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 high since it often takes hours to complete an execution. In this paper we present EDGE^{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^{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 [14] is one of them, applied in the field of Logic Programming.

In [2, 13, 8] 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 programming to DLs.

EDGE [10], 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 execution of this algorithm becomes rather

expensive from a computational point of view, taking a few 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.

In order to reduce EDGE running time, we developed EDGE$^{\mathrm{MR}}$, which represents a MapReduce implementation of EDGE.

MapReduce [5] is a model for processing data with a parallel algorithm on a cluster. In this model the work is distributed among mapper and reducer workers. The mappers take the data and return a set of (key, value) pairs. These sets are then grouped according to the key 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 the map and reduce functions for EDGE were relatively easy to implement with MPI and we wanted the overhead to be as small as possible.

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 reveal that EDGE$^{\mathrm{MR}}$ effectively reduces EDGE running time. Nevertheless, due to the overhead of the communication tasks, the reached speedup is less than linear.

The paper is structured as follows. Section 2 introduces Description Logics while Section 3 summarises the DISPONTE semantics and an inference system for probabilistic DLs. Section 4 briefly describes EDGE. In Section 5 EDGE$^{\mathrm{MR}}$ is presented. Section 6 shows the results of the experiments for evaluating EDGE$^{\mathrm{MR}}$. Finally, Section 7 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 [1].

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. For the sake of simplicity we consider and describe $\mathcal{ALC}$, but the proposed algorithm can work with $\mathcal{SROIQ}(\mathbf{D})$ DLs as well.

We use $\mathbf{A}$, $\mathbf{R}$ and $\mathbf{I}$ to indicate *atomic concepts*, *atomic roles* and *individuals*, respectively. A *role* is an atomic role $R \in \mathbf{R}$. Each $A \in \mathbf{A}$, $\bot$ and $\top$ are concepts. If C, C_1 and C_2 are concepts and $R \in \mathbf{R}$, then $(C_1 \sqcap C_2)$, $(C_1 \sqcup C_2)$ and $\neg C$ are concepts, as well as $\exists R.C$ and $\forall R.C$. Let C and D be concepts, R be a role and a and b be individuals, a *TBox* $\mathcal{T}$ is a finite set of *concept inclusion axioms* $C \sqsubseteq D$, while an *ABox* $\mathcal{A}$ is a finite set of *concept membership axioms* $a : C$ and

role membership axioms $(a, b) : R$. A *knowledge base* (KB) $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ consists of a TBox $\mathcal{T}$ and an ABox $\mathcal{A}$.

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 all concepts as:

$$\begin{aligned} \top^{\mathcal{I}} &= \Delta^{\mathcal{I}} & \bot^{\mathcal{I}} &= \emptyset \\ (\neg C)^{\mathcal{I}} &= \Delta^{\mathcal{I}} \setminus C^{\mathcal{I}} & (C_1 \sqcap C_2)^{\mathcal{I}} &= C_1^{\mathcal{I}} \cap C_2^{\mathcal{I}} \\ (C_1 \sqcup C_2)^{\mathcal{I}} &= C_1^{\mathcal{I}} \cup C_2^{\mathcal{I}} & (\forall R.C)^{\mathcal{I}} &= \{x \in \Delta^{\mathcal{I}} | R^{\mathcal{I}}(x) \subseteq C^{\mathcal{I}}\} \\ (\exists R.C)^{\mathcal{I}} &= \{x \in \Delta^{\mathcal{I}} | R^{\mathcal{I}}(x) \cap C^{\mathcal{I}} \neq \emptyset\} \end{aligned}$$

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 [2] applies the distribution semantics to probabilistic ontologies [14]. 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 on value 1 and the set of certain axioms.

The 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 formula). 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 [13, 9, 11] 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. A *set of explanations* K for a query Q is a set of composite choices that identify a set of sets of worlds which entail 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 and the probability that $f_K(\mathbf{X})$ takes on value 1 is equal to 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.

Explanations are found by means of the Pellet reasoner [15] and are then translated into a BDD that allows to compute the probability of Q with a dynamic programming algorithm in polynomial time in the size of the diagram [4].

Example 1. Let us consider the following knowledge base, inspired by the ontology `people+pets` proposed in [7]:

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

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 choices: $\{(E_1, 1), (E_2, 0), (E_3, 1)\}, \{(E_1, 0), (E_2, 1), (E_3, 1)\}, \{(E_1, 1), (E_2, 1), (E_3, 1)\}$. 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 the random variables X_1 with the 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.

4 Parameter Learning for Probabilistic DLs

EDGE [10] adapts the algorithm EMBLEM [3], developed for learning the parameters for probabilistic logic programs, to the case of probabilistic DLs un-

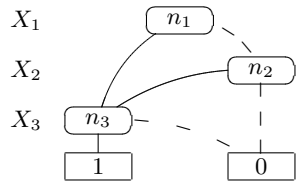


Fig. 1. BDD representing the set of explanations for the query of Example 1.

der the DISPONTE semantics. Inspired by [6], 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 on 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
```

Finally, $P(X_i = x|Q)$ is given by $\frac{P(X_i=x,Q)}{P(Q)}$. In this procedure we use $\eta^x(i)$ to indicate $P(X_i = x, Q)$. 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

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

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, represents the probability mass of each path passing through each node associated with X_i and going down its x-branch. We use the notation $e^x(n)$ to indicate the expression inside the sum.

Computing the two types of probability in the nodes requires two traversals of the graph, so its cost is linear in the number of nodes.

Procedure GETFORWARD computes the value of the 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 the backward

probability 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 as $Prob$ in Algorithm 2.

When the calls of GetBackward for both children of a node n return, we compute the $e^x(n)$ and $\eta^x(i)$ values for non-deleted paths. An array ς is used to store the contributions of the deleted paths. See [10] 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.

EDGE is written in Java, hence it is highly portable. For further information about EDGE please refer to [10].

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 method in order to compute the parameters. 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 is done by means of the Message Passing Interface (MPI), specifically we use 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 all BDDs during all iterations. 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. Thereafter 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 slaves. Each slave 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. 5.2 for details.

Expectation-Maximization After all the nodes have built the BDDs for all the queries, EDGE$^{\text{MR}}$ starts the Expectation-Maximization cycle. During the Expectation step all the slaves traverse their BDDs and calculate their

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

local set of $\eta^x(i)$. Then the master gathers all the $\eta^x(i)$ from the slaves and aggregates them by summing the vectors component-wise. Then it calls the Maximization procedure in which it estimates the parameters and sends them to the slaves. The cycle is repeated until one of the stopping criteria is reached.

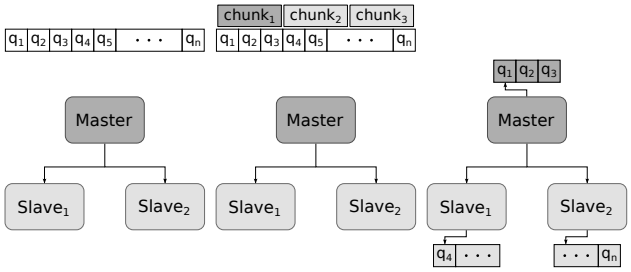
Algorithm 3 Function EDGE$^{\mathrm{MR}}$

```
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
```

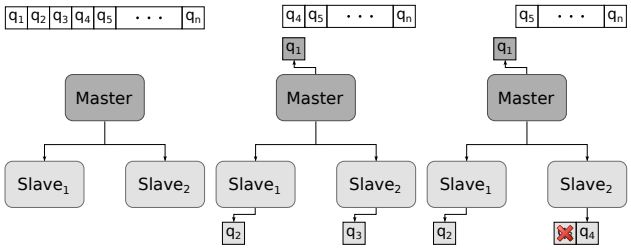
5.2 Scheduling Techniques

In a distributed context the performances depend on the scheduling strategy. We evaluated two scheduling strategies, *single-step scheduling* and *dynamic schedul-*

ing, during the generation of the BDDs for each query, 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 begins to compute its queries while, for each other chunk of queries, the master starts a thread which takes care to send a chunk 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$^{\mathrm{MR}}$ 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 may require a different amount of time. Therefore with single-step scheduling it could happen that a slave takes a lot more time than another slave to deal with its chunk of queries. Hence the master and some slaves could be idle. Dynamic scheduling mitigates this issue. At first each machine is assigned a query in order. Then if the master ends handling a query it just takes the next one, instead, if a slave ends handling a query, it asks the master for a new query and the master sends the next query to the slave. During this phase the master runs a thread listener that

waits for the slaves' requests of new queries and for each request starts a new thread that sends a query to the slave which has done the request. When all the queries are evaluated, EDGE^{MR} starts the EM cycle. An example of dynamic scheduling with two slaves is displayed in Fig. 2(b).

6 Experiments

In order to evaluate the performances of EDGE^{MR}, four datasets were selected: Mutagenesis, Carcinogenesis, an extract of DBpedia and `education.data.gov.uk`. The last three datasets are the same as in [12]. 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 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.

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.

We performed the experiments with 1, 3, 5, 9 and 17 nodes, where 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 number of nodes, i.e. the parameters found with 1 node are the same as those found with n nodes, and of the type of scheduling. Table 2 shows the running time in seconds for parameter learning on the three datasets with different configurations.

Figure 3 shows the speedup obtained 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

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.

sublinear, showing that a certain amount of overhead (the resources, and thereby the time, spent for the MPI communications) is present. The dynamic scheduling technique has generally better performance than single-step scheduling.

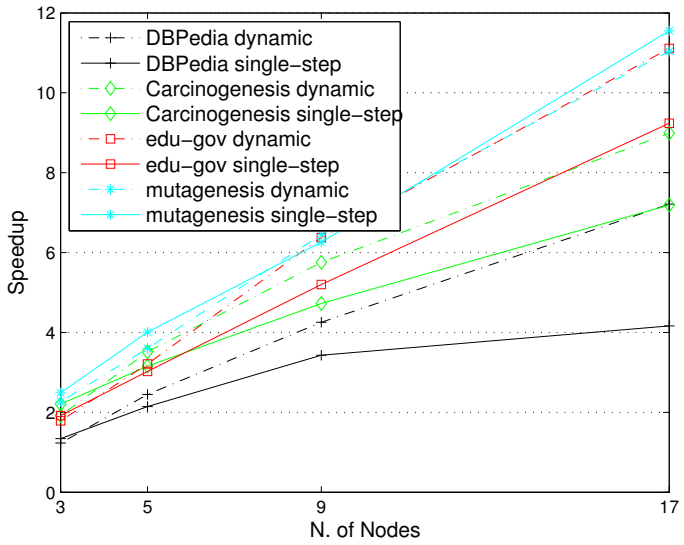


Fig. 3. Speedup of EDGE^{MR} relative to EDGE with single-step and dynamic schedulings.

7 Conclusions

EDGE is an algorithm for learning the parameters of probabilistic knowledge bases under the DISPONTE semantics. In this paper we presented EDGE^{MR}, which is 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 in a sublinear trend. The sublinearity is caused by the overhead.

We are currently working on a way to distribute structure learning of probabilistic knowledge bases under DISPONTE semantics. In particular we would like to develop a MapReduce version of LEAP [12].

References

1. 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)
2. Bellodi, E., Lamma, E., Riguzzi, F., Albani, S.: A distribution semantics for probabilistic ontologies. CEUR Workshop Proceedings, vol. 778, pp. 75–86. Sun SITE Central Europe (2011)
3. Bellodi, E., Riguzzi, F.: Expectation Maximization over Binary Decision Diagrams for probabilistic logic programs. Intell. Data Anal. 17(2), 343–363 (2013)
4. 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)
5. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. Commun. ACM 51(1), 107–113 (2008)
6. 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)
7. Patel-Schneider, P, F., Horrocks, I., Bechhofer, S.: Tutorial on OWL (2003)
8. 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)
9. 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)
10. 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, Heidelberg, Germany (2013)
11. 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)
12. 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.) URSW III, pp. 63–78. LNCS, Springer International Publishing (2014)
13. 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, Heidelberg, Germany (2013)
14. 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)
15. 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)