AIC-PRAiSE User Guide

Rodrigo de Salvo Braz
Artificial Intelligence Center
SRI International

June 2, 2017

Contents

1 Introduction 3

2 Installing PRAiSE 3

3 Using PRAiSE 3
- 3.1 Higher-Order Graphical Models (HOGMs) . . . 4
  - 3.1.1 Main Definitions . . . 4
  - 3.1.2 Currently allowed boolean formulas . . . 7
  - 3.1.3 Semantics of HOGMs . . . 8
  - 3.1.4 A Subtlety in HOGMs . . . 9
- 3.2 PRAiSE Demo . . . 10
- 3.3 PRAiSE command-line interface . . . 11
- 3.4 Queries via the PRAiSE API . . . 12

4 Future Directions 13

1 Introduction

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

HOGMs is a more abstract and compact representation than the usual tabular (and sometimes procedural or decision diagram-based) representations typical in graphical models. For example, if `i` and `j` are bounded integers in $\{1, \ldots, 1000\}$ and we want to express the fact that `j` is less than `i`, we may represent the appropriate factor with the HOGM expression `if j < i then 1 else 0`. Then appropriate algorithms can manipulate this expression *without* iterating over the 10^6 entries in the corresponding table, instead taking advantage of the simple structure of this factor.

The current implementation of PRAiSE realizes the ideas presented in [dOGD16]. That paper describes how to perform probabilistic inference on languages in a particular theory. Currently, the theories available are propositional logic, equality on categorical types, difference arithmetic on bounded integers, and linear real arithmetic. We will provide more details in the next sections.

2 Installing PRAiSE

PRAiSE can be run directly from the browser at `http://aic-sri-international.github.io/aic-praise/` (Java Web Start required). The complete source code and instructions for installing a development version can be found at `https://github.com/aic-sri-international/aic-praise/`.

3 Using PRAiSE

PRAiSE can be used in three different forms:

- a graphical user interface offering a way for editing models and executing queries, explained in Section 3.2;
- a command-line interface reading models, query and evidence from files and outputting query answers, explained in Section 3.3;
- `jar` files exposing the PRAiSE library API for use by Java Virtual Machine (JVM) projects, explained in Section 3.4.

All three pieces of software take as input models specified in the HOGM language, detailed in the next Section, 3.1.

3.1 Higher-Order Graphical Models (HOGMs)

3.1.1 Main Definitions

The HOGM language is composed of a sequence of sort declarations, a sequence of symbol declarations, and a sequence of potential statements.

A **sort declaration** is used to define a categorical type, and is of the (Backus-Naur) form:

sort declaration

`sort` *identifier* `:` `[` *size* `,]` *constantIdentifier* `[ (,` *constantIdentifier* `)*];`

where *identifier* is a string starting with a letter and containing alphanumeric characters, _ and ' that represents the name of the sort, *size* is an optional integer constant indicating the size of the sort, and *constantIdentifier* are non-capitalized identifiers introducing constants belonging to the sort.

Examples are sort declarations are:

```
sort People : 10000, bob, mary;  // 10,000 people, including Bob, and Mary
sort Cats : bob, tom, mary;  // an unknown number of cats, including Bob, Tom, Mary
sort Dogs : 100;  // 100 dogs, with no known constants
```

Unknown sort sizes, if relevant for query results, appear as a cardinality function. For example, the unknown size of sort `Cats` above appears as `|Cats|`.

Sort declarations are optional because there are already built-in types (see below).

A **symbol declaration** is of the form:

symbol declaration

`(random | constant)` *identifier* `:` *type*`;`

where *identifier* is a string starting with a letter and containing alphanumeric characters, _ and ', and *type* is either a declared sort name, `Boolean`, `Integer`, n`..`m for n and m integer constants denoting an integer interval, `Real` or `[`a `;` b`]`, for a and b real constants, denoting a real interval (with variations `[`a`;` b`[`, `]`a `;` b`]` and `]`a `;` b`[` for open intervals).

A symbol prefixed by `random` denotes the value of a random variable of the given type, while a symbol prefixed by `constant` denotes a constant of the given type and unknown value. Such constants are useful for representing uninstantiated evidence, or a free model parameter, and appear in query results as such.

Examples:

```
random mypet       : Cats;
constant mySpouse : People;
random happy     : Boolean;
constant active : Boolean;
random counter  : Integer;
random counter' : -10..30;
random x : Real;
```

```
random y : [0;100];
```

Potential statements are used to specify conditional probability distributions (and, more generally in the case of factor graphs, factors). They are real-valued expressions ending in `;`.

Potential statements

The following operators can be used in potential expressions:

- `if-then-else` conditional expressions
- arithmetic operators, including `^` for power
- boolean operators `and`, `or`, `not`, `=>`, `<=>`
- `for all` and `there exists` quantifiers following the format
  `(for all I in 1..10 : I > J and I - 2 > -10)`,
  `there exists I in Integer : I > 10`,
  `there exists P in Boolean : P => Q`, etc.
- equality and disequality `=` and `!=` on real, integers and sort-typed variables or constants
- inequalities between integer-valued expressions: `<`, `>`, `<=`, `>=`; integer literals must belong to different arithmetic, that is, involve at most two variables with opposite signs (when on the same hand side of literal)
- inequalities between real-valued expressions: `<`, `>`, `<=`, `>=`; literals involving real-valued variables and constants must involve only *linear* expressions.

As an example, let us see how to represent the following generative model (which can also be seen as a Bayesian network) as a HOGM:

$$NFriends \in I, \quad Happy \in \{false, true\}$$

$$P(NFriends = nFriends) = \begin{cases} 1/100, \text{ if } 0 \leq nFriends < 100 \\ 0, \text{ otherwise} \end{cases}$$

$$P(Happy = happy | NFriends = nFriends) = \begin{cases} 1, \text{ if } nFriends > 10 \wedge happy \\ 0, \text{ if } nFriends > 10 \wedge \neg happy \\ 0.3, \text{ if } nFriends \leq 10 \wedge happy \\ 0.7, \text{ if } nFriends \leq 10 \wedge \neg happy \end{cases}$$

In HOGMs, we only need to declare the symbols, and express the *right-hand* side of the conditional probabilistic distributions:

```
random nFriends : Integer;
random happy : Boolean;
```

```
if 0 <= nFriends and nFriends < 100
     then 1/100
     else 0;

if nFriends > 10
     then if happy then 1 else 0
     else if happy then 0.3 else 0.7;
```

As a convenience, HOGMs offers **weighted boolean formulas**, by which expressions of the form

weighted boolean formulas

`if` ϕ `then` α `else` $1 - \alpha$,
for ϕ a boolean expression and $0 \leq \alpha \leq 1$,

can be abbreviated by the equivalent
$\phi \quad \alpha$.

This means that `if happy then 0.3 else 0.7` is equivalent to `happy 0.3` and the HOGM can be more clearly expressed as:

```
random nFriends : Integer;
random happy : Boolean;

if 0 <= nFriends and nFriends < 100
     then 1/100
     else 0;

if nFriends > 10
     then happy 1
     else happy 0.3;
```

A further convenience automatically adds the **default potential** 1 to a boolean formula appearing where a real-typed expression is expected, so the HOGM can be even further simplified as

default potential 1

```
random nFriends : Integer;
random happy : Boolean;

if 0 <= nFriends and nFriends < 100
     then 1/100
     else 0;
```

```
if nFriends > 10
    then happy       // no need for a potential
    else happy 0.3;
```

The default potential of 1 is convenient for declaring observations (evidence). For example, if we want to state that `happy` is observed to be true, we can add the potential statement:

```
happy;
```

which will rule out any global assignments in which happy is not true, having the effect of conditioning the model on that observation.

This allows the conditioning on arbitrary boolean formulas. For example, we can condition on the more complex formula

```
(happy and nFriends = 50) or not (nFriends = 5);
```

Adding a potential to such an statement turns it into a soft evidence statement. If we have probability 0.8 that `happy` is true, we can add the statement

```
happy 0.8;
```

3.1.2 Currently allowed boolean formulas

The *atoms* of the boolean formulas in HOGMs, that is, the boolean formulas that are *not* applications of boolean connectives, but applications of operators such as `=`, `!=`, `<`, etc, must be atoms recognized by the currently supported theories. Currently, the demo and command-line interfaces support equalities and disequalities of integers and categorical types, propositions (boolean variables), and inequalities on bounded integers.

Equalities, disequalities and inequalities on *integers* are supported only for **difference arithmetic**, which means that all their occurrences must be equivalent to a form either i *op* c or form $i - j$ *op* c, where i and j are variables, *op* is one of `=`, `!=`, `<`, `>`, `<=`, `>=`, and c is an integer numeric constant.

difference arithmetic

Examples of valid difference arithmetic conditions (where variables are `Integer`s) are

```
x != 1
x - 1 != 0
x > y
x - y <= 10
-y + x <= 10
```

Examples of atoms that are *not* difference arithmetic atoms are

```
x + y != 1
x > y + z
```

```
2*x - y <= 10
```

Equalities, disequalities and inequalities on *reals* are supported only for **linear real arithmetic**, in which variables can only be multiplied by constants, but an arbitrary number of such terms are allowed.

linear real arithmetic

Examples of valid linear real arithmetic conditions (where variables are `Reals`) are

```
x != 1
2*x != 1
2*x != 1 - 3*y
x + 2*y + 3*z < 0
x > y
x - y <= 10
-y + x <= 10
```

Examples of atoms that are *not* linear real arithmetic atoms are

```
x*z + y != 1
x^2 > y + z
```

IMPORTANT: PRAiSE treats evidence by simply adding extra statements to the model. For example, if a Boolean variable `happy` is observed to be true, it is dealt with by assuming that the model was augmented with the statement `if happy then 1 else 0;`. However, when real-valued variables are present, this method does not work because it renders the joint probability ill-defined (always equal to 0 instead of summing up to 1). For example, if we want to observe `x = 3`, adding `if x = 3 then 1 else 0` to the model renders the joint probability ill-defined because the measure of the set of possible worlds in which `x = 3` is 0, and all possible worlds not satisfying it also become 0 (doe to the else clause). We will address this in future versions, but for now please consider such cases undefined. Meanwhile, conditioning real-valued variables to a given value can be approach up to a precision with statements of the form `x > 3 and x <= 3.01`, for example.

The non-Boolean expressions involving integers and reals are allowed to be multivariable polynomials such as

```
y*x^2 + z*y
(x + y + z)^3 - 1/2*(x^4)
```

To see more examples of HOGMs, please check the examples provided within the PRAiSE demo.

3.1.3 Semantics of HOGMs

The semantics of HOGMs is similar to that of graphical models, which we assume readers to be familiar with.

Given a collection of potential statements, the probability of an assignment to random variable is *proportional* to the product of all potentials.

While the potential statements can be any collection of expressions, we highly recommend that users only write potential statements corresponding to conditional probabilities coming from a Bayesian network. Otherwise, the HOGM will correspond to an undirectly graphical model, the potentials of which are much harder to make intuitive sense of.

To conform to a Bayesian network (or a directed model), each potential statement should describe the probability of a *child* random variable in terms of zero or more *parent random variables* and, for each assignment to the parents, the sum of potential statement for all values of the child random variable must be 1. A directed graph linking parents to children must be acyclic.

Note how this holds for the previously seen model

```
if 0 <= nFriends and nFriends < 100
    then 1/100
    else 0;

if nFriends > 10
    then if happy then 1 else 0
    else if happy then 0.3 else 0.7;
```

in which `nFriends` has no parents and 100 possible values, and `happy` has parent `nFriends`. The conditional probability (a prior) potential statement of `nFriends` sums up to 1 if accumulated for all 100 values, and given an assignment to `nFriends`, the potential assignment of `happy` also sums up to 1 for its two possible values (false and true).

After that is done, further potential statements may be provided to represent evidence (assigning non-zero potentials to assignments conforming to observations).

3.1.4 A Subtlety in HOGMs

The previous section defines the basic working of the HOGM language. We now detail a common pitfall when using it regarding non-binary random variables such as integer-typed and categorical typed variables. It is sometimes tempting to write

```
random nFriends : 0..99;
if nFriends = 10 then 0.2 else 0.8;
```

or the equivalent

```
random nFriends : 0..99;
nFriends = 10   0.2;
```

assuming that this will mean a prior probability for `nFriends` assigning probability 0.2 for its value 10, and probability 0.8 distributed over its remaining values. However, remember

that this really stands for

$$P(NFriends = nFriends) = \begin{cases} 0.2, \text{ if } nFriends = 10 \\ 0.8, \text{ otherwise} \end{cases}$$

which applies to *each value* $nValue$ and therefore does not sum up to 1, but to $0.2+0.8\times 99$, since 99 values of the variable are not equal to 10. What we really want to write instead is

```
random nFriends : 0..99;
if nFriends = 10 then 0.2 else 0.8/99;
```

Consider this more complex case involving inequalities and a conditional probability statement:

```
random friendly : Boolean;
friendly 0.4;
random nFriends : 0..99;
if friendly then if nFriends >= 40 then 0.7/60 else 0.3/40;
             else if nFriends >= 30 then 0.2/70 else 0.8/30;
```

And, for good measure, let us see an example involving categorical type:

```
sort Companies : 10, ibm;
sort People : 100, bob, mary;
random company : Companies;
random boss : People;
company = ibm;
if company = ibm then if boss = mary or boss = bob then 0.4/2 else 0.6/98;
                    else 1/100;
```

3.2 PRAiSE Demo

The PRAiSE demo (screenshot shown in Fig. 3.2) allows users to run inference on HOGMs, evidence and queries edited in the demo itself. It can be run from an installed version of AIC-PRAiSE, or directly from the browser at http://aic-sri-international.github.io/aic-praise/ (Java Web Start required).

These are the main elements of the PRAiSE demo interface:

- the main panel is an editing area that allows the user to write a HOGM.
- the Play button (Ctrl-r) executes the query present in the query box (Ctrl-u), and the result is shown in the area below it.

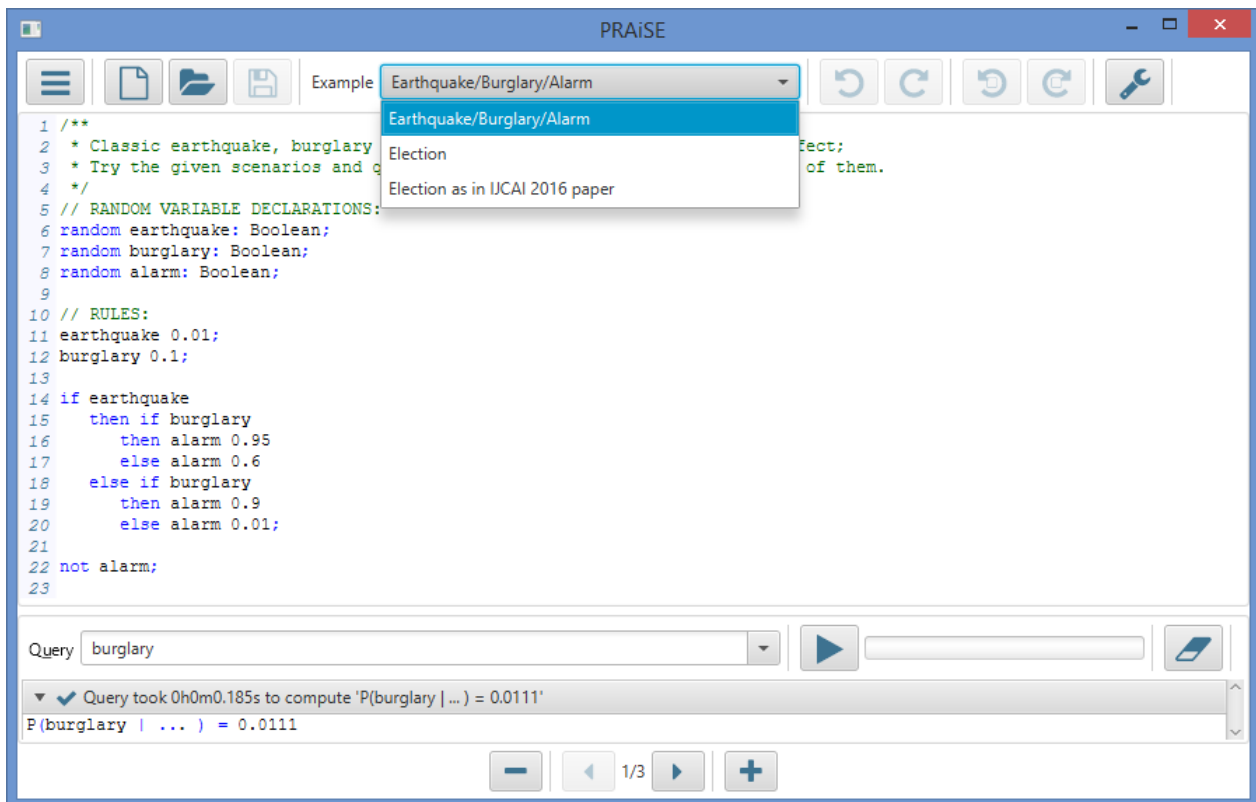

Figure 1: A screenshot of the PRAiSE demo.

- the results area (right below the query box) shows a list of all previous query results.
- the Eraser button clears the results area.
- the Example drop-down box allows the user to load pre-specified example models.
- the buttons at the very bottom of the window add, remove and navigate between multiple pages. This is useful for didactically demonstrating a sequence of changes to a model, and some of the pre-loaded examples may make use of this.
- the save and load buttons save and load the content of the demo to and from files. A single file stores all pages in the demo.
- the Blank Document button clears the demo contents.
- the left-corner menu button offers commands for saving and loading from models in the UAI conference format.

3.3 PRAiSE command-line interface

There are two possible ways of running a command-line interface of PRAiSE inference:

- running the `jar` available at https://github.com/aic-sri-international/aic-praise/releases/tag/20160518_latest_praise_cli
- installing the source of PRAiSE and running the class `com.sri.ai.praise.sgsolver.cli.PRAiSE`

Running the class with a `--help` option, the programs explains its require input and further options. Essentially, it takes either plain text files or files saved by the PRAiSE demo as input, containing HOGMs and queries, executes them, and outputs the results.

3.4 Queries via the PRAiSE API

PRAiSE can also be used as a Java library, providing query results directly to client code, as shown in the following example (present in the `testAPI` method in `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,
// as opposed to summing over the entire joint probability distribution).

InferenceForFactorGraphAndEvidence inferencer =
new InferenceForFactorGraphAndEvidence(
new ExpressionFactorsAndTypes(modelString),
isBayesianNetwork ,
evidence,
exploitFactorization,
null /* default constraint theory */);

Expression queryExpression;
Expression marginal;

queryExpression = parse("not earthquake");
// can be any boolean expression, or any random variable
marginal = inferencer.solve(queryExpression);
System.out.println("Marginal is " + marginal);

queryExpression = parse("earthquake");
marginal = inferencer.solve(queryExpression);
System.out.println("Marginal is " + marginal);
```

4 Future Directions

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

- New theories for relational random variables (that is, random functions), and algebraic data types.
- 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

[dOGD16] R. de Salvo Braz, C. O'Reilly, V. Gogate, and V. Dechter. Probabilistic Inference Modulo Theories. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*, New York, USA, 2016.

[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.

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