Distributed Parameter Learning for Probabilistic Ontologies

Giuseppe Cota[1], Riccardo Zese[1], Elena Bellodi[1], Fabrizio Riguzzi[2], and Evelina Lamma[1]

[1] Dipartimento di Ingegneria – University of Ferrara
Via Saragat 1, I-44122, Ferrara, Italy
[2] Dipartimento di Matematica e Informatica – University of Ferrara
Via Saragat 1, I-44122, Ferrara, Italy
[giuseppe.cota,riccardo.zese,elena.bellodi,evelina.lamma,
fabrizio.riguzzi]@unife.it

Abstract. Representing uncertainty in Description Logics has recently received an increasing attention because of its potential to model real world domains. EDGE for "Em over bDds for description loGics paramEter learning" is an algorithm for learning the parameters of probabilistic ontologies from data. However, the computational cost of this algorithm is significant since it may take hours to complete an execution. In this paper we present EDGE$^{\text{MR}}$, a distributed version of EDGE that exploits the MapReduce strategy by means of the Message Passing Interface. Experiments on various domains show that EDGE$^{\text{MR}}$ significantly reduces EDGE running time.

Keywords: Probabilistic Description Logics, Parameter Learning, MapReduce, Message Passing Interface.

1 Introduction

Representing uncertain information is becoming crucial to model real world domains. The ability to describe and reason with probabilistic knowledge bases is a well-known topic in the field of Description Logics (DLs). In order to model domains with complex and uncertain relationships, several approaches have been proposed that combine logic and probability theories. The distribution semantics [18] from logic programming is one of them. In [3, 17, 12] the authors proposed an approach for the integration of probabilistic information in DLs called DISPONTE (for "DIstribution Semantics for Probabilistic ONTologiEs"), which adapts the distribution semantics for probabilistic logic programs to DLs.

EDGE [14], for "Em over bDds for description loGics paramEter learning", is an algorithm for learning the parameters of probabilistic DLs following the DISPONTE semantics. EDGE was tested on various datasets and was able to find good solutions. However, the algorithm may take hours on datasets of the order of MBs. In order to efficiently manage larger datasets in the era of Big

Data, it is of foremost importance to develop approaches for reducing the learning time. One solution is to distribute the algorithm using modern computing infrastructures such as clusters and clouds.

Here we present EDGE$^{\mathrm{MR}}$, a MapReduce version of EDGE. MapReduce [6] is a model for processing data in parallel on a cluster. In this model the work is distributed among mapper and reducer workers. The mappers take the input data and return a set of (key, value) pairs. These sets are then grouped according to the key into couples (key, set_of_values) and the reducers aggregate the values obtaining a set of (key', aggregated_value) couples that represents the output of the task.

Various MapReduce frameworks are available, such as Hadoop. However we chose not to use any framework and to implement a much simpler MapReduce approach for EDGE$^{\mathrm{MR}}$ based on the Message Passing Interface (MPI). The reason is that we adopted a modified MapReduce approach where the map and reduce workers are not purely functional in order to better adapt to the task at hand.

A performance evaluation of EDGE$^{\mathrm{MR}}$ is provided through a set of experiments on various datasets using 1, 3, 5, 9 and 17 computing nodes. The results show that EDGE$^{\mathrm{MR}}$ effectively reduces EDGE running time, with good speedups.

The paper is structured as follows. Section 2 introduces Description Logics while Section 3 summarizes the DISPONTE semantics and an inference system for probabilistic DLs. Section 4 briefly describes EDGE and Section 5 presents EDGE$^{\mathrm{MR}}$. Section 6 shows the results of the experiments for evaluating EDGE$^{\mathrm{MR}}$, while Section 8 draws conclusions.

2 Description Logics

DLs are a family of logic based knowledge representation formalisms which are of particular interest for representing ontologies in the Semantic Web. For a good introduction to DLs we refer to [2].

While DLs are a fragment of first order logic, they are usually represented using a syntax based on concepts and roles. A concept corresponds to a set of individuals of the domain while a role corresponds to a set of couples of individuals of the domain. The proposed algorithm can deal with $\mathcal{SROIQ}(\mathbf{D})$ DLs which we now describe.

We use $\mathbf{A}$, $\mathbf{R}$ and $\mathbf{I}$ to indicate *atomic concepts*, *atomic roles* and *individuals*, respectively. A *role* could be an atomic role $R \in \mathbf{R}$, the inverse R^- of an atomic role $R \in \mathbf{R}$ or a complex role $R \circ S$. We use $\mathbf{R}^-$ to denote the set of all inverses of roles in $\mathbf{R}$. Each $A \in \mathbf{A}$, $\bot$ and $\top$ are concepts and if $a \in \mathbf{I}$, then $\{a\}$ is a concept called *nominal*. If C, C_1 and C_2 are concepts and $R \in \mathbf{R} \cup \mathbf{R}^-$, then $(C_1 \sqcap C_2)$, $(C_1 \sqcup C_2)$ and $\neg C$ are concepts, as well as $\exists R.C$ and $\forall R.C$ and $\geq nR.C$ and $\leq nR.C$ for an integer $n \geq 0$.

A *knowledge base* (KB) $\mathcal{K} = (\mathcal{T}, \mathcal{R}, \mathcal{A})$ consists of a TBox $\mathcal{T}$, an RBox $\mathcal{R}$ and an ABox $\mathcal{A}$. An RBox $\mathcal{R}$ is a finite set of *transitivity axioms* $Trans(R)$,

role inclusion axioms $R \sqsubseteq S$ and *role chain axioms* $R \circ P \sqsubseteq S$, where $R, P, S \in \mathbf{R} \cup \mathbf{R}^-$. A *TBox* $\mathcal{T}$ is a finite set of *concept inclusion axioms* $C \sqsubseteq D$, where C and D are concepts. An *ABox* $\mathcal{A}$ is a finite set of *concept membership axioms* $a : C$ and *role membership axioms* $(a, b) : R$, where C is a concept, $R \in \mathbf{R}$ and $a, b \in \mathbf{I}$.

A KB is usually assigned a semantics using interpretations of the form $\mathcal{I} = (\Delta^\mathcal{I}, \cdot^\mathcal{I})$, where $\Delta^\mathcal{I}$ is a non-empty *domain* and $\cdot^\mathcal{I}$ is the *interpretation function* that assigns an element in $\Delta^\mathcal{I}$ to each individual a, a subset of $\Delta^\mathcal{I}$ to each concept C and a subset of $\Delta^\mathcal{I} \times \Delta^\mathcal{I}$ to each role R. The mapping $\cdot^\mathcal{I}$ is extended to complex concepts as follows (where $R^\mathcal{I}(x, C) = \{y | \langle x, y\rangle \in R^\mathcal{I}, y \in C^\mathcal{I}\}$ and $\#X$ denotes the cardinality of the set X):

$$
\begin{aligned}
\top^\mathcal{I} &= \Delta^\mathcal{I} \\
\bot^\mathcal{I} &= \emptyset \\
\{a\}^\mathcal{I} &= \{a^\mathcal{I}\} \\
(\neg C)^\mathcal{I} &= \Delta^\mathcal{I} \setminus C^\mathcal{I} \\
(C_1 \sqcup C_2)^\mathcal{I} &= C_1^\mathcal{I} \cup C_2^\mathcal{I} \\
(C_1 \sqcap C_2)^\mathcal{I} &= C_1^\mathcal{I} \cap C_2^\mathcal{I} \\
(\exists R.C)^\mathcal{I} &= \{x \in \Delta^\mathcal{I} | R^\mathcal{I}(x) \cap C^\mathcal{I} \neq \emptyset\} \\
(\forall R.C)^\mathcal{I} &= \{x \in \Delta^\mathcal{I} | R^\mathcal{I}(x) \subseteq C^\mathcal{I}\} \\
(\geq nR.C)^\mathcal{I} &= \{x \in \Delta^\mathcal{I} | \#R^\mathcal{I}(x, C) \geq n\} \\
(\leq nR.C)^\mathcal{I} &= \{x \in \Delta^\mathcal{I} | \#R^\mathcal{I}(x, C) \leq n\} \\
(R^-)^\mathcal{I} &= \{(y, x) | (x, y) \in R^\mathcal{I}\} \\
(R_1 \circ ... \circ R_n)^\mathcal{I} &= R_1^\mathcal{I} \circ ... \circ R_n^\mathcal{I}
\end{aligned}
$$

A query over a KB is an axiom for which we want to test the entailment from the KB.

3 Semantics and Reasoning in Probabilistic DLs

DISPONTE [3] applies the distribution semantics to probabilistic ontologies [18]. In DISPONTE a *probabilistic knowledge base* $\mathcal{K}$ is a set of certain and probabilistic axioms. *Certain axioms* are regular DL axioms. *Probabilistic axioms* take the form $p :: E$, where p is a real number in $[0, 1]$ and E is a DL axiom.

The idea of DISPONTE is to associate independent Boolean random variables to the probabilistic axioms. By assigning values to every random variable we obtain a *world*, i.e. the union of the set of probabilistic axioms whose random variables take value 1 with the set of certain axioms.

Probability p can be interpreted as an *epistemic probability*, i.e., as the degree of our belief in axiom E. For example, a probabilistic concept membership axiom $p :: a : C$ means that we have degree of belief p in $C(a)$. The statement that Tweety flies with probability 0.9 can be expressed as $0.9 :: tweety : Flies$.

Let us now give the formal definition of DISPONTE. An *atomic choice* is a pair (E_i, k) where E_i is the ith probabilistic axiom and $k \in \{0, 1\}$. k indicates whether E_i is chosen to be included in a world ($k = 1$) or not ($k = 0$). A *composite choice* κ is a consistent set of atomic choices, i.e., $(E_i, k) \in \kappa, (E_i, m) \in \kappa \Rightarrow k = m$ (only one decision for each axiom). The probability of a composite

choice κ is $P(\kappa) = \prod_{(E_i,1)\in\kappa} p_i \prod_{(E_i,0)\in\kappa}(1-p_i)$, where p_i is the probability associated with axiom E_i. A *selection* σ is a composite choice that contains an atomic choice (E_i, k) for every probabilistic axiom of the theory. A selection σ identifies a theory w_σ called a *world* in this way: $w_\sigma = \{E_i|(E_i, 1) \in \sigma\}$. Let us indicate with $\mathcal{S}_\mathcal{K}$ the set of all selections and with $\mathcal{W}_\mathcal{K}$ the set of all worlds. The probability of a world w_σ is $P(w_\sigma) = P(\sigma) = \prod_{(E_i,1)\in\sigma} p_i \prod_{(E_i,0)\in\sigma}(1-p_i)$. $P(w_\sigma)$ is a probability distribution over worlds, i.e. $\sum_{w\in\mathcal{W}_\mathcal{K}} P(w) = 1$. We can now assign probabilities to queries. Given a world w, the probability of a query Q is defined as $P(Q|w) = 1$ if $w \models Q$ and 0 otherwise. The probability of a query can be defined by marginalizing the joint probability of the query and the worlds:

$$P(Q) = \sum_{w\in\mathcal{W}_\mathcal{K}} P(Q, w) = \sum_{w\in\mathcal{W}_\mathcal{K}} P(Q|w)P(w) = \sum_{w\in\mathcal{W}_\mathcal{K}:w\models Q} P(w) \qquad (1)$$

The system BUNDLE [17, 13, 15] computes the probability of a query w.r.t. KBs that follow the DISPONTE semantics by first computing all the explanations for the query and then building a Binary Decision Diagram (BDD) that represents them. An explanations κ for a query Q is a composite choices that identifies a set of worlds which all entail Q. Given the set K of all explanations for a query Q, we can define the Disjunctive Normal Form (DNF) Boolean formula f_K as $f_K(\mathbf{X}) = \bigvee_{\kappa\in K} \bigwedge_{(E_i,1)} X_i \bigwedge_{(E_i,0)} \overline{X_i}$. The variables $\mathbf{X} = \{X_i|(E_i, k) \in \kappa, \kappa \in K\}$ are independent Boolean random variables with $P(X_i = 1) = p_i$ and the probability that $f_K(\mathbf{X})$ takes value 1 gives the probability of Q. A BDD for a function of Boolean variables is a rooted graph that has one level for each Boolean variable. A node n has two children: one corresponding to the 1 value of the variable associated with the level of n, indicated with $child_1(n)$, and one corresponding to the 0 value of the variable, indicated with $child_0(n)$. When drawing BDDs, the 0-branch - the one going to $child_0(n)$ - is distinguished from the 1-branch by drawing it with a dashed line. The leaves store either 0 or 1.

BUNDLE finds the set K of all explanations for a query Q by means of the Pellet reasoner [19]. Then it builds a BDD representing function f_K. From it, the probability $P(f_K = 1)$, and thus the probability of Q, can be computed with a dynamic programming algorithm in polynomial time in the size of the diagram [5].

Example 1. Let us consider the following knowledge base, inspired by the ontology `people+pets` proposed in [11]:

$$\exists hasAnimal.Pet \sqsubseteq NatureLover$$
$$(kevin, fluffy) : hasAnimal$$
$$(kevin, tom) : hasAnimal$$
$$(E_1)\ 0.4 :: fluffy : Cat$$
$$(E_2)\ 0.3 :: tom : Cat$$
$$(E_3)\ 0.6 :: Cat \sqsubseteq Pet$$

Individuals that own an animal which is a pet are nature lovers and *kevin* owns the animals *fluffy* and *tom*. *fluffy* and *tom* are cats and cats are pets with the specified probability. This KB has eight worlds and the query axiom $Q = kevin : NatureLover$ is true in three of them, corresponding to the following selections: $\{(E_1, 1), (E_2, 0), (E_3, 1)\}, \{(E_1, 0), (E_2, 1), (E_3, 1)\}, \{(E_1, 1), (E_2, 1), (E_3, 1)\}$. The probability is $P(Q) = 0.4 \cdot 0.7 \cdot 0.6 + 0.6 \cdot 0.3 \cdot 0.6 + 0.4 \cdot 0.3 \cdot 0.6 = 0.348$. If we associate random variables X_1 with axiom E_1, X_2 with E_2 and X_3 with E_3, the BDD representing the set of explanations is shown in Figure 1.

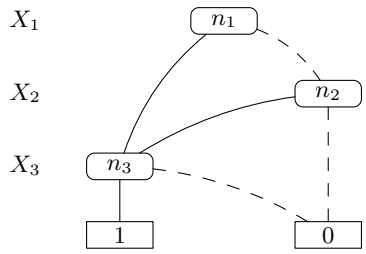


Fig. 1. BDD representing the set of explanations for the query of Example 1.

4 Parameter Learning for Probabilistic DLs

EDGE [14] adapts the algorithm EMBLEM [4], developed for learning the parameters for probabilistic logic programs, to the case of probabilistic DLs under the DISPONTE semantics. Inspired by [8], it performs an Expectation-Maximization cycle over Binary Decision Diagrams (BDDs).

EDGE performs supervised parameter learning. It takes as input a DL KB and a number of positive and negative examples that represent the queries in the form of concept assertions, i.e., in the form $a : C$ for an individual a and a class C. Positive examples represent information that we regard as true and for which we would like to get high probability while negative examples represent information that we regard as false and for which we would like to get low probability.

First, EDGE generates, for each query, the BDD encoding its explanations using BUNDLE. Then, EDGE starts the EM cycle in which the steps of Expectation and Maximization are iterated until a local maximum of the log-likelihood (LL) of the examples is reached. The LL of the examples is guaranteed to increase at each iteration. EDGE stops when the difference between the LL of the current iteration and that of the previous one drops below a threshold ϵ or when this difference is below a fraction δ of the previous LL. Finally, EDGE returns the reached LL and the probabilities p_i of the probabilistic axioms. EDGE's main procedure is illustrated in Alg. 1.

Algorithm 1 Function EDGE

```
function EDGE(K, P_E, N_E, ε, δ)                ▷ P_E, N_E: positive and negative examples
    Build BDDs                                   ▷ performed by BUNDLE
    LL = −∞
    repeat
        LL_0 = LL
        LL = EXPECTATION(BDDs)
        MAXIMIZATION
    until LL − LL_0 < ε ∨ LL − LL_0 < −LL_0 · δ
    return LL, p_i for all probabilistic axioms
end function
```

Function EXPECTATION (shown in Algorithm 2) takes as input a list of BDDs, one for each example Q, and computes the expectations $\mathbf{E}[c_{i0}|Q]$ and $\mathbf{E}[c_{i1}|Q]$ for all axioms E_i directly over the BDDs. c_{ix} represents the number of times a Boolean random variable X_i takes value x for $x \in \{0, 1\}$ and $\mathbf{E}[c_{ix}|Q] = P(X_i = x|Q)$. Then it sums up the contributions of all examples: $\mathbf{E}[c_{ix}] = \sum_Q \mathbf{E}[c_{ix}|Q]$.

Algorithm 2 Function EXPECTATION

```
function EXPECTATION(BDDs)
    LL = 0
    for all i ∈ Axioms do
        E[c_i0] = E[c_i1] = 0
    end for
    for all BDD ∈ BDDs do
        for all i ∈ Axioms do
            η^0(i) = η^1(i) = 0
        end for
        for all variables X do
            ς(X) = 0
        end for
        GETFORWARD(root(BDD))
        Prob=GETBACKWARD(root(BDD))
        T = 0
        for l = 1 to levels(BDD) do
            Let X_i be the variable associated with level l
            T = T + ς(X_i)
            η^0(i) = η^0(i) + T · (1 − p_i)
            η^1(i) = η^1(i) + T · p_i
        end for
        for all i ∈ Axioms do
            E[c_i0] = E[c_i0] + η^0(i)/Prob
            E[c_i1] = E[c_i1] + η^1(i)/Prob
        end for
        LL = LL + log(Prob)
    end for
    return LL
end function
```

In turn, $P(X_i = x|Q)$ is given by $\frac{P(X_i=x,Q)}{P(Q)}$. In Algorithm 2 we use $\eta^x(i)$ to indicate $\mathbf{E}[c_{ix}]$. EXPECTATION first calls procedures GETFORWARD and GETBACKWARD that compute the forward and the backward probability of nodes and $\eta^x(i)$ for non-deleted paths only. These are the paths that have not been deleted when building the BDDs. Forward and backward probabilities in each node represent the probability mass of paths from the root to the node and that

of the paths from the node to the leaves respectively. The expression

$$P(X_i = x, Q) = \sum_{n \in N(Q), v(n)=X_i} F(n)B(child_x(n))\pi_{ix}$$

represents the total probability mass of each path passing through the nodes associated with X_i and going down its x-branch, with $N(Q)$ the set of BDD nodes for query Q, $v(n)$ the variable associated with node n, $\pi_{i1} = p_i$, $\pi_{i0} = 1 - p_i$, $F(n)$ the forward probability of n, $B(n)$ the backward probability of n.

Computing the two probabilities in the nodes requires two traversals of the graph, so its cost is linear in the number of nodes. Procedure GetForward computes forward probabilities for every node. It traverses the diagram one level at a time starting from the root level, where $F(root) = 1$, and for each node n computes its contribution to the forward probabilities of its children. Then the forward probabilities of both children are updated. Function GetBackward computes backward probabilities of nodes by traversing recursively the tree from the leaves to the root. It returns the backward probability of the root corresponding to the probability of the query $P(Q)$, indicated with $Prob$ in Algorithm 2.

When the calls of GetBackward for both children of a node n return, we have the $e^x(n)$ and $\eta^x(i)$ values for non-deleted paths only. An array ς is used to store the contributions of the deleted paths that is then added to $\eta^x(i)$. See [14] for more details.

Expectations are updated for all axioms and finally the log-likelihood of the current example is added to the overall LL.

Function Maximization computes the parameters' values for the next EM iteration by relative frequency. Note that $\eta^x(i)$ values can be stored in a bi-dimensional array $eta[a, 2]$ where a is the number of axioms.

EDGE is written in Java for portability and ease of interface with Pellet. For further information about EDGE please refer to [14].

5 Distributed Parameter Learning for Probabilistic DLs

The aim of the present work is to develop a parallel version of EDGE that exploits the MapReduce strategy. We called it EDGE$^{\text{MR}}$ (see Algorithm 3).

5.1 Architecture

Like most MapReduce frameworks, EDGE$^{\text{MR}}$ architecture follows a master-slave model. The communication between the master and the slaves adopts the Message Passing Interface (MPI), in particular we used the OpenMPI[3] library which provides a Java interface to the native library. The processes of EDGE$^{\text{MR}}$ are not purely functional, as required by standard MapReduce frameworks such as Hadoop, because they have to retain in main memory the BDDs during the

[3] `http://www.open-mpi.org/`

whole execution. This forced us to develop a parallelization strategy exploiting MPI.

EDGE$^{\text{MR}}$ can be split into three phases: *Initialization*, *Query resolution* and *Expectation-Maximization*. All these operations are executed in parallel and synchronized by the master.

Initialization During this phase the data is replicated and a process is created on each machine. Then each process parses its copy of the probabilistic knowledge base and stores it in main memory. The master, in addition, parses the files containing the positive and negative examples (the queries).

Query resolution The master divides the set of queries into subsets and distributes them among the workers. Each worker generates its private subset of BDDs and keeps them in memory for the whole execution. Two different scheduling techniques can be applied for this operation. See Subsec. 5.3 for details.

Expectation-Maximization After all the nodes have built the BDDs for their queries, EDGE$^{\text{MR}}$ starts the Expectation-Maximization cycle. During the Expectation step all the workers traverse their BDDs and calculate their local *eta* array. Then the master gathers all the *eta*'s from the workers and aggregates them by summing the arrays component-wise. Then it calls the Maximization procedure in which it updates the parameters and sends them to the slaves. The cycle is repeated until one of the stopping criteria is satisfied.

5.2 MapReduce View

Since EDGE$^{\text{MR}}$ is based on MapReduce, it can be split into three phases: *Initialization*, *Map* and *Reduce*.

Initialization Described in Subsection 5.1.

Map This phase can be seen as a function that returns a set of *(key, value)* pairs, where *key* is an example identifier and *value* is the array *eta*.

- *Query resolution*: each worker resolves its chunks of queries and builds its private set of BDDs. Two different scheduling techniques can be applied for this operation. See Subsec. 5.3 for details.
- *Expectation Step*: each worker calculates its local *eta*.

Reduce This phase is performed by the master (also referred to as the "reducer") and it can be seen as a function that returns pairs (i, p_i), where i is an axiom identifier and p_i is its probability.

- *Maximization Step*: the master gathers all the *eta* arrays from the workers, aggregates them by summing component wise, performs the Maximization step and sends the newly update parameters to the slaves.

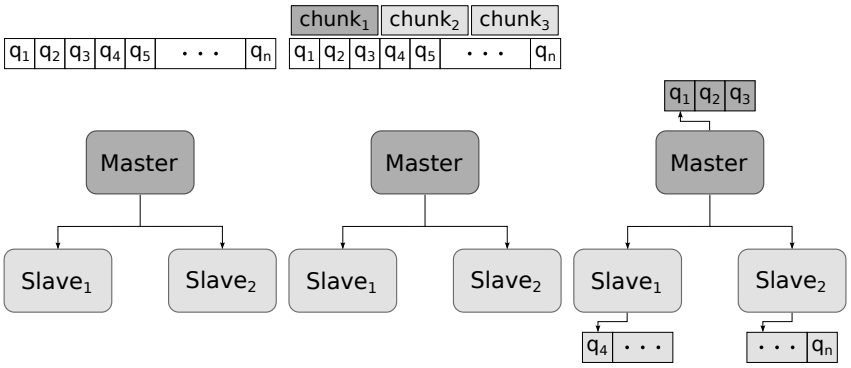
The Map and Reduce phases implement the functions Expectation and Maximization respectively, hence they are repeated until a local maximum is reached. It is important to notice that the Query Resulution step in the Map phase is executed only once because the workers keep in memory the generated BDDs for the whole execution of the EM cycle, what changes among iterations are the parameters of random variables.

Algorithm 3 Function EDGE$^{\mathrm{MR}}$

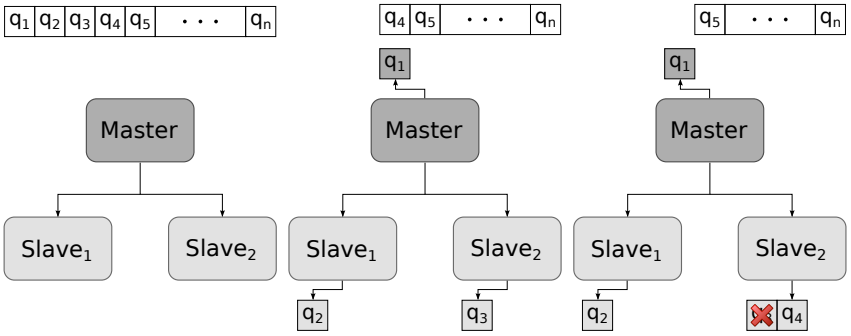
```
function EDGE^MR(K, P_E, N_E, S, ε, δ) ▷ P_E, N_E: pos. and neg. examples, S: scheduling method
    Read knowledge base K
    if MASTER then
        Identify examples E
        if S == dynamic then
            Send an example e_j to each slave
            Start thread listener        ▷ This thread sends an example to the slave at every request
            c = m − 1                                          ▷ c counts the computed examples
            while c < |E| do
                c = c + 1
                Build BDD_c for example e_c                              ▷ performed by BUNDLE
            end while
        else                                                              ▷ single-step scheduling
            Split examples E into n subsets E_1, ..., E_n
            Send E_m to each worker m, 2 ≤ m ≤ n
            Build BDDs_1 for examples E_1
        end if
        LL = −∞
        repeat
            LL_0 = LL
            Send the parameters p_i to each worker m, 2 ≤ m ≤ n
            LL = Expectation(BDDs_1)
            Collect LL_m and the expectations from each worker m, 2 ≤ m ≤ n
            Update LL and the expectations
            Maximization
        until LL − LL_0 < ε ∨ LL − LL_0 < −LL · δ
        Send STOP signal to all slaves
        return LL, p_i for all i
    else                                                                  ▷ the j-th slave
        if S == dynamic then
            while c < |E| do
                Receive e_j from master
                Build BDD_j for example e_j
                Request another example to the master
            end while
        else                                                              ▷ single-step scheduling
            Receive E_j from master
            Build BDDs_j for examples E_j
        end if
        repeat
            Receive the parameters p_i from master
            LL_j = Expectation(BDDs_j)
            Send LL_j and the expectations to master
        until Receive STOP signal from master
    end if
end function
```

5.3 Scheduling Techniques

In a distributed context the scheduling strategy influences significantly the performances. We evaluated two scheduling strategies, *single-step scheduling* and *dynamic scheduling*, during the generation of the BDDs for the queries, while the initialization and the EM phases are independent of the chosen scheduling method.



(a) Single-step scheduling



(b) Dynamic scheduling

Fig. 2. Scheduling techniques of EDGE$^{\mathrm{MR}}$.

Single-step Scheduling if N is the number of the slaves, the master divides the total number of queries into $N+1$ chunks, i.e. the number of slaves plus the master. Then the master starts $N+1$ threads, one building the BDD for its queries while the others sending the other chunks to the corresponding slave. After the master has terminated dealing with its queries, it waits for the results from the slaves. When the slowest slave returns its results to the

master, EDGE^{MR} proceeds to the EM cycle. Figure 2(a) shows an example of single-step scheduling with two slaves.

Dynamic Scheduling is more flexible and adaptive than single-step scheduling. Handling each query chunk may require a different amount of time. Therefore, with single-step scheduling, it could happen that a slave takes much more time than another one to deal with its chunk of queries. This may cause the master and some slaves to wait. Dynamic scheduling mitigates this issue. The user can establish a chunk dimension, i.e. the number of examples in each chunk. At first, each machine is assigned a chunk of queries in order. When it finishes handling the chunk, it takes the following chunk. So if the master ends handling its chunk, it just picks the next one, while if a slave ends handling its chunk, it asks the master for another one. During this phase the master runs a listener thread that waits for slaves' requests of new chunks. For each request, the listener starts a new thread that sends a chunk to the requesting slave (to improve the performances this is done through a thread pool). When all the BDDs for queries are built, EDGE^{MR} starts the EM cycle. An example of dynamic scheduling with two slaves and a chunk dimension of one example is displayed in Fig. 2(b).

6 Experiments

In order to evaluate the performances of EDGE^{MR}, four datasets were selected:

- **Mutagenesis** [21], which describes the mutagenicity of chemical compounds.
- **Carcinogenesis**[4] [20], which describes the carcinogenicity of chemical compounds.
- an extract of **DBPedia**[5] [9], a knowledge base obtained by extracting the structured data of Wikipedia.
- `education.data.gov.uk`[6], which contains information about school institutions in the United Kingdom.

The last three datasets are the same as in [16]. All experiments have been performed on a cluster of 64-bit Linux machines with 2 GB (max) memory allotted to Java per node. Each node of this cluster has 8-cores Intel Haswell 2.40 GHz CPUs.

For the generation of positive and negative examples, we randomly chose a set of individuals from the dataset. Then, for each extracted individual a, we sampled three named classes: A and B were selected among the named classes to which a explicitly belongs, while C was taken from the named classes to which a does not explicitly belong but that exhibits at least one explanation for the query $a : C$. The axiom $a : A$ was added to the KB, while $a : B$ was considered as a positive example and $a : C$ as a negative example. Then both

[4] http://dl-learner.org/wiki/Carcinogenesis
[5] http://dbpedia.org/
[6] http://education.data.gov.uk

the positive and the negative examples were split in five equally sized subsets and we performed five-fold cross-validation for each dataset and for each number of workers. Information about the datasets and training examples is shown in Table 1. We performed the experiments with 1, 3, 5, 9 and 17 nodes, where

Dataset	# of all axioms	# of probabilistic axioms	# of pos. examples	# of neg. examples	Fold size (MiB)
Carcinogenesis	74409	186	103	154	18.64
DBpedia	5380	1379	181	174	0.98
education.data.gov.uk	5467	217	961	966	1.03
Mutagenesis	48354	92	500	500	6.01

Table 1. Characteristics of the datasets used for evaluation.

the execution with 1 node corresponds to the execution of EDGE. Furthermore, we used both single-step and dynamic scheduling in order to evaluate the two scheduling approaches. It is important to point out that the quality of the learning is independent of the type of scheduling and of the number of nodes, i.e. the parameters found with 1 node are the same as those found with n nodes. Table 2 shows the running time in seconds for parameter learning on the four datasets with the different configurations. Figure 3 shows the speedup obtained

Dataset	EDGE	EDGE^{MR}							
		Dynamic				Single-step			
		3	5	9	17	3	5	9	17
Carcinogenesis	847	441.8	241	147.2	94.2	384	268.4	179.2	117.8
DBpedia	1552	1259.8	634	364.6	215.2	1155.6	723.8	452.6	372.6
education.data.gov.uk	6924.2	3878.2	2157.2	1086	623.2	3611.6	2289.6	1331.6	749.4
Mutagenesis	1439.4	635.8	399.8	223.2	130.4	578.2	359.2	230	124.6

Table 2. Comparison between EDGE and EDGE^{MR} in terms of running time (in seconds) for parameter learning.

as a function of the number of machines (nodes). The speedup is the ratio of the running time of 1 worker to the running time of n workers. We can note that the speedup is significant even if it is sublinear, showing that a certain amount of overhead (the resources, and therefore the time, spent for the MPI communications) is present. The dynamic scheduling technique has generally better performance than single-step scheduling.

7 Related Work

The pervasiveness of Internet, the availability of sensor data, the dramatically increased storage and computational capabilities provide the opportunity to gather

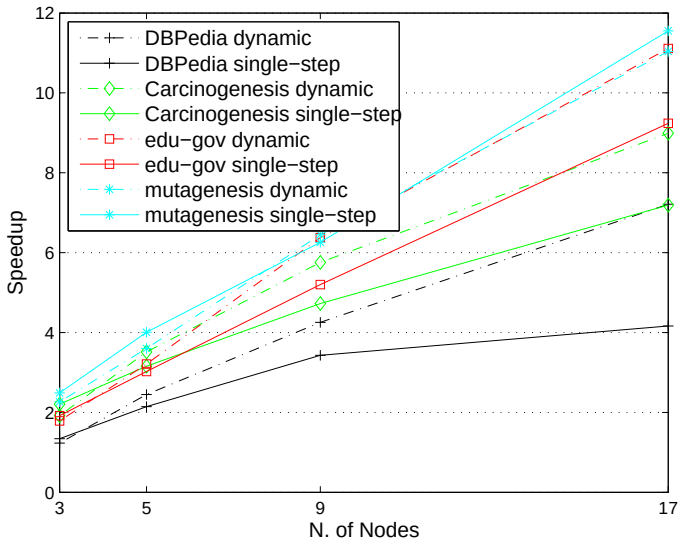


Fig. 3. Speedup of EDGE$^{\mathrm{MR}}$ relative to EDGE with single-step and dynamic schedulings.

huge sets of data. This large amount of information is known by the name *Big Data*. The ability to process and perform learning and inference over massive data is one of the major challenges of the current decade. Big data is strictly intertwined with the availability of scalable and distributed algorithms.

In [1] the authors developed a method to parallelize inference and training on a probabilistic relational model. They show that (loopy) belief propagation can be lifted and that lifting is MapReduce-able. In addition they show that the MapReduce approach improves performances significantly. For parameter learning, they propose an approximate method which is MapReduce-able as well. In order to train large models, they shatter the factor graph into smaller pieces which can be elaborated locally in a distributed fashion by exploiting the MapReduce approach.

Other specific distributed implementations have been developed for various machine learning methods, such as support vector machines [22], robust regression [10] and extreme learning machines [7].

8 Conclusions

EDGE is an algorithm for learning the parameters of DISPONTE probabilistic knowledge bases. In this paper we presented EDGE$^{\mathrm{MR}}$, a distributed version of EDGE based on the MapReduce approach.

We performed experiments over four datasets with an increasing number of nodes. The results show that parallelization significantly reduces the execution time, even if with a sublinear trend due to overhead.

We are currently working on a way to distribute structure learning of DISPONTE probabilistic knowledge bases. In particular, we aim at developing a MapReduce version of LEAP [16]. Moreover, we plan to investigate the possibility of parallelizing and distributing also the inference.

References

1. Ahmadi, B., Kersting, K., Mladenov, M., Natarajan, S.: Exploiting symmetries for scaling loopy belief propagation and relational training. Machine Learning 92(1), 91–132 (2013)
2. Baader, F., Calvanese, D., McGuinness, D.L., Nardi, D., Patel-Schneider, P.F. (eds.): The Description Logic Handbook: Theory, Implementation, and Applications. Cambridge University Press, New York, NY, USA (2003)
3. Bellodi, E., Lamma, E., Riguzzi, F., Albani, S.: A distribution semantics for probabilistic ontologies. In: International Workshop on Uncertainty Reasoning for the Semantic Web. CEUR Workshop Proceedings, vol. 778, pp. 75–86. Sun SITE Central Europe (2011)
4. Bellodi, E., Riguzzi, F.: Expectation Maximization over Binary Decision Diagrams for probabilistic logic programs. Intell. Data Anal. 17(2), 343–363 (2013)
5. De Raedt, L., Kimmig, A., Toivonen, H.: ProbLog: A probabilistic Prolog and its application in link discovery. In: 20th International Joint Conference on Artificial Intelligence. vol. 7, pp. 2462–2467. AAAI Press (2007)
6. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. Commun. ACM 51(1), 107–113 (2008)
7. He, Q., Shang, T., Zhuang, F., Shi, Z.: Parallel extreme learning machine for regression based on mapreduce. Neurocomputing 102, 52–58 (2013)
8. Ishihata, M., Kameya, Y., Sato, T., Minato, S.: Propositionalizing the EM algorithm by BDDs. In: Late Breaking Papers of the International Conference on Inductive Logic Programming. pp. 44–49 (2008)
9. Lehmann, J., Isele, R., Jakob, M., Jentzsch, A., Kontokostas, D., Mendes, P.N., Hellmann, S., Morsey, M., van Kleef, P., Auer, S., Bizer, C.: DBpedia - a large-scale, multilingual knowledge base extracted from wikipedia. Semantic Web Journal 6(2), 167–195 (2015)
10. Meng, X., Mahoney, M.: Robust regression on mapreduce. In: Proceedings of The 30th International Conference on Machine Learning. pp. 888–896 (2013)
11. Patel-Schneider, P, F., Horrocks, I., Bechhofer, S.: Tutorial on OWL (2003)
12. Riguzzi, F., Bellodi, E., Lamma, E., Zese, R.: Epistemic and statistical probabilistic ontologies. In: Uncertainty Reasoning for the Semantic Web. CEUR Workshop Proceedings, vol. 900, pp. 3–14. Sun SITE Central Europe (2012)
13. Riguzzi, F., Bellodi, E., Lamma, E., Zese, R.: Computing instantiated explanations in OWL DL. In: AI*IA 2013. LNAI, vol. 8249, pp. 397–408. Springer (2013)
14. Riguzzi, F., Bellodi, E., Lamma, E., Zese, R.: Parameter learning for probabilistic ontologies. In: Faber, W., Lembo, D. (eds.) RR 2013. LNCS, vol. 7994, pp. 265–270. Springer Berlin Heidelberg (2013)

15. Riguzzi, F., Bellodi, E., Lamma, E., Zese, R.: Probabilistic description logics under the distribution semantics. Semantic Web - Interoperability, Usability, Applicability 6(5), 447–501 (2015)
16. Riguzzi, F., Bellodi, E., Lamma, E., Zese, R., Cota, G.: Learning probabilistic description logics. In: Bobillo, F., Carvalho, R.N., Costa, P.C., d'Amato, C., Fanizzi, N., Laskey, K.B., Laskey, K.J., Lukasiewicz, T., Nickles, M., Pool, M. (eds.) Uncertainty Reasoning for the Semantic Web III, pp. 63–78. LNCS, Springer International Publishing (2014)
17. Riguzzi, F., Lamma, E., Bellodi, E., Zese, R.: BUNDLE: A reasoner for probabilistic ontologies. In: Faber, W., Lembo, D. (eds.) RR 2013. LNCS, vol. 7994, pp. 183–197. Springer Berlin Heidelberg (2013)
18. Sato, T.: A statistical learning method for logic programs with distribution semantics. In: Proceedings of the 12th International Conference on Logic Programming. pp. 715–729. MIT Press (1995)
19. Sirin, E., Parsia, B., Cuenca-Grau, B., Kalyanpur, A., Katz, Y.: Pellet: A practical OWL-DL reasoner. J. Web Semant. 5(2), 51–53 (2007)
20. Srinivasan, A., King, R.D., Muggleton, S., Sternberg, M.J.E.: Carcinogenesis predictions using ILP. In: Lavrac, N., Dzeroski, S. (eds.) 7th International Workshop on Inductive Logic Programming. LNCS, vol. 1297, pp. 273–287. Springer Berlin Heidelberg (1997)
21. Srinivasan, A., Muggleton, S., Sternberg, M.J.E., King, R.D.: Theories for mutagenicity: A study in first-order and feature-based induction. Artif. Intell. 85(1-2), 277–299 (1996)
22. Sun, Z., Fox, G.: Study on parallel svm based on mapreduce. In: International Conference on Parallel and Distributed Processing Techniques and Applications. pp. 16–19 (2012)