet.of.models") ;
[1] TRUE
>   printModel("last")
 Model no.  15 [[AC][AD][AE][BC][CE][DE][F]]
[1] TRUE
>   editModel(action = "join.of.models") ;
[1] TRUE
>   printModel("last")
 Model no.  16 [[ACE][ADE][AB][AF][BC][BD][BE][BF][CD][CF][DF][EF]]
[1] TRUE
```

Returning the formula for the MLE for discrete models

For pure discrete data the index of the graph can be returned for decomposable models from CoCo. For non-decomposable models the returned values are extended with specification of non-decomposable irreducible components. The returned values is a structure with 3 components: a text string with sets for complete irreducible components and separators and sets and generating classes of non-decomposable irreducible components, the index of the sets for complete components and separators, and a real for the variables specified in the data, but not in the model.

```
>   read.model("[[F][BC][AD][DE][AC][CE]];")
[1] "[[F][BC][AD][DE][AC][CE]];"
>
>   printFormula()
 P [[CE][AC][DE][AD][BC][F]] ( I [ABCDEF] ) =
   N ( I []       ) ^  -1 *
   N ( I [F]      ) ^   1 *
   N ( I [C]      ) ^  -1 *
   N ( I [BC]     ) ^   1 *
   F [[AC][CE][DE][AD]] ( I [ACDE] ) *
   1.0000000
[1] TRUE
>
>   return.decomposition(model = "current", type = "expression")
[[1]]
[1] "[][F][C][BC]/{[ACDE],[AD][DE][CE][AC]}//"

[[2]]
[1] -1  1 -1  1

[[3]]
```

```
[1] 1
```

Returning junction trees for mixed models

Junction trees of a model can only be returned from the module CoCoCg (where the above index not is available). The returned value can be a structure with a tree, the leafs being sets or models (in a text string). The following example is on Edwards (2000).

```
> library(CoCoCg)
Loading required package: methods
Loading required package: CoCoCore
Loading required package: CoCoObjects
Loading required package: CoCoRaw
Loading required package: CoCo
>
> read.model("ab/abx,aby/abxy ; ", coco.id = Rats)
Recovering CoCo-object: ' Rats '.

 No-Name   -    A program for estimation (etc.) in
 Mixed Interaction Models and small CGregressions.
 Version(0.00.a.)  Mon Jan  6 11:00:05 CET 2003
 Compiled with cc or gcc, a C compiler for ...
 Copyright (c) 1996, by Jens Henrik Badsberg
 Licensed to ...

Setting slot-value  .specification  of ' Rats '.
Setting slot-value  .observations  of ' Rats '.
   24 cases read.
[1] "ab/abx,aby/abxy ; "
> read.model("[[ab]] / [[y][ax]] / [[xy]]")
[1] "[[ab]] / [[y][ax]] / [[xy]]"
> test()
 Test of [[xy][ax][ab]]
 against [[abxy]]

            Statistic       Asymptotic       Adjusted
 -2log(Q) =   76.0848   P =  6.299E-06
 Power    =         -   P =          -
 X^2      =         -   P =          -
 DF.      =                        24
 F-stat.  =   17.1092  Df =   24,  18   P =  4.470E-08
[1] TRUE
> printModel("all")
 Model no.    2 [[ab]] / [[ax][y]] / [[xy]] /CURRENT/
 Model no.    1 [[ab]] / [[aby][abx]] / [[abxy]] /BASE/
```

```
[1] TRUE
> printFormula()
 P [[ab]] / [[ax][y]] / [[xy]] ( I [abxy] ) =
   1.000000000
  /   12/  Modl is homogeneous: [[ab]] / [[ax][y]] / [[xy]]
   /    6/  Model is continuous: [[xy]]
    F ( I [xy]    % [[]] / [[x][y]] / [[xy]] ) [m][d][l][q]
   /    9/  Modl is homogeneous: [[ab]] / [[ax]] / [[x]]
    /    5/  Modl is homogeneous: [[a]] / [[ax]] / [[x]]
     F ( I [ax]    % [[a]] / [[ax]] / [[x]] ) [m][d][l][q]
    /    6/  Model is discrete: [[ab]]
     F ( I [ab]    % [[ab]] / [] / [] ) [m][d][l][q]
    COV ( [a]     )  ^ -1
   COV ( [x]     )  ^ -1
[1] TRUE
>
> # Junction tree, with generators:
> return.junction.tree(model = FALSE, type = "junction.tree.components",
+                      split.models = FALSE, split.generators = FALSE,
+                      omit.generators = FALSE)
$left
$left$variables
[1] "[xy]"

$left$model
[1] "[[]]/[[x][y]]/[[xy]]"

$separator
[1] "<[x]>"

$right
$right$left
$right$left$variables
[1] "[ax]"

$right$left$model
[1] "[[a]]/[[ax]]/[[x]]"

$right$separator
[1] "<[a]>"

$right$right
$right$right$variables
[1] "[ab]"

$right$right$model
```

```
[1] "[[ab]]/[]/[]"

> # Junction tree split in structure,
> # without generators, e.i. the models:
> return.junction.tree(model = FALSE, type = "junction.tree.components",
+                      split.sets = TRUE, omit.generators = TRUE)
$left
[1] "[xy]"

$separator
[1] "x"

$right
$right$left
[1] "[ax]"

$right$separator
[1] "a"

$right$right
[1] "[ab]"
```

References

Asmussen, S. and Edwards, D. (1983). Collapsibility and response variables in contingency tables, *Biometrika* **70**: 567–578.

Badsberg, J. H. (1996a). Algorithms for collapsing log linear models onto sets containing a given set, *Research Report R 94–2033*, Department of Mathematics and Computer Science, Aalborg University, Denmark.

Badsberg, J. H. (1996b). Decomposition of graphs and hypergraphs with identification of conformal hypergraphs, *Research Report R 94–2032*, Department of Mathematics and Computer Science, Aalborg University, Denmark.

Badsberg, J. H. (2001). A guide to CoCo, *Journal of Statistical Software* **6**(4): 1–178. http://www.jstatsoft.org/v06/.

Darroch, J. N., Lauritzen, S. L. and Speed, T. P. (1980). Markov fields and log-linear interaction models for contingency tables, *Annals of Statistics* **8**: 522–539.

Edwards, D. (1990). Hierarchical interaction models (with discussion), *Journal of the Royal Statistical Society, Series B* **52**: 3–20 and 51–72.

Edwards, D. (2000). *Introduction to Graphical Modelling, Second Edition*, Springer-Verlag, New York.

Edwards, D. and Havránek, T. (1987). A fast model selection procedure for large families of models, *Journal of the American Statistical Association* **82**: 205–213.

Haberman, S. J. (1972). Log-linear fit for contingency tables, Algorithm AS 51, *Appl. Statist.* **21**: 218–227.

Jensen, F. V. (1988). Junction trees and decomposable hypergraphs, *Research report*, JUDEX Ltd., Aalborg.

Jiroušek, R. (1991). Solution of the marginal problem and decomposable distributions, *Kybernetike* **27**(5): 403–412.

Lauritzen, S. L. (1996). *Graphical Models*, Oxford University Press, Oxford.

Lauritzen, S. L., Speed, T. P. and Vijayan, K. (1984). Decomposable graphs and hypergraphs, *J. Austral. Math. Soc. (Series A)* **36**: 12–29.

Leimer, H.-G. (1993). Optimal decomposition by clique separators, *Discrete Mathematics* **113**: 90–123.

Rose, D. J., Tarjan, R. E. and Lueker, G. S. (1976). Algorithmic aspects of vertex elimination on graphs, *SIAM Journal on Computing* **5**: 266–283.

Tarjan, R. E. (1985). Decomposition by clique separators, *Discrete Mathematics* **55**: 221–232.

Tarjan, R. E. and 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.