Grappa: R functions for probability propagation

Peter J. Green[*]
University of Bristol, UK.

November 20, 2005

1 Introduction

Grappa is a suite of functions in **R** for calculating marginal and conditional probability distributions on collections of variables with finite state spaces, typically satisfying parsimonious conditional independence assumptions. Typical applications are to expert systems and forensic genetics. It does part of the job to the familiar Hugin system – for a summary of some relative advantages and disadvantages, see Section 4. This document describes use of Grappa; familiarity with using **R** is assumed.

Here is a familiar motivating example: the fictitious 'Asia' expert system from Lauritzen and Spiegelhalter (1988), in Grappa:

```
query('asia',c(0.01,0.99))
query('smoke')
tab(c('tb','asia'),,c(.05,.95,.01,.99),c('yes','no'))
tab(c('cancer','smoke'),,c(.1,.9,.01,.99),c('yes','no'))
tab(c('bronc','smoke'),,c(.6,.4,.3,.7),c('yes','no'))
or('tbcanc','tb','cancer')
tab(c('xray','tbcanc'),,c(.98,.02,.05,.95),c('yes','no'))
tab(c('dysp','tbcanc','bronc'),,c(.9,.1,.8,.2,.7,.3,.1,.9),c('yes','no'))

pnmarg('cancer')

prop.evid('asia','yes')
prop.evid('dysp','yes')
prop.evid('xray','no')
pnmarg('cancer')
```

The code above is divided into 3 parts: in the first, conditional probability tables are defined, implicitly linking all 8 variables into a DAG. The two values for each variable are named as `yes` and `no` (tacitly for '`asia`', '`smoke`' and '`tbcanc`'). In the second, the (prior) distribution of `cancer` is output: the request to do so has automatically forced compilation and initialisation of the so-called potential representation of the model. In the third part, evidence that three variables take the indicated values: `asia=yes`, `dysp=yes` and `xray=no` is inserted; this evidence is propagated through the graph, and the new (posterior) marginal distribution of `cancer` is output.

The following section of this guide describes how to perform each of these phases of the calculation on a general problem.

[*]Department of Mathematics, University of Bristol, Bristol BS8 1TW, UK.
Email: `P.J.Green@bristol.ac.uk`.

2 Using Grappa

2.1 Starting up Grappa

Grappa is supplied as an **R** source file, `Grappa.R`. Put this in your current working directory for **R**, start up **R**, and type: `source('Grappa.R')`.

2.2 Setting up conditional probability tables

Before starting to define a new model, you should delete any existing tables by running `rmtables()` or `clean()`.

The primitive function for setting up a conditional probability table (CPT) is `tab`. The specification for this is:

```
tab(vars,levels=dim(probs),probs,values,putsw=T)
```

The variables (`vars`) are typically specified as a character vector, for example, `c('cancer','smoke')`, although numerical 'names' are also permitted. The probability values, the conditional distribution of the first-named variable given each possible setting of the others, are specified in the argument `probs`, as a vector, or a matrix or array of dimensions appropriate to the number of variables. The variables correspond to dimensions in the same order, and a vector of probabilities is used in the usual **R** fashion, that is, 'first subscript varies fastest'. If a matrix or array is given, `levels` can be omitted and is then set to `dim(probs)`. Otherwise, `levels` must be given explicitly, subject to a default setting of `c(2,2,...)`, that is $2 \times 2 \times \cdots$, a table with all binary variables.

`tab()` returns the table as value, and if `putsw` is `TRUE` (the default) also assigns it to the table database (see Subsection 3.9).

The set of values for a variable can be specified using the argument `values`; the default is $(1, 2, \ldots)$. These values are used primarily in the annotation of tables that are output, but the default integer indexing remains available.

The function `query` is a useful shorthand for setting up binary variables with no parents: `query(var,probs,values)` is equivalent to `tab(var,2,probs,values)` with the default for `probs` being $(0.5, 0.5)$ and for values being `c('yes','no')`.

Example. Assuming that the variable `smoke` has values already set to `c('yes','no')`, the code

```
tab(c('cancer','smoke'),,c(.1,.9,.01,.99),c('yes','no'))
```

sets up a 2×2 table for `cancer` given `smoke`, with the conditional probabilities $p(\texttt{cancer} = \texttt{yes}|\texttt{smoke} = \texttt{yes}) = 0.1$, $p(\texttt{cancer} = \texttt{no}|\texttt{smoke} = \texttt{yes}) = 0.9$, $p(\texttt{cancer} = \texttt{yes}|\texttt{smoke} = \texttt{no}) = 0.01$ and $p(\texttt{cancer} = \texttt{no}|\texttt{smoke} = \texttt{no}) = 0.99$.

This example could have been written
`tab(c('cancer','smoke'),,matrix(c(0.1,0.9,0.01,0.99),2,2),c('yes','no'))` with equal effect.

The table set up by these commands has the form

```
> t.cancer.smoke
           smoke=yes smoke=no
cancer=yes       0.1     0.01
cancer=no        0.9     0.99
```

2.3 Reading off marginal distributions

To print the (normalised) marginal distribution of a variable at any stage, just type `pnmarg(var)`; for example, `pnmarg('cancer')`.

Marginals for all variables simultaneously can be obtained using the function `fq()`, or for a specified set of variables by, e.g. `fq(c('var1','var2',var3'))`. This prints out a compact table of probabilities, and also returns the table invisibly as a numerical matrix, one column for each variable. It takes options overriding its default behaviour: `print=F` suppresses printing, `trans=T` transposes the table, and `values=T` annotates the output with the assigned value sets for each variable (only when `trans=T`). The probabilities are `round`ed to `digits` digits, default 5.

Joint marginals for 2 or more variables in the same clique are also easily obtained, by calling for example `joint(c('var1','var2',var3'))`. Joint distributions for variables in different cliques cannot be obtained directly.

2.4 Inserting evidence and propagating its effect

If data (or 'evidence') is introduced, we want to recompute all probabilities conditional on this (and conditional on all evidence already introduced). This is done with the `prop.evid` function:

```
prop.evid(var,value,usevs=F,quiet=!getOption('verbose'))
```

For example, `prop.evid('asia','yes')`. The value of the observed variable may be given either as an integer index, or as the corresponding element of the value set for that variable, `vs.var`. If there is ambiguity, that is, if the elements of the value set are numerical, then the given value is treated as an index, unless the argument `usevs` is set to `TRUE` to force use of the value set.

Entries of the table for one of the cliques containing `var` are updated to reflect the evidence, and this is then propagated through the junction tree, permanently modifying the potential representation (see the next section for more about these technical terms). This is done silently if `quiet` is `TRUE`.

3 Some more technical details

3.1 Compiling

To compile your graph, simply type

```
compile()
```

By default, this is done automatically when it is needed, under most circumstances: see Section 3.3.

This computation has two steps, which can be executed separately if desired:

`makeadj()` – the tables database is scanned, the variables in the model are listed in `var.names`, and the adjacency matrix of the moral graph created in `j.adj`.

`mcwh()` – the adjacency graph is csanned, eliminating vertices one-by-one using the minimum clique weight heuristic (Kjaerulff, 1990); cliques and separators are identified, and indexed in `j.cq` and `j.sp`. If the graph is not decomposable, this will be discovered, and edges added to the graph to make it decomposable if necessary. Cliques and separators are processed, forming a junction tree in `j.tree`. This function also checks that the graph is connected, and exits with an error message if not.

Optionally, the call to `mcwh()` is replaced by calls to:

`mcs()` – maximum cardinality search is performed using the data in this matrix: cliques and separators are identified, and indexed in `j.cq` and `j.sp`. If the graph is not decomposable, this will be discovered, and edges added to the graph to make it decomposable if necessary: these are listed in `j.joined`, and

`makejt()` – the cliques and separators are processed, forming a junction tree in `j.tree`. This function also checks that the graph is connected, and exits with an error message if not.

To force use of `mcs()` and `makejt()`, set `options(usemcs=TRUE)`.

All these steps create and use global variables, overwriting any previous contents: `j.adj`, `j.cq`, `j.sp`, `j.tree`, `var.names`, and `var.nvals`. The original probability tables are not altered by these calculations.

Further, if the probabilities in the CPTs (or even the number of values of variables, or the names of those values) are changed subsequently, but the topology of the graph left unaltered, there is no need to repeat the `compile()` call. This is useful in applications where the tables are repeatedly changed within a loop.

These functions produce output reporting their activity, under the control of the logical argument `quiet`, whose default is `!getOption('verbose')`; `quiet` is the first argument to each of `compile`, `makeadj`, `mcs` and `makejt`. So with the usual default setting of **R** option `verbose` to `FALSE`, `compile()` completes the whole process silently, while following the call `options(verbose=TRUE)`, full details will be reported.

Following compilation, a compact representation of the junction tree, including the membership of all cliques and separators, can be obtained by typing `jt()`.

3.2 Initialising the potential representation

The system is 'equilibrated' by initialising the tables for the clique/separator potential representation of the joint distribution, and passing messages around the junction tree until the clique and separator tables hold the respective marginals. These two steps are accomplished by typing

```
equil()
```

By default, this is done automatically when it is needed, under most circumstances: see Section 3.3.

This computation has two steps, which can be executed separately if desired:

`initcliqs()` – sets up `var.nvals`, then the clique and separator tables in the global lists `tcq` and `tsep` respectively, overwriting any previous contents. Subsequent propagation commands manipulate these tables.

`trav()` – uses the 'collect/distribute' propagation strategy of Jensen, Olesen and Andersen(1990), centred on an arbitrary root-clique, to equilibrate the potential representation.

All of `equil`, `initcliqs` and `trav` also take the `quiet` argument, as in Subsection 3.1 above.

3.3 Automatic compilation and equilibration

Under most circumstances, the compilation and equilibration phases of the computation are initiated automatically. Automatic initiation is suppressed if the option `auto` is set to `FALSE` by typing `options(auto=FALSE)`.

Every time a CPT is inserted into the table database, a logical variable `needequil` is set to `TRUE`; unless it replaces a previous table involving the same variables, it also sets `needcomp` to `TRUE`.

Functions such as `joint`, `nm`, `pnmarg`, `prop.evid` and `simulate` which require the potential representation to be complete and equilibrated, initiate compilation and equilibration as required.

3.4 Using compiled low-level code

Except in small models, the computation involved to create the cliques, and to (re-)equilibrate the potential representation can be time-consuming in **R**. Fortran code for these operations is also provided, and if an appropriately-named dynamically linked library found in the current working directory, this is loaded and used. If it is not found, then the **R** version of the code is used instead. The results are identical with either version, but the Fortran version provides a considerable speed-up on large graphs.

The library is called `Grappa.dll` in Windows, `Grappa.so` in Linux, and in general will be `Grappa` with the extension `.Platform$dynlib.ext`.

The functions `compile()`, `equil()`, `mcs()` and `trav()` have an argument `uselib`: if `TRUE`, the library versions of these two functions are used. If `uselib` is missing, the default value is that set by `options(uselib=...)`, and if that is not set, `TRUE` is assumed. These defaults also apply to compillation and equilibration automatically invoked.

There is (currently) no **R** version of `mcwh()`, so lack of access to the compiled code forces use of `mcs()` and `makejt()` instead.

See the **R** documentation for instructions on how to compile the library.

Under Windows, the command is

```
Rcmd SHLIB *.f -o Grappa.dll
```

Under Cygwin in Windows, do the same, having created a file `Makevars.win` with the line

```
PKG_LIBS = -mno-cygwin
```

in the same directory.

Under Linux, type

```
R SHLIB *.f -o Grappa.so
```

3.5 Retracting evidence

There is no direct way to retract evidence, unlike in some other programs for probability propagation – it is necessary either

to re-initialise (as in Subsection 3.2), and re-enter any evidence that is not retracted, or

to use the stack manipulation functions in the following section, which allow the current state of the clique and separator tables to be saved and returned to later, so that the effect of alternative evidence values can be investigated.

3.6 Saving and restoring values of global variables

One disadvantage of the reliance of Grappa on global variables with pre-fixed names is that it can be awkward to explore several models together, or investigate various parameter settings or alternative evidence values.

A simple device for saving current values of selected global variables on a stack, and recovering these values later, is provided to help to circumvent this.

Variables are 'pushed' (inserted) onto the stack (which is a global variable, a list called `stack`) using the function `push(`var_1,var_2,`...)`. The default is `push(tcq,tsep)`, thus saving the current clique and separator tables. Later, these variables can be 'popped' (recovered) from the stack using `pop()`; this overwrites any more recent values for the saved global variables, and (by default) removes the saved values from the stack. The stack operates on the usual 'last in, first out' basis.

The current contents of the stack are summarised by calling `peek()`; items most recently pushed onto the stack are listed first.

In fact, the storage structure is a little more general than a simple stack. The function `pop()` has two arguments: `which` to specify which item to pop, default the most recently pushed, and `keep`, default `FALSE`, to specify whether the item should be left on the 'stack' after it is recovered.

3.7 Functions for setting up conditional probability tables for genetics applications

Applications in genetics tend to have features that can be exploited to produce particularly succinct model specifications. These features include: 'gene' nodes usually having the same value (allele) sets, 'genotype' nodes having values determined by the gene values, 'mixture' nodes representing mixed traces (mixtures of two or more genotypes), founder nodes often all having the same distributions (the population gene frequencies), and many of the CPTs representing simple genetic laws and logical relationships, such as that between a child's gene and those of its parents (Mendel's law), and the rule determining a genotype from the two genes.

All of these features are provided using the following functions in Grappa:

```
founder(g,freq)
genotype(gt,mg,pg,nall)
mendel(cg,tmg,tpg,nall)
select(tfg,pfg,tfeqpf,freq)
mix(mix,agt,bgt,nall)
or(p,q,r,qv=c(T,F),rv=c(T,F))
and(p,q,r,qv=c(T,F),rv=c(T,F))
by(v,v1,v2,...)
which(v,w,lw=1:ll,...)
```

Before using these, it is useful to set two key global variables: `vs.alleles` and `gene.freq`. For example:

```
vs('alleles',c('8','10','11','x'))
gene.freq<<-c(.184884,.134884,.233721,.446511)
```

The default value of the `freq` argument of the functions above is `gene.freq`, and that of the 'number of alleles' argument `nall` to the length of `gene.freq` or `vs.alleles`. Also all nodes are given default value sets based on the allele names.

The purpose of the functions should be self-evident; one possible exception is `select(tfg,pfg,tfeqpf,freq)`, which is useful in paternity testing and in models with simple mutation: if `tfeqpf` takes its first value, `tfg` is a copy of `pfg`, otherwise it is drawn randomly from `freq`.

The functions `or` and `and` create tables in which `p` is a deterministic function of `q` and `r` – boolean 'or' and 'and' respectively. The `qv` and `rv` arguments specify the boolean values associated with each of the values of the variables `q` and `r`. For example, if `q` and `r` have 3 and 4 values respectively, and default value sets, then `and(p,q,r,c(F,T,T),c(T,F,F,T))` specifies that `p` is `TRUE` if '`q` is 2 or 3, and `r` is 1 or 4', and `FALSE` otherwise.

The function `by` is useful in defining a variable to be used as an inferential 'target'. The 'output' variable `v` takes a different value for each combination of values of the input variables `v1,v2,v3,...`; these values are used in the usual first-variable-varies-fastest fashion. These variables must have already been introduced in a previously created CPT, so that their numbers of values are known.

For example, if `v1` and `v2` both have three values, `by(v,v1,v2)` creates a table introducing a variable `v` with 9 values, determined by `v1,v2`, with, for example, `v`=4 if `v1`=1 and `v2`=2.

The function `which` is a table specification function, used for multiple indexing or selection. It has various uses, depending on the arguments given. The general form is `which(v, w,..., lw=1:ll)`; this makes a table for the (degenerate) conditional distribution of node `v`, given `w` and any other `(...)` node arguments. A common use of `which` is to make binary selections, e.g. `which('trace','guilty','suspectgt','altsuspectgt')`.

If other node arguments `(...)` are given, but `lw` is absent, variable `v` is deterministically the `lv[l]`'th of the remaining arguments. e.g. `which('a','i','b1','b2','b3')` means that `a` is "`bi`". If `lw` is also present, the selection is made indirectly through the index values in `lw`, for example, `which('smg','pattern',lv=c(3,3,3,3,2,1,3),'spg','aspg','pool')` sets `'smg'` to be equal to the value of `'aspg'` if `'pattern'` is 6, to that of `'spg'` if `'pattern'` is 5, and otherwise (`'pattern'` is 1, 2, 3, 4 or 7) to that of `'pool'`.

If there are no additional parent nodes `(...)`, `which` makes a table where `v` is `lw[i]` when `w` is `i` e.g.

```
> t.a
  a=A    B    C    D
 0.25 0.25 0.25 0.25
> which('b','a',lw=c(3,1,2,1))
> t.a.b
    a=A B C D
b=1   0 1 0 1
  2   0 0 1 0
  3   1 0 0 0
```

3.8 Simulating data from models in Grappa

You can generate values from the current model using

```
    simulate(nobs=1)
```

for example `x<-simulate(1000)`. This uses the current clique and separator tables, and thus generates values conditional on all evidence inserted so far.

The function returns values as a matrix with `nobs` rows, and one column for each variable in the model. The variables are in the same order as they are in the globals variables `var.names`.

Empirical marginal distributions computed from such a matrix `x` can be obtained using the function `fq(x)`, whose other arguments are described above in Section 2.3.

3.9 Data structures

Grappa uses variables in the global environment. This avoids problems with scope in programming, but does require some discipline to avoid overwriting important data. The following variables are used:

Name	Purpose	Created/modified by
`t.`*var*$_1$.*var*$_2$...	CPTs	`tab()`, `query()`, etc.
`vs.`*var*	Value set for variable	`vs()`, etc.
`var.names`	Names of variables	`makeadj()`
`j.adj`	Adjacency matrix	`makeadj()`
`j.cq`	Clique memberships	`mcwh()`, `mcs()`
`j.sp`	Separator memberships	`mcwh()`, `mcs()`
`j.tree`	Junction tree	`mcwh()`, `makejt()`
`var.nvals`	Numbers of values of variables	`initcliqs()`
`tcq`	Clique tables	`initcliqs()` and `trav()`
`tsep`	Separator tables	`initcliqs()` and `trav()`
`vs.alleles`, `gene.freq`	See Subsection 3.7	

3.10 Other reserved symbols

Users should take care not to overwrite definitions of the functions and global variables in Grappa. The current complete list, including those described in this document, various auxiliary routines, and utility functions, is

```
acliq      and        by         clean      compile
cs         equil      evid       fast.trav  fetch
fns        founder    fq         ftrcs      ftrjt
genotype   Grappa.dir gtvals     initcliqs  join
joint      jt         make       makeadj    makejt
marg       mcs        mcsf       mcsr       mcwh
mendel     mix        mult       needcomp   needequil
nm         norm       nvals      or         pass
passf      peek       pl.adj     pnmarg     pop
print.tab  prop.evid  prvs       push       put
query      rdargs     rdir       rmtables   se
select     set        si         sim        simn
simulate   sizes      ss         stack      tab
tables     trav       travf      travr      vars
vs         which
```

3.11 Programming using Grappa

3.11.1 Systematic naming of variables, and looping

Example:

```
for(x in c('mother','father','son','daughter'))
        genotype(cs(x,'gt'),cs(x,'mg'),cs(x,'pg'))
```

3.11.2 Use of built-in R functions for distributions

Example:

```
p<-c(.04,.98)
P<-array(0,c(5,2,4))
for(i in 1:2) for(n in 1:4) P[,i,n]<-dbinom(0:4,n,p[i])
tab(c('test','disease','number'),c(5,2,4),P)
```

4 Comparisons with Hugin

	Advantages of Grappa	Disadvantages of Grappa
Compared to Hugin Lite	Larger problem sizes Programmability	No graphical interface No continuous variables No decision or utility nodes No retraction of evidence
Compared to Hugin Professional	Cost	See above

5 Changes in Grappa

Since Grappa was first released, numerous facilities have been added, particularly those for automatically initiating compilation and equilibration, for using a dynamically linked library for the major computation, and for faster compliation by elimination, using minimum clique weight heurstic. The only changes that would possibly prevent an earlier Grappa computation from working are:

1. The argument `putsw` to the function `tab()` is now in 5th place, not 4th.
2. The change of the default value for the `quiet` argument in the compilation and propagation functions from `FALSE` to `!getOption('verbose')`. This only affects the form of the reports from these functions.
3. You should no longer specify the numbers of values of the input variables to the function `by()`

6 Obtaining Grappa

The source code for Grappa, this user guide, a file of examples (in a plain text file so that they can be copy-pasted into an **R** session), and background material can all be downloaded from
`http://www.stats.bris.ac.uk/~peter/Grappa/`

7 Examples

The following complete examples illustrate some of the ideas above. In each case, the input is shown, assuming all tables have been deleted before starting. The output is not shown completely, and not in sequence, but extracts from it are shown after the `=>` symbol.

7.1 Simple illustration of Bayes' theorem: 'blood tests'

There are two diseases, 'harmless' and 'serious' with prevalences 95% and 5% among patients presenting with a certain symptom. A blood test is available to diagnose the serious disease, with a 4% false positive rate and a 2% false negative rate. The test can be applied up to 4 times, results being independent. What are the posterior probabilities of the serious disease given one positive test result, out of one test? Or for one positive out of 3 tests?

```
tab('disease',,c(0.95,0.05),c('harmless','serious'))
tab('number',4,rep(0.25,4))
p<-c(.04,.98)
```

```
P<-array(0,c(5,2,4))
for(i in 1:2) for(n in 1:4) P[,i,n]<-dbinom(0:4,n,p[i])
tab(c('test','disease','number'),c(5,2,4),P,0:4)

prop.evid('number',1)
prop.evid('test',1,usevs=T)
pnmarg('disease')

equil()
prop.evid('number',3)
prop.evid('test',1,usevs=T)
pnmarg('disease')

=>
  disease=harmless disease=serious
        0.4367816       0.5632184
                         likrat= 0.7755102

  disease=harmless disease=serious
        0.9994406    0.0005593543
                         likrat= 1786.776
```

7.2 An ordinary Markov chain

This makes use of numerical variable 'names', and also uses the function `fq()` to extract and print normalised marginal distributions.

```
tab(0,5,rep(0.2,5))
for(i in 1:20) tab(c(i,i-1),c(5,5),c(
 .7,.3,0,0,0,
 .3,.4,.3,0,0,
 0,.3,.4,.3,0,
 0,0,.3,.4,.3,
 0,0,0,.3,.7))
prop.evid(10,2)
fq(0:20,tr=T)

=>

      [,1]    [,2]    [,3]    [,4]    [,5]
0  0.26574 0.24110 0.20058 0.15926 0.13332
1  0.27397 0.24653 0.20100 0.15409 0.12441
2  0.28308 0.25273 0.20171 0.14833 0.11416
3  0.29303 0.25986 0.20291 0.14194 0.10226
4  0.30369 0.26816 0.20498 0.13492 0.08826
5  0.31470 0.27799 0.20850 0.12726 0.07155
6  0.32520 0.29020 0.21450 0.11880 0.05130
7  0.33300 0.30700 0.22500 0.10800 0.02700
8  0.33000 0.34000 0.24000 0.09000 0.00000
```

```
9  0.30000 0.40000 0.30000 0.00000 0.00000
10 0.00000 1.00000 0.00000 0.00000 0.00000
11 0.30000 0.40000 0.30000 0.00000 0.00000
12 0.33000 0.34000 0.24000 0.09000 0.00000
13 0.33300 0.30700 0.22500 0.10800 0.02700
14 0.32520 0.29020 0.21450 0.11880 0.05130
15 0.31470 0.27799 0.20850 0.12726 0.07155
16 0.30369 0.26816 0.20498 0.13492 0.08826
17 0.29303 0.25986 0.20291 0.14194 0.10226
18 0.28308 0.25273 0.20171 0.14833 0.11416
19 0.27397 0.24653 0.20100 0.15409 0.12441
20 0.26574 0.24110 0.20058 0.15926 0.13332
```

7.3 A mixed-trace problem in forensic genetics

A crime scene blood sample contains alleles 8, 10 and 11 at a certain STR marker, so must be a mixture of blood from two (or more) individuals. A body is found, elsewhere, with genotype 8-10; a suspect is 8-11. If the population gene frequencies for 8, 10 and 11 are (.184884,.134884,.233721), what is the likelihood ratio for the hypothesis: the trace is a mixture from the suspect and victim, vs. that for: the trace is a mixture from the victim and an unknown member of the population, assuming Hardy-Weinberg and linkage equilibrium apply, and that there are exactly two individuals' blood present in the mixture? (For further explanation of this example, see Mortera (2002).)

```
vs('alleles',c('8','10','11','x'))
gene.freq<<-c(.184884,.134884,.233721,.446511)
founder('vmg')
founder('vpg')
genotype('vgt','vmg','vpg')

founder('smg')
founder('spg')
genotype('sgt','smg','spg')

query('T2eqv',,c('V','U'))
query('T1eqs',,c('S','U'))
by('target','T2eqv','T1eqs')

select('T2mg','vmg','T2eqv')
select('T2pg','vpg','T2eqv')

select('T1mg','smg','T1eqs')
select('T1pg','spg','T1eqs')

genotype('T2gt','T2mg','T2pg')
genotype('T1gt','T1mg','T1pg')

mix('mix','T2gt','T1gt')

prop.evid('vgt','8-10')
prop.evid('sgt','8-11')
prop.evid('mix','8-10-11')
pnmarg('target')

=>
 target=VS  target=US target=VU  target=UU
 0.7278388 0.09543417 0.1485508 0.02817623
```

7.4 Mixtures with an unknown number of contributors

With a larger, or unknown, number of contributors to the mixed trace, a different formulation, due to Mortera, Dawid and Lauritzen (2002) proves more efficient. Here is the Grappa code for the HBGG marker case of data in Tables 1 and 5 of Mortera, Dawid and Lauritzen, but with the maximum number of unknown contributors (`nmax`) allowed to vary freely (`nmax` is 2 in the tables cited, but set to 4 in the code below):

```
vs('alleles',c('A','B','C'))
gene.freq<<-c(.566,.429,.005)
nmax<-4

for(id in c('v','s',paste('u',1:nmax,sep='')))
      {founder(cs(id,'mg'));  founder(cs(id,'pg'))}
query('v.in.mix',,c('V','U'))
query('s.in.mix',,c('S','U'))
tab('n.unknown',nmax+1,rep(1/(nmax+1),nmax+1),nmax:0)
by('target','v.in.mix','s.in.mix','n.unknown')
for(al in vs.alleles)
      {
      for(id in c('v','s',paste('u',1:nmax,sep='')))
            or(cs(id,'has',al),cs(id,'mg'),cs(id,'pg'),
                  al==vs.alleles,al==vs.alleles)
      for(id in c('s','v'))
            and(cs(al,id),cs(id,'has',al),cs(id,'.in.mix'))
      for(n in 1:nmax)
            {
            id<-cs('u',n)
            and(cs(al,id),cs(id,'has',al),'n.unknown',,(nmax:0)>=n)
            }
      or(cs(al,'m',0),cs(al,'v'),cs(al,'s'))
      for(n in 1:nmax)
            or(cs(al,'m',n),cs(al,'m',n-1),cs(al,'u',n))
      }

prop.evid(cs('Am',nmax),'yes')
prop.evid(cs('Bm',nmax),'yes')
prop.evid(cs('Cm',nmax),'no')

prop.evid('vhasA','yes')
prop.evid('vhasB','yes')
prop.evid('shasA','yes')
prop.evid('shasB','no')
prop.evid('shasC','no')

pnmarg('target')
joint(c('v.in.mix','s.in.mix'))
fq('n.unknown',tr=T,va=T)

=>
 target=VS4 target=US4 target=VU4 target=UU4 target=VS3 target=US3 target=VU3
 0.05841203 0.05777163 0.05841203 0.05770188 0.05900056 0.05700155 0.05900056
 target=UU3 target=VS2 target=US2 target=VU2 target=UU2 target=VS1 target=US1
 0.05662253  0.0595949 0.05335493  0.0595949  0.0512955 0.06019511 0.04071695
 target=VU1 target=UU1 target=VS0 target=US0 target=VU0 target=UU0
 0.06019511 0.02952696 0.06080143          0 0.06080143          0

          s.in.mix=S s.in.mix=U
v.in.mix=V  0.2980040  0.2980040
v.in.mix=U  0.2088451  0.1951469

               4       3       2       1      0
n.unknown 0.2323 0.23163 0.22384 0.19063 0.1216
```

References

Kjærulff, U. (1990). *Triangulation of graphs - algorithms giving small total state space.* Technical report R90-09, Institute of Electronic Systems, Aalborg University.

Jensen, F. V., Olesen, K. G. and Andersen, S. K. (1990). An algebra of Bayesian belief universes for knowledge-based systems. *Networks*, **20**, 637–659.

Lauritzen, S. L. and Spiegelhalter, D. J. (1988). Local computations with probabilities on graphical structures and their application to expert systems (with discussion). *Journal of the Royal Statistical Society* B, **50**, 157–224.

Mortera, J. (2003). In *Highly structured stochastic systems*, eds P. J. Green, N. L. Hjort and S. Richardson. Oxford: OUP.

Mortera, J., Dawid, A. P. and Laurizten, S. L. (2002). *Probabilistic expert systems for DNA mixture profiling.* Research report 225, Department of Statistical Science, University College, London.