ENFrame: A Framework for Processing Probabilistic Data

Dan Olteanu and Sebastiaan J. van Schaik, University of Oxford

This article introduces ENFrame, a framework for processing probabilistic data. Using ENFrame, users can write programs in a fragment of Python with constructs such as loops, list comprehension, aggregate operations on lists, and calls to external database engines. Programs are then interpreted probabilistically by ENFrame. We exemplify ENFrame on three clustering algorithms: k-means, k-medoids, and Markov clustering; and one classification algorithm: k-nearest neighbour.

A key component of ENFrame is an event language to succinctly encode correlations, trace the computation of user programs, and allow for computation of discrete probability distributions for program variables. We propose a family of sequential and concurrent, exact and approximate algorithms for computing the probability of interconnected events. Experiments with k-medoids clustering and k-nearest neighbour show orders-of-magnitude improvements of exact processing using ENFrame over naïve processing in each possible world, of approximate over exact, and of concurrent over sequential processing.

Categories and Subject Descriptors: H.2.4 [**Database Management**]: Systems; G.3 [**Mathematics of Computing**]: Probability and Satistics; D.1.3 [**Software**]: Concurrent Programming

General Terms: Algorithms, Systems

Additional Key Words and Phrases: data processing, probabilistic inference, data mining

ACM Reference Format:
Dan Olteanu and Sebastiaan J. van Schaik, 2015. ENFrame: A Framework for Processing Probabilistic Data *ACM Trans. Datab. Syst.* 1, 1, Article 1 (January 2015), 43 pages.
DOI: http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

Probabilistic data management has gone a long, fruitful way in the last decade [Suciu et al. 2011]. We have a better understanding of the space of possible probabilistic relational and XML data models and its implication on query tractability. The community already delivered several open-source systems that exploit the first-order structure of database queries for scalable inference, e.g., MystiQ [Boulos et al. 2005], Trio [Widom 2008], MayBMS/SPROUT [Huang et al. 2009], and PrDB [Sen et al. 2009] to name very few, as well as applications in the space of web data management [Fink et al. 2011] and probabilistic knowledge bases [Dong et al. 2014]. Significantly less effort has been spent on supporting complex data processing beyond mere querying, such as general-purpose programming or even data mining.

There is a growing need for computing frameworks that allow users to build applications feeding on uncertain data without worrying about the underlying uncertain nature of such data or the computationally hard inference task that comes along

The authors acknowledge the financial support from the European Commission under the FP7 grant HiPerDNO, the UK Engineering and Physical Sciences Research Council under the grant ADEPT, het Prins Bernhard Cultuurfonds, and het De Breed Kreiken Innovatiefonds. The authors would also like to thank Robert Fink for his contribution to ENFrame.

DOI: http://dx.doi.org/10.1145/0000000.0000000

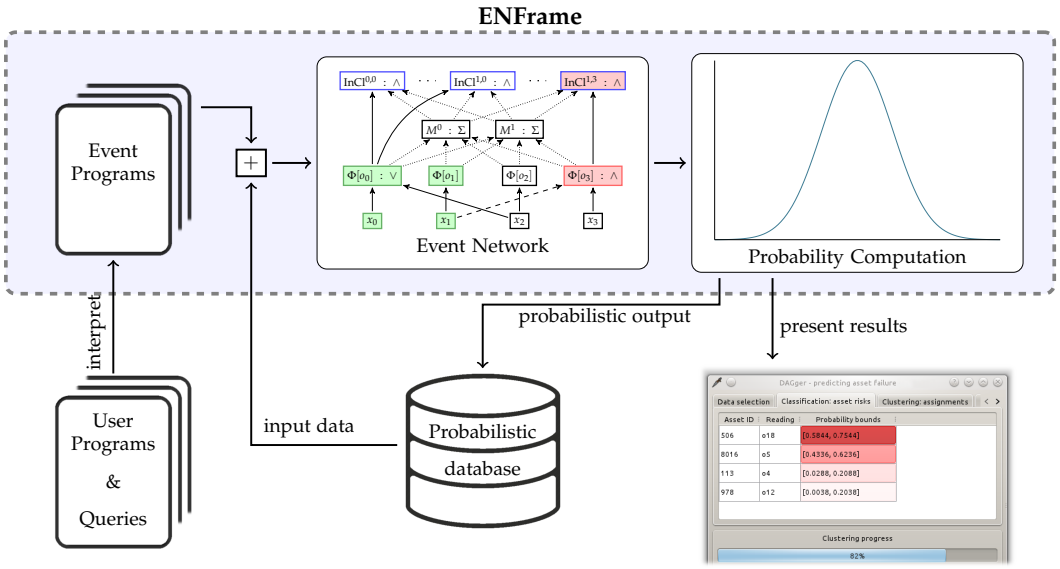


Fig. 1. The architecture of ENFrame.

with it. For tasks that only need to query probabilistic data, existing probabilistic database systems offer a viable solution [Suciu et al. 2011]. Similarly, there are approaches developed for specific data mining tasks, such as clustering or frequent pattern mining [Aggarwal 2009]. For more complex user-defined tasks, however, existing approaches are not applicable and successful development requires a high level of expertise in probabilistic databases, which hinders the adoption of existing technology as well as communication between potential users and experts.

A similar observation on the growing need for programming with probabilistic data has been recently made in the areas of machine learning [DARPA 2013] and programming languages [Gordon et al. 2014]. Developing probabilistic programming languages that allow probabilistic models to be expressed concisely has become a popular research topic [Roy 2015]. Such languages allow to express probability distributions via generative stochastic models, to draw values at random from such distributions, and to condition values of program variables on observations. For inference, the programs are grounded to Bayesian networks and fed to Markov Chain Monte Carlo methods [Milch and et al 2005]. In the area of databases, MCDB [Jampani et al. 2011] and SimSQL [Cai et al. 2013] have been visionary in enabling stochastic analytics in the database by coupling Monte Carlo simulations with declarative SQL extensions and parallel database techniques. Recent work extends Datalog with probability distributions for declarative statistical modelling [Bárány et al. 2016].

The thesis of this article is that one can build powerful and useful probabilistic data programming frameworks that *leverage* existing work on probabilistic databases. We describe ENFrame, a framework that aims to fulfil this vision. The architecture of this framework is depicted in Figure 1. Given an input user program and a probabilistic database, ENFrame computes a probability distribution over possible program results, which can then be presented back to the user [Olteanu and van Schaik 2012]. Its main ingredients are: a language for user programs, a specification of probabilistic events that capture program traces, and a suite of probabilistic inference algorithms that take as input the events and output their probability distributions.

ENFrame's programming language is a fragment of Python with loops, list comprehension, aggregates, Boolean and real-valued variables, variable assignments, and query calls to external database engines. A user program can express tasks such as clustering and classification intermixed with querying. Section 2 introduces this Python fragment and exemplifies it using clustering algorithms (k-means, k-medoids, and Markov clustering) and classification tasks (k-nearest neighbour).

Separation of data and program. The users (programmers) are oblivious to the probabilistic nature of the input data: They program as if the input data were deterministic, with no uncertainty. It is the job of ENFrame to understand probabilities and input correlations, thus allowing users without expert knowledge of probabilistic models to program over uncertain data.

Unifying semantics for data and processing. ENFrame adheres to the possible worlds semantics for its whole processing pipeline. Under this semantics, the input is a probability distribution over a finite set of possible worlds, with each world defining a database. The result of a user program is equivalent to executing it within each world and is a probability distribution over possible outcomes of the program variables.

Event language. ENFrame uses a language of probabilistic events over discrete random variables to express arbitrary correlations occurring in the input data, *e.g.*, modelled on Bayesian networks [Pearl 1989] and probabilistic c-tables [Suciu et al. 2011], and in the result of the user program, *e.g.*, co-occurrence of data points in the same cluster, and trace the program state at any time. By annotating each computation in the program with events, we effectively translate it into an event program: program variables become random variables whose possible outcomes are conditioned on events. Section 3 introduces the event language. It features negation, aggregates, and definitions within an algebraic formalism reminiscent of provenance semimodules [Amsterdamer et al. 2011] and their use in a probabilistic setting [Fink et al. 2012].

Inference algorithms. A key challenge faced by ENFrame is to compute the probabilities of all events capturing the traces of a program. These events are organised in a directed graph called an *event network*, where expressions common to several events are represented once. Event networks for user programs can be very repetitive and highly interconnected due to the combinatorial nature of the programs. For instance, for clustering the events at each iteration are expressions over the events at the previous iteration and have the same structure at each iteration. Moreover, the event networks can be cyclic, so as to account for program loops.

The complex nature of ENFrame's event networks sets the inference problem apart from earlier work in probabilistic databases, *e.g.*, [Fink et al. 2013], which only considers one propositional event in disjunctive normal form at a time. We perform probability computation for the entire event network by decomposing the network into simpler networks using repeated elimination of random variables and by incrementally refining lower and upper bounds on the probabilities of all events in the simpler networks. The approximation algorithms presented in Section 4 use an error budget to avoid a large number of decomposition steps that can only gain a small probability mass. We introduce several approximation approaches, each with a different strategy for spending the error budget. We also present a concurrent approximation approach that distributes the decomposition task over multiple CPU cores.

While the computation time can grow exponentially in the number of input random variables in worst case (since the inference problem is #P-hard), the algorithms exploit the structure of programs, and thus of correlations of events capturing program traces, for more efficient inference. This is in line with our prior work on query evaluation in probabilistic databases [Olteanu et al. 2009; Fink et al. 2013].

Section 5 reports on experiments with k-medoids clustering and k-nearest neighbour classification and shows orders-of-magnitude performance improvements of ENFrame's exact algorithm over the naïve approach that executes the user program in each possible world, of approximate over exact, and of concurrent over sequential processing. Experiments also confirm that ENFrame's clustering accuracy is identical to that of the naïve approach and that clustering in the top-p most probable worlds for p close to the overall number of worlds does not reach a good accuracy while being more expensive than ENFrame.

Differences to existing trends. There are key differences that set ENFrame apart from the myriad of recent probabilistic programming [Roy 2015] and probabilistic data processing [Aggarwal 2009] approaches:

— ENFrame uses the possible worlds semantics and a complete representation system for probabilistic data for its entire processing pipeline from the input data to the program result. This system is compatible with those used for query processing in probabilistic databases, *e.g.* [Fink et al. 2013]. ENFrame's input can consist of data with arbitrarily correlated (and not only independent) probabilistic events. This enables processing pipelines mixing programming (via ENFrame) and querying (via, *e.g.*, SPROUT). This is a key difference to the bulk of existing data mining approaches that use expected values and simple probabilistic models [Aggarwal 2009].
— The users need not be aware of the probabilistic nature of data and their programs are written as if the input data is plain and not probabilistic. One implication is that probability distributions can only be supplied as input data and not in the actual program. This is a key difference to probabilistic programming approaches, where probability distributions are an essential part of the language. Consequently, the users are required to understand probabilistic models and semantics.
— In contrast to (Monte Carlo) sampling-based frameworks for inference [Jampani et al. 2011] and in the spirit of intensional query processing in probabilistic databases, the probabilistic event language used by ENFrame can enable rich and useful computation beyond inference such as result explanation, sensitivity analysis [Kanagal et al. 2011], and incremental maintenance of the program output in the face of updates to the input (this is subject to future work). The events used by ENFrame can be (exponentially) more succinct than the propositional events used so far in probabilistic database formalisms such as probabilistic c-tables [Suciu et al. 2011].

An extended abstract of this article has been already published [van Schaik et al. 2014; Olteanu and van Schaik 2014]. Besides a more in-depth treatment of each aspect of ENFrame, this article also discusses the case of k-nearest neighbour classification, introduces a concurrent approximate inference approach, an accuracy measure for probabilistic clustering, as well as experiments for all these contributions.

2. ENFRAME'S USER LANGUAGE

This section introduces the user language supported by ENFrame. Its design is grounded in three main desiderata:

(1) It should naturally express common data mining algorithms, allow to issue queries, and manipulate their results.
(2) User programs must be oblivious to the deterministic or probabilistic nature of the input data and to the probabilistic formalism considered.
(3) It should be simple enough to allow for an intuitive and straightforward probabilistic interpretation.

We settled on a subset of Python that can naturally express several clustering algorithms. In line with query languages for probabilistic databases, where a Boolean query Q is a map $Q : D \to \{true, false\}$ for deterministic databases and a Boolean random variable for probabilistic databases, every user program has a sound semantics for both deterministic and probabilistic input data. In the former case, the result of a program is a deterministic assignment of values to program variables, in the latter case it is a probability distribution over possible values for each program variable. For instance, in a clustering program the user can define a Boolean variable stating whether two objects belong to the same cluster. In a deterministic setting, the variable can take values *true* or *false*. In a probabilistic setting, its value would be a probability distribution over the two possible Boolean outcomes, and it would state how likely it is for the two objects to co-occur together in the same cluster. After program execution, the program result, *i.e.*, the values of program variables, and can be presented to the user or serve as input for subsequent processing using a probabilistic DBMS or ENFrame program.

The user language comprises the following syntactic constructs:

Variables and Assignments. Program variables can be of a scalar type (real, integer, or Boolean), or a list thereof. We allow assignments of values to variables, *e.g.*, `V = 2`, `W = V`, `M[2] = True`, or `M[i] = W`. Scalar variables can be multiplied, exponentiated (`pow(B, r)` for B^r), and inverted (`invert(B)` for $1/B$). The multiplication of a list with a scalar yields a list where each component of the list is multiplied with the scalar. Lists must be initialised, *e.g.*, for list `M` of length `n`: `M = [None] * n`. The expression `range(m, n)` specifies the list `[m,...,n-1]`.

Loops. Loops take the form: `for` i `in range(`b`,`e`)`, where b and e are constants or variables of type integer.

Reduce. Lists of scalar values can be *reduced* to a scalar value using one of the predefined functions `reduce_or`, `reduce_and`, `reduce_sum`, `reduce_mult`, `reduce_count`. For instance, for a list `B` of Booleans, the expression `reduce_and(B)` computes the conjunction of the truth values in `B`, and the expression `reduce_count(B)` computes the number of elements in `B`.

List comprehension. Inside a `reduce`-function, anonymous lists may be defined using list comprehension. For example, given a list `B` of Booleans of size `n`, the expression `reduce_sum([1 for i in range(0,n) if B[i]])` counts the `true` values in `B`.

User-defined functions. We allow user-defined functions (udfs) that take as input (possibly empty tuples of) program variables and return scalar or list values, or tuples of such values. For instance, the function `dist(A,B)` is a distance measure on the feature space between the lists `A` and `B` of reals. The function `loadData()` can be used to load the records from a file or to issue queries to an external database. ENFrame supports positive relational algebra queries with aggregates on probabilistic data via a user-defined function that calls the SPROUT probabilistic query engine [Fink et al. 2012]. The functions `loadParams()` and `init()` can be used to set parameters such as the number of iterations and clusters of a clustering algorithm. All user-defined functions are implemented externally to ENFrame.

The clustering programs discussed later in this section exemplify user-defined functions for breaking ties. Many data mining tasks require explicit handling of ties: For instance, if two objects are equidistant to two cluster medoids in k-medoids, we have to decide which cluster the object will be assigned to. For instance, the membership of objects to clusters can be encoded using a list `InCl` such that `InCl[i][l]` is true if and only if object `l` is in cluster `i`. A tie would be a configuration of `InCl` in which for a fixed

⟨*loop*⟩ ::= {⟨*decl*⟩} { **for** ⟨*var*⟩ **in** ⟨*range*⟩ **:** { ⟨*loop*⟩ } }

⟨*decl*⟩ ::= ⟨*var*⟩ **=** (⟨*expr*⟩ | **[None]*** ⟨*var*⟩) | **[(** {⟨*var*⟩**,**} ⟨*var*⟩ **) =]** ⟨*udf*⟩

⟨*expr*⟩ ::= ⟨*const*⟩ | ⟨*var*⟩ | ⟨*reduce*⟩ **(** ⟨*lcompr*⟩ **)**
| Boolean or arithmetic expression over ⟨*expr*⟩
| numerical comparison of two ⟨*expr*⟩
| user-defined function ⟨*udf*⟩ over ⟨*expr*⟩ returning scalar values

⟨*lcompr*⟩ ::= **[** ⟨*expr*⟩ **for** ⟨*var*⟩ **in** ⟨*range*⟩ **if** ⟨*expr*⟩ [**else** ⟨*expr*⟩] **]**

⟨*range*⟩ ::= **range(** ⟨*var*⟩ , ⟨*var*⟩ **)**

⟨*reduce*⟩ ::= **reduce_and** | **reduce_or** | **reduce_sum** | **reduce_mult** | **reduce_count**

⟨*var*⟩ ::= a variable identifier for scalars and list entries

⟨*const*⟩ ::= a scalar (Boolean, integer, float)

Fig. 2. High-level grammar of the user language supported by ENFrame.

object `l`, `InCl[i][l]` is true for more than one cluster `i`. We explicitly break such ties using the (user-defined) function `breakTies2(M)`. For each fixed value `i` of the second dimension (hence the '2' in the function name) of the 2-dimensional list `M`, we iterate over the first dimension of `M` and sets all but the first true value of `M[i][l]` to *false*. Similarly, the function `breakTies1(M)` fixes the first dimension and breaks ties in the second dimension of `M`, while `breakTies(M)` breaks ties in a one-dimensional list.

The grammar in Figure 2 highlights the major syntactic constructs of the user language currently supported by ENFrame. An ENFrame user program consists of a sequence of declarations and loop blocks. The language allows to define variables whose values can be scalar, are given by expressions, or lists. An expression can be: a Boolean, integer, or floating-point constant; a variable identifier; the Boolean result of a comparison between two expressions; a propositional formula on expressions with logical connectors `and`, `or`, `not`; the numerical result of an operation such as summation, multiplication, inversion, and exponentiation of expressions; the numerical result of a *reduce* operation on an anonymous list of scalars created via list comprehension; the numerical result of the application of user-defined functions on expressions, *e.g.*, for breaking ties or selecting the top-k largest values in a list.

To keep it simple, typing and standard semantic constraints are not modelled in the grammar, though, *e.g.*, Boolean and number expressions cannot be freely mixed and certain operators and functions only make sense for either Booleans, numbers, or lists. In addition, programs have to satisfy the following restrictions specific to ENFrame, which are imposed by ENFrame's capability for probabilistic interpretation of user programs:

— The variables used to initialise, specify, or iterate over lists can only be integers and bounded to constants from the input data that are the same in all possible worlds; in particular, they cannot be the result of program computation, since such results can become probabilistic.
— List comprehension can only be used to construct one-dimensional lists of scalar types.
— The implementation of user-defined functions is required to work on both deterministic and probabilistic inputs. For instance, the distance-measure function `dist`

```
(O, n) = loadData()          # list and no. of objects
(k, it) = loadParams()       # no. of clusters and it.
M = init()                   # initialise centroids

for t in range(0,it):        # clustering iterations
 InCl = [None] * k           # assignment phase:
 for i in range(0,k):        # for every cluster i ...
  InCl[i] = [None] * n
  for l in range(0,n):       # and for every object l:
   InCl[i][l] = reduce_and( # assign l to closest centroid
    [dist(O[l],M[i]) <= dist(O[l],M[j])
     for j in range(0,k)]
  )
 InCl = breakTies2(InCl)     # each object in one cluster

 M = [None] * k              # update phase:
 for i in range(0,k):        # for every cluster i ...
  M[i] = scalar_mult(        # centroid at cluster centre
   invert(reduce_count(
    [1 for l in range(0,n) if InCl[i][l]]
   )), reduce2_sum(
    [O[l] for l in range(0,n) if InCl[i][l]]
   )
```

$\vec{o}_0 \ldots \vec{o}_{n-1}$: n object vectors in the feature space
$\Phi(o_0) \ldots \Phi(o_{n-1})$: n propositional object events
k, it: number of clusters and iterations

$\forall i \; in \; 0..n-1 : O^i \equiv \Phi(o_i) \otimes \vec{o}_i$
$M^0_{-1} \equiv \vec{o}_0; \ldots; M^{k-1}_{-1} \equiv \vec{o}_{(k-1)}$

$\forall t \; in \; 0..it-1 :$

$\forall i \; in \; 0..k-1 :$

$\forall l \; in \; 0..n-1 :$

$$InCl^{i,l}_t \equiv \bigwedge_{j=0}^{k-1} \left[dist\left(O^l, M^i_{t-1}\right) \leq dist\left(O^l, M^j_{t-1}\right) \right]$$

Encoding of `breakTies2` omitted

$\forall i \; in \; 0..k-1 :$

$$M^i_t \equiv \left(\sum_{l=0}^{n-1} InCl^{i,l}_t \otimes 1 \right)^{-1} \cdot \left(\sum_{l=0}^{n-1} InCl^{i,l}_t \otimes O^l \right)$$

Fig. 3. User and event programs for k-means clustering

mentioned above would return a real for deterministic input lists and a probability distribution over reals for inputs representing probability distributions over lists.

2.1. Data Mining Algorithms in ENFrame

We illustrate ENFrame's user language with four data mining algorithms: k-means, k-medoids, Markov clustering, and k-nearest neighbour classification. Figures 3–6 list user programs for these algorithms; we next discuss each of them. The event programs listed alongside the user programs are discussed in Section 3.

k-means clustering. The k-means algorithm partitions a set of n records, or data points, $o_0, \ldots, o_{n-1}$ into k groups of similar data points. We initially choose a centroid M^i for every cluster ($0 \leq i < k$), *i.e.*, a data point representing the cluster centre (initialisation phase). In successive iterations, each data point is assigned to the cluster with the closest centroid (assignment phase), after which the centroid is recomputed for each cluster (update phase). The algorithm terminates after a given number of iterations or after reaching convergence. Note that our user language does not support fixpoint computation, and hence checking convergence.

Figure 3 gives the user program for k-means. The set `O` of n input objects is retrieved using a `loadData` call. Each object is represented by a feature vector (*i.e.*, list) of reals. We then load the parameters `k`, the number `it` of iterations, and initialise cluster centroids `M`. The initialisation phase has a significant influence on the clustering outcome and convergence. We assume that initial centroids have been chosen, for example by using a heuristic. Subsequently, a list `InCl` of Booleans is computed such that `InCl[i][l]` is `True` if and only if `M[i]` is the closest centroid to object `O[l]`; every object is then assigned to its closest cluster. Since two clusters may be equidistant to an object, ties are broken using the `breakTies2` call; it fixes an order of the clusters and enforces that each object is only assigned to the *first* of its potentially multiple closest clusters. Next, the new cluster centroids `M[i]` are computed as the centroids of each cluster. The assignment and update phases are repeated `it` times.

```
1  (O, n) = loadData()          # list of objects
2  (k, it) = loadParams()       # no. of clusters and it.
3  M = init()                   # initialise medoids
4
5  for t in range(0,it):        # clustering iterations
6   InCl = [None] * k           # assignment phase:
7   for i in range(0,k):        # for every cluster i ...
8    InCl[i] = [None] * n
9    for l in range(0,n):       # and for every object l:
10    InCl[i][l] = reduce_and(  # closest medoid to l
11     [(dist(O[l],M[i]) <= dist(O[l],M[j]))
12      for j in range(0,k)]
13    )
14  InCl = breakTies2(InCl)     # one cluster per object
15
16  M = [None] * k              # update phase:
17  for i in range(0,k):        # for every cluster i ...
18   DistSum = [None] * n
19   for l in range(0,n):       # and every object l:
20    DistSum[l] = reduce_sum(  # compute distance sum
21     [dist(O[l],O[p]) for p in range(0,n) if InCl[i][p]]
22    )
23
24   IsMedoid = [None] * n;
25   for l in range(0,n):       # object with min. sum ...
26    IsMedoid[l] = reduce_and( # is medoid of cluster i ...
27     [DistSum[i][l] <= DistSum[i][p]
28      for p in range(0,n) if InCl[i][p]]
29    ) and InCl[i][l]          # if l is in cluster i
30   IsMedoid = breakTies(IsMedoid)
31
32   M[i] = reduce_sum(
33    [O[l] for l in range(0,n) if IsMedoid[l]]
34   )
```

$\vec{o}_0 \ldots \vec{o}_{n-1}$: n object vectors in the feature space
$\Phi(o_0) \ldots \Phi(o_{n-1})$: n propositional object events
k, it: number of clusters and iterations

$\forall i \; in \; 0..n-1 : O^i \equiv \Phi(o_i) \otimes \vec{o}_i$
$M_{-1} \equiv initialiseMedoids()$

$\forall t \; in \; 0..it-1 :$

$\forall i \; in \; 0..k-1 :$

$\forall l \; in \; 0..n-1 :$

$$InCl_t^{i,l} \equiv \bigwedge_{j=0}^{k-1} \left[dist\left(O^l, M_{t-1}^i\right) \leq dist\left(O^l, M_{t-1}^j\right) \right]$$

Encoding of breakTies2 omitted

$\forall i \; in \; 0..k-1 :$

$\forall l \; in \; 0..n-1 :$

$$DistSum_t^{i,l} \equiv \sum_{p=0}^{n-1} \left[InCl_t^{i,p} \otimes dist\left(O^l, O^p\right) \right]$$

$\forall l \; in \; 0..n-1 :$

$$IsMedoid_t^{i,l} \equiv \bigwedge_{p=0}^{n-1} \left[DistSum_t^{i,l} \leq \left(InCl_t^{i,p} \otimes DistSum_t^{i,p} \right) \right] \wedge InCl_t^{i,l}$$

Encoding of breakTies omitted

$$M_t^i \equiv \sum_{l=0}^{n-1} \left[IsMedoid_t^{i,l} \otimes O^l \right]$$

Fig. 4. User and event programs for k-medoids clustering

k-medoids clustering. The k-medoids algorithm is almost identical to k-means, but elects k cluster *medoids* rather than centroids: these are cluster members that minimise the sum of distances to all other objects in the cluster. Figure 4 gives the user program for k-medoids clustering. The assignment phase is the same as for k-means, while the update phase is more involved: we first compute a list `DistSum` of sums of distances between each cluster medoid and all other objects in its cluster, then find one object in each cluster that minimises this sum, and finally elect these objects as the new cluster medoids `M`. The last step uses `reduce_sum` to select exactly one of the objects in a cluster as the new medoid, since for each cluster `i` only one value in `IsMedoid`, which corresponds to one object, is `True` due to the tie-breaker.

Markov clustering. The Markov clustering algorithm (MCL) is a fast and scalable unsupervised cluster algorithm for graphs based on simulation of stochastic flow [van Dongen 2000]. Natural clusters in a graph are characterised by the presence of many edges within a cluster and few edges across clusters. MCL simulates random walks within a graph by alternating two operations: *expansion* and *inflation*. Expansion corresponds to computing random walks of higher length. It associates new probabilities with all pairs of nodes, where one node is the point of departure and the other is the destination. Since higher length paths are more common within clusters than between different clusters, the probabilities associated with node pairs lying in the same cluster will, in general, be relatively large as there are many ways of going from one to the other. Inflation has the effect of boosting the probabilities of intra-cluster walks

```
(n, M) = loadData()      # M: stoch. n*n matrix
(r, it) = loadParams()   # r: inflation parm.

for t in range(0,it):
 N = [None] * n          # expansion phase
 for i in range(0,n):
  N[i] = [None] * n
  for j in range(0,n):   # square matrix:
   N[i][j] = reduce_sum(
    [M[i][k] * M[k][j] for k in range(0,n)]
   )

 M = [None] * n          # inflation phase
 for i in range(0,n):
  M[i] = [None] * n
  for j in range(0,n):   # inflate and re-normalise
   M[i][j] = pow(N[i][j],r) *
       invert(reduce_sum(
        [pow(N[i][k],r) for k in range(0,n)]))

InCl = [None] * n        # interpret M
for i in range (0,n):
 InCl[i] = [None] * n
 for j in range (0,n):   # determine cluster for j
  InCl[i][j] = M[i][j] > 0
```

```
# M: stochastic n × n matrix with edge weights
# Φ(o_0) ... Φ(o_{n-1}): n propositional object (vertex) events
# r, it: inflation parameter and number of iterations
∀i in 0..n−1 : ∀j in 0..n−1 : M^{i,j}_{-2} ≡ (Φ(o_i) ∧ Φ(o_j)) ⊗ M^{i,j}
∀i in 0..n−1 : ∀j in 0..n−1 : M^{i,j}_{-1} ≡ M^{i,j}_{-2} · ( Σ_{k=0}^{n-1} M^{i,k}_{-2} )^{-1}

∀t in 0..it−1 :

  ∀i in 0..n−1 :

    ∀j in 0..n−1 :
      N^{i,j}_t ≡ Σ_{k=0}^{n-1} [ M^{i,k}_{t-1} · M^{k,j}_{t-1} ]

  ∀i in 0..n−1 :

    ∀j in 0..n−1 :
      M^{i,j}_t ≡ (N^{i,j}_t)^r · ( Σ_{k=0}^{n-1} (N^{i,k}_t)^r )^{-1}

∀i in 0..n−1 :

  ∀j in 0..n−1 :
    InCl^{i,j} ≡ M^{i,j}_{it-1} > 0
```

Fig. 5. User and event programs for Markov clustering

```
# nu = no. of unlabelled points
# nl = no. of labelled points
# C = classes of labelled points, nc = no. of classes
# SLN[u] = sorted list of labelled neighb. for unlabelled u
(nu, nl, C, SLN) = loadData()
(k) = loadParams()           # k: no. of neighb. to consider

Votes = [None] * nl;         # votes from labelled objects
for l in range (0, nl):      # iterate over labelled objects
 Votes[l] = [None] * nc      # votes from object l
 for i in range (0, nc):     # iterate over possible classes
  Votes[l][i] = 1 + (1 if C[l] == i else 0) # vote for class i

ClassAssign = [None] * nu;   # stores class assignments
for u in range (0, nu):      # iterate over unlabelled points
 VoteSum = [None] * nc       # sum of votes neighbours
 for i in range (0, nc):     # iterate over possible classes
  NearestVotes = [None] * nc # sort votes:
  for m in range (0, nl):    # vote from mth nearest neighbour
   NearestVotes[m] = Votes[SLN[u][m]][i]

  # sum first k votes
  VoteSum[i] = reduce_sum(select_first(k, NearestVotes))

 ClassAssign[u] = [None] * k      # class assignm. for u
 for i in range (0, nc):          # iterate over classes
  ClassAssign[u][i] = reduce_and( # assignment to class i
   VoteSum[i] >= VoteSum[p] for p in range (0, nc)
  )
 ClassAssign[u] = breakTies(ClassAssign[u])   # break ties
```

```
# n_u  = number of unlabelled data points
# n_l  = number of labelled data points
# n_c  = number of classes
# C^l  = class of labelled data point l  (0 ≤ C^l < n_c)
# N^m_u = m^th closest neighbour of unlabelled point o_u

∀l in 0 ... (n_l − 1) :

  ∀i in 0 ... (n_c − 1) :
    Votes^{l,i} ≡ Φ[o_l] ⊗ ( 1 + ( (C^l = i) ⊗ 1 ) )

∀u in 0 ... (n_u − 1) :

  ∀i in 0 ... (n_c − 1) :

    ∀m in 0 ... (n_l − 1) :
      NearestVotes^m_u ≡ Votes^{N^m_u, i}

  VoteSum^i_u ≡ Σ F_k (NearestVotes_u)

  ∀i in 0 ... (n_c − 1) :
    ClassAssign^i_u ≡ ⋀_{p=0}^{n_c-1} [ VoteSum^i_u ≥ VoteSum^p_u ]

# definition of breakTies omitted
```

Fig. 6. User and event programs for k-nearest neighbour classification

$\langle eid \rangle$ ::= event identifier | $x = i$ for random variable x and possible outcome i

$\langle bool \rangle$ ::= $\langle eid \rangle$ | Boolean expression over $\langle bool \rangle$ | $\langle val \rangle$ comp. operator $\langle val \rangle$

$\langle val \rangle$ ::= constant | $\langle bool \rangle \otimes \langle val \rangle$ | arithmetic or Boolean expression over $\langle val \rangle$
| F ({$\langle val \rangle$,} $\langle val \rangle$) for user-defined function F | [{$\langle val \rangle$,} $\langle val \rangle$]

Fig. 7. High-level grammar of event expressions supported by ENFrame.

and demoting inter-cluster walks. This is achieved without a priori knowledge of cluster structure. Figure 5 presents the user and event programs for Markov clustering. Expansion coincides with taking the power of a stochastic matrix `M` using the normal matrix product (*i.e.* matrix squaring). Inflation corresponds to taking the Hadamard power of a matrix (taking powers entry-wise). It is followed by a scaling step to maintain the stochastic property, *i.e.*, the matrix elements correspond to probabilities that sum up to 1 in each column.

k-nearest neighbour classification. The k-nearest neighbour classification algorithm (or simply: k-nn) is a supervised data mining technique that assigns a class to one or more *unlabelled* data points o_l, based on the class labels of the k labelled data points that are closest to o_l. The unlabelled data point is assigned the class which occurs most in its direct neighbourhood.

Figure 6 contains the user and event programs for k-nn classification. The algorithm requires, amongst other parameters, the classes of all labelled data points, and a sorted list of nearest neighbours for every unlabelled object. Every object is allowed to cast votes for every single of the the `nc` classes. An object `l` contributes a vote value of `Votes[l][i] = 1` if it does not belong to class `i`, and votes `Votes[l][i] = 2` in case it does have class label `i`.

Every unlabelled object `u` considers the votes from its `k` nearest neighbours. For every possible class label, the votes from the nearest neighbours are added up. Subsequently, object `u` is the class label with the largest number of votes.

3. TRACING COMPUTATION BY EVENTS

To evaluate the user program on probabilistic data, we first ground the program by computing its execution traces on the input data and then use these traces to compute the probability distribution of the program result. This section shows how to capture the execution traces of user programs on probabilistic data using probabilistic events. Section 4 then shows how to compute the probability distributions of entire collections of such events.

We next introduce ENFrame's language of events and connect it with existing probabilistic database formalisms, give examples of events and explain the event programs for the clustering and k-nn user programs from Section 2.1, and finally show how to derive the event program from a user program.

3.1. Event Expressions

In our probabilistic model, the input data is a list of probabilistic events, where each event is a discrete random variable with Boolean, integer, or real-valued outcomes.

Syntax of Events Expressions. Syntactically, an event is an expression over a finite set of independent discrete random variables. The grammar for event expressions is highlighted in Figure 7. For the sake of notational clarity, it does not distinguish between scalars and lists and assumes that expressions are well-typed.

Boolean events are denoted by $\langle bool \rangle$. In their simplest form, they are assignments of random variables, *e.g.*, $x = i$, and Boolean expressions over such assignments. Besides Boolean events, ENFrame uses so-called conditioned values, or c-values for short; they are denoted by $\langle val \rangle$ in the grammar. The c-value $\Phi \otimes v$ represents a (Boolean, numerical, or list) value v conditioned on the Boolean event Φ in the sense that it takes the value v in the possible worlds where Φ is *true* and a "neutral" value otherwise; by convention, the neutral value does not occur in the input data. Additionally, user-defined functions and arithmetic and Boolean expressions on c-values and lists of c-values are supported. Our example programs (Figures 3–6) use functions to compute the distance between c-values and to select the k closest c-values to a given value. Furthermore, Boolean events may refer to previously-defined events by their identifiers and may encode comparisons between c-values.

Semantics of Events Expressions. The semantics of event expressions follows the possible worlds semantics. This is standard for Boolean propositional events over discrete random variables [Suciu et al. 2011]: Each total assignment ν of the input variables defines a possible world and any Boolean event is either *true* or *false* in any given world. Variable assignments can be extended to c-values. A c-value $\Phi \otimes v$ for a Boolean event Φ and a value v has the value v in those worlds defined by variable assignments that map Φ to *true* and a neutral value otherwise. By convention, the neutral value does not occur in the input data. Like scalars and lists, c-values can be added, multiplied, or exponentiated. We extend the domains of numerical (Boolean) values and their operations $+$, $\cdot$, $()^{-1}$ by a special element u to stand for the neutral value such that $0^{-1} = u$. The meaning of summation, multiplication, and neutral value depend on the value domain. For numbers (Booleans), the neutral value would be 0 (*false*), whereas summation (disjunction) and multiplication (conjunction) are the standard ones for numbers (Booleans). Operators $+, \cdot$ propagate u as $u + x = x$ and $u \cdot x = u$ for any value x. For any other values x, y, the operators $+$ and $\cdot$ are as usual. Similarly, we extend the lists by an element $\mathbf{u}$. For lists of numbers (Booleans), we use pointwise summation (disjunction) and multiplication (conjunction). For any scalar value a and list $\mathbf{x}$, u and $\mathbf{u}$ are propagated as follows: $u \cdot \mathbf{x} = \mathbf{u}$, $\mathbf{u} + \mathbf{x} = \mathbf{x}$, $a \cdot \mathbf{u} = \mathbf{u}$, and $\mathbf{u} \cdot \mathbf{x} = u$. For instance, the sum of c-values $\Phi \otimes v + \Psi \otimes w$ evaluates to $v + w$ if Φ and Ψ are *true*, to v if Φ is *true* and Ψ is *false*, to w if Φ is *false* and Ψ is *true*, and to the neutral value u in the domain of v and w otherwise.

We extend an assignment ν from variables to c-values and Boolean events as follows (the equations for neutral values are given above and not repeated below):

$$\nu(\Phi \otimes v) = \begin{cases} v, & \text{if } \nu(\Phi) = \top \\ u \text{ (}\mathbf{u}\text{, resp.)} & \text{otherwise} \end{cases}$$

$$\nu(\Phi_1 \otimes v_1 \text{ op } \Phi_2 \otimes v_2) = \nu(\Phi_1 \otimes v_1) \text{ op } \nu(\Phi_2 \otimes v_2), \text{ where op} \in \{+, \cdot\}$$

$$\nu((\Phi \otimes v)^w) = (\nu(\Phi \otimes v))^w$$

$$\nu(\Psi \text{ op } (\Phi \otimes v)) = \nu(\Psi) \text{ op } \nu(\Phi \otimes v), \text{ where op} \in \{\wedge, \vee\}$$

$$\nu(\Phi_1 \otimes v_1 \ \ \theta \ \ \Phi_2 \otimes v_2) = \begin{cases} \top, & \text{if } \nu(\Phi_1 \otimes v_1) = u \text{ or } \nu(\Phi_2 \otimes v_2) = u \text{ or } \nu(\Phi_1 \otimes v_1) \ \ \theta \ \ \nu(\Phi_2 \otimes v_2) \\ \bot, & \text{otherwise} \end{cases}$$

$$\text{where } \theta \in \{\leq, \geq, =, <, >\}$$

$$\nu(\text{dist}(\Phi_1 \otimes v_1, \Phi_2 \otimes v_2)) = \begin{cases} u, & \text{if } \nu(\Phi_1 \otimes v_1) = \mathbf{u} \text{ or } \nu(\Phi_2 \otimes v_2) = \mathbf{u} \\ \text{dist}(\nu(\Phi_1 \otimes v_1), \nu(\Phi_2 \otimes v_2)), & \text{otherwise} \end{cases}$$

The last equation is an example of how user-defined functions can be added to our framework; the function `dist` computes the (Euclidean) distance between two c-values.

We next give a probabilistic interpretation of event expressions that explains how they can be understood as random variables: Boolean event expressions give rise to Boolean random variables, and c-values give rise to random variables with numerical outcomes.

Let us fix a set of random variables $\mathbf{X}$. For every random variable $x \in \mathbf{X}$, we denote by $P[x = i]$ the probability that x takes the value i from the set of its possible outcomes $outcomes(x)$. Let $\Omega = \{\nu : x \mapsto outcomes(x) \mid x \in \mathbf{X}\}$ be the set of possible mappings from the random variables in $\mathbf{X}$ to their outcomes.

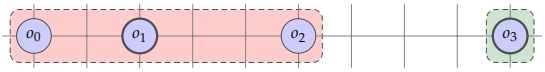
Definition 3.1. The probability mass function $\Pr(\nu) = \prod_{x \in \mathbf{X}} P[x = \nu(x)]$ for every sample $\nu \in \Omega$, and the probability measure $\Pr(E) = \sum_{\nu \in E} \Pr(\nu)$ for $E \subseteq \Omega$ define a probability space $(\Omega, 2^{\Omega}, \Pr)$ that we call the *probability space induced by* $\mathbf{X}$.

An event expression E is a random variable over the probability space induced by $\mathbf{X}$ with probability distribution

$$P[E = s] = \Pr\big(\{\nu \in \Omega \mid \nu(E){=}s\}\big) = \sum_{\nu \in \Omega : \nu(E) = s} \Pr(\nu).$$

By virtue of this definition, every Boolean event expression becomes a Boolean random variable, and real-valued (list-valued) c-values become random variables over the reals (the lists).

Examples of Events and Event Programs for Clustering. Consider an example of k-medoids clustering with objects $o_0, \ldots, o_3$ as depicted below. They can be clustered into two clusters with medoids o_1 and o_3:



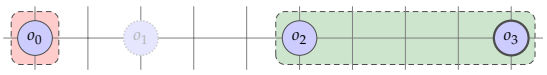
Now assume a probability space defined by a set of independent Boolean random variables $x_1, \ldots, x_4$. Each of the 2^4 total assignments of these variables to possible outcomes true ($\top$) and false ($\bot$) defines a possible world whose probability is the product of the probabilities of the individual variable assignments.

A simple example of a probabilistic event is the propositional formula $\Phi_0 = (x_1 = \top \wedge x_3 = \top)$ with outcomes $\{\top, \bot\}$. The probability of this formula being true is given by the sum of probabilities of the total assignments that satisfy it: $\Pr[\Phi_0 = \top] = Pr[x_1 = \top] \cdot \Pr[x_3 = \top]$. The event Φ_0 can be associated with the object o_0 in the following sense: o_0 exists in exactly those possible worlds where $\Phi_0 = \top$ and its probability of existence is $\Pr[\Phi_0 = \top]$. In all other worlds, o_0 does not exist (technically, its value is neutral for the given domain of objects). We write down this association between the Boolean event Φ_0 and the value o_0 as the c-value $\Phi \otimes v$. In the probabilistic database literature [Suciu et al. 2011], this association is implicit and not denoted by an explicit algebraic construct as in ENFrame. A scenario where such a c-value may occur considers a value v aggregated from readings of two different sensors whose reliability is approximated by the Boolean random variables x_1 and x_3.

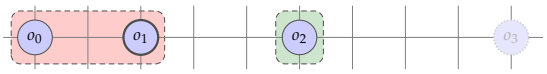
We consider the following events for the objects $o_0, \ldots, o_3$:

$$\Phi_0 = (x_1 = \top \wedge x_3 = \top) \qquad \Phi_2 = (x_3 = \top)$$
$$\Phi_1 = (x_2 = \top) \qquad \Phi_3 = (x_2 = \bot \wedge x_4 = \top)$$

Distinct worlds can have different clustering results. For instance, the world defined by the assignment $\{x_1 = \top, x_2 = \bot, x_3 = \top, x_4 = \top\}$ consists of objects o_0, o_2, and o_3, for which k-medoids clustering yields:



The worlds defined by $\{x_1 = \top, x_2 = \top, x_3 = \top\}$ and any assignment for x_4 yield:



Equipped with c-values, an initialisation of k-means with $k = 2$ can be written in terms of two expressions $M^0 = \Phi_0 \otimes o_0 + \neg\Phi_0 \otimes o_2$ and $M^1 = \top \otimes 0.5 \cdot (o_1 + o_3)$: centroid M^0 is set to object o_0 if Φ_0 is true and to o_2 if Φ_0 is false; M^1 is the geometric centre of o_1 and o_3.

In the assignment phase, each object is assigned to its nearest centroid. The condition $\text{InCl}^{i,l}$, which states whether o_l is closest to M^i, can be written as the following Boolean event, which encodes that the distance from o_l to centroid M^0 is smaller than the distance to centroid M^1:

$$\text{InCl}^{i,l} \equiv \bigwedge_{j=0}^{1} \left[\text{dist}\left(\Phi(o_l) \otimes o_l, M^i\right) \leq \text{dist}\left(\Phi(o_l) \otimes o_l, M^j\right)\right]$$

Given the Boolean events $\text{InCl}^{i,l}$, we can represent the centroid of cluster i for the next iteration by the following expression:

$$\left(\sum_{l=0}^{3} \text{InCl}^{i,l} \otimes 1\right)^{-1} \cdot \left(\sum_{l=0}^{3} \text{InCl}^{i,l} \otimes o_l\right)$$

This specifies a random variable over possible cluster centroids conditioned on the assignments of objects to clusters as encoded by $\text{InCl}^{i,l}$. For n objects, this expression is exponentially more succinct than an equivalent purely propositional encoding of centroids, since the latter would require one Boolean expression for each subset of the objects.

The event programs corresponding to the four user programs for k-means, k-medoids, MCL, and k-nn are given on the right side of Figures 3–6.

3.2. Connection to existing probabilistic database formalisms

Boolean and c-value events are commonplace in mainstream probabilistic database models, such as the tuple-independent, block-independent disjoint, the probabilistic c-table, and the probabilistic vc-table models [Suciu et al. 2011; Fink et al. 2012].

The probabilistic web data repositories such as NELL [Carlson et al. 2010] and Knowledge Vault (KV) [Dong et al. 2014] consist of tuple-independent tables, where each record is an explicit representation of a probability distribution of an independent random variable. For instance, the record (t, p) encodes that the represented random variable takes the value t with probability p and is otherwise undefined, i.e., it does not appear in the table, with probability $1 - p$. We can capture this information in our formalism using a c-value $x \otimes t$, where x is a Boolean random variable whose probability for true is p and the interpretation of the neutral value for the domain of t is that of a record not appearing in the database.

The block-independent disjoint tables [Suciu et al. 2011] and x-relations [Agrawal et al. 2006] represent sets of blocks of alternative values. Any data source with conflicting data yields this type of probabilistic data, *e.g.*, geolocation data [Mokbel et al. 2006], optical character recognition, and automated data extraction [Dong et al. 2009; Bleiholder and Naumann 2009]. An example of a block is $\{(t_1, p_1), \ldots, (t_n, p_n)\}$, where

the values p_1 to p_n are probabilities such that $\sum_{i=1}^{n} p_i \leq 1$ and the values t_1 to t_n are mutually exclusive. In other words, at most one of these values t_i can exist in any of the possible worlds and it does so with probability p_i. If the sum of the probabilities is less than 1, then there are worlds where this block is undefined and their overall probability is $1 - \sum_{i=1}^{n} p_i$. Such a block encodes a discrete probability distribution. An important use of c-values is to explicitly define such input probability distributions. The above distribution can be modelled in our formalism using a sum of c-values $(x = 1) \otimes t_1 + \cdots + (x = n) \otimes t_n$, where x is an integer-valued random variable and the Boolean events $x = 1, \ldots, x = n$ are by definition mutually exclusive and have probabilities $p_1, \ldots, p_n$.

The results of relational queries on tuple-independent or block-independent disjoint tables encode more complex events that are conjunctions or disjunctions of input events in the spirit of c-tables [Suciu et al. 2011]. A restricted form of c-values has been previously used to capture events representing aggregates in probabilistic value-conditioned tables [Fink et al. 2012], where the c-values adhere to algebraic laws of semi-modules representing the tensor product between a semiring, *e.g.*, the Boolean semiring, and an aggregate monoid, *e.g.*, the monoid of reals with the count operation.

Machine learning approaches also generate probabilistic data and as long as their output can be formulated as instances of our probabilistic model, ENFrame can work on them. The two approaches mentioned above, NELL (Never Ending Language Learner) and KV (Knowledge Vault) are prime examples. Furthermore, Markov chains and Bayesian networks can be encoded straightforwardly as event programs. For instance, consider a Markov chain of n nodes $o_1, \ldots, o_n$: $o_1 \rightarrow \cdots \rightarrow o_n$. We can express this using a sequence of events $\Phi_1, \ldots, \Phi_n$ as follows. The event Φ_i for node o_i for $1 < i \leq n$ is defined as

$$\Phi_i = (\Phi_{i-1} \wedge x_i^{\mathrm{t}} = \top) \vee (\neg\Phi_{i-1} \wedge x_i^{\mathrm{f}} = \top)$$

and $\Phi_1 = (x_1^{\mathrm{t}} = \top)$. This is a disjunction of two Boolean events, for the cases that o_{i-1} exists or not. Consequently, two new Boolean random variables x_i^{t} and x_i^{f} are introduced for every node o_i.

To sum up, all these existing probabilistic models are subsumed by our event language and instances of these models can readily serve as input to ENFrame programs.

3.3. From User Programs to Event Programs

In this section, we explain how to translate a user program into an event program. An event program is an imperative specification of a sequence of event declarations of the form $\langle eid \rangle \equiv \langle val \rangle$ or $\langle eid \rangle \equiv \langle bool \rangle$. The semantics of an event program is that of the set of all events declared in the program, where occurrences of event identifiers in expressions are recursively resolved and replaced by the referenced expressions.

To mirror the language for user programs, ENFrame also supports higher-order events with loops. However, loops are not necessary as they can be unfolded and events can be generated for each loop iteration. This is possible since the range for each loop is given by a constant (following our restriction for loop ranges in user programs). We require that event declarations are immutable, *i.e.*, each distinct event identifier may only be declared once.

The translation of user to event programs needs to consider:

(1) Resolving user defined functions for loading the data and the various parameters of the program;
(2) Translating mutable variables and lists in the user program to immutable events in the event program;
(3) Translating function calls such as `reduce_*`;

Loading the data. We first resolve the user defined functions for loading the data and various parameters, *e.g.*, `loadData()`, `loadParams()`, and `init()` in our example programs. By this, we populate a list of c-values of the form $\Phi_i \otimes o_i$, where Φ_i is a Boolean event with a known probability distribution (also given in the input) and o_i is usually a multi-dimensional data point (record of arbitrary arity). The parameters needed for the program are also loaded from the input; these are constants that stay the same in all possible worlds.

From Mutable Variables to Immutable Events. It is natural to reassign variables in user language programs, for example when updating k-means centroids in each iteration based on the cluster configuration of the previous iteration. In contrast, events in event programs are *immutable*, *i.e.*, can be assigned only once. The translation from the user program to the event program generates for each user program variable M a sequence of unique event identifiers whose lexicographic order reflects the sequence of assignments of M. The idea is to first unfold the nested loop blocks of the user program (the loop ranges are restricted to allow this). We then maintain a counter for each distinct variable symbol M and each assignment of that variable.

Since lists in user programs have a known fixed size, their translation is straightforward: A k-dimensional array $M[n_1], \ldots, [n_k]$, which can be encoded using lists in ENFrame, translates to $\prod_i n_i$ distinct identifiers $M^{0,\ldots,0}, \ldots, M^{n_1-1,\ldots,n_k-1}$.

Reduce operations can only be applied to anonymous lists created via list comprehension. The expression `reduce_and([`Ψ `for` v `in range(`b`,` e`) if` Φ`]` is translated to the Boolean event $\bigwedge_{v=b}^{e-1} \Phi \wedge \Psi$. Similarly, `reduce_or` translates to $\bigvee$, `reduce_sum` to $\sum$, and `reduce_mult` to $\prod$. The expression `reduce_count([`Ψ `for` v `in range(`b`,` e`) if` Φ`])` translates to the event $\sum_{v=b}^{e-1} \Phi \otimes 1$.

3.4. Event Networks

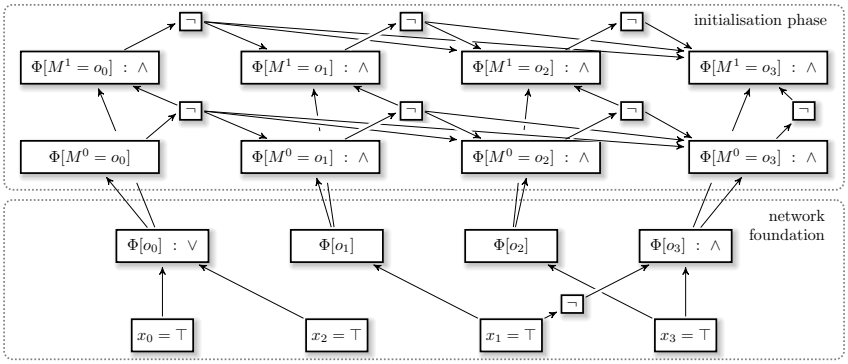
We represent the interconnected events from an event program in an *event network*. This is a graph representation of the event program, in which nodes are random variables, Boolean connectives, comparisons, aggregates, and c-values. The advantage of this representation is that common sub-expressions are shared among the events, which means less storage needed for the event program and also shared probability computation.

Example 3.2. Consider the list of four input events $\Phi[o_0] \otimes o_0, \ldots, \Phi[o_3] \otimes o_3$, where

$$\Phi[o_0] = (x_0 = \top \vee x_2 = \top) \qquad \Phi[o_1] = (x_1 = \top)$$
$$\Phi[o_2] = (x_3 = \top) \qquad \Phi[o_3] = (x_1 = \bot \wedge x_3 = \top).$$

The following event network depicts in the bottom layer the graph representation of the Boolean events associated with the four objects $o_0, \ldots, o_3$. The upper layer depicts (an excerpt of) the initialisation phase of 2-medoids clustering for these objects using the program in Figure 4. A directed edge from a node A to a node B means that A is used to express B. For instance, the node $\Phi[M^0 = o_0]$, which defines the Boolean event stating that the object o_0 is the medoid M^0, is only pointed at by Φ_0, which means that $\Phi[M^0 = o_0] = \Phi[o_0]$ and that we initialise the medoid M^0 with o_0; clearly, the medoid M^0 can be only be initialised with o_0 in those worlds where $\Phi[o_0]$ is true. The event $\Phi[M^0 = o_1]$ states that the object o_1 is the initial medoid M^0 and is defined by the network fragment rooted at its corresponding node: $\Phi[M^0 = o_1] = (\neg\Phi[M^0 = o_0] \wedge \Phi[o_1])$. This reads as follows: The medoid M^0 can be initialised with o_1 in those worlds where it is not initialised with o_0 and where o_1's condition $\Phi[o_1]$ is satisfied. The remaining initialisations follow a similar pattern.

There are further layers for each assignment and update steps of the k-medoid clustering algorithm; these are not depicted here for lack of space.



□

Section 4 introduces probability computation algorithms for event programs represented as event networks.

4. PROBABILISTIC INFERENCE FOR EVENT NETWORKS

The inference problem is #P-hard already for simple classes of events and uniform distributions, such as positive bipartite propositional formulas in disjunctive normal form over Boolean random variables with uniform probability distributions [Provan and Ball 1983]. In ENFrame, we need to compute probabilities of a large number of interconnected events that are beyond propositional ones. We approach this problem with three complementary techniques, which are presented in this section: (1) bulk-compilation of an event network into a decomposition tree that allows for efficient probability computation, (2) approximation techniques to prune large, not-yet-explored fragments of the tree, and (3) a recipe for concurrent compilation that assigns disjoint fragments of the tree to workers running on distinct cores of a machine. These techniques bring increasingly larger performance benefits, as presented in Section 5.

4.1. Bulk-compilation of event networks

Recall that all events of a program are represented by an event network, which is a graph representation shared between the parse trees of the events in the program. The goal of the ENFrame probabilistic inference is to compute exact or approximate probabilities for the top nodes in the network; we subsequently refer to these nodes as *targets*. These nodes represent events such as "objects o_i and o_j occur in the same cluster" or "object o_i is the closest to object o_j."

The bulk-compilation procedure is based on Boole's expansion theorem [Boole 1854], which is also referred to as Shannon expansion [Shannon 1949] or simply decomposition: select an input Boolean random variable x and partially evaluate a Boolean target Φ to $\Phi|_{x=\top}$ by setting x to *true* ($\top$) and to $\Phi|_{x=\bot}$ by setting x to *false* ($\bot$). Then, we have that

$$\Phi = (x = \top) \wedge \Phi|_{x=\top} \vee (x = \bot) \wedge \Phi|_{x=\bot}$$

and the probability of Φ is defined by

$$P[\Phi] = P[x = \top] \cdot P[\Phi|_{x=\top}] + P[x = \bot] \cdot P[\Phi|_{x=\bot}].$$

where $\Phi|_{x=\top}$ and $\Phi|_{x=\bot}$ are simpler events since they do not contain occurrences of x. Furthermore, standard simplifications involving constants are applied to $\Phi|_{x=\top}$ and $\Phi|_{x=\bot}$: $\top \wedge \phi = \phi$, $\bot \vee \phi = \phi$, $\top \vee \phi = \top$, and $\bot \wedge \phi = \bot$ for any Boolean event ϕ.

This decomposition procedure can be trivially generalised to networks of discrete events. Given a network Ψ and a random variable x with a finite set D of possible (Boolean or real-valued) outcomes, we have that

$$\Psi = \sum_{i \in D} (x = i) \cdot \Psi|_{x=i}$$

where the sum is interpreted as disjunction for Booleans and as standard addition for reals. The probability of an outcome v for Ψ is then given by:

$$P[\Psi = v] = \sum_{i \in D} P[x = i] \cdot P[\Psi|_{x=i} = v].$$

By repeating this decomposition and thus incrementally eliminating variables, we eventually end up with events that are constants and have probability 1. The trace of this repeated decomposition is a tree, which we call a *decomposition tree*; for Boolean variables, this would be a decision tree. A decomposition tree for an event network Ψ is defined by Ψ and the order of elimination of its variables along each branch. Different decomposition trees may be obtained for different variable elimination orders. In worst case, the tree has one branch for each possible assignment of the variables and there can be exponentially many such assignments in the number of variables. The probability of an event Φ is the probability of any of its decomposition trees, which is the sum of the probabilities of its branches. Since a branch is a (possibly partial) assignment ν of random variables, its probability is the product of the probabilities of the individual variable assignments in ν.

All our inference algorithms rely on decomposition trees of event networks, though with two key improvements of practical relevance: the decomposition tree is not materialised and the (partially evaluated) networks $\Phi|_{x=i}$ are not constructed explicitly. Algorithm 1 sketches the main idea behind these algorithms. Exact inference is obtained by calling the main procedure with error $\epsilon = 0$ and would correspond to a simplified algorithm without the framed, blue pseudocode. We next explain this simpler algorithm and later return to approximate inference.

Instead of materialising the tree, we explore it in some order, *e.g.*, in depth-first order in Algorithm 1, and collect the probabilities of all visited branches as well as record for each event Φ and outcome v the probability $L(\Phi = v)$ of those visited branches that satisfy $\Phi = v$; initially, $L(\Phi = v) = 0$ for any possible outcome[1] v of Φ. At any time, $L(\Phi = v)$ and $1 - \Sigma_{v' \in \text{outcomes}(\Phi), v' \neq v} L(\Phi = v')$ represent lower and respectively upper bounds on the probability of $\Phi = v$. These bounds eventually converge to the exact probability when the tree is explored completely. If approximate probabilities suffice, then the bounds need not meet and potentially large fragments of the tree remain unexplored. This can improve the performance of bulk-compilation significantly.

Instead of constructing the (partially-evaluated) network $\Phi|_{x=i}$ explicitly, we record the value of each node in the network for any variable assignment $x = i$. This together with the original network uniquely defines the network for $\Phi|_{x=i}$. The process of computing this information is called *masking*, cf. Algorithm 2. We perform masking by traversing the original network bottom-up and recording which nodes are masked, *i.e.*, evaluated to constants given the values of their children. Initially, all nodes in

[1]The implementation only records lower bounds for outcomes that are actually computed during compilation and not for all permissible values in the domain as that would be impractical.

ALGORITHM 1: Inference algorithm for event networks (framed code is for approximation).

BULKCOMPILE(NETWORK D [,ERROR ε])
  foreach $n \in D$ **do**
    $M[n].val \leftarrow$ unknown ▷ initial value of n is unknown, which is not a possible outcome
    $M[n].lval \leftarrow \min(n)$ ▷ for $\mathbb{R}$-valued nodes n, initial lval is the lowest value that n can get
    $M[n].uval \leftarrow \max(n)$ ▷ for $\mathbb{R}$-valued nodes n, initial uval is the highest value that n can get
  end
  foreach $\Phi \in$ *targets*(D) **do**
    foreach $v \in$ *outcomes*(Φ) **do** $L(\Phi = v) \leftarrow 0$ ▷ initial probability lower bound of 0 for target Φ
    [$B[\Phi] \leftarrow 2\varepsilon$] ▷ initial error budget for target Φ
  end
  DFS($D, M, \emptyset$ [, B]) ▷ start exploring the decomposition tree with empty assignment of probability 1

DFS(NETWORK D, MASKS M, ASSIGNMENT ν [, ERROR BUDGETS B])

  if $\forall \Phi \in$ *targets*$(D) : \ \Sigma_{v \in outcomes(\Phi)} L(\Phi = v) \leq 2\varepsilon$ **then**
    return B ▷ approximation reached for all targets; we are done
  end
  if $\forall \Phi \in$ *targets*$(D) : \ (B[\Phi] \geq \Pr[\nu]$ **or** $M[\Phi].val \neq unknown)$ **then** ▷ sufficient error budget to trim branch ν
    foreach $\Phi \in$ *targets*(D) **do** ▷ decrease available budget for unmasked targets Φ
      if $M[\Phi].val = unknown$ **then** $B[\Phi] \leftarrow B[\Phi] - \Pr[\nu]$
    end
    return B ▷ return updated error budgets
  end
  $R \leftarrow 0$ ▷ initial residual error budget from one branch to the next
  $x \leftarrow nextVariable(D, \nu)$ ▷ select next variable to eliminate
  foreach $i \in$ *outcomes*(x) **do**
    $\nu' \leftarrow \nu \cup \{x = i\}$ ▷ extend the current assignment
    $M_{\nu'} \leftarrow$ copy(M) ▷ get a working copy of the current mask
    $M_{\nu'}[x].val \leftarrow i$ ▷ extend the current mask with the assignment for x
    $M_{\nu'} \leftarrow$ MASK$(D, M_{\nu'}, \nu', x,$ NULL$)$ ▷ propagate the mask $M_{\nu'}$ in the network D
    foreach $\Phi \in$ *targets*(D) **do** $B_{\nu'}[\Phi] \leftarrow \frac{B[\Phi]}{|\text{outcomes}(x)|} + R$ ▷ error budget for branch ν'
    [$R \leftarrow$] DFS$(D, M_{\nu'}, \nu'$ [, $B_{\nu'}$]) ▷ recurse on branch ν' and return the residual error budget R
  end
  return [R]

the network have an *unknown* value, *i.e.*, they are unmasked. Nodes that represent monotonically-increasing real-valued events (*e.g.*, conditional values and sums of non-negative numbers such as distance-sums in clustering) are masked using lower and upper bound values that will eventually converge as more variables are eliminated. Bounds on real-valued events can also help to evaluate quickly the Boolean outcomes of comparison events such as $r_1 < r_2$.

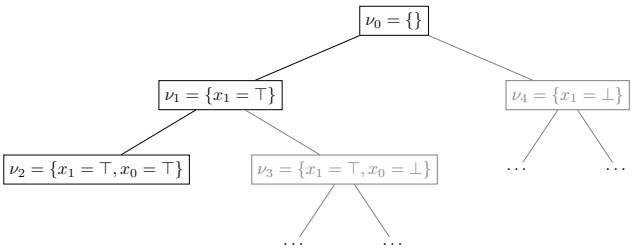
When a target Φ is eventually masked to a value v by a variable assignment ν, the probability $\Pr(\nu)$ is added to the probability lower bound of the event $\Phi = v$, which also means that it is subtracted from the probability upper bounds for all events $\Phi = v'$ with $v' \neq v$. If one or more targets are left unmasked, a next variable x' is eliminated and the masking process is repeated with $\nu' = \nu \cup \{x' = c\}$, where c is a possible outcome of x'. In case ν already masked the target Φ, then its extension ν' cannot contribute anymore to the probability Φ (as it already did via ν), although it may contribute to further targets. Once all targets are masked, we backtrack and select a different outcome for the most recently chosen variable whose outcomes are not exhausted. If all branches of the tree have been investigated, the probability bounds of the targets have necessarily converged and we terminate.

ALGORITHM 2: Masking an event network.

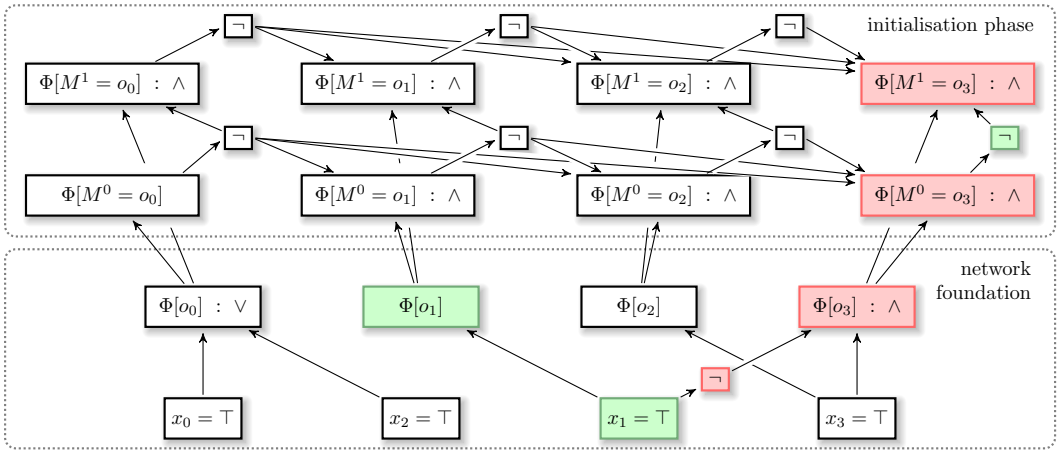
```
MASK(NETWORK D, MASKS M, VARIABLE ASSIGNMENT ν, NODE n, CHILD c)
  if M[n].val = unknown then                                              ▷ n is not yet masked
    switch nodetype(n) do
      case ¬  : M[n].val ← not M[c].val                                   ▷ M[c].val ∈ 𝔹, M[n].val ∈ 𝔹
      case ∧  :                                                           ▷ M[n].val ∈ 𝔹
        if ν ⊭ (c = M[c].val) then M[n].val ← ⊥
        else if (∀c ∈ children(n) : ν ⊨ (c = M[c].val)) then M[n].val ← ⊤
      end
      case ∨  :                                                           ▷ M[n].val ∈ 𝔹
        if ν ⊨ (c = M[c].val) then M[n].val ← ⊤
        else if (∀c ∈ children(n) : ν ⊭ (c = M[c].val)) then M[n].val ← ⊥
      end
      case ⊗  :                                                           ▷ c-value: M[n].val, v ∈ ℝ
        Let n = Φ ⊗ v                                                     ▷ Φ and v are children of n
        if ν ⊨ Φ then                                                     ▷ update lower and upper bound and value of c-value
          M[n].lval ← v;   M[n].uval ← v;   M[n].val ← v
        end
      end
      case Σ  :                                                           ▷ sum of c-values: M[n].val ∈ ℝ
        Let n = c_1 + ... + c_k                                           ▷ c_1, ..., c_k are c-values and children of n
        M[n].lval ← M[c_1].lval + ... + M[c_k].lval                       ▷ update lower bound of n
        M[n].uval ← M[c_1].uval + ... + M[c_k].uval                       ▷ update upper bound of n
        if M[n].lval = M[n].uval then M[n].val ← M[n].lval                ▷ lower and upper bound coincide
      end
      case <  :                                                           ▷ M[n].val ∈ 𝔹, Similarly for Θ ∈ {<, ≤, >, ≥, =}
        Let n = (c_l < c_r)                                               ▷ c_l and c_r are the left and resp. right children of n
        if M[c_l].uval < M[c_r].lval then
          M[n].val ← ⊤
        else
          if M[c_l].lval ≥ M[c_r].uval then M[n].val ← ⊥
        end
      end
    endsw
    if n ∈ targets(D) and M[n].val ≠ unknown then                         ▷ n is a target and has just been masked
      L(n = M[n].val) ← L(n = M[n].val) + Pr[ν]                           ▷ add probability mass to the masked node
    end
  end
  foreach p ∈ parents(n) do                                               ▷ propagate mask to yet unmasked parents of n
    if M[p].val = unknown then M ← MASK(D, M, ν, p, n)
  end
  return M
```

Variable order for decomposition trees. Decomposition trees may not be necessarily balanced and the order in which the variables are processed has a significant influence on their size and height. For instance, the tree for the Boolean event $\Phi = (x_0 = \top \vee x_1 = \top \vee x_2 = \top) \wedge x_2 = \top$ has height three, if constructed using the variable order x_0, x_1, x_2 for all branches, and height one if x_2 comes first in the order (since Φ is equivalent to $x_2 = \top$). ENFrame employs a heuristic to find a good variable ordering that aims to choose a next variable x' such that it affects (and thus partially evaluates) as many events as possible. Whenever a node n is reached during masking, yet it cannot be masked to a constant, we initiate a back-propagation procedure (top-down in the network) via its unmasked descendants to select the next variable to eliminate. We select a variable that is reached most times during back-propagation from all such unmasked nodes.

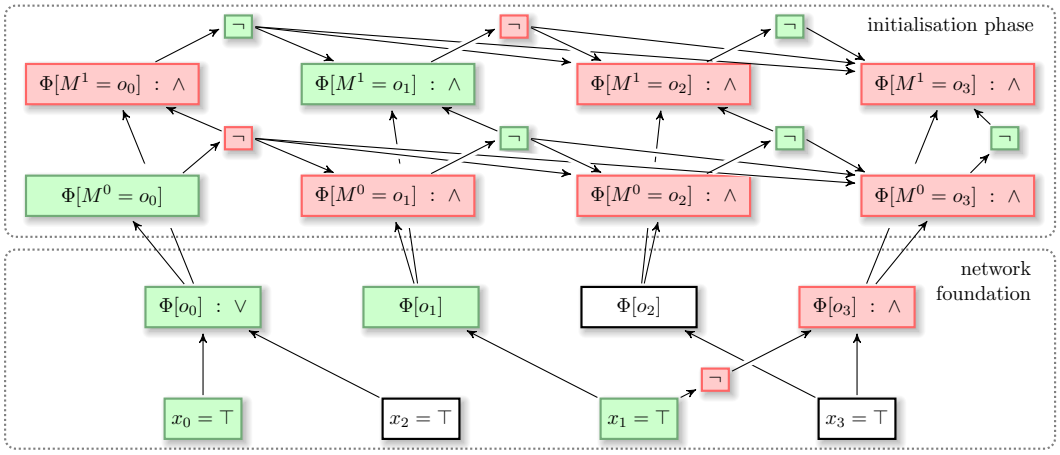
Example 4.1. Recall the event network from Example 3.2, consisting of the foundation and initialisation phase for k-medoids clustering and assume the medoid selection events as targets. We next illustrate the consecutive masking for assignments ν_1 and ν_2 from the following decomposition tree with the variable order x_1, x_0, x_2, x_3:



After masking for ν_1, we obtain the following network:



The assignment $\nu_1 = \{x_1 = \top\}$ propagates to the node $\Phi[o_1]$ masking it *true* (green background) and to its parents, which remain unmasked since they are conjunctions with multiple children not yet masked. The assignment is also propagated to its second parent node, which is a negation and gets masked *false* (red background). This masking further propagates upwards to conjunctions that also get masked *false*. As a consequence, the object o_3 cannot be an initial medoid to any cluster in the possible worlds defined by the assignment ν_1. We next extend the assignment ν_1 to ν_2 with $x_0 = \top$ and obtain the following masked network:



The propagation of the assignment $x_0 = \top$ triggers a wave of new masks in the network, which results in all initial medoid nodes being masked. It is now established that, in the possible worlds defined by ν_2, the objects o_0 and o_1 are the initial medoids M^0 of cluster C_0 and respectively M^1 of cluster C_1.

In this particular example, there is no need to further explore the tree branch by extending ν_2: all targets (*i.e.*, the initial medoid nodes) have been masked, the assignments of x_2 and x_3 (and, by extension, the existence of o_2) are irrelevant. When compared with the naïve approach of running the user program in each world, our decomposition-based approach can be much faster as it shares computation across worlds: all four worlds defined by ν_2, *i.e.*, all four possible assignments to x_2 and x_3, yield the same medoids.

The exploration of the decomposition tree then continues with $\nu_3 = \{x_1 = \top, x_0 = \bot\}$ and the masking of ν_2 is discarded.

□

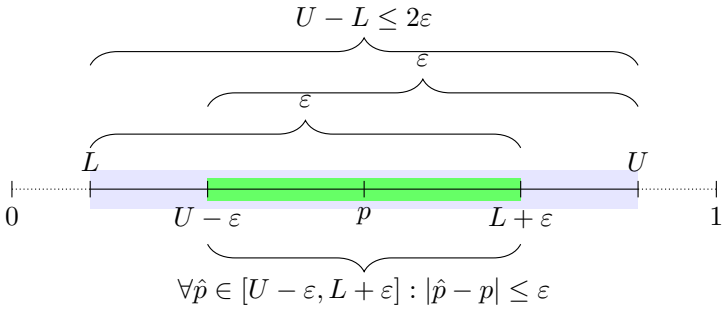
4.2. Approximate inference with error guarantees

We previously explained how to compute the exact probabilities of targets in an event network. Algorithm 1 can also compute approximate probabilities with error guarantees. Recall that while exploring the decomposition tree, we add the probabilities of the visited branches to the lower bounds of targets. For each target Φ and possible outcome v_i of Φ, Algorithm 1 incrementally refines lower bounds for $\Phi = v_i$. As explained before, $L(\Phi = v_i)$ and $U(\Phi = v_i) = 1 - \Sigma_{1 \leq j \leq l, j \neq i} L(\Phi = v_j)$ represent lower and respectively upper bounds on the probability of $\Phi = v_i$. At any time we thus have correct intervals for the probabilities of these targets. We can stop the exploration once the bounds of all targets are sufficiently tight.

We use the following notion of approximate probability distribution.

Definition 4.2. Given a fixed error $0 \leq \varepsilon \leq 1$ and an event Φ with probability distribution $(v_1 : p_1, \ldots, v_n : p_l)$ where $p_i = P[\Phi = v_i]$, an absolute ε-approximation is a distribution $(v_1 : \hat{p_1}, \ldots, v_n : \hat{p_n})$ such that $\forall 1 \leq i \leq l : p_i - \varepsilon \leq \hat{p}_i \leq p_i + \varepsilon$. □

We can further adapt Definition 4.2 to our needs and derive a relationship between the pair (L, U) of lower and upper bounds for a given event $\Phi = v_i$ and the given error ε. We next depict this relationship [Olteanu et al. 2010]:



As depicted, any probability $\hat{p}$ in the interval $[U - \varepsilon, L + \varepsilon]$ is an ε-approximation of the true probability p: indeed, when $U - L = 2\varepsilon$ and thus $\hat{p} = U - \varepsilon = L + \varepsilon$ at one extreme, then $\hat{p}$ is not further than ε away from p, which sits between L and $L + \varepsilon$ or U and $U - \varepsilon$; likewise, when $U - L = 0$ and thus $\hat{p} = U = L$ at the other extreme, then $\hat{p} = p$. When lifting this condition to the entire distribution of Φ, we obtain that an

absolute ε-approximation can be defined by any distribution $(v_1 : \hat{p_1}, \ldots, v_n : \hat{p_n})$ such that $\forall 1 \leq i \leq n : U(\Phi = v_i) - \varepsilon \leq \hat{p}_i \leq L(\Phi = v_i) + \varepsilon$.

We thus need to run the algorithm until $U(\Phi = v_i) - L(\Phi = v_i) \leq 2\varepsilon$ for each target Φ; once this condition is achieved for any outcome v_i of Φ, it will also be achieved by all other outcomes of Φ, since contributions to the lower bound $L(\Phi = v_i)$ must be subtracted from the upper bounds $U(\Phi = v_j)$ of all other outcomes v_j for $j \neq i$.

We denote by $B[\Phi] = 2\varepsilon$ the *error budget* of Φ. Algorithm 1 trims unexplored fragments of the decomposition tree whose probability mass is less than the current error budget. With every such trimming, the error budget decreases accordingly.

There exist multiple strategies for investing this error budget for every target. At each branching point, *i.e.*, when choosing outcomes for a new variable x to eliminate, Algorithm 1 evenly divides the current error budget over all branches, *i.e.*, over all possible outcomes of x. Before descending along a branch ν, we first check whether the approximation precision has been reached. If not, we check whether the current branch need not be explored since the upper bound $\Pr[\nu]$ on its probability is smaller than the current error budget. If this is the case, then we can just discard that branch and subtract its upper bound $\Pr[\nu]$ from the error budget. Any residual error budget R is added to the budget of the next branch.

We denote by BALANCED this strategy for investing the error budget. To better understand it, we contrast it with two simpler strategies (see the experiments in Section 5): GREEDY, an eager strategy that uses the error budget as soon as possible; and POSTPONED, a lazy strategy that saves the error budget until the end and uses it to trim the last branches in the decomposition tree.

GREEDY is effective for unbalanced decomposition trees with long branches on the left-hand side since it eagerly cuts such branches that would otherwise provide a very small probability mass (recall that the longer a branch the smaller its probability, which is the product of the variable assignments along the branch). This strategy can also benefit from prioritising for each variable its outcomes with small probability over the other outcomes as it again leads to branches of small probability first.

POSTPONED spends the error budgets by pruning the last (rightmost) branches of the decomposition tree. It works well for unbalanced trees with relatively short left branches and longer right branches. Such trees can be obtained for instance for positively correlated events, such as when each event is a disjunction $x_1 = \top \vee \cdots \vee x_k = \top$. The branches representing satisfying assignments are of increasing length and decreasing probability: $\{x_1 = \top\}, \{x_1 = \bot, x_2 = \top\}, \ldots, \{x_1 = \bot, \ldots, x_{k-1} = \bot, x_k = \top\}$. POSTPONED performs poorly if the decomposition tree is deeper on the left than on the right (reverse mirroring the poor scenario for GREEDY).

Both GREEDY and POSTPONED perform rather poorly on balanced trees, such as for sets of conditional events and for events that are grouped into sets such that any two events are mutually exclusive within a set and independent across sets. BALANCED strikes a balance between GREEDY and POSTPONED by distributing the error budget evenly over the branches of the tree. Other approximation techniques can be added to ENFrame, *e.g.*, the randomised ε-approximation Oracle average, and probability computation using top-p most probable worlds.

Example 4.3. Consider the decomposition tree and distribution of error budgets illustrated in Figure 8. Furthermore, assume that there is one target $\Phi = (x_0 = \top \wedge x_1 = \top) \vee x_0 = \bot$, and $P[x_0 = \top] = 0.9, P[x_1 = \top] = 0.05, \varepsilon = 0.1$.

We start by initialising the lower bounds $L[\Phi = \top] = 0$ and $L[\Phi = \bot] = 0$. Algorithm 1 explores the decomposition tree as following:

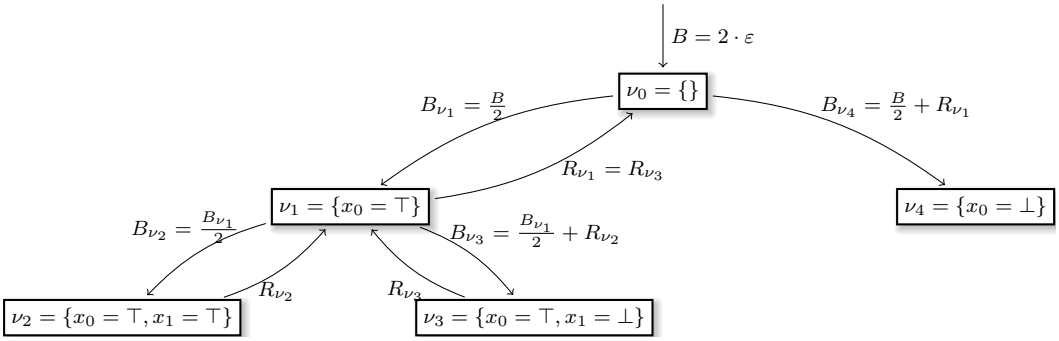


Fig. 8. Schematic error budget distribution in a decomposition tree for a Boolean event $\Phi = (x_0 = \top \wedge x_1 = \top) \vee x_0 = \bot$ as done by BALANCED. Forward (downwards) edges indicate budget assignment to a child node, back edges represent the residual error budget returned to parent nodes.

ν_0: The probability of the tree rooted at this node is $\Pr[\nu_0] = 1$ by definition. The budget $B_{\nu_0} = 2\varepsilon = 0.2$ is not sufficient to prune this tree. The assignment $\nu_0 = \{\}$ does not satisfy Φ either. We choose to eliminate the variable x_0:

- ν_1: The budget $B_{\nu_1} = \frac{B_{\nu_0}}{2} = 0.1$ is not sufficient to prune the subtree rooted at ν_1, since $\Pr[\nu_1] = 0.9$. The assignment ν_1 does not satisfy Φ. We next eliminate the variable x_1:
  - ν_2: The budget $B_{\nu_2} = \frac{B_{\nu_1}}{2} = 0.05$ is sufficient to prune the subtree rooted at ν_2, since $\Pr[\nu_2] = 0.9 \cdot 0.05 = 0.045$. The branch is pruned, and the residual budget $R_{\nu_2} = B_{\nu_2} - \Pr[\nu_2] = 0.005$ is returned. Under ν_2, Φ is satisfied so instead of using the error budget to prune this branch, we could have contributed with the quantity $\Pr[\nu_2]$ to $\Pr[\Phi]$. This was indeed our initial approach, i.e., first check (un)satisfiability of the event and then use the error budget. However, for large event networks, such as those we used in the experiments, masking is the most expensive operation and required to check (un)satisfiability. We therefore use the error budget, if possible, before masking.
  - ν_3: The budget $B_{\nu_3} = \frac{B_{\nu_1}}{2} + R_{\nu_2} = 0.05 + 0.005 = 0.055$ is not sufficient to prune the subtree rooted at ν_3 since $\Pr[\nu_3] = 0.855$. The assignment ν_3 falsifies Φ, therefore $L[\Phi = \bot]$ is increased by $\Pr[\nu_3]$ and thus the upper bound of $U[\Phi = \top]$ is decreased to $1 - 0.855 = 0.145$. The residual error budget $R_{\nu_3} = B_{\nu_3} = 0.055$ is returned.

  The residual error budget R_{ν_1} is equal to the residual budget of its right child: $R_{\nu_1} = R_{\nu_3} = 0.055$. Observe that this is equal to the original budget B_{ν_1} for the subtree rooted by ν_1, minus the budget used in this subtree (0.045, for pruning ν_2): $R_{\nu_1} = B_{\nu_1} - 0.045 = 0.1 - 0.045 = 0.055$.
- ν_4: The budget $B_{\nu_4} = \frac{B_{\nu_0}}{2} + R_{\nu_1} = 0.1 + 0.055 = 0.155$ is sufficient to prune the subtree rooted at ν_4 since $\Pr[\nu_4] = 0.1$. The branch is pruned, the residual error budget is irrelevant as this was the last branch of the tree. The algorithm finishes with probability bounds $L[\Phi = \top] = 0$ and $U[\Phi = \top] = 0.145$; they also translate into the probability bounds $L[\Phi = \bot] = 0.855$ and $U[\Phi = \bot] = 1$.

We thus have that any probability in the interval $[0.145 - 0.1, 0 + 0.1] = [0.045, 0.1]$ is within 0.1 from the true probability $P[\Phi = \top] = 0.045 + 0.1 = 0.145$. The algorithm would actually have terminated after the assignment ν_3 resulted in a sufficiently tight pair of probability bounds: $U(\Phi = \top) - L(\Phi = \top) = 0.145 \leq 2 \cdot \varepsilon = 0.2$.

□

4.3. Concurrent inference

In this section we show how the inference algorithms introduced in the previous sections can be distributed on many-core architectures. The key idea is to split the task of exploring the decomposition tree in subtasks that can be executed concurrently by threads on different cores of the underlying hardware.

A major challenge when introducing concurrency to algorithms is dealing with thread synchronisation [Dijkstra 1965; Pacheco 2011], critical sections of code that require mutual exclusion amongst the threads. Decomposition-based probabilistic inference is naturally suitable for concurrent computation: the branches $(\nu_1, \ldots, \nu_k)$ rooted at a node ν can be explored independently of each other. However, ENFrame's approximation algorithms introduce a dependency between tree branches: the error budget of ν_j's branch depends on the residual error budgets of the previous branches. Synchronisation of the threads that process such branches would completely undo any possible gain from multi-core processing, as threads would end up spending most time waiting for each other. By modifying the way error budgets are calculated, the dependency between threads can be broken – this gives rise to the BALANCED-C algorithm for concurrent approximate probabilistic inference.

We first introduce notation useful to define our strategy for concurrently spending the error budget of a target Φ. This strategy assumes the decomposition tree for Φ be balanced, *i.e.*, it has depth given by the number $|\mathbf{X}|$ of variables in Φ and also $k^{|\mathbf{X}|}$ leaves, where k is the maximum domain size for a variable in Φ. This simplification makes it easy to distribute the error budget over several threads in a way that does not require too much synchronisation.

— Let $\rho[\Phi]$ denote the total remaining error budget for target Φ, *i.e.*, initially $\rho[\Phi] = B[\Phi] = 2\varepsilon$ and is reduced whenever some of Φ's error budget is used.
— Let π denote the progress of the exploration of the decomposition tree in terms of the fraction of leaf nodes that have been considered (pruned or visited, $0 \leq \pi \leq 1$). The value of π is continuously updated.
— Let also $d(\nu)$ denote the depth of branch ν in the decomposition tree.
— The size of the subtree rooted at ν is then defined as $\mathit{treesize}(\nu) = k^{|\mathbf{X}|-d(\nu)}$, which is an upper bound on the number of leaves in the full subtree rooted at ν.
— The number of leaves that still need to be considered depends on the progress π. An upper bound is given by $\lambda = (1-\pi) \cdot k^{|\mathbf{X}|}$.
— Furthermore, σ_ν expresses the size of the subtree of ν relative to λ: $\sigma_\nu = \frac{\mathit{treesize}(\nu)}{\lambda}$.
— Then, the error budget $B_\nu[\Phi]$ available to a node rooted at ν for approximating Φ at any time during the bulk-compilation procedure is defined as follows:

$$B_\nu[\Phi] = \rho[\Phi] \cdot \sigma_\nu = \rho[\Phi] \cdot \frac{\mathit{treesize}(\nu)}{\lambda} = \rho[\Phi] \cdot \frac{k^{|\mathbf{X}|-d(\nu)}}{(1-\pi) \cdot k^{|\mathbf{X}|}} = \rho[\Phi] \cdot \frac{k^{-d(\nu)}}{1-\pi}.$$

Using the above error budget B_ν for a node rooted at any variable assignment ν, BALANCED-C computes the approximate probability distributions of all targets with an absolute error of ε. This error budget is now a closed-form expression that no longer depends directly on the result and residual error budgets of other branches of the decomposition tree. Instead, the remaining budget $\rho_\nu[\Phi]$ for a target Φ is divided over the nodes of the decomposition tree. A node defined by an assignment ν receives a budget that depends on (i) the potential maximum size of its subtree, (ii) the progress of the inference algorithm, and (iii) the remaining error budget. A few small critical sections remain:

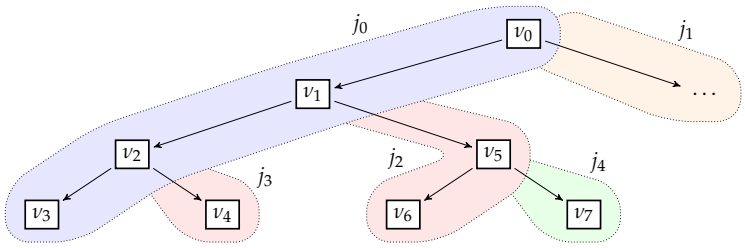


Fig. 9. Schematic illustration for concurrent processing of a decomposition tree using BALANCED-C .

— Verifying whether sufficient budget is available to prune a branch and subsequently updating the remaining error budgets ρ.
— Updating π whenever a branch is pruned or a leaf node is identified.
— Updating probability bounds during the masking process.
— Queuing and dequeueing processing jobs.

Every single one of these critical sections are limited to updating one (or a constant number) of variables, rather than containing computationally complex instructions.

BALANCED-C employs a combination of the producer-consumer pattern [Ben-Ari 1990] and the recursive split pattern for concurrent computation (sometimes referred to as the 'divide and conquer' pattern). Whenever a node defined by an assignment ν in the tree is split by some thread $\mathcal{T}_a$ to construct branches ν_1 to ν_k, all but the first of these branches are pushed onto a queue as independent jobs to be picked up by idle consumer threads $\mathcal{T}_b$ ($\mathcal{T}_a = \mathcal{T}_b$ is possible). In the meantime, $\mathcal{T}_a$ continues down the branch ν_1. A thread $\mathcal{T}_i$ becomes idle when the branch it works on is not extended further because either all targets are masked or the branch is pruned. When a thread becomes idle, it picks up the next job from the queue or waits for one to be queued. This strategy for concurrent processing is illustrated in Figure 9. The coloured jobs $j_0, \ldots, j_4$ are listed in the order in which they are generated and illustrate how the tree is split up into tasks that can be processed independently. The threads can propagate masks into the event networks simultaneously; the progress and budget variables σ and ρ are updated by all threads in synchronised code blocks.

4.4. Extensions

In the end of this section, we cover several extensions of the proposed inference algorithms: how to deal with certain data, how to encode and process iterations in event networks, and finally masking of higher-order events.

Certain data. Input certain data, *i.e.*, data that occurs in all possible worlds, is associated with events represented by Boolean random variables whose probability for true is 1. Before running the inference algorithms presented in this section, ENFrame identifies such events (by inspecting the probability distributions of all input variables) and masks the network for an assignment that maps all these variables to $\top$. If all input data is certain, then this initial masking completely evaluates the network and computes the result of the user program. If only *some* input data is certain, then this initial masking corresponds to a partial evaluation of the user program that is done once for all possible worlds.

Iterations in user programs. User programs encoding data mining tasks and the corresponding event programs often contain loops. Event networks for event programs with loops can be represented in two distinct ways.

The default representation is *unfolded*: all events that occur inside loops are explicitly represented as distinct nodes in the event network. For instance, a `for`-loop with i iterations (*e.g.*, in k-means or k-medoids clustering) and m events in every iteration leads to $i \cdot m$ events in the resulting event network.

For bounded-range loops, the repetitive events are amenable to a *folded* encoding. All events in the body of such a `for`-loop are captured in a single group of nodes in the event network. Bulk-compilation of event networks with such *iterative* groups requires to simulate looping for inference computation.

Algorithm 1 for inference and Algorithm 2 for masking assume unfolded networks. Several minor modifications are required to make them aware of folded event networks. Firstly, the masks data structure M becomes two-dimensional to allow for storing masks for all nodes n in any loop iteration i: $M[i][n]$. Secondly, to facilitate the transition between different iterations of a loop, a *loop node* type $\triangledown$ is introduced: When the current node n is such a loop node, then in Algorithm 2 its masking is propagated to its corresponding node at the next iteration. Finally, a target in an iterative group is only considered *reached* when the group has reached its final iteration. For iterative programs, a folded event network results in a smaller memory footprint for the event network. The overhead of probability computation remains however the same.

Masking of higher-order events. Higher-order events can be used to reduce the size of the event network by transferring logic from event expressions into native C++ code. Regular network nodes can be masked using Boolean or real-valued masks, following the semantics of the events. Higher-order nodes require higher-order masks, which are propagated following a distinct mechanism, which is often more complex than for the simple network nodes supported by default. This mechanism needs to be manually predefined by the programmer in native C++ code.

For instance, a single higher-order event can represent the assignment of objects to a cluster i in k-means or k-medoids, rather than explicitly representing one event for every combination of object and cluster. The mask information for such a higher order event is more complex as it needs to encode (i) which objects are definitely assigned to cluster i; (ii) objects that cannot yet be assigned due to a currently incomplete variable assignment. This information needs to be passed on to the parent nodes in the event network each time a change occurs.

ENFrame is extensible by design so new types of complex nodes can be added to event networks. Further work is however required on better understanding when to encapsulate groups of events into a higher-order node versus when to expose these events in the network. The trade-off is space, which can be orders of magnitude less in case of higher-order nodes, versus functionality, which decreases once the events are hidden behind complex nodes.

5. EXPERIMENTAL EVALUATION

This section describes an experimental evaluation of the performance of ENFrame. The focus of this evaluation is a performance benchmark of the probabilistic inference algorithms introduced in Section 4, used for two programs: k-medoids clustering and k-nearest neighbours classification.

ENFrame's output for any input program is the same as if the program would be executed in each of the possible worlds. Our focus in this section is on probabilistic inference for user programs and not on quality of data mining. As such, we do not aim to compare the quality of data mining tasks implemented in ENFrame to other single-purpose algorithms for mining of uncertain data. Such a comparison would invariably yield discriminatory associations. Firstly, ENFrame is a generic framework for expressing a variety of data mining tasks and not a specific data mining algorithm,

as investigated in the literature. Secondly, the framework's support for correlations and possible worlds semantics sets it apart from the bulk of single-purpose data mining algorithms presented in the literature, which rely on (over)simplifying independence assumptions (*e.g.*, [Chau et al. 2006; Ngai et al. 2006b], surveyed in [Volk et al. 2009]) and expected values for *e.g.*, nearest neighbours and cluster medoids. Specific solutions reported in the literature might outperform implementations of data mining algorithms in ENFrame, at the cost of producing results that are possibly inaccurate when compared to mining in every possible world, due to a fundamental difference in probabilistic semantics.

This section does, however, introduce the first steps towards quality evaluation of clustering probabilistic data. We used it to empirically verify the claims regarding the error bounds for the approximation algorithms and to compare ENFrame's approximations to the quality of probability computation in the top-p most probable possible worlds. In the following, we describe the experimental setup, summarise our findings, and present these findings in more detail.

5.1. Experimental setup

5.1.1. Data and correlations. The data used in the experimental evaluation was obtained from sensors in energy distribution networks, kindly provided by UK Power Networks (UKPN) as part of the HiPerDNO FP7 research project [Taylor et al. 2011]. The data describes network load and occurrences of *partial discharge* in energy distribution networks. Partial discharge is an electrical discharge that does not fully bridge the insulation between two conducting electrodes, and has recently been identified as one of the major causes of long-term degradation and eventual failure of cables.

In order to minimise the number of lost customer-minutes, energy distribution network operators are currently deploying sensors to monitor partial discharge activity in the distribution network to be able to act preemptively [Michel 2007; Michel and Eastham 2011]. Unfortunately, monitoring partial discharge is not a straightforward task: the phenomenon is hard to detect, sensors often report spurious measurements and are prone to failure (as are the transmission channels).

The partial discharge occurrence count is aggregated over the duration of an hour, and subsequently paired with average network load readings (or records) during that time. The resulting data set consists of 1300 records, which is used to demonstrate the framework's ability to process non-synthetic data.

To experimentally evaluate ENFrame's ability to operate on various types of correlations, the data set has been augmented with three commonly occurring correlation patterns [Agrawal et al. 2006; Sen and Deshpande 2007; Suciu et al. 2011]. All three schemes consider correlated events $\Phi(o_i)$ (or lineage) associated with records o_i that are built up using Boolean random variables and define the worlds containing o_i. We further partitioned the records in groups of four with identical events.

— *Positive* correlations, in which any two records o_i and o_j are either positively correlated or independent. Queries (union, projection, or aggregates) often produce this type of correlated events [Koch and Olteanu 2008], as does time-series data from sensor networks [Lian and Chen 2010]. In this correlation scheme, each event $\Phi(o_i)$ for a record o_i is Boolean and a disjunction of l distinct positive Boolean variable assignments, *e.g.*, $x_1 = \top \vee \cdots \vee x_l = \top$. In the experiments, we vary the number of variables v to study its effect on performance.
— *Mutually exclusive* correlations, in which the records are partitioned in *mutex* sets of cardinality (at most) m such that any two records are mutually exclusive within a set and independent across sets. Any data source with conflicting data yields this type of correlation, *e.g.*, geolocation data [Mokbel et al. 2006], optical character recognition,

and automated data extraction [Dong et al. 2009; Bleiholder and Naumann 2009]. The number of variables v depends on the number of objects and on the size of the mutex sets.

— *Conditional* correlations, which model uncertainty as a Markov chain with one node per record: $o_1 \to \cdots \to o_n$. The event $\Phi(o_i)$ of record o_i for $1 < i \leq n$ is defined as

$$\Phi(o_i) = (\Phi(o_{i-1}) \wedge x_i^{\mathrm{t}} = \top) \vee (\neg\Phi(o_{i-1}) \wedge x_i^{\mathrm{f}} = \top)$$

and $\Phi(o_1) = (x_1^{\mathrm{t}} = \top)$. This is a disjunction of two Boolean events, for the cases that o_{i-1} exists or not. Consequently, two new Boolean random variables x_i^{t} and x_i^{f} are introduced for every record o_i. The number of variables v depends on the number of objects.

The Boolean random variables used in the various correlation schemes have randomly generated probabilities for true in the range $[0.5, 0.8]$ (the case of $[0.2, 0.5]$ is symmetric). This range was chosen to prevent events with probabilities too close to (or, for approximation, within the used absolute error from) either zero or one, which are not representative for the non-trivial work regime of the proposed probabilistic inference algorithms as they can be approximated much faster than in the general case.

5.1.2. Probabilistic inference algorithms. We report on performance benchmarks for probability computation for k-medoids clustering and k-nearest neighbour classification using five algorithms:

— The sequential (*i.e.*, single-threaded) exact algorithm EXACT;
— The sequential approximation algorithms: GREEDY, POSTPONED, and BALANCED;
— The concurrent approximation algorithm BALANCED-C.

Furthermore, two reference implementations are used:

— The NAÏVE algorithm iterates over all possible worlds and executes the program in every world.
— The TOP-P algorithm iterates over the top-p most probable worlds and executes the program in each of these worlds.

The approximation algorithms were set to compute probabilities within an (absolute) error bound of $\varepsilon = 0.1$. For k-medoids clustering, the targets are the events that represent medoid selection; for k-nn, the classification events are used as targets. To facilitate comparison, k-medoids clustering experiments were run with three iterations and $k = 3$ clusters. The experiments with k-medoids used a folded event network with higher-order events.

5.1.3. Machines. With the exception of Figure 14, all experiments were carried out on an Intel Xeon X5660 (2.80GHz, 4 cores) machine with 4GB of RAM, running Ubuntu with Linux kernel 3.5. The results of thread-scalability presented in Figure 14 were obtained on an Intel Xeon E5-2690 (2.90GHz, 16 cores).

The framework was implemented in C++ and compiled using GCC 4.7.2. Each of the plots depicts averages (with min/max ranges where applicable) of five runs with randomly generated event expressions and variable probabilities.

5.2. Observations on performance

5.2.1. Sequential algorithms. Figures 10 and 11 show that all of ENFrame's inference algorithms outperform NAÏVE by up to six orders of magnitude for each data set with more than 10 variables. Furthermore, BALANCED can be up to four orders of magnitude faster than EXACT. NAÏVE is only competitive for a very small number of possible worlds, in particular as defined by up to ten Boolean random variables. For more

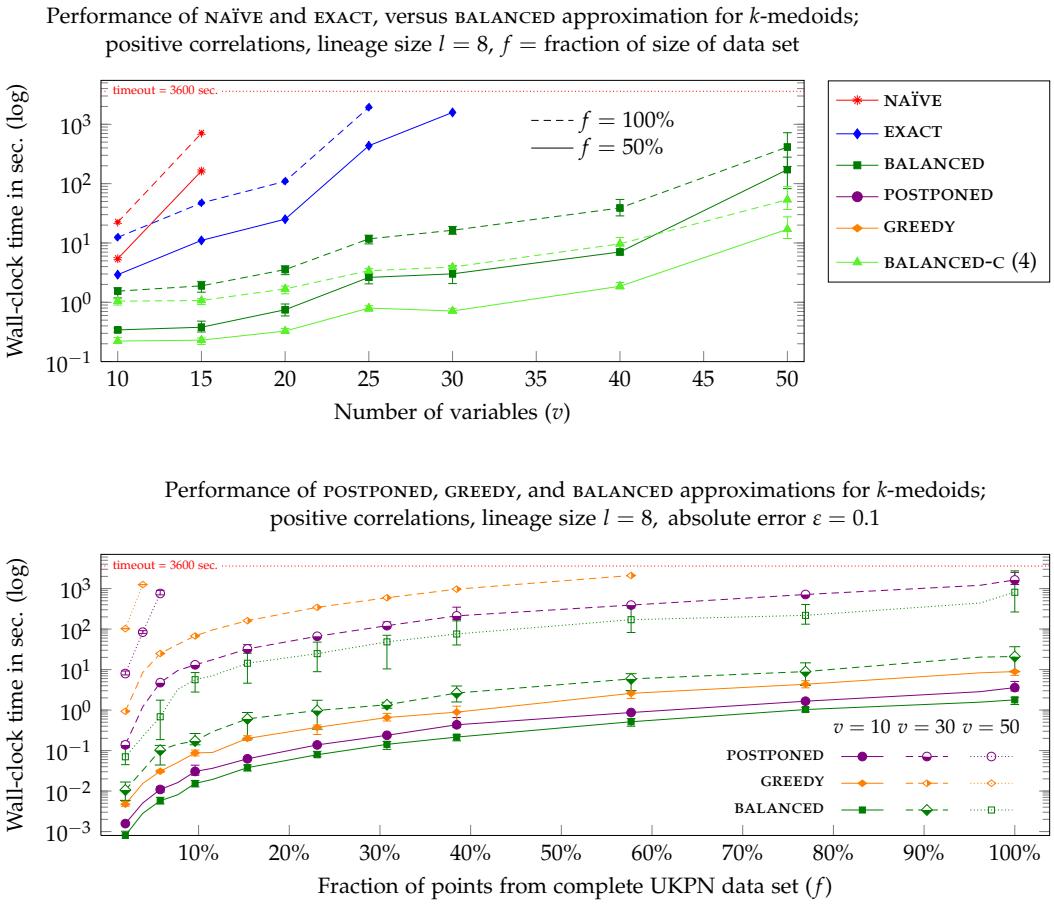


Fig. 10. Probability computation for k-medoids on positively correlated data. Top: scalability in terms of variables. Bottom: scalability of approximations in terms of size of the data set (BALANCED-C not shown for readability reasons).

worlds, ENFrame performs better as it exploits commonalities among possible worlds and executes fragments of the program once for many worlds. NAÏVE results in a time-out for over 20 variables ($\approx$ 1 million possible worlds) regardless of the correlation scheme.

The reason why the approximation algorithms outperform EXACT is as follows. For a given depth d, there are up to 2^d nodes in the decomposition tree that contribute to the probability mass of values of targets in the event network. The contributed mass decreases exponentially with an increase in depth, reflecting that most nodes in the tree only contribute a small fraction of the total mass. Depending on the desired error bound, a shallow exploration of the tree may be enough to obtain a sufficiently large probability mass.

Among the approximation algorithms, BALANCED performs best; it outperforms EXACT for clustering (Figures 10 and 11) and classification (Figures 12 and 13) by up to four orders of magnitude since it does not traverse the decomposition tree to its full depth. The other two methods (GREEDY and POSTPONED, only used for k-medoids clus-

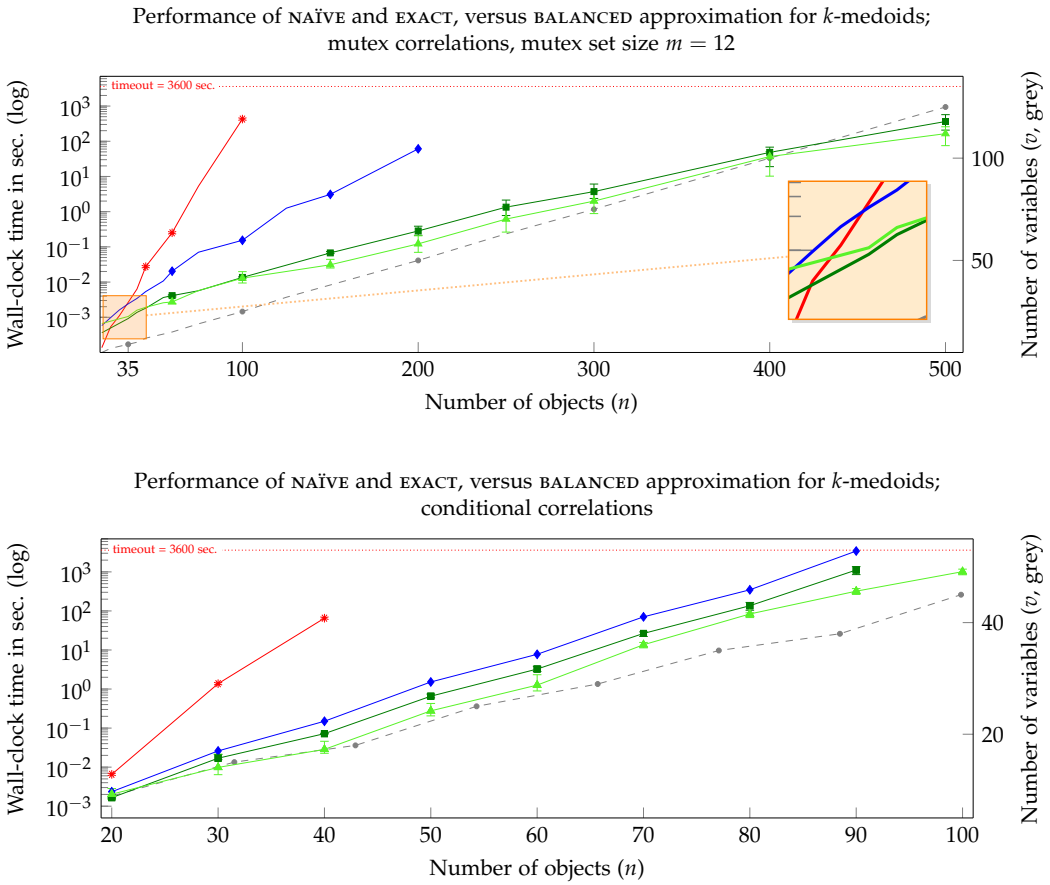


Fig. 11. Probability computation for k-medoids on data with mutually exclusive (top) and conditional correlations (bottom). GREEDY and POSTPONED overlap with EXACT and are not shown. Grey dashed line indicates number of variables v on right y-axis (depends on number n of objects). Legend: see Figure 10.

tering in the experiments) use the available error budget to respectively cut the first and last branches, while exploring other branches in full depth.

POSTPONED performs remarkably well for positive correlations, because the decomposition tree for the disjunctive events is unbalanced. The left branches of the tree correspond to assignments of variables to *true*, which immediately satisfies the disjunctive events, and thus allows for targets to be masked quickly. Further to the right of the tree, branches correspond to assignments of variables to *false*. More variables need to be set to fully evaluate a disjunctive event, thus leading to longer branches. POSTPONED invests the error budget towards the end of its exploration of the decomposition tree and can therefore prune the deep right branches whilst maintaining the ε-approximation. The decomposition trees for the mutex and conditional correlation schemes are more balanced. As a result, the performance of POSTPONED and GREEDY degrades and is within a factor two of EXACT. To improve the readability of the plots, POSTPONED and GREEDY are not shown in Figure 11.

5.2.2. Concurrent algorithm. By distributing the probability computation over multiple CPU cores, ENFrame's performance improves significantly. Figures 10, 11, 12, and 13

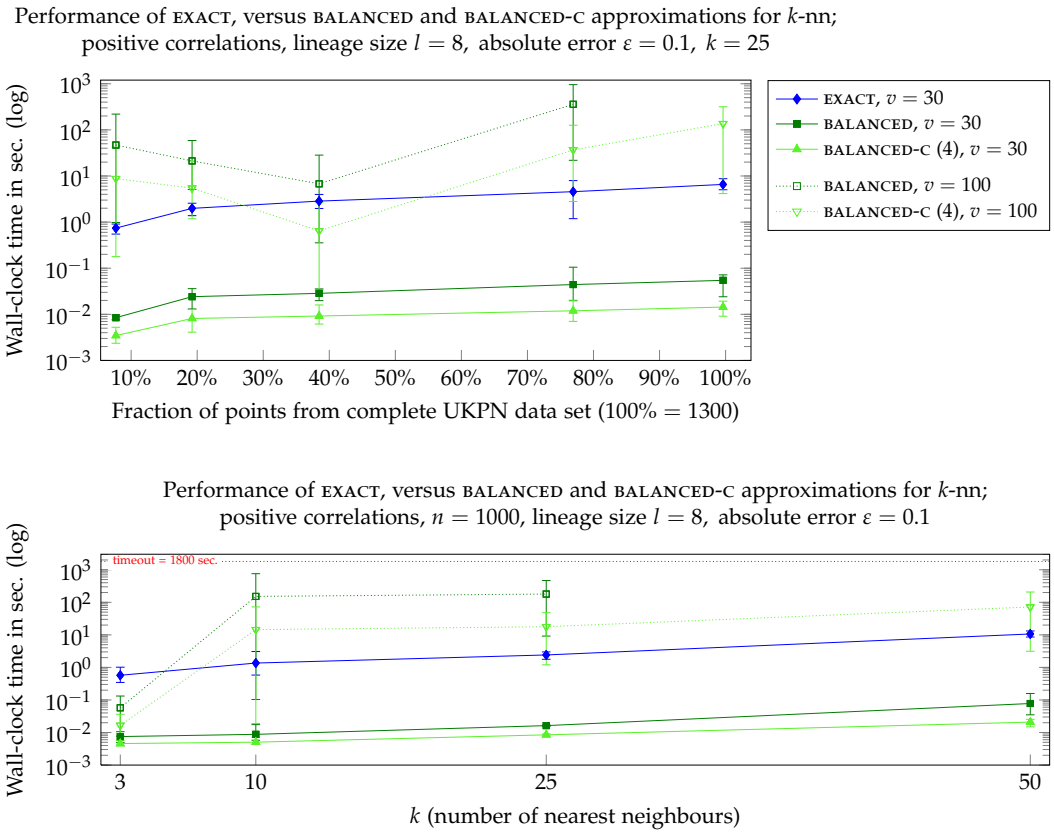


Fig. 12. Probability computation for k-nearest neighbour on positively correlated data, and varying values for n (number of records), v (number of variables) and k.

show timings for BALANCED-C, the algorithm for concurrent probability computation. Regardless of the type of program (clustering or classification), a smaller number of variables yield fewer possible worlds and hence a more shallow decision tree. Under those circumstances, concurrent probability computation only yields a factor 2-3 improvement over BALANCED due to the fact that a sufficiently large number of jobs cannot be generated quickly enough to fully use all four available workers (threads). However, as soon as the number of variables approaches 30, the efficiency of the concurrent algorithm becomes clear: for almost all data sets and algorithms, regardless of their parameters, the improvement over BALANCED ranges from a factor 3.5 to a factor 10 (*e.g.* k-medoids in Figure 10 for $v = 50$, and k-nearest neighbour in Figure 12 for $v = 100$).

This large performance gain (factor 10, using only 4 threads) can be attributed to two causes. The first reason is a better collaborative use of the error allowance by threads that investigate concurrently multiple branches of the decision tree. The available error allowance for a thread traversing a long branch b_1 can be increased due to a simultaneous successful traversal of a shallow branch b_2 by a different thread. The sequential algorithm BALANCED spends time traversing b_1 to its full depth first, leaving some of the error budget unused after a quick traversal of b_2. This effect is especially relevant for positive correlations, which have unbalanced trees. The second cause is BALANCED's approach to keeping track of the error budgets: the sequential algorithm

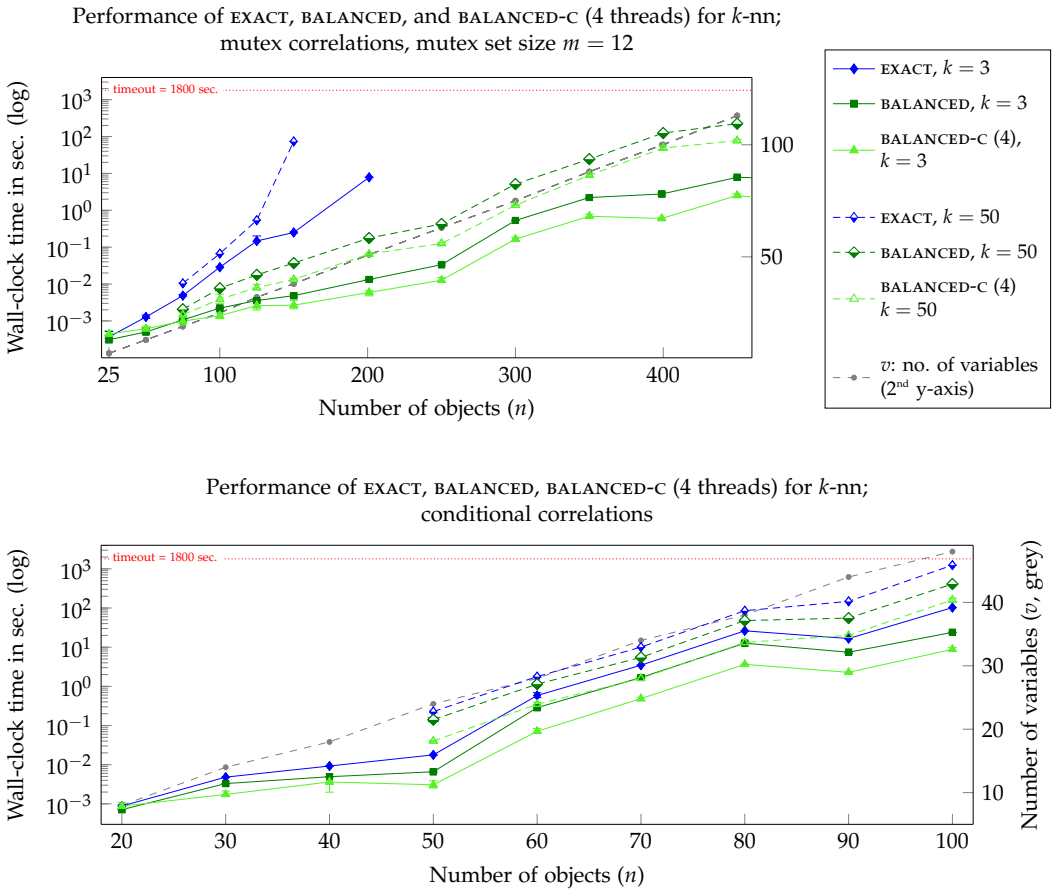


Fig. 13. Probability computation for k-nn classification on data with mutex and conditional correlations for varying values of n (objects) and k.

explicitly stores the error budgets for all targets for the current branch at each depth of the tree to be able to transfer the budget from one branch to another, and spend unused budget locally. The BALANCED-C algorithm recalculates the budget at every depth using the overall progress to minimise the need for thread synchronisation. This effect is not significant for shallow trees, but becomes apparent for deeper trees when the number of variables increases (*e.g.* $v = 100$).

The thread scalability of BALANCED-C has been evaluated using k-nn clustering, the results of which are presented in Figure 14. For large numbers of variables ($v = 150$, left), the actual scalability in the number of threads t is near optimal (*i.e.* t threads yield a factor t performance improvement). For reasons explained above, the performance improvement is sometimes more than a factor t. On the right, experiments with different values for v (number of variables) and k are shown.

5.3. Experiments with k-medoids clustering

A synthetically generated data set was used to investigate the influence of *certain* (*i.e.* with probability 1) objects (or records) on the scalability of probability computation of k-medoids clustering. Figure 15 presents the performance of BALANCED and BALANCED-C probability computation on this data for 0% certain objects (*i.e.*, fully

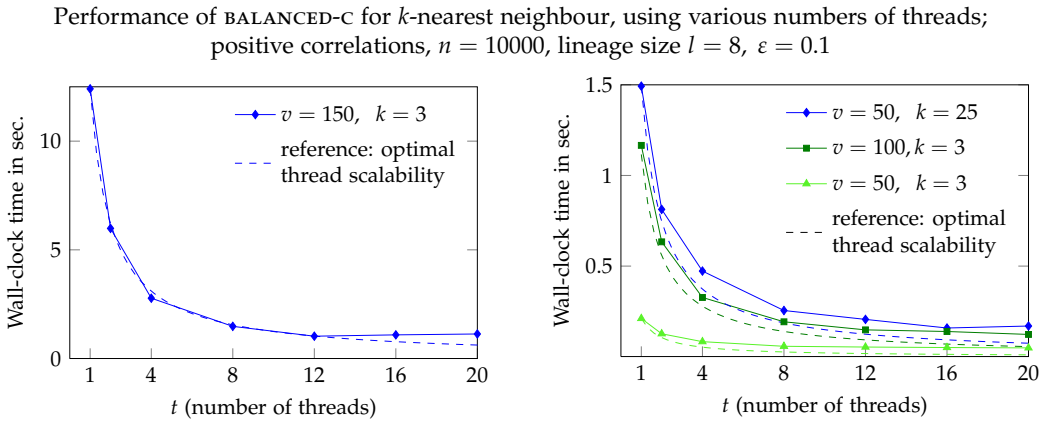


Fig. 14. Concurrent probability computation for k-nn classification with varying v, k.

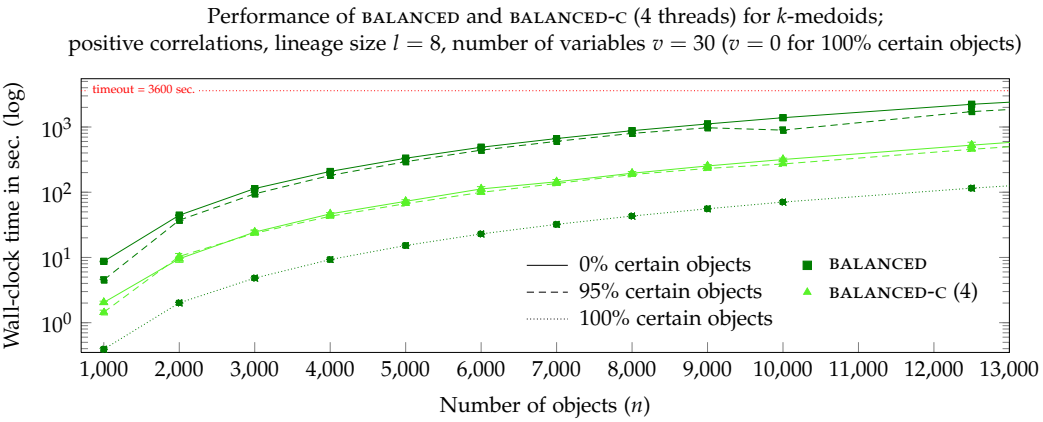


Fig. 15. Probability computation for k-medoids with BALANCED and BALANCED-C on large-scale generated data sets with certain data points.

probabilistic data), 95% certain objects, and 100% certain objects (*i.e.*, fully certain data for which $v = 0$).

For both algorithms, the performance improves slightly when the fraction of certain objects is increased from 0% to 95%. The marginal speed-up in such cases is explained by the fact that the distance-sums of medoids to data points in a cluster become less complex and can be initialised using the distances to objects that are certain. Consequently, fewer variable assignments are needed to decide on a cluster medoid, resulting in a shallower decomposition tree and improved performance.

Performance of BALANCED on a data set with 100% certain objects is provided as a reference point. For fully certain data (*i.e.*, $v = 0$ variables), no decomposition tree is constructed. Therefore, only one thread is used for computation, and BALANCED and BALANCED-C have identical performance.

Further experiments investigated the influence of program-specific parameters (such as the number of dimensions, data point coordinates, the numbers of iterations) on performance. The number of dimensions has no influence on the computation time, as the algorithm uses a precomputed distance measure on the feature space. The num-

ber of clustering iterations has a linear effect on the running time of the algorithm in ENFrame.

The number of targets (including those representing co-occurrence queries) has a minor influence on performance. Due to the combinatorial nature of k-medoids, events are mostly satisfied in bulk. It is thus very rare that one event alone is satisfied at any one time. This also explains why experiments with other types of targets (*e.g.*, object-cluster assignment, pairwise object-cluster assignment) show no difference in performance.

The memory footprint of the event networks was monitored during probability computation and was up to 1GB for the experiments presented in this section.

5.4. Experiments with k-nn classification

The k-nn classification problem shows a more localised complexity than clustering: only the k nearest objects are of importance. As can be seen in Figures 12 and 13, the correlation scheme has a large influence on which objects are considered to be possible k-nearest neighbours in an uncertain setting. For positive correlations, as the distance between the unlabelled object and a possible nearest neighbour increases, the probability of that neighbour being one of the k nearest neighbours rapidly decreases. As a result, BALANCED performs up to four orders of magnitude better than EXACT on positive correlations (Figure 12 for $n = 1000, k = 10$), and both algorithms scale sub-linearly in n. BALANCED-C (with four threads) yields yet another order of magnitude performance improvement.

Figure 12 (bottom) shows that the value of k has a significant influence on performance: increasing the number of nearest neighbours increases the size of the search space for possible candidates, which in turn increases the probabilistic inference time. There are many techniques to determine the right value of k for a data set [Devroye et al. 1996], all of which can be applied directly to k-nn in ENFrame. For our experiments we decided to use a wide range of values to investigate their influence on the performance of probability computation. Investigating the influence of k on the quality of k-nn classification is beyond the scope of this work.

The mutex and conditional correlation schemes yield a rather different behaviour, cf. Figure 13. Under both schemes, the probability of a distant object being one of the k nearest neighbours is larger, which dramatically increases the number of objects that need to be considered. As a result, BALANCED cannot trim as many branches of the decomposition tree as in the case of positive correlations. Concurrent probability computation with four threads yields a consistently better performance with very little synchronisation overhead.

5.5. Quality evaluators for processing probabilistic data

There is a wealth of literature on techniques for measuring the quality of clustering and classification of traditional *certain* data. We introduce here a quality evaluator for probabilistic clustering and refer the reader to the thesis of the second author for a similar evaluator for classification [van Schaik 2015].

Quality evaluators for clustering can be divided into two categories:

— External evaluators, which compare an output (*e.g.* clustering) against an oracle that defines the desired output. Examples of external evaluators include the Rand measure [Rand 1971], and the F-measure.
— Internal evaluators which measure notions of *compactness* and *separation* of clusters. Examples include the Davies-Bouldin Index [Davies and Bouldin 1979] and the Dunn Index [Dunn 1973].

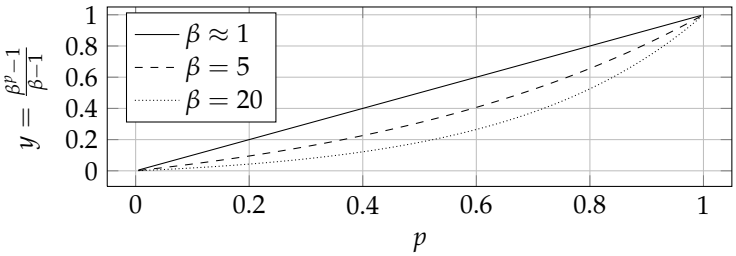


Fig. 16. Effect of parameter β in PAPM calculations

Probabilistic clustering results are fundamentally different from traditional (certain) results: the output is a probability distribution over the possible results. One way of using the existing evaluators on a probabilistic result might be to apply them in every possible world and then use a meaningful aggregator over the exponentially many results. The computational effort required by such an approach make it rather impractical for very large sets of possible worlds.

Our quality evaluator for probabilistic clustering can take the correlations in the input probabilistic data into account. Furthermore, it does not make assumptions about the number of objects and clusters, as not all k clusters and n objects are guaranteed to exist in all possible worlds. Finally, it does not rely on cluster labels, because some cluster j in some possible world w might contain the exact same set of objects as a cluster j' in another world w'.

Many of the (internal and external) clustering quality evaluators described in the literature are based on some measure of the assignment of objects to clusters. For example, such quality measures use the intra-cluster distances as a measure of separation, or the inter-cluster distances as a measure of compactness, or compare clusterings based on the cluster assignment of objects. These evaluators do not share the properties listed above. The pairwise assignment of objects to clusters, however, is a suitable measure: it is defined for both probabilistic and deterministic clusterings, it allows for taking correlations between objects into account, and it does not rely on the number of clusters or their labels.

The *Pairwise Assignment Probability Measure* (PAPM) is an external evaluator for probabilistic clustering, inspired by the Rand measure [Rand 1971]: it compares two clusterings, $\mathcal{D}_1$ and $\mathcal{D}_2$, using the pairwise assignment probabilities of objects to clusters. PAPM has two variants: PAPM_{avg} and PAPM_{max}.

Definition 5.1. Let $\mathcal{D}_1$ and $\mathcal{D}_2$ be clustering results, with k clusters $C_0, \ldots, C_{k-1}$ for a data set with n objects $o_0, \ldots, o_{n-1}$. We define PAPM_{avg} and PAPM_{max} as follows:

$$\Pr_{\mathcal{D}}[o_a \sim o_b] = \sum_{0 \le i < k} \Pr_{\mathcal{D}}[o_a \in C_i \wedge o_b \in C_i]$$

$$\Delta_{\mathcal{D}_1,\mathcal{D}_2}(o_a, o_b) = \left| \Pr_{\mathcal{D}_1}[o_a \sim o_b] - \Pr_{\mathcal{D}_2}[o_a \sim o_b] \right|$$

$$\text{PAPM}_{\text{avg}}(\mathcal{D}_1, \mathcal{D}_2) = \operatorname*{avg}_{o_a,o_b} \left(\frac{\beta^{\Delta_{\mathcal{D}_1,\mathcal{D}_2}(o_a,o_b)} - 1}{\beta - 1} \right), \text{ where } \beta > 1$$

$$\text{PAPM}_{\text{max}}(\mathcal{D}_1, \mathcal{D}_2) = \max_{o_a,o_b} \left(\frac{\beta^{\Delta_{\mathcal{D}_1,\mathcal{D}_2}(o_a,o_b)} - 1}{\beta - 1} \right), \text{ where } \beta > 1.$$

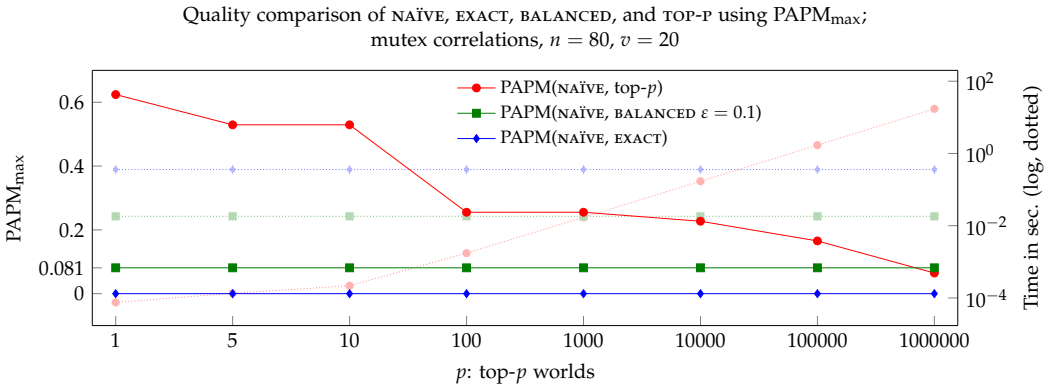


Fig. 17. Accuracy experiment with k-medoids: comparing NAÏVE (golden standard) to EXACT, BALANCED, and TOP-P clustering ($\beta \approx 1$). The dotted lines indicate the performance (in seconds, on the right y-axis) of TOP-P, BALANCED, EXACT, respectively.

The values of both PAPM measures range from zero to one. A score close to zero indicates that two clustering results $\mathcal{D}_1$ and $\mathcal{D}_2$ are very similar: the average (in case of PAPM_{avg}) or maximum (PAPM_{max}) difference between pairwise assignment probabilities is small. A score close to 1 means that the two clusterings are very dissimilar, with large differences in pairwise assignment probabilities.

The term $\Pr[o_a \sim o_b]$ represents the probability of pairwise assignment of two objects o_a and o_b to the same cluster. The events for the object-to-cluster assignment for one object o_a for multiple clusters C_i are mutually exclusive; consequently, the pairwise assignment probability can be computed as the sum of the events in which both objects o_a and o_b are assigned to the same cluster. The term $\Delta_{\mathcal{D}_1,\mathcal{D}_2}(o_a, o_b)$ represents the difference in pairwise assignment probability $\Pr[o_a \sim o_b]$ between two clusterings $\mathcal{D}_1$ and $\mathcal{D}_2$ with regard to objects o_a and o_b.

The constant $\beta > 1$ is a non-linear scaling constant similar to the constant used in the F-measure. It has no influence when set close to 1, but for larger values it reduces the contribution of object pairs with small differences in assignment probabilities to the overall score. The effect of this optional correction is illustrated in Figure 16.

We used PAPM to compare the quality of probabilistic clustering as computed by our exact and approximate inference algorithms and by NAÏVE and TOP-P, where NAÏVE produces by definition the best quality as it performs exact clustering in every possible world. Using PAPM_{max} ($\beta \approx 1$), we verified that indeed EXACT has the same best quality as NAÏVE and that the quality of all approximation algorithms is within the given error bounds and much better than TOP-P. This is shown in Figure 17: the PAPM_{max} score of zero (blue plot line) indicates that ENFrame's EXACT clustering is equivalent to clustering in all possible worlds (NAÏVE). Furthermore, the difference between BALANCED and NAÏVE (green plot line) never exceeds 0.1: the average PAPM_{max} score over five runs is approximately 0.081. On the other hand, the accuracy of clustering in the top-p most probable worlds is far off from NAÏVE, even for large p.

The dotted plot lines in Figure 17 indicate the performance (in seconds) of TOP-P, BALANCED, and EXACT on the right y-axis. Already for $p > 1000$, TOP-P is outperformed by BALANCED. Only when p is almost equal to the number of worlds (*i.e.*, $p \approx 2^v \approx 1000000$), the quality of TOP-P surpasses BALANCED. However, for such large values of p, ENFrame's approximation algorithms are about five orders of magnitude faster, while providing solid error guarantees. The size of the data set for this experiment was restricted due to the (very) limited scalability of the NAÏVE algorithm.

6. RELATED WORK

Our work is at the confluence of several research areas: probabilistic data management, data analytics platforms, and provenance data management. Section 1 highlights the key design differences between ENFrame and existing probabilistic programming and data processing approaches, including: the use of possible worlds semantics throughout the whole processing pipeline, from input data to the result probability distribution; arbitrary correlations in input and computation traces captured via probabilistic events; separation of probabilistic data and program, which enables low-entry programming oblivious of probabilistic models and inference. We next highlight related work in uncertain data mining and querying, data analytics, probabilistic programming, and provenance.

Probabilistic data mining and querying. Our work adds to a wealth of literature on this topic [Aggarwal 2009; Suciu et al. 2011] along two directions: distributed probability computation techniques and a unified formalisation of several clustering and classification algorithms in line with existing work on probabilistic databases.

Distributed probability computation has been approached in the context of the SimSQL/MCDB system, where approximate query results are computed by Monte Carlo simulations [Jampani et al. 2011; Cai et al. 2013]. This contrasts with our approach in that MCDB does not support approximate computation with error guarantees and does not exploit the type of event correlations supported by ENFrame.

Early approaches to mining uncertain data are based on imprecise (fuzzy) data, for example using intervals, and produce fuzzy (soft) or hard output. Follow-up work shifted to representation of uncertainty by (independent) probability density functions per data point. In contrast, we allow for arbitrarily correlated discrete probability distributions. The importance of correlations has been previously acknowledged for clustering [Volk et al. 2009] and frequent pattern mining [Sun et al. 2010]. A further key aspect of our approach that is not shared by existing uncertain data mining approaches is that we follow the possible worlds semantics throughout the whole mining process. This allows for exact and approximate computation with error guarantees and sound semantics of the mining process that is compatible with probabilistic databases. This is not achieved by existing work; for instance, most existing k-means clustering approaches for uncertain data define cluster centroids using *expected distances* between data points [Chau et al. 2006; Ngai et al. 2006a; Gullo et al. 2008; Kriegel and Pfeifle 2005; Gullo et al. 2008; Kao et al. 2010] or the *expected variance* of all data points in the same cluster [Gullo et al. 2010]; they also compute hard clustering where the centroids are deterministic. The recently introduced UCPC approach to k-means clustering [Gullo and Tagarelli 2012] is the first work to acknowledge the importance of probabilistic cluster centroids. However, it assumes independence in the input and does not support correlations.

ENFrame's approach to approximate inference is based on earlier work on approximate query evaluation in probabilistic databases, which has been shown to be effective [Olteanu et al. 2010; Fink et al. 2013]. This is directly related to numerous approaches for (weighted) model counting (#SAT) problem [Gomes et al. 2009]. Mainstream approaches to model counting use Monte Carlo sampling, *e.g.*, [Wei and Selman 2005]. Although these approaches have been shown to be effective in many cases, they depend on uniform sampling [Gomes et al. 2006]. Additionally, Monte Carlo sampling methods only provide probabilistic guarantees (*i.e.*, only with probability $p < 1$ is a sampling result an ε-approximation, where p is preset), and do not exploit the structure of probabilistic events [Olteanu and Wen 2012] as done by ENFrame.

In contrast to sampling, ENFrame's inference algorithms exploit the structure of the events and decompose them using Boole's expansion theorem [Boole 1854], which

is also referred to as Shannon expansion [Shannon 1949]. A key component of decomposition is the choice of which variable to eliminate next. The order in which the variables are eliminated in the decomposition tree has a significant influence on its depth [Rudell 1993]. The problems of finding the optimal variable order, i.e., an order under which the decomposition tree has a minimal number of nodes, or even approximating it by a constant factor are NP-hard [Bollig and Wegener 1996; Sieling 2002]. Significant efforts have been made to construct heuristics for finding a favourable variable order for propositional expressions [Rice and Kulhari 2008]. Many such heuristics rely on topological properties of the expression (circuit) to define a distance measure on variables, and subsequently try to order variables in such a way that variables at a short distance from each other are grouped together in the decision tree (*e.g.*, [Aloul et al. 2004; Drechsler 1996; Fujita et al. 1993]). Other methods attempt to identify the most influential variables for early processing in the decision tree (*e.g.*, [Malik et al. 1988; Rice and Kulhari 2008]).

Probabilistic programming. There is a significant body of work on extending programming languages, and in particular query languages, with probabilistic constructs. They can be classified into four broad categories [Bárány et al. 2016]: imperative specifications over logical structures, programming over probabilistic databases, indirect specifications over the Herbrand base, and purely declarative probabilistic programming languages. Some of these formalisms belong to more than one category.

The first category includes imperative probabilistic programming languages [Roy 2015] and declarative specifications of Bayesian networks, such as BLOG [Milch and et al 2005] and P-log [Baral et al. 2009] that rely on an imperative execution model. BLOG can express probability distributions over logical structures via generative stochastic models that can draw values at random from numerical distributions, and condition values of program variables on observations. In contrast to closed-universe languages such as SQL, logic programs, or ENFrame's user language, BLOG considers open-universe probability models that allow for uncertainty about the existence and identity of objects. MCDB [Jampani et al. 2011] and SimSQL [Cai et al. 2013] propose SQL extensions with for-loops and probability distributions as first class citizens.

The formalisms in the second category assume that the input probabilistic data is generated by a mechanism external to the program. A user program, such as a Prolog [Kimmig et al. 2011] or Datalog [Abiteboul et al. 2014] program, is then evaluated over this input data. This approach has been taken by PRISM [Sato and Kameya 1997], the Independent Choice Logic [Poole 2008], and to a large extent by probabilistic databases [Suciu et al. 2011] and their semi-structured counterparts [Kimelfeld and Senellart 2013]. ENFrame is best positioned in this category.

This second category also contains uncertainty-aware query languages for probabilistic data such as TriQL [Widom 2008], I-SQL, and world-set algebra [Antova et al. 2007a; 2007b]. The latter two are natural analogs to SQL and relational algebra for the case of incomplete information and probabilistic data [Antova et al. 2007a]. Similar to ENFrame's user language, these languages cannot explicitly specify probability distributions, yet they may simulate a specific categorical distribution indirectly using specialised language constructs. Examples of such constructs are `repair-key`, `choice-of`, `possible`, and `group-worlds-by` that can construct possible worlds representing all repairs of a relation w.r.t. key constraints, close the possible worlds by unioning or intersecting them, or group the worlds into sets with the same results to sub-queries. World-set algebra has been extended to (world-set) Datalog, fixpoint, and while-languages [Deutch et al. 2010] to define Markov chains.

Key properties of world-set algebra are conservativity over relational algebra [Antova et al. 2007a], genericity, expressiveness [Koch 2009], and efficient processing [An-

tova et al. 2007b]. World-set algebra captures exactly second-order logic over finite structures, or equivalently, the polynomial hierarchy [Koch 2009]. Moreover, it is closed under composition [Koch 2009]. ENFrame's user language also shares flavours of these features: It is generic in the sense that different equivalent representations of the input probabilistic data lead to semantically equivalent program results. It is expressive in the sense that it features common language constructs such as loops, assignments, aggregates, which are essential to data mining tasks. It is conservative over the Python language in the sense that when executed over deterministic data, the ENFrame user program yield the same result. It is closed under composition in the sense that the composition of two programs, *i.e.*, the output of the first program becomes the input to the second program, can be expressed as one large program and their sequential execution is equivalent to the execution of the large program. Finally, ENFrame's inference mechanisms outperform a suite of known approaches, as demonstrated experimentally.

Formalisms in the third category are indirect specifications of probability spaces over the Herbrand base and includes Markov Logic Networks (MLNs) [Domingos and Lowd 2009; Niu et al. 2011], where the logical rules are used as a compact and intuitive way of defining factors. This approach is applied in DeepDive [Niu et al. 2012], where a database is used for storing relational data and extracted text, and database queries are used for defining the factors of a factor graph. Further formalisms in this category are probabilistic Datalog [Fuhr 2000], probabilistic Datalog+/- [Gottlob et al. 2013], and probabilistic logic programming (ProbLog) [Kimmig et al. 2011].

The fourth category includes a recently developed probabilistic programming Datalog (PPDL), where probabilities are first-class citizens [Bárány et al. 2016]. In contrast to the previously mentioned probabilistic Datalog approaches, PPDL has a robust declarative semantics, *i.e.*, semantic independence from the algorithmic evaluation of Datalog rules and semantics invariance under logical program transformations.

Data analytics platforms. Support for iterative programs is essential in many applications including data mining, web ranking, graph analysis, and model fitting. This has recently led to a surge in data-intensive computing platforms with built-in iteration capability. REX supports iterative distributed computation along database operations in which changes are propagated between iterations [Mihaylov et al. 2012]. MADlib is an open-source library for in-database analytics [Hellerstein et al. 2012]. Similarly, Bismarck is an architecture for in-database analytics [Feng et al. 2012], while UDA-GIST provides efficient in-database support of probabilistic models and computation [Li et al. 2015]. ProbKB is a probabilistic knowledge base designed to infer missing facts in a scalable, probabilistic, and principled manner using a relational DBMS [Chen and Wang 2014]. GraphLab [Low et al. 2010] uses graph representations for scalable parallel programming. The Iterative Map-Reduce-Update programming abstraction for machine learning compiles programs into declarative Datalog code [Borkar et al. 2012]. Infer.NET [Minka et al. 2012] is a probabilistic programming platform with probability distributions as first-class citizen in the user language.

Provenance in database and workflow systems. To enable probability computation, we trace fine-grained provenance of the user computation. This is in line with a wealth of work in probabilistic databases [Suciu et al. 2011]. Our event language is influenced by work on provenance semirings [Green et al. 2007] and semimodules [Amsterdamer et al. 2011; Fink et al. 2012] that capture provenance for positive queries with aggregates in relational databases. The construct $\Phi \otimes v$, where Φ is a Boolean formula and v a real number, is not captured by a semimodule such as those investigated in prior work [Amsterdamer et al. 2011]. Firstly, we allow negation in Boolean events, which is not captured by semirings. Secondly, even for positive events, the tensor product $\mathbb{B}[\mathbf{X}] \otimes \mathbb{R}$ of the Boolean semiring $\mathbb{B}[\mathbf{X}]$ freely generated by the variable set $\mathbf{X}$ and of

the SUM monoid over the real numbers $\mathbb{R}$ is not a semimodule since it violates the following law: $(\Phi_1 \vee \Phi_2) \otimes v = \Phi_1 \otimes v + \Phi_2 \otimes v$. Indeed, under an assignment that maps both Φ_1 and Φ_2 to $\top$, the left side of the equality evaluates to v, whereas the right side becomes $v + v$. Furthermore, our event language allows to define events via iterations, as needed to succinctly trace data mining computation.

7. CONCLUSION

This article introduces ENFrame, a platform for processing probabilistic data. It highlights the design principles behind ENFrame and focuses on a family of probabilistic inference algorithms used by ENFrame to compute the probability distributions over the possible results of user programs. ENFrame is showcased for clustering and classification of probabilistic data. Experiments confirm the quality of the proposed inference algorithms and show that, among these algorithms, the concurrent approximate one is up to several orders of magnitude faster than sequential exact or approximate ones, which on their turn exhibit a similar performance gap when compared with naïve clustering in all possible worlds.

There are several exciting directions for future work. We would like to understand the trade-offs between the expressiveness of the user language and efficiency of inference and between the functionality of fine-grained events and performance brought by compilation of event networks to efficient C++ code. We would also like to investigate additional constructs in the event language needed to support a library of common algorithms for data analysis. Although not discussed in this article, current work on ENFrame looks at how to exploit the structure of the user program, in addition to query structure, for efficient inference. A first step is static analysis of user programs to isolate program fragments that are semantically equivalent to known tractable queries. For practical reasons, it is desirable to better integrate queries and program constructs with relations and allow program data structures (e.g., multi-dimensional arrays) that can be used interchangeably in both programs and queries. Finally, the performance of ENFrame inference needs improvements and one direction of research is to employ large-scale distributed systems.

REFERENCES

Serge Abiteboul, Daniel Deutch, and Victor Vianu. 2014. Deduction with Contradictions in Datalog. In *ICDT*. 143–154.

Charu Aggarwal. 2009. *Managing and Mining Uncertain Data*. Advances in Database Systems, Vol. 35. Kluwer. 1–41 pages.

Parag Agrawal, Omar Benjelloun, Anish Das Sarma, Chris Hayworth, Shubha U. Nabar, Tomoe Sugihara, and Jennifer Widom. 2006. Trio: A System for Data, Uncertainty, and Lineage. In *VLDB*. 1151–1154.

Fadi A. Aloul, Igor L. Markov, and Karem A. Sakallah. 2004. MINCE: A Static Global Variable-Ordering Heuristic for SAT Search and BDD Manipulation. *J. Universal Comp. Sci.* 10, 12 (2004), 1562–1596.

Y. Amsterdamer, D. Deutch, and V. Tannen. 2011. Provenance for aggregate queries. In *PODS*. 153–164.

Lyublena Antova, Christoph Koch, and Dan Olteanu. 2007a. From complete to incomplete information and back. In *SIGMOD*. 713–724.

Lyublena Antova, Christoph Koch, and Dan Olteanu. 2007b. Query language support for incomplete information in the MayBMS system. In *VLDB*. 1422–1425.

Chitta Baral, Michael Gelfond, and Nelson Rushton. 2009. Probabilistic Reasoning with Answer Sets. *Theory Pract. Log. Program.* 9, 1 (2009), 57–144.

Vince Bárány, Balder ten Cate, Benny Kimelfeld, Dan Olteanu, and Zografoula Vagena. 2016. Declarative Probabilistic Programming with Datalog. In *ICDT*. To appear.

Mordechai Ben-Ari. 1990. *Principles of Concurrent and Distributed Programming* (1 ed.). Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Jens Bleiholder and Felix Naumann. 2009. Data fusion. *ACM Comput. Surv.* 41, 1 (2009), 1:1–1:41.

Beate Bollig and Ingo Wegener. 1996. Improving the Variable Ordering of OBDDs Is NP-Complete. *IEEE Trans. on Comp.* 45, 9 (1996), 993–1002.

George Boole. 1854. *An Investigation of the Laws of Thought: On which are Founded the Mathematical Theories of Logic and Probabilities*. Walton and Maberly.

Vinayak R. Borkar, Yingyi Bu, Michael J. Carey, Joshua Rosen, Neoklis Polyzotis, Tyson Condie, Markus Weimer, and Raghu Ramakrishnan. 2012. Declarative Systems for Large-Scale Machine Learning. *IEEE Data Eng. Bull.* 35, 2 (2012), 24–32.

Jihad Boulos, Nilesh N. Dalvi, Bhushan Mandhani, Shobhit Mathur, Christopher Ré, and Dan Suciu. 2005. MYSTIQ: a system for finding more answers by using probabilities. In *SIGMOD*. 891–893.

Zhuhua Cai, Zografoula Vagena, Luis Perez, Subramanian Arumugam, Peter J. Haas, and Christopher M. Jermaine. 2013. Simulation of database-valued Markov chains using SimSQL. In *SIGMOD*. 637–648.

Andrew Carlson, Justin Betteridge, Bryan Kisiel, Burr Settles, Estevam Junior Hruschka, and Tom Mitchell. 2010. Toward an Architecture for Never-Ending Language Learning. In *AAAI*.

Michael Chau, Reynold Cheng, Ben Kao, and Jackey Ng. 2006. Uncertain Data Mining: An Example in Clustering Location Data. In *PAKDD*. 199–204.

Yang Chen and Daisy Zhe Wang. 2014. Knowledge expansion over probabilistic knowledge bases. In *SIGMOD*. 649–660.

DARPA. 2013. Probabilistic Programming for Advancing Machine Learning. (April 2013). DARPA-BAA-13-31.

David L. Davies and Donald W. Bouldin. 1979. A Cluster Separation Measure. *IEEE Trans. on Pattern Analysis and Machine Intelligence* 1, 2 (1979), 224–227.

Daniel Deutch, Christoph Koch, and Tova Milo. 2010. On probabilistic fixpoint and Markov chain query languages. In *PODS*. 215–226.

L. Devroye, L. Györfi, and G. Lugosi. 1996. *A Probabilistic Theory of Pattern Recognition*.

Edsger Wiebe Dijkstra. 1965. Solution of a Problem in Concurrent Programming Control. *CACM* 8, 9 (1965).

Pedro Domingos and Daniel Lowd. 2009. *Markov Logic: An Interface Layer for Artificial Intelligence*. Morgan & Claypool Publishers.

Xin Dong, Evgeniy Gabrilovich, Geremy Heitz, Wilko Horn, Ni Lao, Kevin Murphy, Thomas Strohmann, Shaohua Sun, and Wei Zhang. 2014. Knowledge vault: a web-scale approach to probabilistic knowledge fusion. In *KDD*. 601–610.

Xin Luna Dong, Laure Berti-Equille, and Divesh Srivastava. 2009. Integrating Conflicting Data: The Role of Source Dependence. *PVLDB* 2, 1 (2009), 550–561.

Rolf Drechsler. 1996. Verification of Multi-Valued Logic Networks. In *ISMVL*. 10–15.

J. C. Dunn. 1973. A Fuzzy Relative of the ISODATA Process and Its Use in Detecting Compact Well-Separated Clusters. *J. Cybernetics* 3, 3 (1973), 32–57.

Xixuan Feng, Arun Kumar, Benjamin Recht, and Christopher Ré. 2012. Towards a unified architecture for in-RDBMS analytics. In *SIGMOD*. 325–336.

Robert Fink, Larisa Han, and Dan Olteanu. 2012. Aggregation in Probabilistic Databases via Knowledge Compilation. *PVLDB* 5, 5 (2012), 490–501.

Robert Fink, Andrew Hogue, Dan Olteanu, and Swaroop Rath. 2011. SPROUT^2: A squared query engine for uncertain web data. In *SIGMOD*. 1299–1302.

Robert Fink, Jiewen Huang, and Dan Olteanu. 2013. Anytime approximation in probabilistic databases. *VLDB J.* 22, 6 (2013), 823–848.

Norbert Fuhr. 2000. Probabilistic datalog: Implementing logical information retrieval for advanced applications. *JASIS* 51, 2 (2000), 95–110.

Masahiro Fujita, Hisanori Fujisawa, and Yusuke Matsunaga. 1993. Variable ordering algorithms for ordered binary decision diagrams and their evaluation. *IEEE Trans. on CAD of Integrated Circuits and Systems* 12, 1 (1993), 6–12.

Carla P. Gomes, Ashish Sabharwal, and Bart Selman. 2006. Model counting: A new strategy for obtaining good bounds. *National Conf. on AI* 21, 1 (2006), 54.

Carla P. Gomes, Ashish Sabharwal, and Bart Selman. 2009. Model Counting. In *Handbook of Satisfiability*. 633–654.

Andrew D. Gordon, Thomas A. Henzinger, Aditya V. Nori, and Sriram K. Rajamani. 2014. Probabilistic programming. In *FOSE*. 167–181.

Georg Gottlob, Thomas Lukasiewicz, MariaVanina Martinez, and Gerardo Simari. 2013. Query answering under probabilistic uncertainty in Datalog+/âĹŠ ontologies. *Annals of Math.& AI* 69, 1 (2013), 37–72. DOI:http://dx.doi.org/10.1007/s10472-013-9342-1

Todd J. Green, Grigoris Karvounarakis, and Val Tannen. 2007. Provenance semirings. In *PODS*. 31–40.

Francesco Gullo, Giovanni Ponti, and Andrea Tagarelli. 2008. Clustering Uncertain Data Via K-Medoids. In *SUM*. 229–242.

F. Gullo, G. Ponti, and A. Tagarelli. 2010. Minimizing the Variance of Cluster Mixture Models for Clustering Uncertain Objects. In *ICDM*. 839–844.

F. Gullo, G. Ponti, A. Tagarelli, and S. Greco. 2008. A Hierarchical Algorithm for Clustering Uncertain Data via an Information-Theoretic Approach. In *ICDM*. 821–826.

Francesco Gullo and Andrea Tagarelli. 2012. Uncertain Centroid based Partitional Clustering of Uncertain Data. *PVLDB* 5, 7 (2012), 610–621.

Joseph Hellerstein, Christopher Ré, Florian Schoppmann, Daisy Zhe Wang, Eugene Fratkin, Aleksander Gorajek, Kee Siong Ng, Caleb Welton, Xixuan Feng, Kun Li, and Arun Kumar. 2012. The MADlib Analytics Library or MAD Skills, the SQL. *PVLDB* 5, 12 (2012), 1700–1711.

Jiewen Huang, Lyublena Antova, Christoph Koch, and Dan Olteanu. 2009. MayBMS: a probabilistic database management system. In *SIGMOD*. 1071–1074.

Ravi Jampani, Fei Xu, Mingxi Wu, Luis Perez, Chris Jermaine, and Peter Haas. 2011. The Monte Carlo Database System: Stochastic analysis close to the data. *ACM TODS* 36, 3 (2011), 1–41.

Bhargav Kanagal, Jian Li, and Amol Deshpande. 2011. Sensitivity analysis and explanations for robust query evaluation in probabilistic databases. In *SIGMOD*. 841–852.

Ben Kao, Sau Dan Lee, Foris K. F. Lee, David Wai-Lok Cheung, and Wai-Shing Ho. 2010. Clustering Uncertain Data Using Voronoi Diagrams and R-Tree Index. *IEEE TKDE* 22, 9 (2010), 1219–1233.

Benny Kimelfeld and Pierre Senellart. 2013. Probabilistic XML: Models and Complexity. In *Advances in Probabilistic Databases for Uncertain Information Management*. Studies in Fuzziness and Soft Computing, Vol. 304. 39–66.

Angelika Kimmig, Bart Demoen, Luc De Raedt, Vitor Santos Costa, and Ricardo Rocha. 2011. On the implementation of the probabilistic logic programming language ProbLog. *Theory and Practice of Logic Programming* 11 (2011), 235–262.

Christoph Koch. 2009. A compositional query algebra for second-order logic and uncertain databases. In *ICDT*. 127–140.

Christoph Koch and Dan Olteanu. 2008. Conditioning probabilistic databases. *PVLDB* 1, 1 (2008), 313–325.

H. Kriegel and M. Pfeifle. 2005. Density-based clustering of uncertain data. In *KDD*. 672–677.

Kun Li, Daisy Zhe Wang, Alin Dobra, and Christopher Dudley. 2015. UDA-GIST: An In-database Framework to Unify Data-Parallel and State-Parallel Analytics. *PVLDB* 8, 5 (2015), 557–568.

Xiang Lian and Lei Chen. 2010. A Generic Framework for Handling Uncertain Data with Local Correlations. *PVLDB* 4, 1 (2010), 12–21.

Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. 2010. GraphLab: A New Framework For Parallel Machine Learning. In *UAI*. 340–349.

Sharad Malik, Albert R. Wang, Robert. K. Brayton, and Alberto Sangiovanni-Vincentelli. 1988. Logic verification using binary decision diagrams in a logic synthesis environment. In *ICCAD*. 6–9.

Matthieu Michel. 2007. Innovative asset management and targeted investments using on-line partial discharge monitoring & mapping techniques. In *CIRED*.

Matthieu Michel and Carl Eastham. 2011. Improving the management of MV underground cable circuits using automated on-line cable Partial Discharge mapping. In *CIRED*.

Svilen Mihaylov, Zachary Ives, and Sudipto Guha. 2012. REX: Recursive, Delta-Based Data-Centric Computation. *PVLDB* 5, 11 (2012), 1280–1291.

B. Milch and et al. 2005. BLOG: Probabilistic models with unknown objects. In *IJCAI*. 1352–1359.

T. Minka, J.M. Winn, J.P. Guiver, and D.A. Knowles. 2012. Infer.NET 2.5. (2012). Microsoft Research Cambridge.

Mohamed F. Mokbel, Chi-Yin Chow, and Walid G. Aref. 2006. The New Casper: Query Processing for Location Services without Compromising Privacy. In *VLDB*. 763–774.

W.K. Ngai, B. Kao, C.K. Chui, R. Cheng, M. Chau, and K.Y. Yip. 2006a. Efficient Clustering of Uncertain Data. In *ICDM*. 436–445.

Wang Kay Ngai, Ben Kao, Chun Kit Chui, Reynold Cheng, Michael Chau, and Kevin Y. Yip. 2006b. Efficient Clustering of Uncertain Data. In *ICDM*. 436–445.

Feng Niu, Christopher Ré, AnHai Doan, and Jude W. Shavlik. 2011. Tuffy: Scaling up Statistical Inference in Markov Logic Networks using an RDBMS. *PVLDB* 4, 6 (2011), 373–384.

Feng Niu, Ce Zhang, Christopher Re, and Jude W. Shavlik. 2012. DeepDive: Web-scale Knowledge-base Construction using Statistical Learning and Inference. In *International Workshop on Searching and Integrating New Web Data Sources (CEUR Workshop Proceedings)*, Vol. 884. CEUR-WS.org, 25–28.

Dan Olteanu, Jiewen Huang, and Christoph Koch. 2009. SPROUT: Lazy vs. Eager Query Plans for Tuple-Independent Probabilistic Databases. In *ICDE*. 640–651.

Dan Olteanu, Jiewen Huang, and Christoph Koch. 2010. Approximate confidence computation in probabilistic databases. In *ICDE*. 145–156.

Dan Olteanu and Sebastiaan J. van Schaik. 2012. DAGger: clustering correlated uncertain data (to predict asset failure in energy networks). In *KDD*. 1504–1507.

Dan Olteanu and Sebastiaan J. van Schaik. 2014. Probabilistic Data Programming with ENFrame. *IEEE Data Eng. Bull.* 37, 3 (2014), 18–25.

Dan Olteanu and Hongkai Wen. 2012. Ranking Query Answers in Probabilistic Databases: Complexity and Efficient Algorithms. In *ICDE*. 282–293.

Peter Pacheco. 2011. *An Introduction to Parallel Programming* (1st ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Judea Pearl. 1989. *Probabilistic reasoning in intelligent systems - networks of plausible inference*. Morgan Kaufmann.

David Poole. 2008. The Independent Choice Logic and Beyond. In *Probabilistic Inductive Logic Programming - Theory and Applications*. 222–243.

J.S. Provan and M.O. Ball. 1983. The Complexity of Counting Cuts and of Computing the Probability that a Graph is Connected. *SIAM J. Comput.* 12, 4 (1983), 777–788.

W.M. Rand. 1971. Objective Criteria for the Evaluation of Clustering Methods. *Journal of ASA* 66, 336 (1971), 846–850.

Michael Rice and Sanjay Kulhari. 2008. *A survey of static variable ordering heuristics for efficient BDD/MDD construction*. Technical Report. University of California, Riverside.

Daniel Roy. 2015. Repository on probabilistic programming languages. (2015). www.probabilistic-programming.org.

Richard Rudell. 1993. Dynamic Variable Ordering for Ordered Binary Decision Diagrams. In *ICCAD*. 42–47.

Taisuke Sato and Yoshitaka Kameya. 1997. PRISM: A Language for Symbolic-Statistical Modeling. In *IJCAI*. 1330–1339.

Prithviraj Sen and Amol Deshpande. 2007. Representing and Querying Correlated Tuples in Probabilistic Databases. In *ICDE*. 596–605.

Prithviraj Sen, Amol Deshpande, and Lise Getoor. 2009. PrDB: managing and exploiting rich correlations in probabilistic databases. *VLDB J.* 18, 5 (2009), 1065–1090.

Claude. E. Shannon. 1949. The Synthesis of Two-Terminal Switching Circuits. *Bell System Tech. J.* 28, 1 (1949), 59–98.

Detlef Sieling. 2002. The Nonapproximability of OBDD Minimization. *J. Inf. and Comput.* 172, 2 (2002), 103–138.

Dan Suciu, Dan Olteanu, Christopher Ré, and Christoph Koch. 2011. *Probabilistic Databases*. Morgan & Claypool.

Liwen Sun, Reynold Cheng, David W. Cheung, and Jiefeng Cheng. 2010. Mining uncertain data with probabilistic guarantees. In *KDD*. 273–282.

Gareth A. Taylor, David C. H. Wallom, Sebastien Grenard, Angel Yunta Huete, and Colin J. Axon. 2011. Recent developments towards novel high performance computing and communications solutions for smart distribution network operation. In *ISTG*. 1–8.

S. van Dongen. 2000. *Graph clustering by flow simulation*. Ph.D. Dissertation. University of Utrecht.

Sebastiaan Johannes van Schaik. 2015. *A Framework for Processing Correlated Probabilistic Data*. Ph.D. Dissertation. University of Oxford.

Sebastiaan J. van Schaik, Dan Olteanu, and Robert Fink. 2014. ENFrame: A Platform for Processing Probabilistic Data. In *EDBT*. 355–366.

Peter Benjamin Volk, Frank Rosenthal, Martin Hahmann, Dirk Habich, and Wolfgang Lehner. 2009. Clustering Uncertain Data with Possible Worlds. In *ICDE*. 1625–1632.

Wei Wei and Bart Selman. 2005. A New Approach to Model Counting. In *SAT*. 324–339.

Jennifer Widom. 2008. Trio: a system for data, uncertainty, and lineage. In *Managing and Mining Uncertain Data*, Charu Aggarwal (Ed.). Springer-Verlag, Chapter 5.

Received January 2015; revised 2015; accepted 2015