A framework for development, teaching and deployment of inference algorithms

Sander Evers, Peter J.F. Lucas
Institute for Computer and Information Sciences
Radboud University Nijmegen
s.evers@cs.ru.nl, peterl@cs.ru.nl

Abstract

We present symfer, a software framework for probabilistic inference algorithms. Each inference algorithm (like variable elimination, junction tree propagation, recursive conditioning) is represented as a symbolic manipulation of factor algebra expressions. In combination with the readability and terseness of Python code, this uniform representation makes the framework very suitable for teaching, as well as for explorative research: using the interactive Python interpreter, one can combine features of different algorithms and examine their effect. Numeric evaluation happens in a separate stage, implemented in Java/C for efficient execution and high portability. We exploit the latter feature in an application that performs inference on a smartphone.

1 Introduction

In popular software tools for inference in probabilistic graphical models such as Hugin, GeNIe/SMILE, SamIam and the MATLAB Bayes Net Toolbox, an inference algorithm (such as variable elimination (Zhang and Poole, 1996), junction tree propagation (Shenoy and Shafer, 1988), Kalman filtering (Kalman, 1960)) is a mostly opaque process. The user supplies a model, a query and evidence and calls the algorithm; the algorithm runs and returns a probability distribution. Processes such as determining a graph triangulation, elimination order or junction tree often remain hidden, and options for influencing this process are limited.

Of course, this is exactly what many users expect from these tools; the decisions made in said processes form a layer of complexity from which they want to be shielded. On the other hand, *researchers* and *teachers* of probabilistic inference methods have contrary requirements: what they desire from software is easy inspection and full control of these processes.

The software framework symfer that we present in this article caters to these needs using a novel architecture: it splits up the inference process into a *symbolic* and a *numeric* stage. The symbolic stage contains high-level inference algorithms; it is programmed in Python, which represents the algorithms in a uniform way which is readable (close to the pseudocode usually found in scientific literature) and easy to adapt. Moreover, the Python interpreter provides an interactive interface suited to explorative research similar to MATLAB.

The second stage performs the actual 'number grinding'. It has very few dependencies on external libraries, which makes it easily portable to other platforms. We have exploited this ourselves by including it in a telemedicine application on an Android phone (van der Heijden et al., 2011), which performs inference in a medical probabilistic model. We expect that this easy portability will also interest other software developers.

2 Architecture

Figure 1 shows the basic architecture of our software. Almost everything of interest happens in the first stage. It consists of a Python library with an API for creating and manipulating *factor algebra* expressions (defined in section 3). These can be based on externally created models (currently we support only the Hugin format) or constructed from scratch in Python. The latter option might be of interest if the model has a repetitive structure, or needs to be generated from a higher level description.

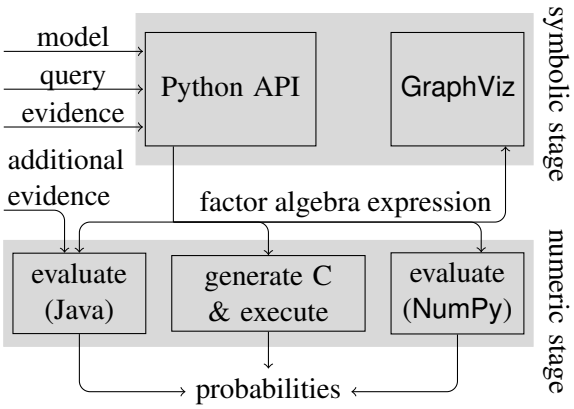

Figure 1: The two-stage architecture of symfer.

The purpose of this symbolic stage is to construct a factor algebra expression, which defines how to perform inference for a specific model, query and evidence combination. For this, the library includes several inference algorithms, of which the source code is meant for easy inspection and adaptation. The user can also use the API to write their own inference algorithms.

Subsequently, there are alternative ways to proceed. To obtain the resulting probabilities, the factor algebra expression (constructed in Python memory) can be evaluated either using Java, Python or C. For the first option, the expression is serialized to YAML (a human-readable textual format similar to JSON) and interpreted by a Java module, which evaluates it numerically. Optionally, the user can supply additional evidence $\mathbf{E}{=}e$ to this module: an index operation $[\mathbf{E}{=}e]$ is then inserted at the appropriate leaves, and the summation operation $\Sigma_{\mathbf{E}}$ is removed (we refer to section 3 for explanation of the operations). This option is practical for scenarios where the same query is repeated several times with different (maybe cumulative) evidence; in these cases, the first stage only needs to be applied once. The Java module is easily ported to other platforms; its only external dependency is the YAML parser.

Evaluation using C is different: dedicated C source code is generated for each operation in the factor algebra expression. This code can then be compiled and executed. The C code is faster, more predictable ('manual' garbage collection), and even more portable than the Java evaluator, but introducing additional evidence is impossible. The third option, evaluating directly from Python (using array processing library NumPy), is easiest to use but not recommended for memory-intensive queries, as garbage collection seems to be inefficient.

To aid in teaching and exploration, the expression can also exported to GraphViz (more specifically, dot) in order to show it as a tree (as in Figure 5).

3 Background: factor algebra

The symbolic expressions created using symfer consist of operations on *factors*. These operations are widely used in inference algorithms, e.g. see (Zhang and Poole, 1996) and (Koller and Friedman, 2009), but their symbolic use, although going back at least to (Shachter et al., 1990), is not widespread.

A factor is much like a mathematical function with multiple arguments, but refers to these arguments by name instead of by position, i.e. $f(X{=}x, Y{=}y)$ instead of $f(x, y)$. Thus, the order does not matter: $f(X{=}x, Y{=}y)$ is the same expression as $f(Y{=}y, X{=}x)$. In the scope of this article, a name like X refers to a probabilistic variable with a finite domain, written $\mathrm{dom}(X)$. Formally, a factor is defined as a single-argument function on instantiations; an *instantiation* $\mathbf{v} = \{V_1{=}v_1, \ldots, V_n{=}v_n\}$ is a function mapping each variable V_i to a value $v_i \in \mathrm{dom}(V_i)$.

A factor defines a specific set of variables $\mathbf{V}$ for which the instantiation must provide values; with a little abuse of terminology this set is called the factor's *domain* and written as $\mathrm{dom}(f) = \mathbf{V}$. Values of a factor are written $f(\mathbf{v})$ or $f(V_1{=}v_1, \ldots, V_n{=}v_n)$, where set braces are omitted for legibility, and are often probabilities. In symfer's numeric stage, a factor is implemented as an *array* of all its function values, with size $\prod_{V_i \in \mathrm{dom}(f)} |\mathrm{dom}(V_i)|$.

Factor algebra provides the tools for manipulating factors:

- *Multiplication:* $f \otimes g$ is a factor with domain $\mathrm{dom}(f) \cup \mathrm{dom}(g)$ and values $(f \otimes g)(\mathbf{v}) = f(\mathbf{v}) \cdot g(\mathbf{v})$.
- *Summation:* $\Sigma_{\mathbf{W}} f$, with $\mathbf{W} \subseteq \mathrm{dom}(f)$, is a factor with domain $\mathbf{U} = \mathrm{dom}(f) \setminus \mathbf{W}$ and values $(\Sigma_{\mathbf{W}} f)(\mathbf{u}) = \sum_{w \in \mathrm{dom}(\mathbf{W})} f(\mathbf{u}, \mathbf{W}{=}w)$.

- *Indexing:* $f[\mathbf{W}{=}w]$, with $\mathbf{W} \subseteq \mathrm{dom}(f)$, is a factor with domain $\mathbf{U} = \mathrm{dom}(f) \setminus \mathbf{W}$ and values $f[\mathbf{W}{=}w](\mathbf{u}) = f(\mathbf{u}, \mathbf{W}{=}w)$.

In the literature, the notations $f^{\downarrow \mathbf{U}}$ for summation and $f|_{\mathbf{W}=w}$ for indexing are often encountered. Furthermore, we use the notation $\mathbb{1}$ for the unit of factor multiplication; factor $\mathbb{1}$ has domain $\emptyset$ and value 1.

Why not use conventional algebra?

Factor algebra is constructed in such a way that its *denotational* semantics are very close to conventional algebra expressions. For example, consider three factors f, g and h, with

$$\begin{aligned} \mathrm{dom}(f) &= \{A\} & \mathrm{dom}(A) &= \{a_1, a_2, a_3\} \\ \mathrm{dom}(g) &= \{A, B\} & \mathrm{dom}(B) &= \{b_1, b_2, b_3, b_4\} \\ \mathrm{dom}(h) &= \{B, C\} & \mathrm{dom}(C) &= \{c_1, c_2, c_3, c_4, c_5\} \end{aligned}$$

Then the result of the factor algebra expression

$$\Sigma_A(f \otimes \Sigma_B(g \otimes \Sigma_C h)) \tag{1}$$

is the same number as the conventional expression

$$\sum_{a \in \mathrm{dom}(A)} f(a) \sum_{b \in \mathrm{dom}(B)} g(a,b) \sum_{c \in \mathrm{dom}(C)} h(b,c). \tag{2}$$

However, now consider the *operational* semantics of expr. (2). First, the outermost summation is expanded, replacing its formal variable a in the rest of the expression by concrete values a_1, a_2 and a_3:

$$\begin{aligned} & f(a_1) \sum_{b \in \mathrm{dom}(B)} g(a_1, b) \sum_{c \in \mathrm{dom}(C)} h(b,c) \\ + & f(a_2) \sum_{b \in \mathrm{dom}(B)} g(a_2, b) \sum_{c \in \mathrm{dom}(C)} h(b,c) \\ + & f(a_3) \sum_{b \in \mathrm{dom}(B)} g(a_3, b) \sum_{c \in \mathrm{dom}(C)} h(b,c) \end{aligned}$$

Then, each of the 3 b-summations are expanded into 4 terms; finally, each of the $3 \cdot 4$ c-summations are expanded into 5 terms. This leaves us with an expression with $3 + 3 \cdot 4$ multiplications and $3 \cdot 4 \cdot 5 - 1$ additions; running time is *exponential in the nesting depth*. Redundant calculations are made: the summation over C is performed for each of the 3 values for a, although its value does not depend on a.

On the other hand, expression (1) is evaluated from the inside out:

1. sum out C from h: $4 \cdot (5 - 1)$ additions
2. multiply with g: $3 \cdot 4$ multiplications
3. sum out B: $3 \cdot (4 - 1)$ additions
4. multiply with f: 3 multiplications
5. sum out A: $3 - 1$ additions

What matters here are not the precise numbers of additions or multiplications, but the fact that they do not depend exponentially on the expression's nesting depth, but rather on the *maximum domain size* (here: 2) of the intermediate factors produced in steps 1–5. Also, no redundant calculations are made this time. The difference between expr. (1) and (2) responsible for this is that in the latter, the concrete elements of $\mathrm{dom}(A)$ are mentioned in the outermost summation, scoping over the complete expression and duplicating operations where such concrete elements are not needed. In expr. (1), all information about concrete domain elements comes from the inside, i.e. from factors f, g and h themselves.

The downside of evaluating factor algebra is that it requires more space. At each evaluation step, all the participating factors are completely present in memory. Thus, the required space is exponential in the maximum intermediate domain size as well.

To conclude: in factor algebra, rewriting the expression matters—expression (1) has a totally different performance than $\Sigma_{ABC}(f \otimes g \otimes h)$—while in conventional algebra, expression (2) has the same nesting depth as $\sum_a \sum_b \sum_c f(a)g(a,b)h(b,c)$. With factor algebra as its basis, symfer enables the optimization of inference by rewriting expressions.

Factor algebra expressions as trees

Factor algebra expressions can be visualized as trees, like the example in Figure 2(a). The factors at the leaves of the tree are usually conditional probability tables that make up a Bayesian network; the branches are formed by $\otimes$, Σ and $[\mathbf{E}{=}e]$ operations. The arrows in the tree illustrate the data flow when evaluating the expression; to evaluate a node, we first need to evaluate its children (in any order). The nodes in the tree can be annotated with properties like its factor domain and corresponding array size, to help understand the cost of evaluation.

The output of any of symfer's symbolic inference algorithms is such a tree; in fact, Figure 2 is the output of variable elimination in the order $[A, B]$

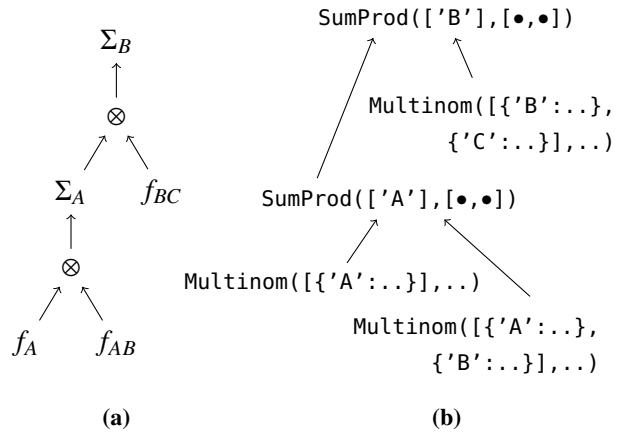

Figure 2: **(a)** Tree representation of factor algebra expression $\Sigma_B((\Sigma_A(f_A \otimes f_{AB})) \otimes f_{BC})$. **(b)** Python representation of this expression (also as tree; fill in the children at the • positions to obtain the flat text).

(given that the factor domains are $\mathrm{dom}(f_A) = \{A\}$, $\mathrm{dom}(f_{AB}) = \{A, B\}$ and $\mathrm{dom}(f_{BC}) = \{B, C\}$).

4 Representation of factors in Python

In the Python module, a factor expression is represented as a Python object. Its class corresponds to the top-level operation in the expression, and the object contains references to the arguments of this operation (themselves also objects representing factor expressions).

The *leaves* of the expression are factors representing conditional probability tables (or, in a Markov network, potentials), and are instances of the class `Multinom`. To construct such an instance, the user has to provide the factor's domain and a list of probabilities. The factor domain is a list with elements such as `{'Sprinkler':['off','on']}`, i.e. a variable name and a variable domain (list of possible values). For example, two tables representing P(`Weather`) and P(`Sprinkler`|`Weather`) can be created as follows:

```
weather       = {'Weather':['sunny','rainy']}
weather_cpd   = Multinom([weather],[0.8,0.2])
sprinkler_cpd = Multinom(
     [{'Sprinkler':['off','on']}, weather],
     [0.6,0.4,0.99,0.01] )
```

The tables are now stored as `weather_cpd` and `sprinkler_cpd`. We used an auxiliary Python variable `weather` to avoid repeating the definition of this domain. Not only does this save typing, but the Python semantics also cause the domain to be *shared* in memory among `weather_cpd` and `sprinkler_cpd`; i.e. both contain a pointer to the same object.

Although it is possible to enter a probabilistic model in Python as shown above, we expect that most users will rather do this using a separate graphical editor. Therefore, `Multinom` instances can also automatically be read from Hugin files.

A *compound* factor algebra expression can be obtained constructing objects of classes `SumProd` and `Index`. The former represents a combination of the operations $\otimes$ and Σ: the expression on the right hand side of the statement

```
sprinkler_marg = SumProd(['Weather'],
                 [weather_cpd,sprinkler_cpd])
```

multiplies the two tables, then sums out the variable `'Weather'`. Of course, it is possible to represent only a product (by using an empty list of variables) or a summation (by using a singleton list of factors).

The result is assigned to Python variable `sprinkler_marg`. More accurately, the name `sprinkler_marg` in the local namespace is bound to the newly created `SumProd` object. This object just holds references to the two `Multinom` objects. Nothing is evaluated yet, but when this will eventually be the case, the `SumProd` object stipulates that the two factors are to be multiplied, and the `'Weather'` variable is to be summed out. In this way, the `SumProd` object is a *symbolic representation* of the factor algebra expression. The correspondence is illustrated in Figure 2.

The Python objects that represent factors, such as `SumProd` and `Multinom` instances, are designed to be immutable: once they are constructed, their 'state' never changes. In this aspect, they are just like mathematical expressions. This design makes it possible to safely share subexpressions. For example, one could define:

```
joint= SumProd([],[weather_cpd,sprinkler_cpd])
sprinkler_marg = SumProd(['Weather'],joint)
weather_marg   = SumProd(['Sprinkler'],joint)
```

where the product of `weather_cpd` and `sprinkler_cpd` is shared among the two expressions that represent the marginals. This product `joint` represents the joint probability distribution over `'Weather'` and `'Sprinkler'`, an immutable

```
1 def ve_order(facs,order):
2     for rv in order:
3         rv_sum = SumProd([rv],[f for f in facs if rv in f.domtypes])
4         other_facs = [f for f in facs if rv not in f.domtypes]
5         facs = [rv_sum] + other_facs
6     return I().product(*facs)
```

Figure 3: Variable elimination with predefined order. In line 6, the expression `I().product(*facs)` calls a convenience function that returns `SumProd([],facs)`, except when `facs` contains only one factor, in which case that factor is returned.

mathematical entity. This stands in contrast to designs where objects represent a current belief over a certain set of variables, as commonly found in software implementations of e.g. the junction tree algorithm.

Indexing (setting probabilistic variables to certain values) is done by creating an instance of class `Index`. For example, the part of the joint probability distribution where the weather is sunny, P(`Sprinkler`, `Weather=sunny`), is obtained using `Index({'Weather':'sunny'},joint)`. This is a factor over the single variable `'Sprinkler'`. Alternatively, we could have defined

```
evidence       = {'Weather':'sunny'}
weather_sunny  = Index(evidence,weather_cpd)
sprinkler_sunny= Index(evidence,sprinkler_cpd)
joint_sunny    = SumProd([],
                 [weather_sunny,sprinkler_sunny])
```

which results in a different factor expression `joint_sunny` for the same distribution. The difference lies in the fact that in the former expression, the whole joint distribution is calculated (requiring, for example, the multiplication of P(`Sprinkler=off`|`Weather=rainy`) and P(`Weather=rainy`)), whereas in `joint_sunny` only the (conditional) probabilities consistent with the evidence are considered.

5 Symbolic inference in symfer

In the previous section, we manually crafted a factor expression for P(`Sprinkler`, `Weather=sunny`) using the Python API; thus, we were performing inference for query `['Sprinkler']`, evidence `{'Weather':'sunny'}` and a model consisting of `weather_cpd` and `sprinkler_cpd`. The symbolic inference algorithms in symfer perform this process automatically. They are defined using the same API. For example, the most simple inference algorithm, variable elimination with a predefined elimination order, is given by the 6 lines in Figure 3. The function `ve_order` defined there is invoked as follows:

```
result = ve_order([weather_cpd,sprinkler_cpd],
                  ['Sprinkler','Weather'])
```

Note that the Python code is very close to mathematical pseudocode. For example, lines 3–5 would correspond to

$$\texttt{rv\_sum} \leftarrow \Sigma_{\texttt{rv}} \bigotimes \{\, f \mid f \in \texttt{facs}, \texttt{rv} \in \mathrm{dom}(f) \,\}$$

$$\texttt{other\_facs} \leftarrow \{\, f \mid f \in \texttt{facs}, \texttt{rv} \notin \mathrm{dom}(f) \,\}$$

$$\texttt{facs} \leftarrow \{\texttt{rv\_sum}\} \cup \texttt{other\_facs}$$

In `ve_order`, evidence is not explicitly considered. There are two approaches for including evidence. Either the user applies `Index` operations *before* invoking the function, i.e.

```
res_sunny = ve_order([weather_sunny,
              sprinkler_sunny], ['Sprinkler'])
```

or the API function `indextree` is invoked on the function *result*:

```
res_sunny = indextree(evidence,result)
```

Here, the two approaches yield the same factor expression; however, when using an elimination heuristic such as minweight, they might result in a different elimination order.

The definition of variable elimination with a triangulation heuristic is not much more complex; see appendix A. Perhaps more surprisingly, junction tree propagation (in the Shenoy–Shafer variant (Shenoy and Shafer, 1988)) can be defined very concisely as well. The key insight is that the `SumProd` expression produced as a result of a variable elimination can function as a clique tree on which the algorithm is based.

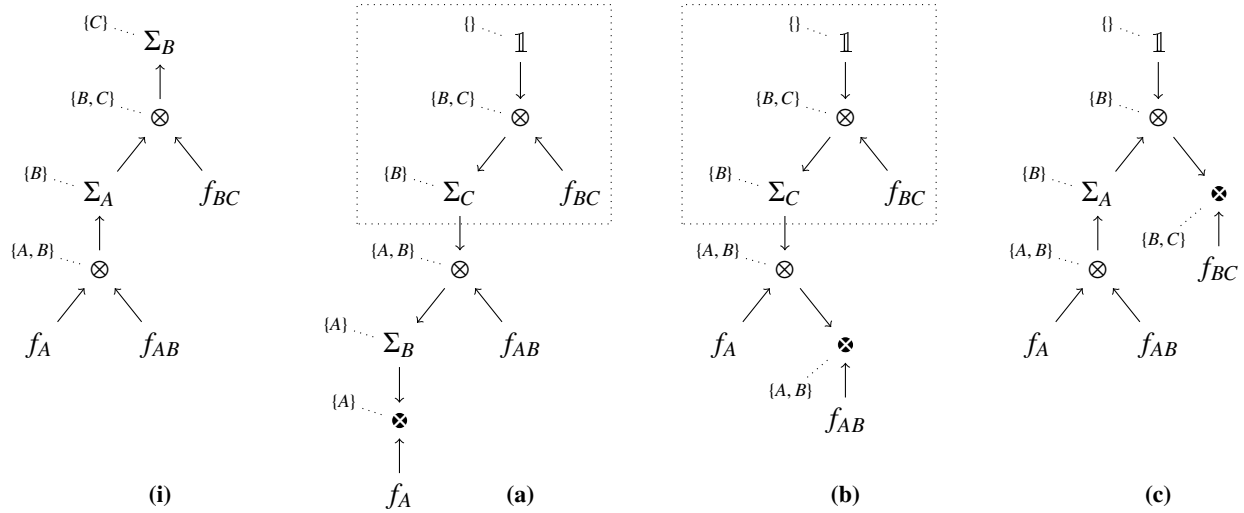

Figure 4: Example input **(i)** and output **(a,b,c)** of the `junctiontree` algorithm. Given a factor expression tree (such as the output of a variable elimination algorithm), `junctiontree` produces a new tree for each leaf (f_A, f_{AB}, f_{BC}) of the input tree. For clarity, we indicate the roots of these trees with ⊗ here, and show the domain for each subtree using dotted lines (these are not part of the tree). For each output tree, a root node is created above the leaf, all the arrows on the path from this node to the original root are reversed, and sums are updated (or added). In memory, common subtrees such as in the dotted box (but also the f_A, f_{AB}, f_{BC} leaves) are shared among the output trees; see Figure 5. For the algorithm itself, see Appendix A.

In our variant, the algorithm creates a factor expression for each leaf of the original tree; this expression calculates the marginal probability distribution over the domain of this leaf. It does this by traversing the path from the original root (which is replaced by $\mathbb{1}$, or `I()` in Python) down to the leaf in question, while creating a new tree in 'reverse order' from the subtrees not on this path; see Figure 4 for an example and Appendix A for the algorithm.

The fact that a junction tree can be created from the structure of a variable elimination procedure, and that junction tree propagation can be interpreted as N instances of variable elimination (one for each marginal) is well known (Shafer, 1996); the often cited advantage of the junction tree algorithm is that it theoretically runs in approximately $2\times$ the time of variable elimination instead of $N\times$. Traditionally, this is accomplished by explicitly storing intermediate results in nodes in the data structure. In our algorithm we achieve essentially the same results (using the same insight) by letting the output trees *share* their subtrees, as shown in Figure 5. As we explained before, this sharing happens quite transparently in Python: unless explicitly instructed, it

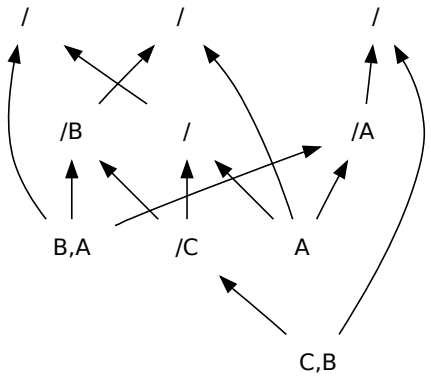

Figure 5: The output trees of the junction tree example in Figure 4, shown with explicit sharing of subtrees, as drawn by symfer and GraphViz. The top row lists, from left to right, the roots of **(b)**, **(a)** and **(c)**. A node starting with a slash represents a `SumProd` expression: it multiplies its children, then sums out the variables after the slash. The other nodes are `Multinom` instances. Multiplication with $\mathbb{1}$ is simplified away.

never copies an object but always a pointer. Crucially, when the factor expressions are evaluated, the numeric stage is aware of this sharing, and evaluates the shared subexpressions only once. The serialization language YAML can represent this expression sharing, which is why it is used instead of JSON.

Although we have not implemented a specific algorithm yet, the principle of *conditioning*, see e.g. (Shachter et al., 1994) and (Darwiche, 2001), can also easily be expressed using factor algebra. For example, to condition on variable B in the expression in Figure 2 (suppose we have named it `veAB`), one could simply do

```
cB = [s.indextree({'B':b},veAB)
      for b in ['b1','b2','b3']]
```

resulting in a list of 3 trees just like Figure 2(a), but without the Σ_B root node and with $[B{=}b_i]$ expressions at the leaves. The numeric stage would have to add the 3 resulting factors together; we will include support for this in a later version.

6 Inference in a telemedicine system

Apart from inference research, we are also putting symfer to use in a telemedicine system that we are developing (van der Heijden et al., 2011) for home use by COPD patients. COPD (chronic obstructive pulmonary disease) is a chronic lung disease, currently affecting some 210 million people worldwide, with considerable health care related costs. The progression of the disease can be slowed by rapid intervention when an exacerbation (acute worsening of the symptoms) is detected. Our system, which consists of a smartphone and two small physiological sensors, enables a patient to perform simple measurements and fill in a questionnaire about the symptoms in a home (or mobile) setting on a daily basis.

Using a probabilistic disease model, the system then determines whether the chance of an exacerbation is high enough to warrant an intervention. It sends its findings to an authorised physician, but can also directly advise the patient. For the system to work without network communication, the probabilistic inference is performed on the Android smartphone itself.

Until recently, we used an inference library on the phone that required us to define the model+query+evidence and run a complete inference algorithm every time the application was started. Currently, we use symfer, in a setup where we provide the symbolic stage with model and query variable and run it only once (on a separate machine). This is possible because in our application, the query does not change. On the smartphone, only the numeric stage is performed. Every time the application is run, the sensors and questionnaire provide new evidence; as we explained in section 2, the numeric stage can insert this evidence in the factor expression just before it evaluates it.

As it has the same structure but starts with smaller arrays at the leaves, the expression with evidence is guaranteed not to require more memory than the original expression without evidence. Thus, if the smartphone has enough memory to run the original expression (which we need to test only once), it can also run the inference with any combination of evidence. This would not have been the case if we ran a variable elimination heuristic each time, as this can change the structure of the expression.

7 Conclusion and prospects

What distinguishes symfer from other inference software is its thorough separation of inference algorithms into a symbolic and numeric stage. We have shown that at the former stage, widely taught algorithms such as variable elimination, junction tree propagation and conditioning can be formulated in a unified manner. This is achieved through the central role of factor algebra, combined with the sharing semantics inherent in Python.

Furthermore, the Python language allowed us to express the algorithms in a notation that is almost as clear and concise as pseudocode. We expect this to facilitate teaching and explorative research. Here, the interactive Python interpreter (Pérez and Granger, 2007) will also help, in the same way as MATLAB does.

Apart from these advantages, we have found the separation of the two stages useful for deploying part of the software on the Android platform, where it is part of a telemedicine application. The reason for this is that the numeric stage is quite small and easily portable. This also means that it can be *optimized* separately from the rest of the software

(for example, by data management researchers, not needing any knowledge of probabilistic models).

We are currently working on support for recursive conditioning, factor indexing (Evers and Lucas, 2011), and efficient handling of deterministic patterns such as Noisy-OR; it is no problem to express these in the factor algebra framework. A somewhat larger challenge will be to include continuous variables, but we expect this to be possible as well.

We make symfer available as open source (BSD-licensed) software, except for the Android numeric evaluator, which we plan to publish with a license for free academic use.

References

Adnan Darwiche. 2001. Recursive conditioning. *Artif. Intell.*, 126(1-2):5–41.

Sander Evers and Peter J. F. Lucas. 2011. Marginalization without summation: Exploiting determinism in factor algebra. In Weiru Liu, editor, *Symbolic and Quantitative Approaches to Reasoning with Uncertainty - 11th European Conference, ECSQARU 2011. Proceedings*, volume 6717 of *Lecture Notes in Computer Science*, pages 251–262. Springer.

R.E. Kalman. 1960. A new approach to linear filtering and prediction problems. *Journal of Basic Engineering*, 82(1):35–45.

D. Koller and N. Friedman. 2009. *Probabilistic graphical models: Principles and techniques*. MIT Press.

Fernando Pérez and Brian E. Granger. 2007. IPython: a System for Interactive Scientific Computing. *Comput. Sci. Eng.*, 9(3):21–29, May.

Ross D. Shachter, Bruce D'Ambrosio, and Brendan Del Favero. 1990. Symbolic probabilistic inference in belief networks. In Howard E. Shrobe, Thomas G. Dietterich, and William R. Swartout, editors, *AAAI*, pages 126–131. AAAI Press / The MIT Press.

Ross D. Shachter, Stig K. Andersen, and Peter Szolovits. 1994. Global conditioning for probabilistic inference in belief networks. In Ramon López de Mántaras and David Poole, editors, *UAI '94: Proceedings of the Tenth Annual Conference on Uncertainty in Artificial Intelligence*, pages 514–522. Morgan Kaufmann.

Glenn Shafer. 1996. *Probabilistic expert systems*. Society for Industrial and Applied Mathematics.

Prakash P. Shenoy and Glenn Shafer. 1988. Axioms for probability and belief-function propagation. In Ross D. Shachter, Tod S. Levitt, Laveen N. Kanal, and John F. Lemmer, editors, *UAI '88: Proceedings of the Fourth Annual Conference on Uncertainty in Artificial Intelligence*, pages 169–198. North-Holland.

Maarten van der Heijden, Bas Lijnse, Peter J. F. Lucas, Yvonne F. Heijdra, and Tjard R. J. Schermer. 2011. Managing COPD exacerbations with telemedicine. In Mor Peleg, Nada Lavrac, and Carlo Combi, editors, *13th Conference on Artificial Intelligence in Medicine, AIME 2011. Proceedings*, volume 6747 of *Lecture Notes in Computer Science*, pages 169–178. Springer.

Nevin Lianwen Zhang and David Poole. 1996. Exploiting causal independence in bayesian network inference. *J. Artif. Intell. Res. (JAIR)*, 5:301–328.

A Python code for inference algorithms

```
def ve_minweight(facs,query):
    remaining = set().union(
           *[set(fac.domlist) for fac in facs])
    remaining -= set(query)
    while remaining:
        cand = None,None,float('inf')
        for rv in remaining:
             rv_facs = [fac for fac in facs
                        if rv in fac.domtypes]
             rv_prod = I().product(*rv_facs)
             rv_weight = len(rv_prod)
             if rv_weight < cand[2]:
                cand = rv,rv_prod,rv_weight
        rv,rv_prod,_ = cand
        remaining.remove(rv)
        rv_sum = rv_prod.sumout([rv])
        other_facs = [fac for fac in facs
                      if rv not in fac.domtypes]
        facs = [rv_sum] + other_facs
    return I().product(*facs)
```

```
def junctiontree(tree,upfac=I()):
    if isinstance(tree,SumProd):
        out = []
        for f in tree.fac:
            prod_others = upfac.product(*[other for other in tree.fac if other is not f])
            facres = junctiontree(f,prod_others.sumto(f.domlist))
            out.extend(facres)
        return out
    elif isinstance(tree,Multinom):
        return [upfac.product(tree)]
```