Structure Learning of Probabilistic Logic Programs by MapReduce

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

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

Abstract. Probabilistic Logic Programming has been shown to be a useful language for Inductive Logic Programming: for instance, the system SLIPCOVER learns high quality theories in a variety of domains. However, the computational cost of SLIPCOVER is sometimes expensive, with a running time of the order of hours. In this paper we present the system SEMPRE for "Structure lEarning by MaPREduce", that implements SLIPCOVER by applying a particularly simple MapReduce strategy, directly implemented with the Message Passing Interface. SEMPRE has been tested on various domains and shown to effectively reduce SLIPCOVER running time, even if the speedup is often sublinear.

Keywords: Probabilistic Logic Programming, Parameter Learning, Structure Learning, MapReduce

1 Introduction

Probabilistic Logic Programming (PLP) represents an interesting language for Inductive Logic Programming (ILP), because it allows algorithms to better deal with uncertain information. The distribution semantics [23] is an approach to PLP that is particularly attractive for its intuitiveness and for the interpretability of the programs. Various algorithms have been proposed for learning the parameters of probabilistic logic programs under the distribution semantics, such as PRISM [24], ProbLog2 [11] and EMBLEM [3]. Recently, systems for learning the structure of these programs have started to appear. Among these, SLIPCASE [2] performs a beam search in the space of possible theories using the log-likelihood (LL) of the examples as the heuristics while SLIPCOVER [4] performs a beam search in the space of clauses using LL as the heuristics again.

These systems demonstrated the ability to learn good quality solutions in a variety of domains [4] but are usually costly, often taking some hours to complete on datasets of the order of MBs. However, we are experiencing a rapid growth in the size of the datasets as testified by the Big Data movement. In order to deal with Big Data, it is fundamental to reduce learning times by exploiting modern computing infrastructures such as clusters and clouds.

MapReduce [10] is an approach for exploiting such infrastructures that distributes the work among a pool of mapper and reducer worker nodes. The computation is performed by dividing the input among mappers, each taking a set of units of information and returning a set of (key, value) pairs. These sets are then given to reducers in the form of pairs (key, list of values) and the reducers compute an aggregate of the values returning a set of (key', aggregated value) couples that represents the output of the task.

In this paper, we propose the system SEMPRE for "Structure lEarning by MaPREduce" that represents a MapReduce implementation of SLIPCOVER. We preferred to parallelize SLIPCOVER over SLIPCASE since it has been shown to give much better results in [4].

MapReduce can be realized using various frameworks, such as Hadoop or [6] that is specifically tailored towards Prolog. However, we decided to avoid using a framework and implement the MapReduce strategy of SEMPRE directly using a Message Passing Interface (MPI): in fact, our mapper workers keep in memory some data structures across MapReduce iteration and the reduce strategy is particularly simple, being realized by a single reducer receiving the output from all mapper jobs.

We experimentally evaluated SEMPRE by running it on various datasets using 1, 8, 16 and 32 nodes. The results show that SEMPRE significantly reduces SLIPCOVER running time, even if the speedup is often less than linear because of a (sometimes) relevant overhead.

The paper is organized as follows. Section 2 summarises PLP under the distribution semantics. Section 3 describes EMBLEM and SLIPCOVER algorithms for parameter and structure learning of probabilistic logic programs. Section 4 describes EMBLEM$^{\mathrm{MR}}$, the MapReduce version of EMBLEM. Section 5 discusses SEMPRE. Section 6 presents the experiments while Section 7 concludes the paper.

2 Probabilistic Logic Programming

We introduce PLP focusing on the distribution semantics. We use Logic Programs with Annotated Disjunctions (LPADs) as the language for their general syntax and we do not allow function symbols; for the treatment of function symbols see [20].

LPADs [27] consist of a finite set of annotated disjunctive clauses C_i of the form $h_{i1} : \Pi_{i1}; \ldots ; h_{in_i} : \Pi_{in_i} : -b_{i1}, \ldots , b_{im_i}$. Here, $b_{i1}, \ldots , b_{im_i}$ are logical literals which form the *body* of C_i, denoted by $body(C_i)$, while $h_{i1}, \ldots h_{in_i}$ are logical atoms and $\{\Pi_{i1}, \ldots , \Pi_{in_i}\}$ are real numbers in the interval $[0, 1]$ such that $\sum_{k=1}^{n_i} \Pi_{ik} \leq 1$. Note that if $n_i = 1$ and $\Pi_{i1} = 1$ the clause corresponds to a non-disjunctive clause. Otherwise, if $\sum_{k=1}^{n_i} \Pi_{ik} < 1$, the head of the annotated disjunctive clause implicitly contains an extra atom *null* that does not appear in the body of any clause and whose annotation is $1 - \sum_{k=1}^{n_i} \Pi_{ik}$. The grounding of an LPAD $\mathcal{T}$ is denoted by $ground(\mathcal{T})$.

An *atomic choice* is a triple (C_i, θ_j, k) where $C_i \in \mathcal{T}$, θ_j is a substitution that grounds C_i and $k \in \{1, \ldots, n_i\}$ identifies a head atom of C_i. It corresponds to an assignment $X_{ij} = k$, where X_{ij} is a multi-valued random variable which correspond to $C_i\theta_j$. A set of atomic choices κ is *consistent* if only one head is selected from a ground clause. In this case it is called *composite choice.* The *probability* $P(\kappa)$ *of a composite choice* κ is computed by multiplying the probabilities of the individual atomic choices, i.e. $P(\kappa) = \prod_{(C_i,\theta_j,k)\in\kappa} \Pi_{ik}$. A *selection* σ is a composite choice that, for each clause $C_i\theta_j$ in $ground(\mathcal{T})$, contains an atomic choice (C_i, θ_j, k). It identifies a *world* w_σ of $\mathcal{T}$, i.e. a normal logic program defined as $w_\sigma = \{(h_{ik} \leftarrow body(C_i))\theta_j | (C_i, \theta_j, k) \in \sigma\}$. Since selections are composite choices, the probability of the worlds is $P(w_\sigma) = P(\sigma)$. We denote by $S_\mathcal{T}$ the set of all selections and by $W_\mathcal{T}$ the set of all worlds of a program $\mathcal{T}$. A composite choice κ identifies a set of worlds $\omega_\kappa = \{w_\sigma | \sigma \in S_\mathcal{T}, \sigma \supseteq \kappa\}$. We define the set of worlds identified by a set of composite choices K as $\omega_K = \bigcup_{\kappa\in K} \omega_\kappa$.

We consider only *sound* LPADs, where each possible world has a total well-founded model, so $w_\sigma \models Q$ means a query Q is true in the well-founded model of the program w_σ. The probability of a query Q given a world w is $P(Q|w) = 1$ if $w \models Q$ and 0 otherwise. The probability of Q is then:

$$P(Q) = \sum_{w\in W_\mathcal{T}} P(Q, w) = \sum_{w\in W_\mathcal{T}} P(Q|w)P(w) = \sum_{w\in W_\mathcal{T}: w\models Q} P(w) \qquad (1)$$

Example 1. The following LPAD $\mathcal{T}$ models the fact that if somebody has the flu and the climate is cold, there is the possibility that an epidemic or a pandemic arises:

$C_1 = epidemic : 0.6; pandemic : 0.3 : -flu(X), cold.$
$C_2 = cold : 0.7.$
$C_3 = flu(david).$
$C_4 = flu(robert).$

T has 18 instances, the query $Q = epidemic$ is true in 5 of them and its probability is $P(epidemic) = 0.6 \cdot 0.6 \cdot 0.7 + 0.6 \cdot 0.3 \cdot 0.7 + 0.6 \cdot 0.1 \cdot 0.7 + 0.3 \cdot 0.6 \cdot 0.7 + 0.1 \cdot 0.6 \cdot 0.7 = 0.588$.

Since in practice enumerating all the worlds where Q is true is unfeasible, inference algorithms find a covering set of *explanations* for Q, i.e. a set of composite choices K such that Q is true in a world w_σ iff $w_\sigma \in \omega_K$. For Example 1, a covering set of explanations is $\{\{(C_1, \{X/david\}, 1), (C_2, \emptyset, 1)\}, \{(C_1, \{X/robert\}, 1), (C_2, \emptyset, 1)\}\}$ where non-disjunctive clauses are omitted.

From the set K, the following Boolean function is built:

$$f_K(\mathbf{X}) = \bigvee_{\kappa\in K} \bigwedge_{(C_i,\theta_j,k)\in\kappa} (X_{ij} = k) \qquad (2)$$

where $\mathbf{X} = \{X_{ij} | C_i$ is a clause and θ_j is a grounding substitution of $C_i\}$ are multi-valued random variables. The domain of X_{ij} is $1, \ldots, n_i$ and its probability distribution is given by $P(X_{ij} = k) = \Pi_{ik}$. The problem of computing $P(Q)$ can be solved by computing the probability that $f_K(\mathbf{X})$ takes on value true. For Example 1, (2) is given by

$$f_K(\mathbf{X}) = (X_{11} \wedge X_{21}) \vee (X_{12} \wedge X_{21}) \tag{3}$$

where X_{11} corresponds to $(C_1, \{X/david\})$, X_{12} corresponds to $(C_1, \{X/robert\})$ and X_{21} corresponds to $(C_2, \emptyset)$.

If we associate a multi-valued variable X_{ij}, corresponding to the ground clause $C_i\theta_j$, having n_i values, with $n_i - 1$ Boolean variables $X_{ij1}, \dots, X_{ijn_i-1}$, the equation $X_{ij} = k$ for $k = 1, \dots n_i - 1$ corresponds with the conjunction $\overline{X_{ij1}} \wedge \dots \wedge \overline{X_{ijk-1}} \wedge X_{ijk}$, while the equation $X_{ij} = n_i$ with $\overline{X_{ij1}} \wedge \dots \wedge \overline{X_{ijn_i-1}}$. Following this approach, which provides good performance [21], $f_K(\mathbf{X})$ in (2) can be translated into a function of Boolean random variables. For Example 1, $X_{11} = 1$ is represented as X_{111} and $X_{11} = 2$ as $\overline{X_{111}} \wedge X_{112}$. Let us call $f'_K(\mathbf{X'})$ the result of replacing multi-valued random variables with Boolean variables in $f_K(\mathbf{X})$. The probability distribution of the Boolean random variables X_{ijk} is computed from that of multi-valued variables as $\pi_{i1} = \Pi_{i1}, \dots, \pi_{ik} = \frac{\Pi_{ik}}{\prod_{j=1}^{k-1}(1-\pi_{ij})}$ up to $k = n_i - 1$, where π_{ik} is the probability that X_{ijk} is true. With this distribution the probability that $f'_K(\mathbf{X'})$ is true is the same as $f_K(\mathbf{X}) = P(Q)$. For Example 1, $f'_K(\mathbf{X'})$ is given by

$$f'_K(\mathbf{X'}) = (X_{111} \wedge X_{211}) \vee (X_{121} \wedge X_{211}) \tag{4}$$

Computing the probability that $f'_K(\mathbf{X'})$ is true is a SUM-OF-PRODUCTS problem and it was shown to be #P-hard [16]. *Knowledge compilation*, that was found to give good results in practice [8], consists of translating $f'_K(\mathbf{X'})$ to a target language that allows answering queries in polynomial time, such as Binary Decision Diagrams (BDD). From a BDD we can compute the probability of the query with a dynamic programming algorithm that is linear in its size [9]. Algorithms that adopt such an approach for inference include [17–19].

A BDD for a function of Boolean variables is a rooted graph that has one level for each Boolean variable. A node n in a BDD has two children: one corresponding to the 1 value of the variable associated with n, and one corresponding to the 0 value of the variable. The leaves store either 0 or 1.

BDDs can be built in practice by highly efficient software packages such as CUDD[3]. A BDD for function (4) is shown in Figure 1.

3 Learning LPADs

BDDs are employed to efficiently perform *parameter learning* of LPADs by the system EMBLEM [3], based on an Expectation Maximization (EM) algorithm (see Algorithm 1). It takes as input a set of interpretations I, i.e., sets of ground facts describing a portion of the domain, and the theory T for which we want to

[3] Available at `http://vlsi.colorado.edu/~fabio/CUDD/`

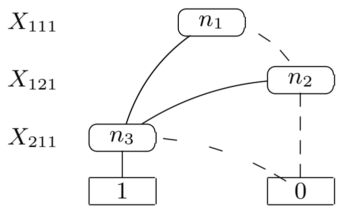

Fig. 1. BDD for function (4). The dashed branch is the one who goes to the child corresponding with the 0 value of the variable.

learn the parameters. It is targeted at discriminative learning, since the user has to indicate which predicate(s) of the domain is/are *target*, the one(s) for which we are interested in good predictions. The interpretations must contain also negative facts for target predicates. All ground atoms for the target predicates (E) will represent the positive and negative examples (*queries*) for which BDDs are built, encoding the disjunction of their explanations.

Algorithm 1. Function EMBLEM

```
1: function EMBLEM($I, T, \epsilon, \delta$)
2:     Identify examples $E$
3:     Build $BDDs$ for the examples $E$ using $T$ and $I$
4:     $LL = -\infty$
5:     repeat
6:         $LL_0 = LL$
7:         $LL = $ Expectation($BDDs$)
8:         Maximization
9:     until $LL - LL_0 < \epsilon \vee LL - LL_0 < -LL \cdot \delta$
10:    return $LL, \pi$
11: end function
```

After building the BDDs, EMBLEM maximizes the LL for the positive and negative target examples with an EM cycle, until it has reached a local maximum. The E-step computes the expectations of the latent variables directly over BDDs and returns the LL of the data that is used in the stopping criterion. The expected counts are then used in the M-step, which updates the parameters π for all clauses for the next EM iteration by relative frequency.

SLIPCOVER [4] (see Algorithm 2) learns the structure of probabilistic logic programs with a two-phase search strategy: (1) beam search in the space of clauses in order to find a set of promising clauses and (2) greedy search in the space of theories. In the first phase SLIPCOVER performs clause search for each target predicate separately. The beam for each target predicate is initialized (Function InitialBeams) with a number of bottom clauses built as in Progol [15]. Then SLIPCOVER generates refinements of the best clause in the beam and evaluates them through LL by invoking EMBLEM. Each clause is then inserted in the new beam of promising clauses and in the sets of target and background clauses ordered according to the LL. This is repeated until the original beam becomes empty. The whole process is repeated at most *NI* steps.

The search in the space of theories starts from an empty theory which is iteratively extended with one target clause at a time from those generated in the previous beam search. The algorithm starts with an empty theory and then iteratively adds a new clause to the theory, runs EMBLEM to compute the corresponding LL and checks whether to keep the clause in the theory or not. If the LL of the new theory decreases, SLIPCOVER removes form the theory the last inserted clause before selecting the new clause to add.

Finally, background clauses, the ones with a non-target predicate in the head, are added en bloc to the theory so built, which is the best theory for target predicates. A further parameter optimization step is executed with EMBLEM and clauses that are never involved in a target predicate goal derivation are removed.

Algorithm 2. Function SLIPCOVER

```
1: function SLIPCOVER($I, NInt, NS, NA, NI, NV, NB, NTC, NBC, \epsilon, \delta$)
2:     $IBs$ =InitialBeams($I, NInt, NS, NA$)                                   ▷ Clause search
3:     $TC \leftarrow []$
4:     $BC \leftarrow []$
5:     for all ($PredSpec, Beam$) $\in IBs$ do
6:         $Steps \leftarrow 1$
7:         $NewBeam \leftarrow []$
8:         repeat
9:             while $Beam$ is not empty do
10:                Remove the first couple (($Cl, Literals$), $LL$) from $Beam$ ▷ Remove the first clause
11:                $Refs \leftarrow$ClauseRefinements(($Cl, Literals$), $NV$)      ▷ Find all refinements $Refs$ of ($Cl, Literals$) with at most $NV$ variables
12:                for all ($Cl', Literals'$) $\in Refs$ do
13:                    ($LL'', \{Cl''\}$) $\leftarrow$EMBLEM($I, \{Cl'\}, \epsilon, \delta$)
14:                    $NewBeam \leftarrow$Insert(($Cl'', Literals'$), $LL''$, $NewBeam, NB$)
15:                    if $Cl''$ is range restricted then
16:                        if $Cl''$ has a target predicate in the head then
17:                            $TC \leftarrow$Insert(($Cl'', Literals'$), $LL''$, $TC, NTC$)
18:                        else
19:                            $BC \leftarrow$Insert(($Cl'', Literals'$), $LL''$, $BC, NBC$)
20:                        end if
21:                    end if
22:                end for
23:            end while
24:            $Beam \leftarrow NewBeam$
25:            $Steps \leftarrow Steps + 1$
26:        until $Steps > NI$
27:    end for
28:    $Th \leftarrow \emptyset$, $ThLL \leftarrow -\infty$                                   ▷ Theory search
29:    repeat
30:        Remove the first couple ($Cl, LL$) from $TC$
31:        ($LL', Th'$) $\leftarrow$EMBLEM($I, Th \cup \{Cl\}, \epsilon, \delta$)
32:        if $LL' > ThLL$ then
33:            $Th \leftarrow Th'$, $ThLL \leftarrow LL'$
34:        end if
35:    until $TC$ is empty
36:    $Th \leftarrow Th \bigcup_{(Cl,LL) \in BC} \{Cl\}$
37:    ($LL, Th$) $\leftarrow$EMBLEM($I, Th, D, NEM, \epsilon, \delta$)
38:    return $Th$
39: end function
```

4 Distributed Parameter Learning

In order to parallelize structure learning, first a MapReduce version of EMBLEM called EMBLEM$^{\mathrm{MR}}$ has been developed, where the Expectation step is performed in parallel following the approach proposed in [5] for applying MapReduce to the EM algorithm.

In particular, EMBLEM$^{\mathrm{MR}}$ (see Algorithm 3) creates n workers indexed from 1 to n. Worker 1 is the "master" and is in charge of splitting work among the "slaves" (the other $n-1$ workers). The Map function is performed by all processes; the Reduce function and the Maximization step are performed by the master (also referred to as the "reducer").

During the Map phase, the input interpretations I and the input theory T whose parameters are to be learned are replicated among all workers, while the examples E are evenly divided into n subsets $E_1, \ldots, E_n$. When splitting examples, E_1 is handled by the master, while $E_2, \ldots, E_n$ are sent to the slaves (also referred to as "mappers"). The m-th subset is sent to mapper m that builds the BDDs for the examples belonging to it. The assignment of subsets of examples to different mappers is possible because each of them stored in main memory I and T and because each example and thus each BDD is independent of the others, allowing one to divide and treat them separately. After that, all the mappers stay active keeping the BDDs in memory, that could not be done with a standard MapReduce framework.

During the learning phase (EM cycle), the Expectation step is executed in parallel by sending the current values of the parameters to each mapper m, which computes the expectations for each of its examples. By keeping the BDDs in memory, the mappers only need to receive the parameters' updated values to accomplish their task. Then, during the Reduce phase, the expectations are aggregated and sent to the reducer, that simply sums up the values obtaining the expected counts. Finally, the Maximization step is performed serially.

This parallelization strategy is implemented using the Message Passing Interface (MPI): we preferred it over a standard MapReduce framework (such as Hadoop) because we wanted to customize the parallelization strategy to better suit our needs: our mappers have side-effects because they have to retain in main memory all the BDDs through all iterations, so they are not purely functional, as should be required by standard MapReduce frameworks.

5 Distributed Structure Learning

SEMPRE (see Algorithm 4) parallelizes three operations of the structure learning algorithm SLIPCOVER by employing n workers, one master and $n-1$ slaves. All the workers initially receive all the input data.

The first operation is the scoring of the clause refinements: when the revisions *Refs* for a clause are generated [line 12], the master process splits them evenly into n subsets $Refs_1, \ldots, Refs_n$ and assigns $Refs_2, ..., Refs_n$ to the slaves. The subset $Refs_1$ is handled by the master. Then, SEMPRE enters the *Map*

Algorithm 3. Function EMBLEM$^{\text{MR}}$

```
function EMBLEM$^{MR}$ $(I, T, n, \epsilon, \delta)$
    if MASTER then
        Identify examples $E$
        Split examples $E$ into $n$ subsets $E_1, \ldots, E_n$
        Send $E_m$ to each worker $m$, $2 \leq m \leq n$
        Build $BDDs_1$ for examples $E_1$ using $T$ and $I$
        $LL = -\infty$
        repeat
            $LL_0 = LL$
            Send the parameters $\pi$ to each worker $m$, $2 \leq m \leq n$
            $LL = $ Expectation($BDDs_1$)
            Collect $LL_m$ and the expectations from each worker $m$, $2 \leq m \leq n$
            Update $LL$ and the expectations
            Maximization
        until $LL - LL_0 < \epsilon \vee LL - LL_0 < -LL \cdot \delta$
        return $LL, \pi$
    else                                                        ▷ the $j$-th slave
        Receive $E_j$ from master
        Build $BDDs_j$ for examples $E_j$ using $T$ and $I$
        $LL = -\infty$
        repeat
            Receive the parameters $\pi$ from master
            $LL_j = $ Expectation($BDDs_j$)
            Send $LL_j$ and the expectations to master
        until $LL - LL_0 < \epsilon \vee LL - LL_0 < -LL \cdot \delta$
    end if
end function
```

phase [lines 20-30], when each worker is listening for requests to score a set of refinements and will return the set of scored refinements with their log-likelihood (LL). Scoring is performed using (serial) EMBLEM which is run over a theory containing only one refinement at a time: since the BDDs built for clauses are usually small, using EMBLEM$^{\text{MR}}$ would imply a too large overhead.

Once the master has received all sets of scored refinements from the workers, it enters the *Reduce phase* [lines 32-35], where it updates the beam of promising clauses (*NewBeam*) and the sets of target and background clauses (TC and BC respectively): the scored refinements are inserted in order of LL into these lists. NTC (NBC) is the maximum size for TC (BC).

The second parallelized operation is parameter learning for the theories. In this phase [lines 45-52], each clause from TC is added to the theory, which is initially empty and then contains all the clauses that improved the its LL (search in the space of theories). In this case, the BDDs that are being built can be quite complex since the theory is incrementally built, so EMBLEM$^{\text{MR}}$ is used.

The third parallelized operation is the final parameter optimization for the theory including also the background clauses [lines 53-54]. All the background clauses are added to the theory previously learned, then the parameters of the theory are learned by means of EMBLEM$^{\text{MR}}$ because the BDDs can be large.

6 Experiments

SEMPRE was implemented in Yap Prolog [22] using the `lammpi` library for interfacing Prolog with the underlying MPI framework. SEMPRE was tested on the following seven real world datasets: Hepatitis [12], Mutagenesis [26], UWCSE [13], Carcinogenesis [25], IMDB [14], HIV [1] and WebKB [7]. All experiments were performed on GNU/Linux machines with an Intel Xeon Haswell E5-2630 v3 (2.40GHz) CPU with 8GB of memory allocated to the job.

Table 1 shows the wall time in seconds taken by SEMPRE to perform learning averaged over the folds (ten for Mutagenesis, four for WebKB and five for all the others). The experiments were performed with 1, 8, 16 or 32 workers. Figure 2 shows the speedup obtained as a function of the number of workers. The speedup for n workers is the fraction of the time with 1 worker over the time for n workers. Ideally, one wants to achieve a linear speedup. The speedup is always larger than 1 and grows with the number of workers achieving the best with 32 workers, except for HIV and IMDB, where there is a slight decrease for 16 and 32 workers due to the overhead caused by the distribution itself; however, these two datasets were the smallest ones and less in need of a parallel solution.

We have evaluated SEMPRE speedup during both distributed parameter and structure learning and seen that it is remarkable in both phases; moreover, we have noted that it spends most time in the beam search of clause refinements: for example, for UWCSE the time for clause search is around 94% of the total time, while for WebKB it is around 96%. The average time to handle each refinement is small, around 23ms for UWCSE and 80ms for WebKB. Therefore, the parallelization decisions taken seem justified: since the refinement handling time is small, it does not make sense to perform distributed parameter learning for clause refinements, while it is more reasonable to distribute the refinements to workers. These results show that SEMPRE is able to exploit the availability of processors in most cases.

	1	**8**	**16**	**32**
Hepatitis	19,867	4,246	2,392	1,269
Mutagenesis	14,784	2,887	2,587	1,579
UWCSE	12,758	5,401	3,152	1,899
Carcinogenesis	170	23	18	16
IMDB	481	104	113	177
HIV	508	118	136	295
WebKB	2,441	486	322	256

Table 1. SEMPRE execution time (in seconds) as the number of slaves varies.

7 Conclusions

The paper presents the algorithm SEMPRE for learning the structure of probabilistic logic programs under the distribution semantics. SEMPRE is a MapReduce implementation of SLIPCOVER, exploiting modern computing infrastructures for performing learning in parallel. SEMPRE has been tested on a number

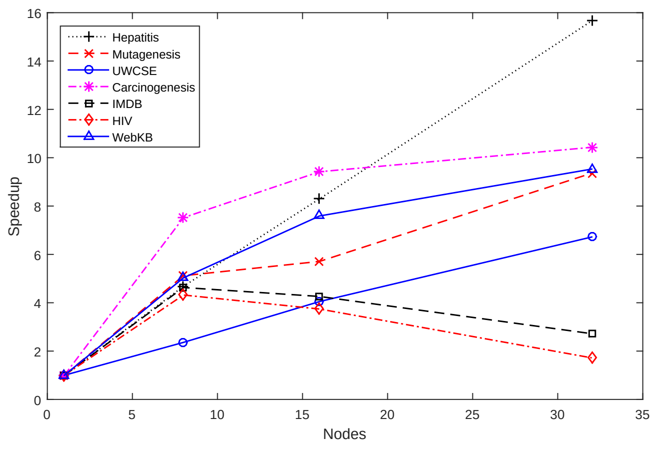

Fig. 2. SEMPRE Speedup referred to Table 1.

of domains with an increasing number of nodes and the results show that parallelization is indeed effective at reducing the running time, even if in some cases the overhead may be significant.

References

1. Beerenwinkel, N., Rahnenführer, J., Däumer, M., Hoffmann, D., Kaiser, R., Selbig, J., Lengauer, T.: Learning multiple evolutionary pathways from cross-sectional data. J. Comp. Biol. 12, 584–598 (2005)
2. Bellodi, E., Riguzzi, F.: Learning the structure of probabilistic logic programs. In: Muggleton, S., Tamaddoni-Nezhad, A., Lisi, F. (eds.) ILP 2012. LNCS, vol. 7207, pp. 61–75. Springer Berlin Heidelberg (2012)
3. Bellodi, E., Riguzzi, F.: Expectation Maximization over Binary Decision Diagrams for probabilistic logic programs. Intell. Data Anal. 17(2), 343–363 (2013)
4. Bellodi, E., Riguzzi, F.: Structure learning of probabilistic logic programs by searching the clause space. Theor. Pract. Log. Prog. 15(2), 169–212 (2015)
5. Chu, C., Kim, S.K., Lin, Y.A., Yu, Y., Bradski, G., Ng, A.Y., Olukotun, K.: Map-reduce for machine learning on multicore. In: Schölkopf, B., Platt, J.C., Hoffman, T. (eds.) 20th Annual Conference on Neural Information Processing Systems. pp. 281–288. MIT Press (2006)
6. Côrte-Real, J., Dutra, I., Rocha, R.: Prolog programming with a map-reduce parallel construct. In: 15th Symposium on Principles and Practice of Declarative Programming. pp. 285–296. ACM (2013)
7. Craven, M., Slattery, S.: Relational learning with statistical predicate invention: Better models for hypertext. Mach. Learn. 43(1-2), 97–119 (2001)
8. Darwiche, A., Marquis, P.: A knowledge compilation map. J. Artif. Intell. Res. 17, 229–264 (2002)
9. 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)

10. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. Commun. ACM 51(1), 107–113 (2008)
11. Fierens, D., den Broeck, G.V., Renkens, J., Shterionov, D.S., Gutmann, B., Thon, I., Janssens, G., De Raedt, L.: Inference and learning in probabilistic logic programs using weighted boolean formulas. Theor. Pract. Log. Prog. 15(3), 358–401 (2015)
12. Khosravi, H., Schulte, O., Hu, J., Gao, T.: Learning compact Markov logic Networks with decision trees. Mach. Learn. 89(3), 257–277 (2012)
13. Kok, S., Domingos, P.: Learning the structure of Markov Logic Networks. In: 22nd international conference on Machine learning. pp. 441–448. ACM (2005)
14. Mihalkova, L., Mooney, R.J.: Bottom-up learning of markov logic network structure. In: 24th International Conference on Machine Learning. pp. 625–632. ACM (2007)
15. Muggleton, S.: Inverse entailment and Progol. New Generat. Comput. 13, 245–286 (1995)
16. Rauzy, A., Châtelet, E., Dutuit, Y., Bérenguer, C.: A practical comparison of methods to assess sum-of-products. Reliab. Eng. Syst. Safe 79(1), 33–42 (January 2003)
17. Riguzzi, F.: Speeding up inference for probabilistic logic programs. Comput. J. 57(3), 347–363 (2014)
18. Riguzzi, F., Swift, T.: Tabling and Answer Subsumption for Reasoning on Logic Programs with Annotated Disjunctions. In: 26th International Conference on Logic Programming. LIPIcs, vol. 7, pp. 162–171 (2010)
19. Riguzzi, F., Swift, T.: The PITA system: Tabling and answer subsumption for reasoning under uncertainty. Theor. Pract. Log. Prog. 11(4–5), 433–449 (2011)
20. Riguzzi, F., Swift, T.: Well-definedness and efficient inference for probabilistic logic programming under the distribution semantics. Theor. Pract. Log. Prog. 13(2), 279–302 (March 2013)
21. Sang, T., Beame, P., Kautz, H.A.: Performing bayesian inference by weighted model counting. In: 20th National Conference on Artificial Intelligence. pp. 475–482 (2005)
22. Santos Costa, V., Rocha, R., Damas, L.: The YAP Prolog system. Theor. Pract. Log. Prog. 12(1-2), 5–34 (2012)
23. Sato, T.: A statistical learning method for logic programs with distribution semantics. In: 12th International Conference on Logic Programming. pp. 715–729. MIT Press (1995)
24. Sato, T., Kameya, Y.: Parameter learning of logic programs for symbolic-statistical modeling. J. Artif. Intell. Res. 15, 391–454 (2001)
25. Srinivasan, A., King, R.D., Muggleton, S., Sternberg, M.J.E.: Carcinogenesis predictions using ILP. In: Lavrac, N., Dzeroski, S. (eds.) ILP 1997. LNCS, vol. 1297, pp. 273–287. Springer Berlin Heidelberg (1997)
26. 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)
27. Vennekens, J., Verbaeten, S., Bruynooghe, M.: Logic Programs With Annotated Disjunctions. In: ICLP 2004. LNCS, vol. 3131, pp. 195–209. Springer Berlin Heidelberg (2004)

Algorithm 4. Function SEMPRE

```
1: function SEMPRE($I, n, NInt, NS, NA, NI, NV, NB, NTC, NBC, \epsilon, \delta$)
2:     $IBs$ =InitialBeams($I, NInt, NS, NA$)                          ▷ Clause search
3:     $TC \leftarrow []$
4:     $BC \leftarrow []$
5:     for all ($PredSpec, Beam$) $\in IBs$ do
6:         $Steps \leftarrow 1$
7:         $NewBeam \leftarrow []$
8:         repeat
9:             while $Beam$ is not empty do
10:                if MASTER then
11:                    Remove the first couple $((Cl, Literals), LL)$ from $Beam$    ▷ Remove the first clause
12:                    $Refs \leftarrow$ClauseRefinements($(Cl, Literals), NV$)  ▷ Find all refinements $Refs$ of $(Cl, Literals)$ with at most $NV$ variables
13:                    Split evenly $Refs$ into $n$ subsets $Refs_1, \ldots, Refs_n$
14:                    for $m = 2$ to $n$ do
15:                        Send $Refs_m$ to worker $m$
16:                    end for
17:                else                                                   ▷ the $j$-th slave
18:                    Receive $Refs_j$ from master
19:                end if
20:                for all $(Cl', Literals') \in Refs_j$ do
21:                    $(LL'', \{Cl''\}) \leftarrow$EMBLEM($I, \{Cl'\}, \epsilon, \delta$)
22:                    $NewBeam \leftarrow$Insert($(Cl'', Literals'), LL'', NewBeam, NB$)
23:                    if $Cl''$ is range restricted then
24:                        if $Cl''$ has a target predicate in the head then
25:                            $TC \leftarrow$Insert($(Cl'', Literals'), LL'', TC, NTC$)
26:                        else
27:                            $BC \leftarrow$Insert($(Cl'', Literals'), LL'', BC, NBC$)
28:                        end if
29:                    end if
30:                end for
31:                if MASTER then
32:                    for $m = 2$ to $n$ do
33:                        Collect the set $\{(LL'', \{Cl''\})|\forall(Cl', Literals) \in Refs_m\}$ from worker $m$
34:                        Update $NewBeam, TC, BC$
35:                    end for
36:                else                                                   ▷ the $j$-th slave
37:                    Send the set $\{(LL'', \{Cl''\})|\forall(Cl', Literals) \in Refs_j\}$ to master
38:                end if
39:            end while
40:            $Beam \leftarrow NewBeam$
41:            $Steps \leftarrow Steps + 1$
42:        until $Steps > NI$
43:    end for
44:    if MASTER then
45:        $Th \leftarrow \emptyset,\ ThLL \leftarrow -\infty$                                  ▷ Theory search
46:        repeat
47:            Remove the first couple $(Cl, LL)$ from $TC$
48:            $(LL', Th') \leftarrow$EMBLEM$^{\mathrm{MR}}$ $(I, Th \cup \{Cl\}, n, \epsilon, \delta)$
49:            if $LL' > ThLL$ then
50:                $Th \leftarrow Th',\ ThLL \leftarrow LL'$
51:            end if
52:        until $TC$ is empty
53:        $Th \leftarrow Th \bigcup_{(Cl,LL)\in BC}\{Cl\}$
54:        $(LL, Th) \leftarrow$EMBLEM$^{\mathrm{MR}}$ $(I, Th, n, \epsilon, \delta)$
55:        return $Th$
56:    end if
57: end function
```