MCMCMS 0.4.2
User Guide

Nicos Angelopoulos James Cussens
Department of Computer Science
University of York, York, UK
{nicos,jc}@cs.york.ac.uk

July 16, 2008

Abstract

We describe a collection of Prolog programs which implements a generic method for applying MCMC over model structures defined by Stochastic Logic Programs (SLPs). In broad terms, an SLP is used to constructively define all possible statistical models that may explain some input data and also to define a prior distribution over these models. MCMC is then run over these possible models by making moves on the tree defining all computation paths of the SLP. The methodology is generic in two ways. Firstly, in being applicable to any statistical model space that can be expressed as an SLP and secondly, in allowing for a variety of alternative proposal moves. For the former, the sole prerequisite is that we have the means of computing the likelihood of each generated model: that is, for computing the probability of data given the model.

Contents

1 Introduction

MCMCMS (*Markov chain Monte Carlo over Model Structures*) is a collection of programs which implement the ideas introduced in (Cussens, 2000), and further analysed in (Angelopoulos & Cussens, 2001). The programs were initially developed under the EPSRC research grant *Induction of Stochastic Logic Programs*, November 2000–October 2001. Currently the development is supported by EPSRC's MATHfit programme, under the grant *Stochastic Logic Programs for MCMC*, September 2003–August 2005. The software can be downloaded from `http://www.cs.york.ac.uk/~nicos/sware/slps/mcmcms/` while most of the papers can be downloaded from `http://www.cs.york.ac.uk/~jc`.

To date we have run experiments over Bayesian networks (BNs), Restricted Acyclic Partially Directed Graphs (RPDAGs), pedigrees and Classification and Regression Trees (C&RTs). The methodology is quite generic and can be applied to any model space, provided an SLP with the desired prior is constructed, and a method for computing the likelihood of constructed models is incorporated.

The remainder of this guide is structured as follows: Section 2 is a tutorial tour of the software. Section 4 describes the basic directory structure and the main files. Section 5 shows how to run MCMC experiments over SLPs. Section 6 gives information on the model spaces for which MCMCMS has been used so far. Section 7 details how the programs can work over new spaces. Finally Section 8 briefly reviews some features of the system which are still to be documented fully.

1.1 Acknowledgements

The development of this software has been supported by two EPSRC grants: *Induction of Stochastic Logic Programs*, November 2000–October 2001 and *Stochastic Logic Programs for MCMC*, September 2003–August 2005.

2 Tutorial

This section aims to show how to use MCMCMS to perform Bayesian inference using very simple examples: the goal here is entirely pedagogic and so unrealistically simple examples are used. Real examples of using MCMCMS can be found in Section 6. All the files needed to run the examples described in this tutorial can be found in the `tutorial` directory of the MCMCMS distribution. The output files which running these examples should give you can be found in the file `tutorial_out.tgz` which is available from the MCMCMS webpage: http://www.cs.york.ac.uk/~nicos/sware/slps/mcmcms/.

2.1 Simple concept learning

Our first example is the 'concept learning' problem illustrated in Fig 1. In concept learning the assumption is that there is an unknown target (or 'true') concept which is simply a subset of some data space. Data consists of positive examples (examples in the target concept) and negative examples (examples outside the target concept). There is also a hypothesis space: sets which the learning system can hypothesise to be the true concept.

Consider a very simple concept learning scenario where the hypothesis space contains a mere five hypotheses, each of which is a rectangle (Fig 1). The data consists of two negative examples (black dots in Fig 1) and a single positive example (the white dot). Clearly, only the b and c rectangles are consistent with the data.

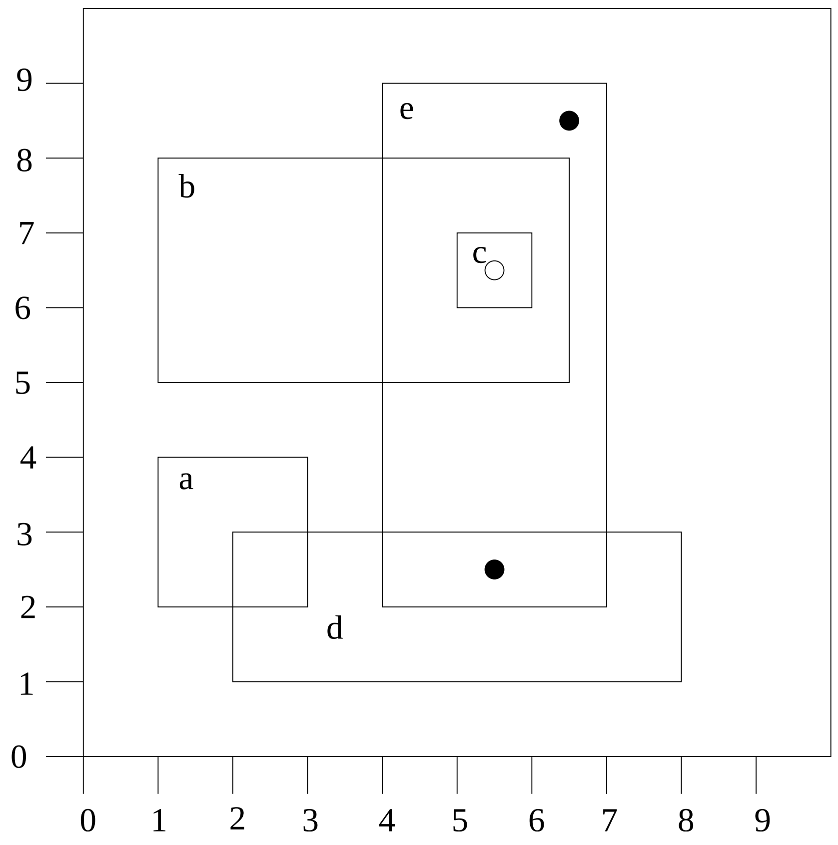

Figure 1: Concept learning with five hypotheses and three examples

In MCMCMS the user is required to define a prior over the hypothesis space using an SLP. Fig 2 shows an SLP defining one possible prior over our tiny hypothesis space. This SLP can be found in `tutorial/slps/tabular.slp`. (This filename, and all others in this section are given relative to the top level of MCMCMS once you have unpacked it.)

Each rectangle is represented by a term in first-order logic which gives the name of the rectangle and the co-ordinates of the bottom-left and top-right corners respectively. Hypotheses can be represented as the user sees fit: the important thing, as we shall see below, is to choose a representation that makes it easy to compute likelihoods.

A full explanation of how SLPs define probability distributions can be found,

```
0.1 :: hyp(rectangle(a,1,2,3,4)).
0.1 :: hyp(rectangle(b,1,5,9,6)).
0.3 :: hyp(rectangle(c,5,6,6,7)).
0.3 :: hyp(rectangle(d,2,1,8,3)).
0.2 :: hyp(rectangle(e,4,2,7,9)).
```

Figure 2: SLP defining a prior over a hypothesis space of five rectangles (in the file tutorial/slps/tabular.slp).

for example, in (Cussens, 2001). In this simple example, each probability label in Fig 2 gives a prior probability for the corresponding rectangle. The choice of predicate symbol *hyp/1* is arbitrary.

Similarly, the choice of data representation is a matter of convenience, although using anything other than Prolog might involve a lot of extra work. Fig 3 shows how we have chosen to represent the three data points of Fig 1: we just give co-ordinates and the class for each datapoint. This data can be found in `tutorial/data/exs.pl`

```
datum((5.5,6.5),pos).
datum((5.5,2.5),neg).
datum((6.5,8.5),neg).
```

Figure 3: Data for concept learning (in the file tutorial/data/exs.pl).

Now that prior and data have been determined we need to consider the likelihood function. To see what sort of likelihood function needs to be defined, consider the form of Bayes theorem relevant to concept learning. Each datapoint E_i is of the form (X_i, C_i), where X_i is the datapoint's position and C_i is its class. A hypothesis H is only required to predict an example's class given its position. It need not make any predictions concerning positions. (In statistical parlance, the co-ordinates giving the position of the example are called either *explanatory variables*, *predictor variables* or *covariates*.) Let X and C represent the vectors of positions and classes of examples, respectively. In the current case this means that $X = ((5.5, 6.5), (5.5, 2.5), (6.5, 8.5))$ and $C = (pos, neg, neg)$. The posterior $P(H|X, C)$ for any hypothesis H can then be calculated as follows:

$$P(H|X, C) = \frac{P(H|X)P(C|H, X)}{P(C|X)} \tag{1}$$

We will assume that each hypothesis attaches a class label to an example independently of other examples we have:

$$P(C|H, X) = \prod_i P(C_i|H, X_i) \tag{2}$$

The LHS of (2) is the likelihood function for this scenario. Since each hypothesis H classifies examples categorically, it is easy to see that each term on the RHS of (2) is either 1, if H classifies (X_i, C_i) correctly, and 0 otherwise. It follows that the overall likelihood $P(C|H, X)$ is 1 if H is consistent with all datapoints and 0 otherwise. Note that the prior $P(H|X)$ is only prior to the *classes* of the examples, their positions can influence the prior. In the current scenario, as Fig 2 shows, this potential influence is not used. See the work on C&RT described in Section 6.3 for a case where the values of predictor variables in the data do affect the prior.

Code for computing likelihoods (in fact, log-likelihoods) is shown in Fig 4. The first clause of predicate *model_llhood/2* declares that a `Model` (i.e. a hypothesis) has log-likelihood $-\infty$ (i.e. likelihood 0) if there exists an example `(X,C)` which is inconsistent with the model. If there are no inconsistent examples, then a log-likelihood of 0 (i.e. a likelihood of 1) is returned by the second clause. Note the use of a cut (!) to ensure that *model_llhood/2* defines a function. This code can be found in the file tutorial/simple_concept_learning.pl.

```
model_llhood(Model,-inf) :-          % Model has zero likelihood if ..
        datum(X,C),                  % .. there exists a datapoint ..
        inconsistent(C,X,Model),     % .. which is inconsistent with Model
        !.
model_llhood(_Model,0).              % hypothesis consistent with the data

inconsistent(pos,(XX,XY),rectangle(_Name,X1,Y1,X2,Y2)) :-
        (XX<X1 ; XX>X2 ; XY<Y1 ; XY>Y2).       % pos outside the rectangle
inconsistent(neg,(XX,XY),rectangle(_Name,X1,Y1,X2,Y2)) :-
        XX>=X1 , XX=<X2 , XY>=Y1 , XY=<Y2.     % neg inside the rectangle
```

Figure 4: Computing the log-likelihood for rectangular hypotheses (in the file tutorial/simple_concept_learning.pl).

The likelihood is the place where the representations of the hypothesis space and the data meet. What is essential is that the definition of *model_llhood/2* is consistent with whatever hypothesis and data representations are chosen.

Since MCMCMS uses the Metropolis-Hastings algorithm it does not require the log-likelihoods directly. All that is needed is that, given any two hypotheses M^* and M_i, the likelihood ratio is computed. In our current example this ratio is simply $P(C|M^*, X)/P(C|M_i, X)$. In some cases it is possible to compute this ratio without going to the trouble of computing the individual likelihoods (or log-likelihoods). However, in our current case there is no such cleverness, we compute the ratio by computing log-likelihoods and then computing the likelihood ratio. The code for doing this is in Fig 5 and can be found in tutorial/simple_concept_learning.pl. Note that `LLi` is instantiated at the time *lhood_canonical/5* is called, so *lhood_canonical/5* only needs to compute `LLst`, the log-likelihood of `MSt`, to get the likelihood ratio.

To run MCMCMS, both *model_llhood/2* and *lhood_canonical/5* must be defined by the user. However, if *lhood_canonical/5* does not use *model_llhood/2*

```
lhood_canonical(Mst,_Mi,LLst,LLi,Ratio) :-
        model_llhood(Mst,LLst),  % compute log-likelihood of Mst
        Ratio is exp(LLst - LLi).
```

Figure 5: Code for computing likelihood ratios (in the file tutorial/simple_concept_learning.pl)

then the latter can be given a 'dummy' definition. See Section 7.1 for further details.

Now that prior, data and likelihood have been defined we can run MCMCMS to produce a sample from the posterior. To do this we start up Prolog (either Sicstus or Yap) and load the file tutorial/simple_concept_learning.pl which is displayed in Fig 6.

```
:- ensure_loaded( '../run.pl' ).

run_test :- run([runs_file('runs/concept_learning_run'),
             results_dir('out/simple_concept_learning_out')]).

exs.   %do nothing with the data, except load it from data/exs.pl

lhood_canonical(Mst,_Mi,LLst,LLi,Ratio) :-
        model_llhood(Mst,LLst),  % compute log-likelihood of Mst
        Ratio is exp(LLst - LLi).


model_llhood(Model,-inf) :-          % Model has zero likelihood if ..
        datum(X,C),                  % .. there exists a datapoint ..
        inconsistent(C,X,Model),     % .. which is inconsistent with Model
        !.
model_llhood(_Model,0).              % hypothesis consistent with the data

inconsistent(pos,(XX,XY),rectangle(_Name,X1,Y1,X2,Y2)) :-
        (XX<X1 ; XX>X2 ; XY<Y1 ; XY>Y2).       % pos outside the rectangle
inconsistent(neg,(XX,XY),rectangle(_Name,X1,Y1,X2,Y2)) :-
        XX>=X1 , XX=<X2 , XY>=Y1 , XY=<Y2.     % neg inside the rectangle
```

Figure 6: Main file (tutorial/simple_concept_learning.pl) for simple concept learning.

Most of the code in tutorial/simple_concept_learning.pl concerns the likelihood and has already been discussed. As for the rest, the first line loads in the main MCMCMS file `../run.pl`. The predicate *run_test/0* is there just to abbreviate the given call to the *run/1* predicate. *run/1* is a MCMCMS predicate (i.e. the user does not define it). The user calls *run/1* with a list of options to perform a given MCMC run. These options are described in detail in Appendix A. In this case we just supply a `runs_file` and specify a directory for output.

A `runs_file` should be a Prolog file which defines a predicate *run_data/11* which sets parameters for a number of runs of the system. Fig 7 shows the *run_data/11* code relevant to our current example. This code can be found in tutorial/runs/concept_learning_run.pl. Detailed documentation for *run_data/11* can be found in Appendix B. The code given in Fig 7 says that the prior is defined in the file tutorial/slps/tabular.slp, where the `slps` directory is relative to the 'main' file for running MCMCMS, that the predicate *hyp/1* defines the prior, that data is in `data/exs.pl` (again `data` is relative to the main file) and that the run should be for 100000 iterations.

```
run_data(
         1,     % id
         uc,    % backtrack strategy
         tabular,       % SLP for prior in slps/tabular.slp
         rm,    % real models, space
         exs,   % (i) 'exs.' is called and (ii) data/exs.pl is loaded
         hyp/[],% hyp(X) is the call to generate a hypothesis X
         tab,   % 'tab' added to output filenames
         100000,% length of Markov chain
[],    % hot chains identifiers- here none
         1,     % selects first random seed from ../auxil/seeds.pl
         all    % do all post-processing operations, if any
        ).
```

Figure 7: *run_data/11* fact for concept learning (found in the file tutorial/runs/concept_learning_run.pl).

The `rm` parameter instructs that only 'real' models should be considered. Whenever a failure is encountered, Prolog's backtracking is used to find the nearest model. Alternatively to this space, when `fm` is used, failures are reported at the top level of the MCMC and the algorithm registers a non-jump iteration. For an example illustrating the differences between `rm` and `fm` see (Angelopoulos & Cussens, 2004, p.5).

The `exs` parameter plays a dual role. Firstly, this parameter determines the name of a file to load. Unless this file name is `null`, it is an error if the file does not exist or is not readable. This file will generally contain the data (it does in our current example), although this is not mandatory. As long as *lhood_canonical/5* is defined to give the correct likelihood ratios, MCMCMS does not mind how the data is found. Secondly, the goal `exs` is called. This is a 'hook' which can be used to do e.g. data pre-processing. As can be seen from Fig 6, in this case `exs` is defined to do nothing. If *exs/0* were not defined the program would still run, but a warning would be generated.

Once tutorial/simple_concept_learning.pl has been compiled, we kick off the run by simply entering `run_test.` at the Prolog prompt. Once the call to *run_test/0* has completed the data from the run can be found in `tutorial/out/simple_concept_learning_out`. Recall that `out/simple_concept_learning_out` was the subdirectory specified in the `results_dir` option given to *run/1*. Fig 8 shows what the MCMCMS run looks like. As Fig 8 indicates, the *run/0* call actually performs two MCMC runs since there are two *run_data/9* facts in tutorial/runs/concept_learning_run.pl. The second experiment will be discussed in Section 2.2.

```
pc275 ~/research/slps/mcmcms-0_1_1/tutorial sicstus
SICStus 3.11.1 (x86-linux-glibc2.3): Fri Feb 20 18:38:25 CET 2004
Licensed to cs.york.ac.uk
| ?- compile(simple_concept_learning).
% compiling /u5/jc/research/slps/mcmcms-0_1_1/tutorial/simple_concept_learning.pl...
%  compiling /u5/jc/research/slps/mcmcms-0_1_1/run.pl...
%   loading /usr/local/pkg/sicstus-3.11.1/lib/sicstus-3.11.1/library/system.po...
%   module system imported into user
%    loading foreign resource /usr/local/pkg/sicstus-3.11.1/lib/sicstus-3.11.1/library/x86-linux-glibc2.3/system.so in module system
%   loaded /usr/local/pkg/sicstus-3.11.1/lib/sicstus-3.11.1/library/system.po in module system, 0 msec 27736 bytes
%   loading /usr/local/pkg/sicstus-3.11.1/lib/sicstus-3.11.1/library/lists.po...%   module lists imported into user
%   loaded /usr/local/pkg/sicstus-3.11.1/lib/sicstus-3.11.1/library/lists.po in
module lists, 0 msec 13776 bytes
%   compiling /u5/jc/research/slps/mcmcms-0_1_1/src/init_lib.pl...
%    compiling /u5/jc/research/slps/mcmcms-0_1_1/src/lib/pl.pl...
%    compiled /u5/jc/research/slps/mcmcms-0_1_1/src/lib/pl.pl in module user, 10 msec 4612 bytes
%    compiling /u5/jc/research/slps/mcmcms-0_1_1/src/lib/sics39_pncms.pl...
yes
| ?- run_test.

running_experiment(1)
Successful extensions: ; end of extensions.
ok('gzip -f out/simple_concept_learning_out/tr_uc_rm_exs_tab_i100K__s1')

running_experiment(2)
Successful extensions: ; end of extensions.
ok('gzip -f out/simple_concept_learning_out/tr_uc_rm_exs_r2304_i100K__s1')

ok('cp /u5/jc/research/slps/mcmcms-0_1_1/tutorial/runs/concept_learning_run.pl /u5/jc/research/slps/mcmcms-0_1_1/tutorial/out/simple_concept_learning_out')
yes
| ?-
```

Figure 8: Running MCMCMS. `sicstus`, `compile(simple_concept_learning).` and `run_test.` are user input.

The models (in this case rectangles) visited during the MCMC run are saved in the file `tr_uc_rm_exs_tab_i100K__s1.gz`. Note that the filename is constructed from the various options defined using *run_data/11*. This file is simply a (gzipped) text-file trace of the models visited and so is 200001 lines long. (There is one extra initial line to record the initial model.) In the trace file lines starting with a 'b' indicate proposed models and those with a 'c' indicate the current model.

Fig 9 shows the start of the trace file which we produced for this MCMCMS run. Rectangle `d` is the initial model (it is chosen using the prior). Rectangle `c` is then proposed and as the third line shows it is accepted and thus becomes the current model. Rectangle `a` is then proposed but the proposal is not accepted, so rectangle `c` remains the current model. In fact, inspection of the trace shows that the chain remains stuck at rectangle `c` until the 22nd iteration.

```
c(rectangle(d,2,1,8,3)).
b(rectangle(c,5,6,6,7)).
c(rectangle(c,5,6,6,7)).
b(rectangle(c,5,6,6,7)).
c(rectangle(c,5,6,6,7)).
b(rectangle(a,1,2,3,4)).
c(rectangle(c,5,6,6,7)).
```

Figure 9: The start of a trace produced by an MCMCMS run (found in the file `tutorial/out/simple_concept_learning_out/tr_uc_rm_exs_tab_i100K__s1.gz`).

In this current tiny example it is trivial to pull out a simple posterior from the trace. Fig 10 shows how to pull out all the counts in the trace file. The last three lines of Fig 10 show the models actually visited (as opposed to merely proposed). The posterior probabilities estimated from the trace for rectangles `b` and `c` are 0.248 and 0.752, respectively, close to the correct values of 0.25 and 0.75. Note that `d` gets an estimated posterior of $1/200001 = 5 \times 10^{-5}$. By chance it happened to be the initial model. A small burn-in (throwing away an early part of the trace) would be enough to delete it entirely. The first five 'b' lines of Fig 10 show that models (rectangles) are being proposed according to the prior distribution.

2.2 A different prior

In the previous section there were only six hypotheses, so the use of MCMC was entirely unnecessary, except for tutorial purposes. In this section we start to move towards more realistic examples. Fig 11 shows an alternative prior over a hypothesis space of $8 \times 8 \times 6 \times 6 = 2304$ rectangles. Rectangles are sampled from this prior by first choosing the X and Y values for the bottom left corner of the rectangle, and then choosing the horizontal and vertical lengths of the

```
zcat tr_uc_rm_exs_tab_i100K__s1.gz | sort | uniq -c
   9934 b(rectangle(a,1,2,3,4)).
   9924 b(rectangle(b,1,5,7,8)).
  29943 b(rectangle(c,5,6,6,7)).
  30091 b(rectangle(d,2,1,8,3)).
  20108 b(rectangle(e,4,2,7,9)).
  24671 c(rectangle(b,1,5,7,8)).
  75329 c(rectangle(c,5,6,6,7)).
      1 c(rectangle(d,2,1,8,3)).
```

Figure 10: Extracting counts from the MCMC trace file

rectangle. The definitions of *x_dist/1* and *y_dist/1* give a bias towards smaller rectangles. The definitions of *bottom_left_x/1* and *bottom_right_x/1* mean that $(2,3)$ and $(6,7)$ are *a priori* the most probable locations for the bottom left-hand corner. Note that now all rectangles get the same name: `dummy`. We keep this dummy argument in our representation of hypotheses to save us the bother of re-writing the likelihood functions. The code displayed in Fig 11 can be found in tutorial/slps/rects2304.slp.

To use this new prior it suffices to have an appropriate *run_data/11* fact in the runs file. Fig 12 shows the relevant fact which can be found in tutorial/runs/concept_learning_run.pl. This is identical to the *run_data/11* fact shown in Fig 7, except that it points to a different file for locating the SLP prior (and uses a different id and output filename string).

Doing:

```
zcattr_uc_rm_exs_r2304_i100K__s1.gz|grep'^c'|sort|uniq-c|sort-rn
```

where `tr_uc_rm_exs_r2304_i100K__s1.gz` is the output file, gives us the frequency with which each rectangle was visited. Fig 13 shows that the most frequently visited rectangles (and hence those with highest estimated posterior probability) are as expected. They are consistent with the data and had high prior probability. However, the least visited rectangle `rectangle(dummy,6,4,8,7)` is not consistent with the data so it might seem odd that it was visited at all—after all it was visited 29 times which leads to an estimated posterior probability of 0.00029 rather than the correct value of 0. What has happened is that, by chance, `rectangle(dummy,6,4,8,7)` is the initial model. For the first 28 iterations another inconsistent model was proposed, and the proposal is rejected. On the 29th iteration `rectangle(dummy,3,4,9,8)`, a consistent rectangle, is eventually proposed. From that point on the chain visits only consistent models. Again, this shows the importance of discarding (for real applications) an initial 'burn-in' part of the sample.

2.3 Changing the likelihood

A slightly more realistic version of concept learning is to assume that each concept (in our case, each rectangle) probabilistically (or 'noisily') classifies

```
hyp(rectangle(dummy,X1,Y1,X2,Y2)) :-
        bottom_left_x(X1),
        bottom_left_y(Y1),
        x_dist(XD),
        y_dist(YD),
        X2 is X1 + XD,
        Y2 is Y1 + YD.


0.1 :: bottom_left_x(1).
0.2 :: bottom_left_x(2).
0.1 :: bottom_left_x(3).
0.1 :: bottom_left_x(4).
0.1 :: bottom_left_x(5).
0.2 :: bottom_left_x(6).
0.1 :: bottom_left_x(7).
0.1 :: bottom_left_x(8).

bottom_left_y(Y) :-
        bottom_left_x(X),
        Y is X + 1.

0.3 :: x_dist(1).
0.2 :: x_dist(2).
0.2 :: x_dist(3).
0.1 :: x_dist(4).
0.1 :: x_dist(5).
0.1 :: x_dist(6).

y_dist(D) :-
        x_dist(XD),
        D is XD + 1.
```

Figure 11: An SLP for a simple prior over 2304 rectangles (found in the file tutorial/slps/rects2304.slp).

```
run_data(
         2,     % id
         uc,    % backtrack strategy
         rects2304,     % SLP for prior in slps/rects2034.slp
         rm,    % real models, space
         exs,   % (i) 'exs.' is called and (ii) data/exs.pl is loaded
         hyp/[],% hyp(X) is the call to generate a hypothesis X
         r2304, % 'r2304' added to output filenames
         100000,% length of Markov chain
[],    % hot chains identifiers- here none
         1,     % selects first random seed from ../auxil/seeds.pl
         all    % do all post-processing operations, if any
        ).
```

Figure 12: Run information for using a more complex prior (found in the file tutorial/runs/concept_learning_run.pl).

```
   1715 c(rectangle(dummy,4,3,6,7)).
   1606 c(rectangle(dummy,5,3,6,7)).
   1288 c(rectangle(dummy,5,6,6,8)).
   1259 c(rectangle(dummy,5,3,7,7)).
   1178 c(rectangle(dummy,4,6,6,8)).
   1169 c(rectangle(dummy,2,6,8,8)).
   1073 c(rectangle(dummy,2,3,8,7)).
   1070 c(rectangle(dummy,5,5,6,7)).
   1067 c(rectangle(dummy,2,3,6,7)).
   1056 c(rectangle(dummy,4,3,7,7)).
...
     82 c(rectangle(dummy,1,5,6,10)).
     49 c(rectangle(dummy,1,6,6,12)).
      4 c(rectangle(dummy,6,4,8,7)).
```

Figure 13: Highest and lowest counts using the more complex prior

examples as positive or negative. This means that a different likelihood function must be used. For example, the likelihood function given in Fig 14 sets $P(C_i|H, X_i) = 0.8$ if X_i in inside H and $C_i = pos$ or if X_i in outside H and $C_i = neg$. Otherwise $P(C_i|H, X_i) = 0.2$. The code displayed in Fig 14 can be found in tutorial/noisy_concept_learning.pl. As before, we assume that the examples are classified independently so that to get $\log P(C|H, X)$ we add the log-likelihoods of each example. In the interests of clarity, the code in Fig 14 is very inefficient. Each time that *model_llhood/2* is called the same list of examples `Data` is constructed and sent to *plod_thru_data/4*. This is a case where some pre-processing, perhaps via *exs/0*, would be more efficient.

Fig 14 contains a clause for the *exec_extension/2* predicate. Users can use such clauses to define post-processing utilities which are run prior to the trace file being gzipped. The two declarations above the clause definition *are* required. In Fig 14 a Unix shell command for producing counts is composed using the predicate *flatoms/2* and then called using the Prolog built-in *shell/1*. When *exec_extension/2* is called, the variable `File` is instantiated to the name of the output file. *flatoms/2* is a predicate which MCMCMS uses internally, but is also available to the user. *flatoms/2* concatenates a list of Prolog atoms together: its definition can be found in `mcmcms/src/lib/flatoms.pl`.

The counts produced by the MCMCMS run for the simple 'six-hypothesis' prior and this 'noisy' likelihood can be found in Fig 15. Recall that the 'b' counts record the frequency with which hypotheses are proposed and the 'c' counts record the frequency with which MCMCMS actually visited each hypothesis. The 'b' counts in Fig 15 are similar to those found in Fig 10. This is as expected: rectangles are being proposed using the prior which is the same in both runs. Turning to the hypotheses actually visited, in Fig 10 only rectangles `b` and `c` are present (except for an initial visit to `d`) because only these two rectangles have non-zero likelihood. In contrast, in Fig 15 `b` and `c` have the highest frequencies, but the other rectangles are still visited: because the likelihood no longer rules them out entirely, they still have some posterior probability of being the 'true' rectangle.

2.4 A mixture prior

Because of the nature of the prior distributions defined by SLPs, it is very easy to formulate mixture priors. Fig 16 contains our original *hyp/1* prior over rectangles, a new *circ_hyp/1* prior over circles and a mixture prior *mix_hyp/1* which mixes the *hyp/1* and *circ_hyp/1* priors to give a distribution over rectangles *and* circles. The likelihood function must now be extended to compute the likelihood of circles: see Fig 17 to see how this is done. Since the mixture prior has a bias towards circles, circles are visited more frequently when MCMCMS is run.

2.5 Using continuous distributions

The examples so far have all concerned entirely discrete spaces. Continuous distributions can be implemented by including calls to Prolog built-ins which

```
:- ensure_loaded( '../run.pl' ).

run_test :- run([runs_file('runs/concept_learning_run'),
             results_dir('out/noisy_concept_learning_out')]).

exs.   %do nothing with the data, except load it from data/exs.pl

lhood_canonical(Mst,_Mi,LLst,LLi,Ratio) :-
        model_llhood(Mst,LLst),  % compute log-likelihood of Mst
        Ratio is exp(LLst - LLi).

model_llhood(Model,LL) :-
        findall(dat(XX,XY,C),datum((XX,XY),C),Data),
        plod_thru_data(Data,Model,0,LL).

plod_thru_data([],_Model,OutLL,OutLL).
plod_thru_data([dat(XX,XY,C)|T],Model,InLL,OutLL) :-
        (
          inconsistent(C,XX,XY,Model)
        ->
          MidLL is log(0.2) + InLL
        ;
          MidLL is log(0.8) + InLL
        ),
        plod_thru_data(T,Model,MidLL,OutLL).

inconsistent(pos,XX,XY,rectangle(_Name,X1,Y1,X2,Y2)) :-
        (XX<X1 ; XX>X2 ; XY<Y1 ; XY>Y2).       % pos outside the rectangle

inconsistent(neg,XX,XY,rectangle(_Name,X1,Y1,X2,Y2)) :-
        XX>=X1 , XX=<X2 , XY>=Y1 , XY=<Y2.     % neg inside the rectangle

:- multifile( exec_extension/2 ).
:- dynamic( exec_extension/2 ).

exec_extension(counts,File) :-
        flatoms(['sort ', File, '| uniq -c >', File, '.counts'],Com),
        shell(Com).
```

Figure 14: The file tutorial/noisy_concept_learning.pl, including a likelihood for 'noisy' concept learning.

```
 9934 b(rectangle(a,1,2,3,4)).
 9924 b(rectangle(b,1,5,7,8)).
29943 b(rectangle(c,5,6,6,7)).
30091 b(rectangle(d,2,1,8,3)).
20108 b(rectangle(e,4,2,7,9)).
 5347 c(rectangle(a,1,2,3,4)).
21740 c(rectangle(b,1,5,7,8)).
66187 c(rectangle(c,5,6,6,7)).
 4044 c(rectangle(d,2,1,8,3)).
 2683 c(rectangle(e,4,2,7,9)).
```

Figure 15: Counts from an MCMCMS run with with the `tabular.slp` prior and a 'noisy' likelihood function (found in the file `tutorial/out/noisy_concept_learning_out/tr_uc_rm_exs_tab_i100K__s1.counts`).

generate floating point values according to various distributions. (In some cases it may be necessary to call C code if the Prolog you are using does not have built-ins for the distributions you need. See Section 4.3 for pointers to the C language interface.)

In Fig 18 there is a prior that adapts the mixture prior (Fig 16) so that the radii of circles is uniformly distributed over the range $[0, 1]$. *random/1* is a predicate of library(random) in both Sicstus and Yap.

```
0.4 :: mix_hyp(Rectangle) :- hyp(Rectangle).
0.6 :: mix_hyp(Circle) :- circ_hyp(Circle).


circ_hyp(circle((X,Y),Radius)) :-
        bottom_left_x(X),
        bottom_left_y(Y),
        radius(Radius).

0.4 :: radius(0.2).
0.3 :: radius(0.3).
0.3 :: radius(0.4).

hyp(rectangle(dummy,X1,Y1,X2,Y2)) :-
        bottom_left_x(X1),
        bottom_left_y(Y1),
        x_dist(XD),
        y_dist(YD),
        X2 is X1 + XD,
        Y2 is Y1 + YD.


0.1 :: bottom_left_x(1).
0.2 :: bottom_left_x(2).
0.1 :: bottom_left_x(3).
0.1 :: bottom_left_x(4).
0.1 :: bottom_left_x(5).
0.2 :: bottom_left_x(6).
0.1 :: bottom_left_x(7).
0.1 :: bottom_left_x(8).

bottom_left_y(Y) :-
        bottom_left_x(X),
        Y is X + 1.

0.3 :: x_dist(1).
0.2 :: x_dist(2).
0.2 :: x_dist(3).
0.1 :: x_dist(4).
0.1 :: x_dist(5).
0.1 :: x_dist(6).

y_dist(D) :-
        x_dist(XD),
        D is XD + 1.
```

Figure 16: Mixture prior over rectangles and circles (found in the file tutorial/slps/mixture.slp).

```
:- ensure_loaded( '../run.pl' ).

run_test :- run([runs_file('runs/mixture_concept_learning_run'),
             results_dir('out/mixture_concept_learning_out')]).

exs.   %do nothing with the data, except load it from data/exs.pl

lhood_canonical(Mst,_Mi,LLst,LLi,Ratio) :-
        model_llhood(Mst,LLst),  % compute log-likelihood of Mst
        Ratio is exp(LLst - LLi).

model_llhood(Model,LL) :-
        findall(dat(XX,XY,C),datum((XX,XY),C),Data),
        plod_thru_data(Data,Model,0,LL).

plod_thru_data([],_Model,OutLL,OutLL).
plod_thru_data([dat(XX,XY,C)|T],Model,InLL,OutLL) :-
        (
          inconsistent(C,XX,XY,Model)
        ->
          MidLL is log(0.2) + InLL
        ;
          MidLL is log(0.8) + InLL
        ),
        plod_thru_data(T,Model,MidLL,OutLL).


inconsistent(pos,XX,XY,circle((X,Y),R)) :-
        SqD is (XX-X)**2 + (XY-Y)**2,
        R2 is R**2,
        R2 < SqD.

inconsistent(neg,XX,XY,circle((X,Y),R)) :-
        SqD is (XX-X)**2 + (XY-Y)**2,
        R2 is R**2,
        R2 >= SqD.

inconsistent(pos,XX,XY,rectangle(_Name,X1,Y1,X2,Y2)) :-
        (XX<X1 ; XX>X2 ; XY<Y1 ; XY>Y2).       % pos outside the rectangle

inconsistent(neg,XX,XY,rectangle(_Name,X1,Y1,X2,Y2)) :-
        XX>=X1 , XX=<X2 , XY>=Y1 , XY=<Y2.     % neg inside the rectangle

:- multifile( exec_extension/2 ).
:- dynamic( exec_extension/2 ).

exec_extension(counts,File) :-
        flatoms(['sort ', File, '| uniq -c >', File, '.counts'],Com),
        shell(Com).
```


Figure 17: The file tutorial/mixture_concept_learning.pl which includes additional clauses to compute ‘noisy’ likelihoods for circle hypotheses

```
0.4 :: mix_hyp(Rectangle) :- hyp(Rectangle).
0.6 :: mix_hyp(Circle) :- circ_hyp(Circle).


circ_hyp(circle((X,Y),Radius)) :-
        bottom_left_x(X),
        bottom_left_y(Y),
        radius(Radius).

radius(X) :-
        random(X).   % random/1 is a Sicstus Prolog built-in.


hyp(rectangle(dummy,X1,Y1,X2,Y2)) :-
        bottom_left_x(X1),
        bottom_left_y(Y1),
        x_dist(XD),
        y_dist(YD),
        X2 is X1 + XD,
        Y2 is Y1 + YD.


0.1 :: bottom_left_x(1).
0.2 :: bottom_left_x(2).
0.1 :: bottom_left_x(3).
0.1 :: bottom_left_x(4).
0.1 :: bottom_left_x(5).
0.2 :: bottom_left_x(6).
0.1 :: bottom_left_x(7).
0.1 :: bottom_left_x(8).

bottom_left_y(Y) :-
        bottom_left_x(X),
        Y is X + 1.

0.3 :: x_dist(1).
0.2 :: x_dist(2).
0.2 :: x_dist(3).
0.1 :: x_dist(4).
0.1 :: x_dist(5).
0.1 :: x_dist(6).

y_dist(D) :-
        x_dist(XD),
        D is XD + 1.
```


Figure 18: A mixture prior with a (truncated) uniform distribution over the radii of circles (found in the file tutorial/slps/continuous.slp).

3 Syntax

Initially an SLP clause was a Prolog clause with the addition of probability label. This label had to be a number between 0 and 1. For instance the program:

```
1/2 :: nat( 0 ).
1/2 :: nat( s(X) ) :-
        nat( X ).
```

fits a geometric distribution over the natural numbers. Posing the query $? - nat(Y)$. 10,000 times will instantiate variable Y to each number with probability similar to that shown in Fig. 19. In the plot, the y-axis plots probability for each instantiation while the x-axis plots the ith instantiation term starting from $i = 1, Y = 0$.

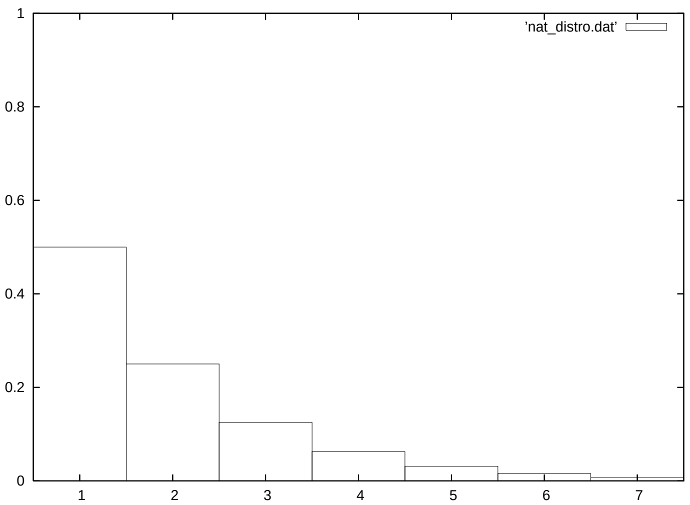

Figure 19: Geometric distribution over naturals.

The restriction to probabilistic labels that are known numbers has been proven too strong in practice. In MCMCMS we allow for functions involving a number or variables and which are evaluated at call-time. For example the following program defines a uniform distribution over element selections from a list.

```
:- pvars( umember(L,_E,_R), [T-length(L,T)] ).

1/L       : [L] : umember( El, [El|T] ).
1 - (1/L) : [L] : umember( El, [_|T] ) :-
                [L-1] : umember( El, T ).
```

The *pvars/2* directive is optional. When it is present, $umember(X, [a, b, c])$ is a valid query, whereas the query $[3] :: umember(X, [a, b, c])$ can be used both when the directive is present and when it is absent.

Returning to the definition of the predicate, note that the base case is selected with probability $1/L$ and the recursive case with probability $1 - 1/L$. When L is the length of the list in the second argument, this predicate will select an element from the list with uniform distribution. An example of this happening is shown in Fig. 20. The probability with which each element is selected, is equal to the product of the edges in the path from the root to its leaf. In the case of $Y = b$, $p(Y = b) = 2/3 * 1/2 = 1/3$. Similarly for the other cases.

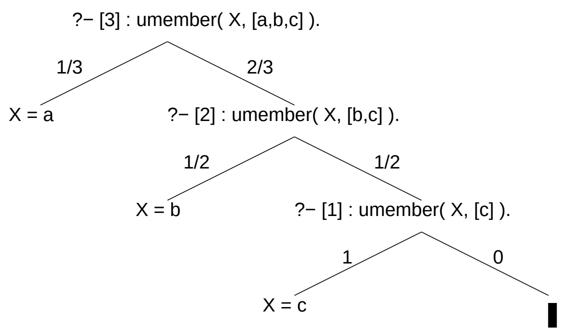

Figure 20: Selecting with uniform distribution.

For the purposes of MCMC the query $umember(X, [a, b, c])$ creates a maximum of 3 choice points. For example when $X = b$ the path is $[2 - 2/3, 1 - 1/2]$, where 1 and 2 are internal clause indices assigned order to the clauses of umember/2 from top to bottom in the order they appear in the source file. The number of, probablitistic, choice points is a crucial dimension in the selection of a bactrack point as described in Section 5.1, also see (Angelopoulos & Cussens, 2004).

It is possible to reduce the number of choice points $umember(X, [a, b, c])$ creates by using the declaration

$$: -s\_tail\_recursive(umember/2).$$

When this is present, the above query will always create a single choice point: $1 - 1/3$ in the case of $X = a$ and $2 - 2/3$ otherwise. The base case clause of the predicate should appear at the top, followed by the recursive clause(s). Note that the declaration is only effective when the body of a recursive clause does not leave any probabilistic choice-points.

The directive, $: -s\_no\_bpoint(Spec)$. forces the probabilistic backtrack points of predicate with specification *Spec* to be ingnored. Only and all the points created at calls to this predicate are ignored. Points left by the body of clauses of such a predicate to other predicates are valid backtracking choices.

4 Files and directories

Once unpacked, the programs can be loaded and run in the Prolog engines supported. Currently these are, the SICStus and Yap systems. MCMCMS was developed on and is supported in the Linux architecture. We use the Slackware distribution, it may thus be necessary to recompile some foreign code used in computing the likelihoods as described in Section 4.3.

4.1 Files

There are two files at the top-level. `mcmcms.pl` is the main entry point for the MCMC algorithm. It defines *mcmcms/11* which only experienced users should call directly. Instead users are expected to call `run(Opts)`, where `Opts` is a list specifying options. Recognised options are listed in Appendix A. *run/1* is defined in the top-level file `run.pl` so this file should always be loaded. *run/1* runs a number of MCMC experiments according to its call options. One of these options specifies a runs_file where a definition for the predicate *run_data/11* is expected. A description of *run_data/11* is given in Appendix B.

4.2 Directories

Directories at the top-level are divided between generic and model specific. The former have code that implement the MCMC algorithm and assist in making the running of experiments an easier task, while the latter hold programs that are only relevant to individual or classes of model spaces. In adddition directory `pbl` contains scripts for re-running experiments that have appeared in various epublications.
Directories in `pbl/`:

`uai01/` Experiments from (Angelopoulos & Cussens, 2001)

`ijcai05/` Experiments from (Angelopoulos & Cussens, 2005)

The generic directories are:

`auxil/` Auxiliary programs such as creating counts and visits for the chain. Also for sampling random models over an SLP. See Section 8.1.

`src/` The Prolog source files that implement the MCMC algorithm. Includes a `lib/` directory of supporting predicates.

The model specific directories are:

`bns/` Files only pertinent to MCMC over BNs. It is worth noting that most SLPs, in directory `slps`, are failure-free.

`bns/rpdags` Programs for running MCMCMS over Restricted Acyclic Partially Directed Graphs. This is a class approximating Markov equivalence classes of BNs. It shares many programs with BNs, thus it is a subdirectory of `bns/`.

`carts/` Programs for running experiments over C&RT models. Two radically different priors and two data sets have been explored.

`tutorial/` Files for the tutorial described in Section 2.

In each of the model specific directories, there might exist subdirectories similar to the top-level generic ones: `src/` and `auxil/`. In addition the following directories have been used:

`data/` This is where programs try to find the (training) data.

`runs/` Repository of scripts. Example files defining *run_data/11*, which is used by *run/1*.

`slps/` Where SLPs are looked for. Generally speaking, we keep all the files for constructing models here. These will normally also include the Prolog programs on which the SLPs are based.

See files `Readme.txt` within each model-specific directory for details on running experiments with the respective models. Also within the directories (`bns/`, `bns/rpdags/`, `carts/bcw/` and `carts/kyphosis/` e.t.c.) the user can find examples of Prolog code that uses *run/1* (see files `run_test.pl` and `run\_disp.pl`). These files ensure appropriate files for experiment-runs, likelihood, backtracking proposal, displaying and post-processing, are loaded before *run/1* is called. The `run_disp.pl` version allows visualisation of current and proposed models. `run_test` and `run_disp` put their output files into `out_test` and `out_disp` respectively. In few cases we have created `run_graphs` scripts. These are normally identical to `out_test` but for creating a pair of plots for each experiment run. These are the stays and visits plots of the chain, which are simple indicators of how much the chain moved in the space of models.

4.3 C Code recompilation

MCMCMS is developed in systems that run the Slackware distribution of Linux. If you need to run on a different distribution or operating system, you might need to recompile the glue that enables Prolog to call the logamma function from C language libraries. This should be a straightforward operation, which is documented in files `bns/src/llhood/logamma_sicstus.pl` and `bns/src/llhood/logamma_yap.pl`. The logamma function is used in computing the likelihood of proposed models given the data. In recent versions of Yap we use the built-in function lgamma/1 so there is no need for linking to the C libraries.

4.4 Development and bug reports

If you develop you own SLPs or want to run your own experiments on the provided SLPs, by far the cleanest and easiest way is to create a new directory. The structure of this directory should follow the structure of the provided experiment directories. Create subdirectories `slps/`, `data/` and `runs/`. Copy or create the appropriate files to these directories. Finally built scripts such as `run_test.pl` and `run_disp.pl` to run your experiments.

If things do not go as expected, you can add the line `:-bb_put(sr,off).` to the top of your script file. When re-executing the run, everything will be printed to the terminal. Failing everything else do not hesitate to send us a message. Preferably, make a tar.gz copy of your directory and put the file in a public location. Then send us an email with the location (see title page for our emails). Please also follow this procedure if you wish to file a bug report.

5 Running experiments

Predicate *run/1*, defined in `mcmcms/run.pl` can be used to run a number of experiments producing a number of output files. We will refer to calls of *run/1* as a 'run'. Each run will execute a number of experiments according to the arguments of *run_data/11*. For each clause of *run_data/11* the MCMC algorithm will be run with the corresponding parameters (see Appendix B). As a result a number of files will be created for each experiment. All filenames associated with a single experiment share a common stem, reflecting the parameters of the experiment. The extension of each file, reveals the kind of output stored in it. The core extentions are:

.gz the gzipped main output of the experiment. This usually records the current and proposed models at each iteration. Possibly along with their likelihoods and model size characteristics.

.stats execution statistics and the internal form of the SLP used are recorded here.

.stays for each model the chain moves to, this records the number of iterations the chain remains stuck at that model. You must load the file `auxil/create_visits.pl` to get this.

.visits the total number of times a single model appears in the chain. You must load the file `auxil/create_visits.pl` to get this.

Additional post-processing on the pre-gzipped output file can be effected by defining `exec_extension(Ext,Stem)`. For instance see `auxil/create_mgraphs.pl` or Fig 14 in the tutorial. Other aspects of the behaviour for *run/1* are controlled by the list of options given as its argument (see Appendix A). Each 'run' prints some simple diagnostics on how things are proceeding. These are written on the error stream which is connected to the terminal.

Throughout the model specific directories we have used simple Prolog programs which (a) make sure all the appropriate files are loaded, and (b) call *run/1*. For example consider file `bns/run\_test.pl` in Table 1. The first directive loads `run.pl` from top-level directory. The second, loads the likelihood to be used and the third the code for displaying Bayesian networks (only included here for illustration). The last four ensure_loaded directives, load predicates that post-process the output files. The commented out directive `bb_put(sr,off)` can be added to force all output to the terminal; this is useful for debugging. Finally *run_test/0*, is a wrap for calling *run_test/1* with *run_data/11* defined in `runs/simple.pl` creating output at 'points' 24, 28 and 29, and putting all resulting files into directory `test/` . (See Appendix A for further information on the use of output 'points'.)

```
:- ensure_loaded( '../run' ).
:- ensure_loaded( 'src/llhood/lhood_canonical_uai01' ).
:- ensure_loaded( 'src/display_model' ).
:- ensure_loaded( '../auxil/create_graphs' ).
:- ensure_loaded( '../auxil/create_visits' ).
:- ensure_loaded( 'auxil/create_counts' ).
:- ensure_loaded( 'auxil/create_blankets' ).

data_format(data(_,_,_,_,_,_,_,_)). % needed by lhood.

% :- bb_put( sr, off ).

run_test :-
   Runs = [
      runs_file('simple'),
      print( [24,28,29] ),
      results_dir(test)
   ],
   run( Runs ).
```

Table 1: bns/run_test.pl

5.1 Backtracking proposals

As noted in (Angelopoulos & Cussens, 2001), a query over an SLP defines a stochastic SLD-tree. Each leaf of the tree corresponds to a model. The MCMC can then be viewed as a walk between leaves of the tree. The method by which a proposed model is reached from the current model defines a proposal distribution. A wide variety of proposals are supported, provided that some weak conditions are met. In the MCMC used here, we have experimented with a number of proposal distributions. Because all proposals over the SLD-tree are concerned with finding a backtrack point on the path of the current model, and then sampling forward from that point, we call these *backtracking proposals*. To make the process clearer consider the general scenario in Fig. 21. Given that the process is at M_i a proposal M_* needs to be reached. This is done by choosing one of the backtrack points (B_j) from the path defined by the root

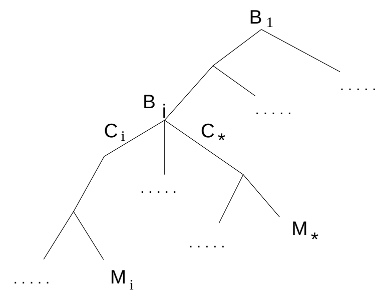

Figure 21: Reaching from M_i in the SLD-tree.

of the tree to M_i. Once such a point (B_i) has been chosen, M_* is reached by sampling forward according to the prior. In this section we detail the different backtracking proposals available in MCMCMS.

In (Angelopoulos & Cussens, 2001) a cyclic probability (CP) backtracking strategy was followed. The algorithm is parametrised by the expected depth *Depth* and its formulated as follows:

00. Let j := 1.

10. Let $p_b = 1 - 2^{-j}$

    20. Backtrack one step.

    25. If at the top of the tree then go to 2, otherwise with probability p_b backtrack one more point and repeat 1, or, with probability $1 - p_b$, go to 2.

    28. Sample for a new leaf M_* by selecting branches with probability proportional to clause labels. On the first choice, do not descend to the branch leading to M_i (branch C_i in Fig. 21).

30. Let $k := j + 1$. If $k > D$ then $j := 1$ else $j := k$

40. Unless iterations limit reached Goto 10.

A simpler version of CP uses a constant, non-cyclic, p_b. In the software this strategy is identified by $pb(PB)$.

One of the shortcomings of CP is that is dependant on M_i . For deterministic strategies, strategies that do not depend on M_i , the calculation of α, the acceptance probability, can be simplified. We have therefore also implemented a cyclic deterministic (CD) strategy. It is again parametrised by a number; the expected *Distance*. The algorithm is much simpler. It simply cycles j from 1 to *Distance*, increasing by one at each MCMC iteration. Backtracking chooses B_j, the choice point j levels up from the leaf M_i , at each proposal.

A third scheme (UC, for uniform choice) randomly picks with a uniform distribution between all choice points between M_i and the root.

Each of the three backtracking proposals can be chosen by selecting the appropriate term for the second argument of *run_data/11* (see Appendix B) which controls the corresponding experiment. $bp(Bp)$ is the backtracking probabilistically proposal with backtracking probability equal to Bp, $cp(D)$ is the cyclic probabilistic backtracking with $Depth = D$, $cd(D)$ is the deterministic strategy with $Distance = D$ and uc for the uniform choice strategy. We have tried to make the backtracking proposal as modular as possible. If you wish to implement a different proposal you should see file `src/bp.pl` .

6 Model spaces

We have constructed SLPs and likelihood functions for three model spaces: Bayesian networks (BNs), restricted acyclic partially directed graphs (RPDAGs) and classification and regression trees (C&RT)s. The following three subsections give details on how to use these programs.

6.1 BNs

BNs are well researched statistical models. In particular, functions for computing the likelihood of models have been known for some time. Therefore, they were chosen to be the first model space tackled in our framework. Our work relates to that of (Friedman & Koller, 2000). However, the benefit of using SLPs for defining the whole model space is that complicated and precise priors including hard constraints and statistically expressed preferences can be defined. This is an area with very little attention in the literature, where it is normal to assume a uniform, uninformative, prior.

BN related files are stored in `mcmcms/bns/` . The generator SLPs are in `slps/` and it is worthwhile noting that most of them have failure-free versions. This is a particularly important for the BP and CP proposals (Section 5.1) which were what we originally used with these SLPs. You can run queries `run_test.` and `run_disp.` from files `run_test.pl` and `run_disp.pl` for small demonstrations. The likelihood used was:

$$\prod_{j \in \mathbf{u}_i} \frac{\Gamma(\alpha_{ij})}{\Gamma(\alpha_{ij} + N_{ij})} \prod_{k \in \mathrm{Dom}(X_i)} \frac{\Gamma(\alpha_{ijk} + N_{ijk})}{\Gamma(\alpha_{ijk})} \tag{3}$$

and it can be found in `bns/src/llhood/lhood_canonical_uai01.pl` . (For details see (Angelopoulos & Cussens, 2001).) Various auxiliary programs for manipulating BNs and the generated output can be found in `bns/auxiliary/`.

6.1.1 Displaying BNs

This is not an essential part of MCMCMS but it is pretty nifty to be able to visualise all those BNs. You need to install `graphviz` http://www.research.att.com/sw/tools/graphviz/download.html then either add to `TCLLIBPATH` the location where Tcldot resides, or copy/move this file, somewhere where your `tcl` looks for lib files. If you are using latest graphviz (currently 1.10) you also need to copy file `$prefix/share/graphviz/demo/doted` to a directory in your `$PATH`. If you are using an older version, such as 1.7, make sure that the wish command in your (bash) `$PATH` is version 8.3 or older.

To display the current BN and the proposed one, use the `display(next_display_at)` *run/1* option. For example `run([display(next_display_at),id(2)])` . This will display the first pair and then ask for a number. Pressing return, or 1 and then return, will display the next pair as well. An integer larger than 1 means display nth step from here, 0 halts Prolog and a negative number $-N$ will skip to the Nth jump from the current position.

To display individual BNs, represented as Prolog terms use,

```
cd auxil; prolog
?-[disp_bn].
?-disp_asia.
```

which should display the ASIA BN. Nodes are colour coded. Red, for nodes with no parents, green for nodes with no children, but with parents, and orange for nodes with both children and parents. Other BNs can be displayed with `disp_bn(BN)` . See `auxil/disp_bns.pl` for the term representation of BNs.

To display bns sampled independently use
`?-['auxil/disp_sampled_bns.pl'].` For example
`?-disp_sbns(bn_un_2p,10,donot_stop,[a,b,c,d,e,f,g,h]).`
displays 10 bns without waiting for user interaction in the between. The arguments of *disp_sbns/4* are as follows:

SLP basename for SLP to be used. Looked for in `'.'` and `../slps/` . The `.slp` extention is not necessary.

HowMany number of bns to generate.

Stop set to `stop` for pausing, on user-interaction, after each generated BN.

Nodes the nodes for the generated BNs.

To compare a learned model against a generator BN (here ASIA is the assumed default generator) do

cd auxil ; prolog
?- [disp_cut_off].
eg (after you have run `run_test.` above)
?- disp_cut_off(0.95, '../test/tr_op0.8_or_3p_i1K_all_s1.counts').

The generic use is `?-disp_cut_off(CutOff,File)` . The first argument is in the range $0 \leq CutOff \leq 1$ which means that the 'learned' model is constructed by taking an edge to be in the model iff the $Times$ the edge appeared over the total number of $Iterations$ is greater that the $CutOff$ point, i.e. $Times/Iterations > CutOff$. The edges in the contrasting BN are coloured as follows

red for edges learned correctly

yellow for false negative (edges present in original only)

pink for false positive (edges present in learned only)

The second argument, $File$, should point to a .counts file.

6.2 RPDAGs

Programs dealing with Restricted Acyclic Partially Directed Graphs (RPDAGs) (Acid & de Campos, 2003) are in `bns/rpdags` since they share a lot of code with BNs. RPDAGs are a compromise between BNs and their equivalence classes. While learning, one ideally will like to learn in the space of equivalence classes since they represent a set of BNs whose structure explains the data equally well. However, it is computationally more expensive to move around in the space of equivalence classes. In response, Acid and de Campos (2003) have proposed RPDAGs. They have the property that many BNs from a single class map to a single RPDAG. It is though also the case, that more than one RPDAGs may map to a single equivalence class. Computationally, RPDAGs are harder to construct than BNs, but easier than Markov equivalence classes.

The SLP generator of all RPDAGs can be found in `rpdags/rpdag.slp` while a Prolog version can be found in `rpdag.pl` . These are based on the additive operations presented in (Acid & de Campos, 2003, p.467,Table 2). The number of RPDAGs according to our programs are shown in Table 2. Also in the table are shown the number of BNs, second row, and number of equivalence classes, in the last row. These are taken from (Gillispie & Perlman, 2001) Postscript files detailing all RPDAGs for three and four nodes can be found in directories `rpdags/doc/three.ps.gz` and `rpdags/doc/four.ps.gz` respectively.

Nodes	1	2	3	4	5	6	7
BNs	1	3	25	543	29281	3781518	1138762420
RPDAGs	1	2	13	258	13521	1713008	496236097
Eq.Class	1	2	11	185	8782	1067825	312510571

Table 2: Number of RPDAGs in relation to BNs and equivalence classes.

The likelihood used is an extension of the likelihood used in the case of BNs. Similarly, RPDAGs are displayed with the same routines as BNs. These have been extended to display unidirectional arcs as links of black colour. To create a small Markov chain you can use query `?-run_test.` defined in file `run_test.pl.` To visualise the models while creating the same chain use `?-run_disp.` from file `run_disp.pl` .

RPDAGs are represented by terms of the form: `rpdag(Nodes,Dag,BiD)`. $Nodes$ is a list of nodes to node information; a triplet containing the children the parents and the neighbours of the node. Neighbours are connected by bidirectional links. Dag is the Directed Acyclic Graph representing the directed part of the RPDAG. BiD is the graph containing the bidirectional links of the RPDAG. Both Dag and BiD are manipulated using the Prolog `ugraph` library.

6.3 C&RTs

Classification and Regression trees are binary decision trees that predict a response Y from a set of features X. If Y is qualitative, then the tree is a classification tree, whereas for quantitative Y we get a regression tree.

The programs implementing experiments over C&RTs are in `carts/`. We have implemented two priors based on work done independently by (Chipman H, 1998), (Denison H. A., 1998) and later by further developed in (Denison, Holmes, Mallick, & Smith, 2002). Three datasets from the two papers are included: The Wisconsin breast cancer data (BCW), the Kyphosis dataset (K) and the Pima Indians (PIMA) Diabetes dataset (ftp://ftp.ics.uci.edu/pub/machine-learning-databases/pima-indians-diabetes/). BCW contains 16 missing data values. Following (Chipman H, 1998) we have simply deleted datapoints which contain missing values. In both cases the machine learning task is binary classification using integer-valued predictors—nine predictors in the case of BCW, and three for K. All splits are binary: made by splitting on some threshold. There are 683 datapoints for BCW and 81 for K.

6.3.1 BCW and K

Experiments with BCW can be run with programs in `carts/bcw/` and with K in `carts/kyphosis`. For the former we used two different priors, the GROWTREE and EDITTREE priors. The GROWTREE prior can be found in `carts/bcw/grow/slps/unify.slp` while the EDITTREE prior is in `carts/bcw/edit/slps/edit.slp`. For K, a much smaller data set, only the GROWTREE is present.

The GROWTREE prior, is essentially that used by (Chipman H, 1998). The stochastic process which defines a GROWTREE prior grows a C&RT tree by starting with a single leaf node and then repeatedly splits each leaf node η with a probability $\alpha(1+d_\eta)^{-\beta}$, where d_η is the depth of node η and α and β are prior parameters set by the user to control the size of trees. Unsplit nodes become leaves of the tree. If a node is split, the splitting rule for that split is chosen uniformly. The source is in `carts/bcw/grow/slps/unify.slp`.

The other type of prior is the EDITTREE prior. Here an 'initial' C&RT model is supplied. This can be any tree, for example the tree produced by the standard greedy algorithm or one manually entered by the user. This tree is then probabilistically edited. This approach is closely related to the proposal mechanisms of (Chipman H, 1998; Denison H. A., 1998). The source is in file `carts/bcw/edit/slps/edit.slp` where, with probability 4/5 the tree is edited. There are three possible editions: growing the tree by splitting a leaf, pruning the tree by merging neighbouring leaves and changing a splitting rule. Our likelihood, file `carts/src/cart_llhood`, is implementing (Chipman H, 1998, Eq.16). C&RTs, are also displayed using *disp_bn/1*. Undirected links are drawn in black, with no arrows in either direction.

6.3.2 PIMA

The PIMA dataset was added recently and is still under development. It can be found in `carts/pima`. Canned examples are provided and the more advanced features can be found in directory `in_progress`. A variety of more advanced GROW like priors were devised for this dataset.

6.4 Pedigrees

We are currently working on constructing pedigree model structures. A pedigree is a special case of a BN where each node has exactly two parents. We have encoded an SLP approximation of the prior used in the Familias program (http://www.math.chalmers.se/ mostad/familias/) (Egeland, Pettera, Mevg, & Stenersen, 2000). Latest code is in `pedigrees/weighed` which includes samples from the approximating prior and some simple mcmc runs.

7 New model spaces

To run experiments over different model spaces one needs to: (a) write a generator SLP, (b) write a Prolog program that implements the likelihood function of a model given some input data, and (c) if desired, write programs for displaying the models on screen with graphics software.

You are strongly advised to create trivial scripts similar to `run_test.pl` and `run_disp.pl` as done in directories `bns/`, `bns/rpdags/`, `carts/bcw/grow` and `carts/kyphosis/`. During debugging you can add directive `:-bb_put(sr,off).` to the script. This will direct all output to the terminal. The current version of MCMCMS can be found with *mcmcms_version/1*.

7.1 Likelihood

To actually hook your likelihood function to MCMCMS you need to define

`lhood_canonical(+MODst,+MODi,-LLst,+LLi,-Ratio).`

and `model_llhood(+Model,-Llhood)`

Figs 6, 14 and 17 in the tutorial provide examples of definitions of these predicates. Variables prefixed by + stand for 'input' variables which are instantiated at the time of the call, while those prefixed by $-$ are 'output' variables which are not instantiated until exit time. $MODst$ is the proposed model, $MODi$ is the current model, $LLst$ is the log-likelihood of $MODst$, LLi is the log-likelihood of $MODi$ and *Ratio* is the likelihood ratio of $MODst$ over $MODi$.

For *model_likelihood/2*, $Model$ is a model, and *Llhood* is its log-likelihood. This predicate should always be defined but if it is not called from *lhood_canonical/5* then it can instantiate *Llhood* to any odd non-numeric value, e.g. `model_llhood(_,dont_know_it).`

7.2 Displaying models

This is not a necessary step for MCMCMS but it is used when visualisation of the current and proposed models is requested. You simply need to define `display_model(+Title,+Model,+Llhood)`.
Title is the title of the model, i.e. where it is the current or the proposal of iteration *i*. *Model* is the representation of the model to be displayed, and *Llhood* is its log-likelihood. When the predicate is entered all three arguments should be instantiated appropriately, with the possible exception of *Llhood*. This depends on whether *model_llhood/2*, see preceeding sub-Section, instantiated the likelihood correctly. Your program should be aware and take appropriate action if that is not the case (e.g. not display the log-likelihood).

The main route for displaying models is via the *run/1* option. This provides a hookable mechanism, the details of which can be found in the sources: `mcmcms/run.pl` . Here we document an implemented hook invoked by `display(next_display_at)` . This always prints the first pair of models (current and proposed) and interrogates the user for when the next display should be. The number 1 and any non integer input will result in the display of the next pair of models, while 0 will halt the execution. A positive integer will display the Nth comparison from the current position. A negative integer will skip the Nth jump from the current position.

7.3 Writing SLPs

Strictly non-stochastic predicates are un-labelled (non-stochastic) predicates that only contain strictly non-probabilistic predicates in their bodies. When writing an SLP that contains such predicates it might be more efficient and cleaner if they are placed in a separate source file. The source file can either be compiled as a module file if you are using the SICStus Prolog system, or loaded without translation (on any of the supported engines). In the former scenario, the module file can be compiled in SICStus and then saved with : `save_modules([Mod],<FILENAME>_npbc)` where *Mod* is the module name. In the latter case, directive `:-els(NonPbcSrc)` can be used to load source file when SLP translation is to be avoided.

The best way to find out how to run experiments are by looking at the experiment scripts in the distribution. For example `pbl/` contains scripts for experimental results which have been published. In scripts that load run.pl or mcmcms.pl, you can use directive `:-elm(RelPos).` This directive loads file *RelPos* whose position is taken relative to the top MCMCMS directory.

8 Other features

Here we briefly review some features of the system which are still to be documented fully.

8.1 Sampling

There are facilities for sampling models from an SLP. Furthermore the output can be displayed in postscript format, and a latex report can be generated. Sampling is not a part of the MCMC algorithms but it can be very useful in providing feedback for newly developed SLPs: in particular, about their digestibility by the software and the distributions they define. For the time being, and until we expand on this section, please look at the source files: `auxil/n_samples.pl` and `auxil/n_samples_tex`

A round-about way to sampling is to run MCMC with a constant ratio of likelihood that is equal to something substantially larger than 1. In this way, the chain will almost certainly consist of the proposed models. (i.e. the proposed model is always accepted.) Such a likelihood can be found in `auxil/lhood_constant.pl`. An example of this technique is in `auxil/from_prior_ex/srun.pl`.

8.2 Global Priors Ratio

It is possible to view the prior of a model, $P(M)$, as consisting of two parts: global and local. The local is the probability of the branch of computation that leading to M and the global is an factor that can be computed after producing the model. As idicated by the name, we want to use the global part in computing the contribution of global features of the model to the prior. An example of the importance of such global features can be found in the prior used by the Familias program (Egeland et al., 2000). MCMCMSwill look into memory for the predicate *global_priors_ratio/3*. If this is present it will be called each time the acceptance probability is computed. The call is: $global\_priors\_ratio(+M_*, +M_i, -Ratio)$. The user-defined predicate is expected to return $Ratio = \frac{P_G(M_*)}{P_G(M_i)}$.

8.3 Tempering

A number of hot chains that are developed in parallel to the main cold chain can be run. Hot chains move on smoother likelihood spaces, so getting less stuck. At each iteration two chains are chosen for a possible swap of their, next-iteration, current models.

Power tempering is the only kind of tempering supported. In the future other kinds might be possible. In power tempering the space is smoothed by raising the likelihood to a power. For hot chain j we sample from stationary distribution $\pi_{\beta_j}(x) \propto \pi_0(x)L(x)^{\beta_j}$ for $0 < \beta_j < 1$. Proposals and acceptances within this chain work exactly as for the cold chain. The only difference is that there is a different likelihood. So, if at model M_i^j we propose model M_*^j then:

$$\alpha_j(M_i^j, M_*^j) = \min\left\{1, \kappa(M_i^j, M_*^j)\left(\frac{L(M_*^j)}{L(M_i^j)}\right)^{\beta_j}\right\}$$

where $\kappa(M_i^j, M_*^j)$ is the combined prior and proposal conttribution.

The chain-swap acceptance probability for moves *between* chains does not depend on the proposal distribution which generates moves *within* chains. So the acceptance probability is the same for all within-chain proposals, namely:

$$\alpha(M_\alpha^{j_1}, M_\alpha^{j_2}) = \min\left\{1, \left[\frac{L(M_\alpha^{j_2})}{L(M_\alpha^{j_1})}\right]^{(\beta_1-\beta_2)}\right\}$$

where $M_\alpha^{j_1}$ is the accepted within-chain model in chain j_1.

An example run-script using power tempering can be found in `carts/pima/run_hot.pl`.

A Options for *run/1*

The following are all optional. If an option term is not present the indicated default value is used.

copy(Copy) copies some of the files used to the results directory. The idea is that results directories can be made self explanatory. `Copy` should be one of 'none' or 'all' or it should be a list of : 'data','runfile','script'(Scr) and 'slp'. The default value is [runfile]. Note that you should substitute the scriptname used in your run of experiments, for `Scr`.

display(DMethod) method controlling display of models. Use 'next_display_at'. If option is absent no models will be displayed.

id(ID) a number of instances of this option are allowed. Each ID should correspond to the first argument of a *run_data/11* clause. If absent, then all experiments described by *run_data/11* will be pursued.

initial(M) registers initial model, M. If it is not present the initial model is sampled from the prior. This option is intended for advanced users only. You need to be very careful that your SLP will be able to reach efficiently to the/a proof of the model. (`mcmcms/carts/bcw/edit/irun.pl` demonstrates the option's usage.)

print(PointsList) points to be recorded to output files. If absent the default value of [24,28,29] is used. Other numbers are undocumented. If more detail is needed, please look at the source files for *dbg/2*. Point 24 prints the initial model, point 28 prints all subsequent visited models in the chain while 29 prints proposed models. The latter is printed in the form of `b(Model)` terms, while current models are deposited as `c(Model)`. On how $Models$ are extracted from the top query, see Appendix B, 5th argument ($Call$).

progress(Atom,Percentile) if present, a progress bar will be displayed for every experiment within the run. *Atom* will be written once for every completed *Percentile* of the number of repetitions in each experiment.

results_dir(ResDir) is the directory for the output files from the experiments. Currently if the directory exists, there will be no complaints, and no clean-up. So unless you know what you are doing it is better if the directory does not exist. If *ResDir* is a path, e.g. of the form `dir1/sub1/test`, which does not exist, then it will be created. Default value: 'test'.

runs_file(RFile) file containing the *run_data/11* definitions. Default: 'run_test'.

source_dir(SDir) a directory relative to which `runs/` `slps/` and `data/` are looked in for. Default value: '.'.

B Arguments for *run_data/11*

1:ID unique id for the experiment. Usually a small positive integer.

2:BP backtracking proposal to be used. One of

bp(P) probabilistic backtracking, with fixed probability; as presented in (Angelopoulos & Cussens, 2001).

cp(D) cyclic probabilistic backtracking, of focus D; as presented in (Angelopoulos & Cussens, 2001).

cd(S) cyclic deterministic backtracking; cycling k from 1 to S. Where $k = 1$ is the root of the SLD-tree.

uc uniformly chosen backtracking, which chooses amongst the available points.

3:SLP the location of the source file for the SLP to be used. Extension .slp is not necessary and files will be looked for at current and `./slps/` directories.

4:MSD model space identifier. Can either be set to 'rm' or to 'fm' (see Appendix C). The former identifies real model spaces- upon failure to reach a model usual Prolog backtracking is employed to explore alternative choices, in a exhaustive shallow-first search. The latter allows failure models to be returned at top level of the MCMC. When this is the case, there will be no attempt to jump to the proposed, imaginary, model. Instead, the count of staying with current model will increase by one.

5:DataCall a callable term. This argument performs two roles. Firstly, file $DataName\{.pl\}$, the predicate name of `DataCall`, is compiled. Secondly, `DataCall` is called as a Prolog goal before the main MCMC sampling begins. It is therefore, a 'hook' which can be useful for any needed pre-processing. `DataCall` is normally defined in $DataName\{.pl\}$ and it should at least succeed. It may have shared arguments with $6 : Query$; so data may provide parameters of the space of all possible models (e.g. C&RT in directory `cart/`). Optionally, 'clean_up_'*DataCall* may also be defined in $DataName\{.pl\}$. This is only necessary when a single run experiment uses more than one data set. See file `bns/data/jc_asia_data_dyn_all.pl` for an example.

6:Query a $Name/Args$ structure, where $Name$ is a predicate symbol and $Args$ is a list. If L is the length of the list $Args$ then *Name/(L+1)* is the top-level predicate for the chain and the predicate should be defined in SLP. The last argument of calls to *Name/(L+1)* will be a fresh variable. The software expects the this variable to be instantiated to a model each time the call to *Name/(L+1)* succeeds. The first L arguments in calls to *Name/(L+1)* will be those supplied in the list $Args$.

7:Contr an atom that will be added to the stem filename of the output files created from this experiment.

8:Repeats a large integer, stating the length of the Markov chain.

9:HotChnsIds a list of hot chain identifiers. For power tampering, which is the only kind currently supported, each identifier should be a power β_j with $0 < \beta_j < 1$. See Section 8.3.

10:RndSeedIdx a random seeds index: an integer between 1 and 1000, see file `auxil/seeds.pl` .

11:Exts the extensions to be created from the main output file. This can be used to restrict the files generated from applications of *exec_extension/2*. The safer value is 'all'. Alternatively a list of extensions matching the first argument of defined *exec_extension/2* clauses can be given. The empty list, and equivalently 'none' are also acceptable values which create no extention files.

C Failure Model (fm) universe vs. Real Model (rm) universe

Some notes on comparing two different 'backtracking' strategies. These, are strategies definiting the model space distributions, as oppose to the proposal backtracking.

The two alternatives examined are: real model space (RM) and failure model space (FM). The former uses native backtracking to always reach a real model. Every time a label is chosen from a list of labels corresponding to matching to unified clauses, a backtrack point is created. Upon failure, the remaining of the labels are tried. (Probabilistictly chosen among the untried ones). The latter does not create such backtrack points. In the event of a failure, the entire call should fail. This is caught by the top level, and the special model '__failure' is outputed. The MCMC will always chose not to jump in this case.

In passing lets note that non-probabilistic predicates need to be (near) deterministic for this to work. To allow more flexibility a special token can be generated instead of failure. This is similar to EM software, but it means that the transformed SLP clauses have to be aware of this special token.

We have tested RM and FM by sampling N ($N = 10000$) independent samples from the prior described in CICLOPS04 ((Angelopoulos & Cussens, 2004)), and by running N iterations of the MCMC with a constant likelihood. An extra argument has been added to $run_data/8$ which should be set to 'rm' or 'fm'.

C.0.1 FM- sampling

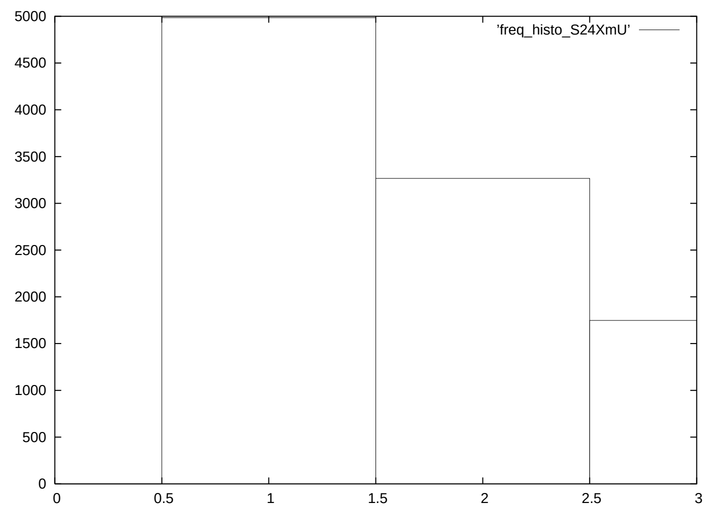

Figure 22: Using fm to draw 10000 samples from ciclops 04 prior- observing X after calling p(X).

The frequencies in detail: 'a-4985' 'c-3267' 'failure-1748'

C.0.2 FM- MCMC

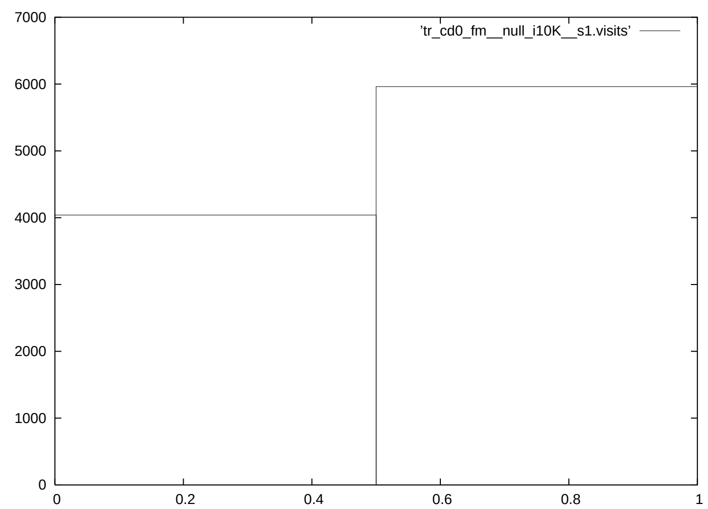

Figure 23: Using FM to do MCMC 10000 iterations from ciclops 04 prior- observing X after calling p(X).

C.0.3 RM- sampling

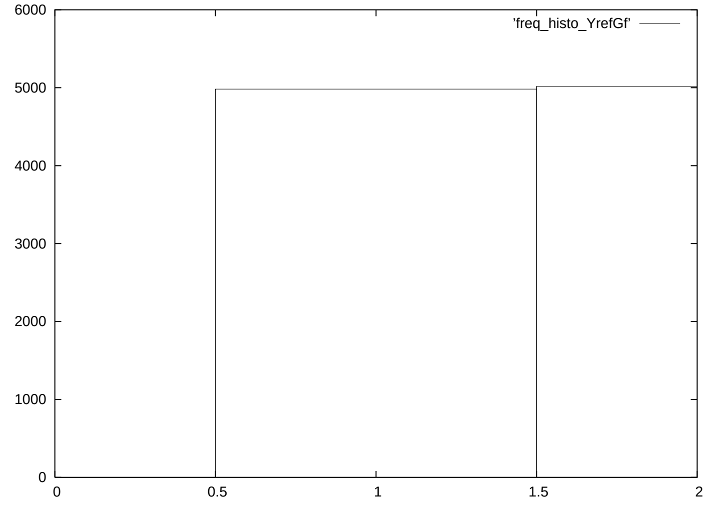

Figure 24: Using fm to draw 10000 samples from ciclops 04 prior- observing X after calling p(X).

The frequencies in detail: 'a-4982 ' 'c-5018 '

C.0.4 RM- MCMC

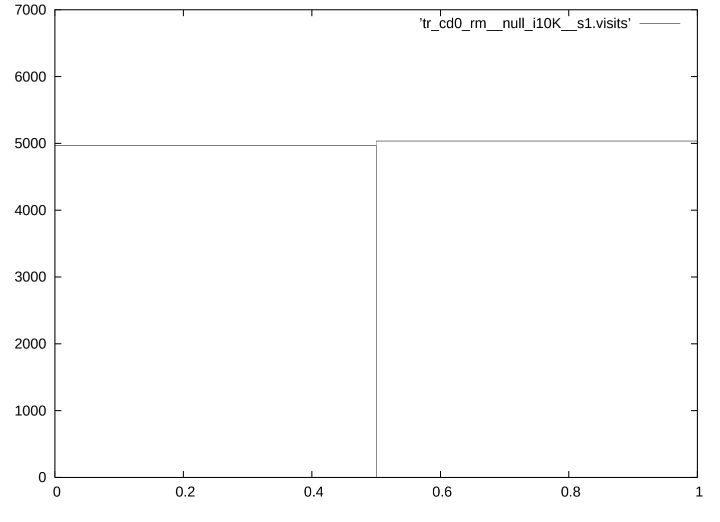

Figure 25: Using RM to do MCMC 10000 iterations from ciclops 04 prior- observing X after calling p(X).

C.1 SLP Program

```
1/2 :: p( a ).
1/2 :: p( X ) :- q(X), t(X).

1/3 :: q( b ).
2/3 :: q( c ).

1 :: t(c).

do_trace( b ) :- trace.
do_trace( c ) :- true.
```

C.2 Transformed Program

```
:- module( slp, [] ).
:- compile(library('../runtime_fm')).
do_trace(1, [1|A]/B, [1|C]/D, b) :-
        user:trace,
        A=B,
        C=D.
do_trace(2, [2|A]/A, [2|B]/B, c).
p([A|B]/C, [D|E]/F, G) :-
        select_id([[a],[H]], [G], [0.5,0.5], 3, A, B, D, I, J),
        (   I=:=3 ->
            [a]=[G],
            user:true,
            J=C,
            E=F
        ;   [H]=[G],
            q(J/K, E/L, H),
            t(K/C, L/F, H)
        ).
q([A|B]/C, [D|E]/F, G) :-
        select_id([[b],[c]], [G], [0.3333333333333333,0.6666666666666666], 5,
                A, B, D, H, I),
        (   H=:=5 ->
            [b]=[G],
            user:true,
            I=C,
            E=F
        ;   [c]=[G],
            user:true,
            I=C,
            E=F
```

```
        ).
t([A|B]/C, [D|E]/F, G) :-
        select_id([[c]], [G], [1], 7, A, B, D, _, H),
        [c]=[G],
        user:true,
        H=C,
        E=F.
```

References

Acid, S., & de Campos, L. M. (2003). Searching for Bayesian network structures in the space of restricted acyclic partially directed graphs. *Journal of Artificial Intelligence Research*, *18*, 445–490.

Angelopoulos, N., & Cussens, J. (2001). Markov chain Monte Carlo using tree-based priors on model structure. In Breese, J., & Koller, D. (Eds.), *Uncertainty in Artificial Intelligence: Proceedings of the Seventeenth Conference (UAI-2001)*, pp. 16–23 Seattle, USA. Morgan Kaufmann.

Angelopoulos, N., & Cussens, J. (2004). On the implementation of MCMC proposals over stochastic logic programs. In *Colloquium on Implementation of Constraint and LOgic Programming Systems. Satellite workshop to ICLP'04* Saint-Malo, France.

Angelopoulos, N., & Cussens, J. (2005). Exploiting informative priors for bayesian classification and regression trees. In *Nineteenth International Joint Conference on Artificial Intelligence* Edinburgh, UK. To appear.

Chipman H, George E, M. R. (1998). Bayesian CART model search (with discussion). *Journal of the American Statistical Association*, *93*, 935–960.

Cussens, J. (2000). Stochastic logic programs: Sampling, inference and applications. In *Sixteenth Annual Conference on Uncertainty in Artificial Intelligence (UAI-2000)*, pp. 115–122 San Francisco, CA. Morgan Kaufmann.

Cussens, J. (2001). Parameter estimation in stochastic logic programs. *Machine Learning*, *44*(3), 245–271.

Denison, D. G. T., Holmes, C. C., Mallick, B. K., & Smith, A. F. M. (2002). *Bayesian Methods for Nonlinear Classification and Regression*. Wiley.

Denison H. A., Mallick B. K., S. A. F. M. (1998). A Bayesian CART algorithm. *Biometrika*, *85*, 363–377.

Egeland, T., Pettera, M., Mevg, B., & Stenersen, M. (2000). Beyond traditional paternity and identification cases. selecting the most probable pedigree. *Forensic Science International*, *110*(1).

Friedman, N., & Koller, D. (2000). Being Bayesian about network structure. In *Uncertainty in Artificial Intelligence: Proceedings of the Sixteenth Conference (UAI-2000)*, pp. 201–210 San Francisco, CA. Morgan Kaufmann Publishers.

Gillispie, S. B., & Perlman, M. D. (2001). Enumerating markov equivalence classes of acyclic digraph models. In *UAI 2001*.