AIC-PRAiSE User Guide

Rodrigo de Salvo Braz
Artificial Intelligence Center
SRI International

June 3, 2015

Contents

1 Introduction

AIC-PRAiSE is SRI's Artificial Intelligence Center system for Probabilistic Reasoning as Symbolic Evaluation. It takes the specification of a probabilistic model in a relational form, Higher-order Graphical Models (HOGM), and computes the marginal probability of a given query.

HOGMs is a formalism similar to, but much more expressive than, Markov Logic Networks (MLNs, [RD06]), although the current implementation of it in AIC-PRAiSE is only slightly more expressive. It is described in Section 3.1.

AIC-PRAiSE performs *lifted* inference, that is, inference performed on the relational, intensional, representation level directly, rather than on the propositional level. It distinguishes itself from other lifted inference systems by doing so through *symbolic evaluation*. At this point, the most visible effect of this approach is the fact that queries may contain unbound logical variables that get partially bound, in a way similar to how Prolog queries are answered.

2 Installing AIC-PRAiSE

AIC-PRAiSE can be run by downloading and running demos as `jar` files on `https://github.com/aic-sri-international/aic-praise/`. This requires a Java runtime environment (JRE) to be installed, which can be obtained at `http://java.com`. Instructions for setting up a development environment are more involved and can be found at the same page.

3 Using AIC-PRAiSE

Using AIC-PRAiSE can be done in four possible ways: through a demo or API using Lifted Belief Propagation, and through a demo or API using SGDPLL(T), a DPLL-based algorithm.

Both demos operate on probabilistic models specified as Higher-Order Graphical Models (HOGMs).

The first and older demo, detailed in Section 3.2, applies the Lifted Belief Propagation algorithm, which is slower, approximate, and retricted to Boolean and Boolean function-valued random variables, but is able to process relational models (that is, models with random functions).

The second demo, detailed in 3.3, applies the Symbolic Generalized DPLL (SGDPLL(T)) algorithm, which is faster, exact, and not restricted to Boolean random variables, but (so far) not able to process relational models. In time, the project will focus on the second demo, extended with an implementation of SGDPLL(T) able to process relational models.

Both demos have a mode in which Church programs ([GMR$^+$12]) are converted to our language for HOGMs, thus providing lifted inference for Church programs. Each demo is restricted to a fragment of Church corresponding to its own fragment of HOGM. For example, the LBP demo can process Church programs with random functions, whereas the SGDPLL(T) demo can deal with non-Boolean types and provides exact results.

Alternatively, one can use AIC-PRAiSE through the corresponding Java APIs, explained in Sections 3.4 and 3.5.

The most important aspect of using AIC-PRAiSE is the language it uses to describe HOGMs, detailed in the next Section, 3.1.

3.1 Higher-Order Graphical Models (HOGMs)

3.1.1 Undirected graphical models

We assume the reader's familiarity with graphical models. This section highlights the concepts we rely on and establishes some notation, so that their relation to Higher-Order Graphical Models (HOGMs) is clearer.

An **undirected graphical model** consists of a set of random variables $X_1, \dots, X_n$ and a set of **factors** F defined on them. Each fator $f\ inF$ is associated to a tuple of random variables X_f (called its **neighbors** and a potential function $\phi_f : \mathcal{D}(X_f) \to \mathbb{R}_0^+$, where $\mathcal{D}$ is a function mapping random variables to their ranges. The undirected graphical model defines a joint probability distribution on $X_1, \dots, X_n$ equal to

undirected graphical model

factors

neighbors

$$P(x_1, \dots, x_n) = \frac{1}{Z} \prod_{f \in F} \phi_f(x_f)$$

where $x_1, \dots, x_n, x_f$ represent the values of the respective (tuples of) random variables, and Z is a normalization constant defined as

$$Z = \sum_{x_1,\dots,x_n} \prod_{f \in F} \phi_f(x_f)$$

Intuitively, each factor represents a dependence between its neighbors. Suppose we wish to represent a dependence between the boolean random variables $Rain$ and Wet such that the probability of their being equal is twice as large as that of their being distinct. A factor f with potential function

$$\phi_f(rain, wet) = \begin{cases} 2 & \text{if } rain = wet \\ 1 & \text{otherwise.} \end{cases}$$

represents this dependence.

The same model could have another factor g on $Rain$ alone representing the fact that it is four times more likely to rain than not:

$$\phi_g(rain) = \begin{cases} 4 & \text{if } rain \\ 1 & \text{otherwise.} \end{cases}$$

The normalized product of f and g defines a joint probability distribution combining these two dependences.

A potential zero indicates a situation that is impossible (a hard constraint).

$$\phi_h(day, menu) = \begin{cases} 0 & \text{if } day = \text{Friday and } menu = \text{beef} \\ 1 & \text{otherwise.} \end{cases}$$

indicates that it is never the case that if day = Friday and $menu$ = beef, while all other combinations are equally likely.

In HOGMs we use often use potentials between 0 and 1, but doing so does not limit their expressivity because, since the product of potentials is normalized, multiplying a potential function by any positive constant does not alter the final joint probability. It's the *relative proportion* between a potential function's entries that matters.

We can represent a Bayesian network as an undirected graphical model by considering each conditional probability table (CPT) in it as a factor. This means that the potentials in an undirected model corresponding to a Bayesian network can be interpreted as conditional probabilities. This will be the case for some of our HOGMs examples.

3.1.2 The AIC-PRAiSE language for Higher-Order Graphical Models (HOGMs)

Higher-Order Graphical Models (HOGMs) is a formal language for specifying graphical models based on a Higher-Order Logic formalism. Its main feature is the ability to specify potential functions succintly over entire classes of random variables, using an expressive language that includes interpreted functions such as arithmetic and list-manipulation functions.

Please note that the HOGMs version presented here is somewhat different and more limited from the full HOGMs language presented elsewhere due to the development stage of AIC-PRAiSE.

Let us consider several examples of HOGMs, from simple to more complex, without concerning ourselves with formal definitions for now.

We start with

```
random rainy: Boolean;

rainy 0.3;
```

The last statement of the model declares a potential function defined on the Boolean random variable `rainy` equal to 0.3 for when `rainy` is true, and 0.7 for when it is false. The 0.7 values comes from the always-present implicit assumption that potential p on a formula is complemented by a potential $1 - p$ on its negation.

The type declaration `random rainy: -> Boolean` states that `rainy` is a Boolean random variable without arguments (propositional).

A potential can be defined not only on a Boolean random variable, but more generally on a Boolean formula:

```
random rainy: Boolean;
random holiday: Boolean;

rainy and holiday 0.3;
```

The last statement declares a potential function on Boolean random variables `rainy` and `holiday` equal to 0.3 for when their conjunction is true, and 0.7 otherwise.

We can use if-then-else expressions (*conditional expressions*) to define more complex potential functions:

```
// Sorts:
random earthquake: Boolean;
random burglary: Boolean;
random alarm: Boolean;

// Rules:

earthquake 0.01;
burglary 0.1;

if earthquake
then if burglary
then alarm 0.95
else alarm 0.6
else if burglary
then alarm 0.9
else alarm 0.01;
```

The above states that $P(\texttt{alarm}|\texttt{earthquake}, \texttt{burglary}) = 0.95$, $P(\texttt{alarm}|\texttt{earthquake}, \texttt{not burglary}) = 0.6$, and so on. (Also, note that `//` introduces the rest of the line as comments as in Java and C++.)

A **default certainty potential** 1 is assumed for formulas `F` without a number; in the following, `rainy and cold` is defined to be true with certainty:

default certainty potential 1

```
random rainy: Boolean;
random cold: Boolean;

rainy and cold;
```

These certainty statements cab be used to declare observed evidence, besides rules involving certainty.

So far, we have only used **propositional** random variables, but HOGMs also supports **random functions** (often referred in the literature as parameterized random variables). The following example illustrates how they are used:

propositional

random functions

```
sort People: 100, bob, dave, rodrigo, ciaran;

random epidemic: Boolean;
random sick: People -> Boolean;
random fever: People -> Boolean;
random rash: People -> Boolean;
random itch: People -> Boolean;

// RULES
// By default, how likely is an epidemic?
epidemic 0.001;

if epidemic then sick(X) 0.6 else sick(X) 0.05;
if sick(X) then fever(X) 0.7 else fever(X) 0.01;
if sick(X) then rash(X) 0.6 else rash(X) 0.07;
if rash(X) then itch(X) 0.9 else itch(X) 0.01;
```

The above HOGM shows the use of *sort* declarations, which includes the domain size and, optionally, some of its named constants. The sort is then used as a domain for random functions. These domains can be Cartesian products, as in the following example:

```
sort People: 10, ann, bob, dave, rodrigo, ciaran;

random friends: People x People -> Boolean;

friends(X,Y) <=> friends(Y,X);
not friends(X,X);

friends(bob, dave);
```

The capitalized symbols in the rules used as arguments to random functions are *logical variables* and are implicitly universally quantified, much like the conventions of Prolog. A rule with logical variables stands for all its instantiations obtained for each possible assigment to its logical variables. This means that the above model is equivalent to

```
sort People: ann, bob, dave, rodrigo, ciaran, person6,
person7, person8, person9, person10;

random friends: People x People -> Boolean;

friends(ann,bob) <=> friends(bob,ann);
friends(ann,dave) <=> friends(dave,ann);
// ...
```

```
friends(bob,ann) <=> friends(ann,bob);
friends(bob,dave) <=> friends(dave,bob);
// ...
friends(person9,person10) <=> friends(person10,person9);

not friends(ann, ann);
not friends(bob, bob);
// ...
not friends(person10, person10);
```

where `// ...` indicates sections that would contain the expected rules.

Needless to say, the relational form is much more succint.

Note also that we assume that distinct capitalized symbols in the same statement are assumed to have distinct values. This means that

```
friends(X, Y) 0.8;
```

is equivalent to

```
if X != Y then friends(X, Y) 0.8;
```

If one wishes to cancel out this assumption, one must write:

```
friends(X, Y) and Y may be same as X 0.8;
```

3.2 Lifted Belief Propagation PRAiSE Demo (LBP-PRAiSE)

The first PRAiSE demo is the Lifted Belief Propagation PRAiSE (LBP-PRAiSE) demo. This demo applies the Lifted Belief Propagation algorithm to HOGMs. It is able to process relational models, but is an approximate algorithm.

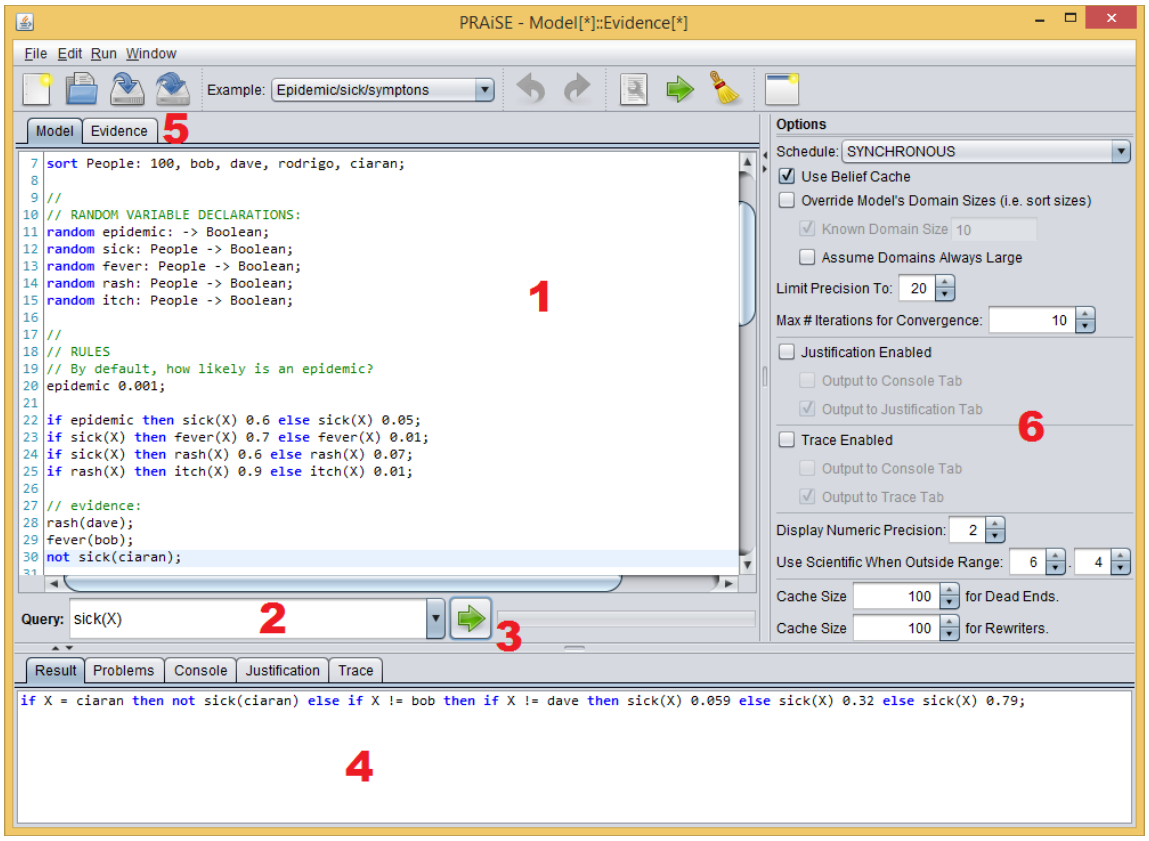
Once started, LBP-PRAiSE shows the window depicted and explained in Fig. 1.

Once a model is specified, the user can pose queries about it to the system. This is done by entering a random variable or a formula in the query box and pressing Enter, or clicking on the arrow button. The result is then shown in the "Result" tab below the query box in the form of a rule. For example, querying the epidemic model defined above with `sick(dave)` provides a result `sick(dave) 0.051;`, meaning that the marginal probability of that random variable for the model is 0.051.

The random variable or formula can contain unbound logical formulas, in which case the answer is provided conditioned on relevant possible values for these variables. If we enter observed evidence into the model by adding a new rule

```
rash(bob);
```

and we query `sick(X)`, we obtain the result



```
sort People: 100, bob, dave, rodrigo, ciaran;

//
// RANDOM VARIABLE DECLARATIONS:
random epidemic: -> Boolean;
random sick: People -> Boolean;
random fever: People -> Boolean;
random rash: People -> Boolean;
random itch: People -> Boolean;

//
// RULES
// By default, how likely is an epidemic?
epidemic 0.001;

if epidemic then sick(X) 0.6 else sick(X) 0.05;
if sick(X) then fever(X) 0.7 else fever(X) 0.01;
if sick(X) then rash(X) 0.6 else rash(X) 0.07;
if rash(X) then itch(X) 0.9 else itch(X) 0.01;

// evidence:
rash(dave);
fever(bob);
not sick(ciaran);
```

Query: sick(X)

```
if X = ciaran then not sick(ciaran) else if X != bob then if X != dave then sick(X) 0.059 else sick(X) 0.32 else sick(X) 0.79;
```

Options
Schedule: SYNCHRONOUS
[✓] Use Belief Cache
[ ] Override Model's Domain Sizes (i.e. sort sizes)
[✓] Known Domain Size 10
[ ] Assume Domains Always Large
Limit Precision To: 20
Max # Iterations for Convergence: 10
[ ] Justification Enabled
[ ] Output to Console Tab
[✓] Output to Justification Tab
[ ] Trace Enabled
[ ] Output to Console Tab
[✓] Output to Trace Tab
Display Numeric Precision: 2
Use Scientific When Outside Range: 6 . 4
Cache Size 100 for Dead Ends.
Cache Size 100 for Rewriters.

Figure 1:
A screenshot of the LBP-PRAiSE demo. It consists of the following major components:

1. The model editing area allows the definition of a HOGMs model in the PRAiSE syntax.
2. The query box accepts a formula (and in particular a random variable) of which one wishes the marginal probability of. Capitalized symbols stand for unbound logical variables.
3. The query execution button allows for starting and stopping of inference.
4. The result box shows the computed answer. Other tabs in the box show problems, console output, and debugging information (justification and trace).
5. An extra "Evidence" tab allows for entering evidence separately from the model. This is optional because evidence is treated as extra rules that can be entered directly in the model editing area.
6. A configuration panel (initially minimized to the right) provides some algorithm, formatting and debugging options.

```
if X != bob then sick(X) 0.052 else sick(X) 0.31;
```

This indicates a marginal probability of 0.052 for `sick(X)` for all values of `X` not equal to `bob`, and 0.31 for when `X` is equal to `bob`. Again, this is similar to the partial binding by Prolog of unbound variables present in a query. (Incidentally, the reader might be asking herself why the probability of `sick(X)` for `X` not equal to `bob` has become slightly higher after this evidence; the answer is that `rash(bob)` makes `epidemic` slightly more likely and it in turn makes others being sick more likely as well.)

Currently, the algorithm used for answering queries is Lifted Belief Propagation. In the presence of cycles, this algorithm only provides an approximate answer. Note also that the potentials in the rules may or may not correspond to probabilities depending on the model structure, in the same way that weights or potentials in Markov networks do not necessarily correspond to probabilities.

The right panel of the LBP-PRAiSE demo allows certain configurations to be set. Just like its regular counterpart, Lifted Belief Propagation can follow a synchronous or asynchronous schedule, and this option is available as a configuration setting (the asynchronous schedule is implemented in two different ways, group or non-group, the former being a lifted interpretation of it). The configurations also include the maximum number of iterations before the algorithm stops in the case convergence does not happen sooner. Other configuration settings involve numeric precision, debugging information, and cache sizes.

3.2.1 Performing inference on Church Programs

The LBP-PRAiSE demo offers a Church perspective (under the Window-¿Open Perspective menu), which allows Church programs to be entered and reasoned about. It also exhibits the translation from Church to HOGMs.

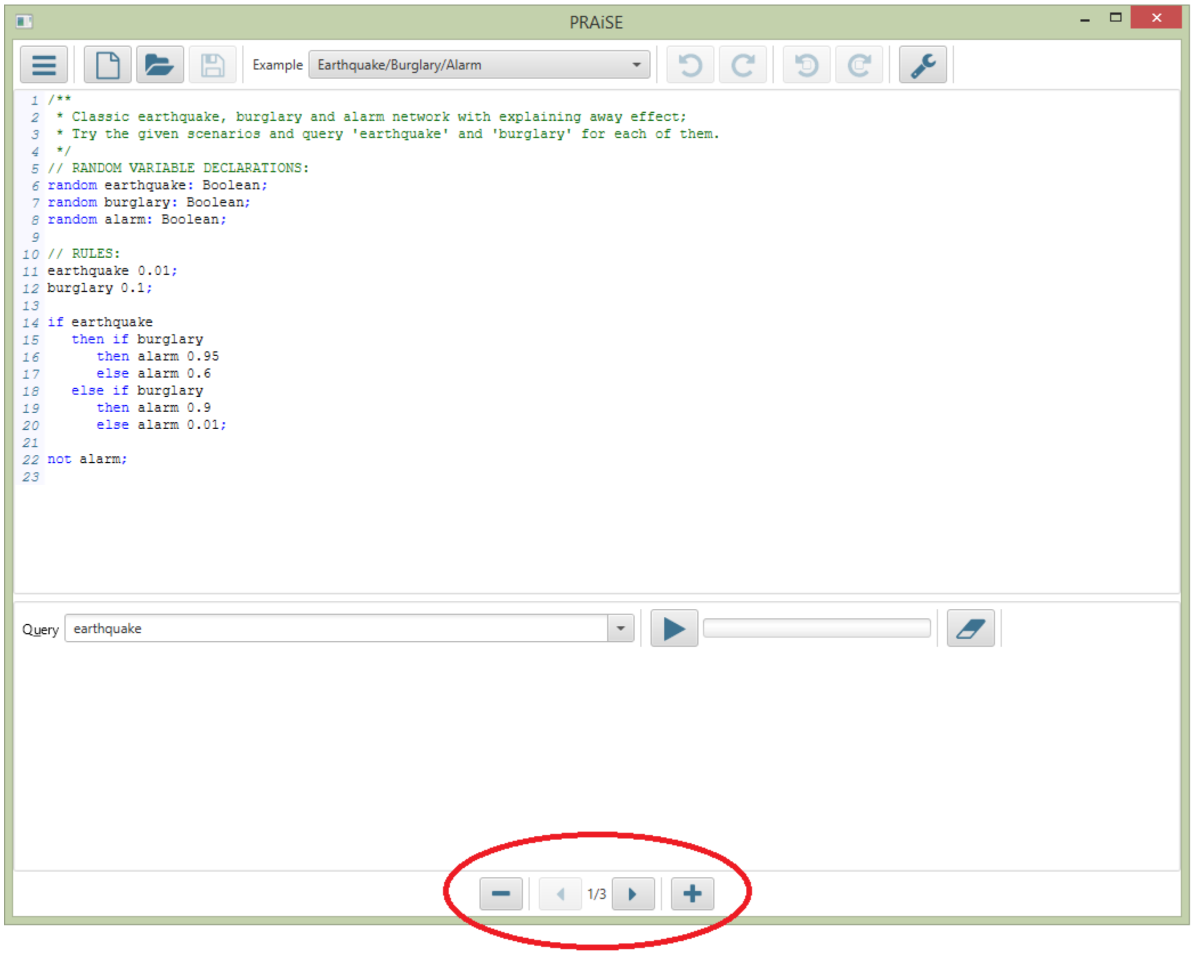
3.3 SGDPLL(T) PRAiSE Demo (SGDPLL(T)-PRAiSE)

The SGDPLL(T)-PRAiSE demo allows users to run inference on HOGMs using inference based on the SGDPLL(T) algorithm. A screenshot of the demo is shown in Fig. 2.

As of June 2015, this demo only works for models without *random* functions. It is however faster, and produces exact results.

3.3.1 Performing inference on Church Programs

The SGDPLL(T)-PRAiSE demo offers a Church perspective (under the left-corner menu button), which allows Church programs to be entered and reasoned about. It also exhibits the translation from Church to HOGMs. Because this demo currently does not process relational models, the Church programs entered cannot contain random functions, but only random variables.



```
/**
 * Classic earthquake, burglary and alarm network with explaining away effect;
 * Try the given scenarios and query 'earthquake' and 'burglary' for each of them.
 */
// RANDOM VARIABLE DECLARATIONS:
random earthquake: Boolean;
random burglary: Boolean;
random alarm: Boolean;

// RULES:
earthquake 0.01;
burglary 0.1;

if earthquake
   then if burglary
      then alarm 0.95
      else alarm 0.6
   else if burglary
      then alarm 0.9
      else alarm 0.01;

not alarm;

```

Figure 2:
A screenshot of the SGDPLL(T)-PRAiSE demo app. It follows a similar layout to that of the LBP-PRAiSE demo. The main difference is that this demo allows multi-page files. The region marked in red contains controls for changing and adding pages to the file.

3.4 Queries via Lifted Belief Propagation API

The Lifted Belief Propagation algorithm can also be used as a Java library, providing query results directly to client code, as shown in the following example (present in the `LBPTest.java` file):

```
Expression query = parse("friends(dave, X)");
Model model = Model.fromRules(
    "sort People: 10, ann, bob, dave, rodrigo, ciaran;" +
    "random friends: People x People -> Boolean;" +
    "friends(X,Y) <=> friends(Y,X);" +
    "not friends(X,X);" +
    "friends(bob, dave);");
```

```
Solver solver = new LiftedBeliefPropagationSolver(true);
actual = solver.marginal(query, model);
```

3.5 Queries via SGDPLL(T)-based API

The SGDPLL(T) algorithm can also be used as a Java library, providing query results directly to client code, as shown in the following example (present in the `InferenceForFactorGraphAndEvidenceTest.java` file):

```
String modelString = ""
+
"random earthquake: Boolean;" +
"random burglary: Boolean;" +
"random alarm: Boolean;" +
"" +
"earthquake 0.01;" +
"burglary 0.1;" +
"" +
"if earthquake" +
"   then if burglary" +
"      then alarm 0.95" +
"      else alarm 0.6" +
"   else if burglary" +
"      then alarm 0.9" +
"      else alarm 0.01;" +
"     " +
"not alarm;"
+ "";

Expression queryExpression = parse("not earthquake");
// can be any boolean expression, or any random variable

Expression evidence = parse("not alarm");
// can be any boolean expression

boolean isBayesianNetwork = true;
// is a Bayesian network, that is, factors are normalized
// and the sum of their product over all assignments to random variables is 1.

boolean exploitFactorization = true;
// exploit factorization (that is, employ Variable Elimination.

InferenceForFactorGraphAndEvidence inferencer =
new InferenceForFactorGraphAndEvidence(
new ExpressionFactorsAndTypes(modelString),
isBayesianNetwork,
evidence,
exploitFactorization);

Expression marginal = inferencer.solve(queryExpression);

System.out.println("Marginal is " + marginal);
```

4 Future Directions

AIC-PRAiSE is being currently developed and several extensions are expected. The main ones are:

- Exact inference for relational models.
- Anytime Lifted Inference ([dSBNB+09]), an approximate inference scheme with a interval guarantees and convergence to exact answer.
- MAP queries (most likely assignment to random variables).
- Tracing data structures for debugging support.
- Translation from other languages such as MLNs ([RD06]).

References

[dSBNB+09] R. de Salvo Braz, S. Natarajan, H. Bui, J. Shavlik, and S. Russell. Anytime lifted belief propagation. In *Statistical Relational Learning Workshop*, 2009.

[GMR+12] Noah D. Goodman, Vikash K. Mansinghka, Daniel M. Roy, Keith Bonawitz, and Daniel Tarlow. Church: a language for generative models. *CoRR*, abs/1206.3255, 2012.

[RD06] Matthew Richardson and Pedro Domingos. Markov logic networks. *Machine Learning*, 62(1-2):107–136, 2006.