Structure Learning of Probabilistic Logic Programs by MapReduce

Fabrizio Riguzzi\textsuperscript{1}, Elena Bellodi\textsuperscript{2}, Riccardo Zese\textsuperscript{2}, Giuseppe Cota\textsuperscript{2}, and Evelina Lamma\textsuperscript{2}

\textsuperscript{1} Dipartimento di Matematica e Informatica – University of Ferrara  
Via Saragat 1, I-44122, Ferrara, Italy
\textsuperscript{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\textsuperscript{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 $h_{i1} : \Pi_{i1} ; \ldots ; h_{in_i} : \Pi_{in_i} : \neg b_{i1} , \ldots , b_{im_i}$. Here, $b_{i1} , \ldots , b_{im_i}$ are logical literals which form the body of $C_i$, denoted by $\text{body}(C_i)$, 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
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:

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

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

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 $w_\sigma$ 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:

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

(2)

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

\[
f_K(X) = (X_{11} \land X_{21}) \lor (X_{12} \land X_{21})
\]

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 \( X_{ij1}, \ldots, X_{ijn_i-1} \), the equation \( X_{ij} = k \) for \( k = 1, \ldots, n_i - 1 \) corresponds with the conjunction \( X_{ij1} \land \ldots \land X_{ijk-1} \land X_{ijk} \), while the equation \( X_{ij} = n_i \) with \( X_{ij1} \land \ldots \land X_{ijn_i-1} \). Following this approach, which provides good performance [21], \( f_K(X) \) 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 \( X_{1111} \land X_{1112} \). Let us call \( f'_K(X') \) the result of replacing multi-valued random variables with Boolean variables in \( f_K(X) \). The probability distribution of the Boolean random variables \( X_{ijk} \) is computed from that of multi-valued variables as \( \pi_i = \Pi_{i1}, \ldots, \pi_{ik} = \frac{P_{ik}}{\prod_{j=1}^{n_i-1}(1-\pi_{ij})} \) up to \( k = n_i - 1 \), where \( \pi_{ik} \) is the probability that \( X_{ijk} \) is true. With this distribution the probability that \( f'_K(X') \) is true is the same as \( f_K(X) = P(Q) \). For Example 1, \( f'_K(X') \) is given by

\[
f'_K(X') = (X_{111} \land X_{211}) \lor (X_{121} \land X_{211})
\]

Computing the probability that \( f'_K(X') \) 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 \( f'_K(X') \) 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 0 value.

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,ϵ,δ)
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: $LL = \text{Expectation(BDDs)}$
8: $\text{Maximization}$
9: until $LL - LL_0 < \epsilon \lor LL - LL_0 < -LL \cdot \delta$
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 $\pi$ 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

```plaintext
1: function SLIPCOVER(I, NInt, NS, NA, NI, NV, NB, NTC, NBC, ϵ, δ)  
2: IBs = INITIALBEAMS(I, NInt, NS)  
3: TC ← []  
4: BC ← []  
5: for all (PredSpec, Beam) ∈ IBs do  
6:   Steps ← 1  
7:   NewBeam ← []  
8:   repeat  
9:     while Beam is not empty do  
10:        Remove the first couple ((Cl, Literals), LL) from Beam  
11:        Refs ← ClauseRefinements((Cl, Literals, NV))  
12:        for all (Cl′, Literals′) ∈ Refs do  
13:           LL′ ← EMBLEM(I, {Cl′}, ϵ, δ)  
14:           NewBeam ← 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 ← Insert((Cl′, Literals′), LL′, TC, NTC)  
18:              else  
19:                 BC ← Insert((Cl′, Literals′), LL′, BC, NBC)  
20:           end if  
21:        end for  
22:     end while  
23: end for  
24: Beam ← NewBeam  
25: Steps ← Steps + 1  
26: until Steps > NI  
27: end for  
28: Th ← ∅, ThLL ← −∞  
29: repeat  
30:   Remove the first couple (Cl, LL) from TC  
31:   (LL′, Th′) ← EMBLEM(I, Th ∪ {Cl}, ϵ, δ)  
32:   if LL′ > ThLL then  
33:      Th ← Th′, ThLL ← LL′  
34:   end if  
35: until TC is empty  
36: Th ← Th ∪ {Cl}  
37: (LL, Th) ← EMBLEM(I, D, NEM, ϵ, δ)  
38: return Th  
39: end function
```
4 Distributed Parameter Learning

In order to parallelize structure learning, first a MapReduce version of EMBLEM called EMBLEM$^\text{MR}$ 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$^\text{MR}$ (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 $\text{Refs}$ for a clause are generated [line 12], the master process splits them evenly into $n$ subsets $\text{Refs}_1, \ldots, \text{Refs}_n$ and assigns $\text{Refs}_2, \ldots, \text{Refs}_n$ to the slaves. The subset $\text{Refs}_1$ is handled by the master. Then, SEMPRE enters the Map
Algorithm 3. Function EMBLEM$_{MR}$

1: function EMBLEM$_{MR}$ ($I$, $T$, $n$, $\epsilon$, $\delta$)  
2: if MASTER then  
3: Identify examples $E$  
4: Split examples $E$ into $n$ subsets $E_1, \ldots, E_n$  
5: Send $E_m$ to each worker $m$, $2 \leq m \leq n$  
6: Build $BDDs_1$ for examples $E_1$ using $T$ and $I$  
7: $LL = -\infty$  
8: repeat  
9: $LL_0 = LL$  
10: Send the parameters $\pi$ to each worker $m$, $2 \leq m \leq n$  
11: $LL = \text{Expectation}(BDDs_1)$  
12: Collect $LL_m$ and the expectations from each worker $m$, $2 \leq m \leq n$  
13: Update $LL$ and the expectations  
14: until $LL - LL_0 < \epsilon \lor LL - LL_0 < -LL \cdot \delta$  
15: return $LL, \pi$  
16: 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$_{MR}$ 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 ($\text{NewBeam}$) 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$_{MR}$ 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$_{MR}$ 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.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>8</th>
<th>16</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hepatitis</td>
<td>19.867</td>
<td>4.246</td>
<td>2.302</td>
<td>1.269</td>
</tr>
<tr>
<td>Mutagenesis</td>
<td>14.784</td>
<td>2.887</td>
<td>2.587</td>
<td>1.579</td>
</tr>
<tr>
<td>UWCSE</td>
<td>12.758</td>
<td>5.401</td>
<td>3.152</td>
<td>1.899</td>
</tr>
<tr>
<td>Carcinogenesis</td>
<td>170</td>
<td>23</td>
<td>18</td>
<td>16</td>
</tr>
<tr>
<td>IMDB</td>
<td>481</td>
<td>104</td>
<td>113</td>
<td>177</td>
</tr>
<tr>
<td>HIV</td>
<td>508</td>
<td>118</td>
<td>136</td>
<td>295</td>
</tr>
<tr>
<td>WebKB</td>
<td>2.441</td>
<td>486</td>
<td>322</td>
<td>256</td>
</tr>
</tbody>
</table>

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

Algorithm 4. Function SEMPRE

1: function SEMPRE(I, n, NInt, NS, NA, NI, NV, NB, NTC, NBC, ϵ, δ)  
2: IBs ← InitialBeams(I, NInt, NS, NA)  
3: TC ← []  
4: BC ← []  
5: for all (PredSpec, Beam) ∈ IBs do  
6: Steps ← 1  
7: NewBeam ← []  
8: repeat  
9: while Beam is not empty do  
10: if MASTER then  
11: Remove the first couple ((Cl, LL), Beam) from Beam  
12: end if  
13: for all (LL′′, {Cl′′}) ∈ Refs, where Refs is the set of all refinements of (Cl, LL) with at most NV variables  
14: Split evenly Refs into n subsets Refs_1, ..., Refs_n  
15: for m = 2 to n do  
16: Send Refs_m to worker m  
17: end for  
18: else  
19: Receive Refs_j from master  
20: end if  
21: for all (Cl′, Literals′) ∈ Refs_j do  
22: NewBeam ← Insert((LL′′, {Cl′′}), LL′′, NewBeam, NB)  
23: if Cl′ is range restricted then  
24: if Cl′ has a target predicate in the head then  
25: TC ← Insert((LL′′, {Cl′′}), LL′′, TC, NTC)  
26: else  
27: BC ← Insert((LL′′, {Cl′′}), 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′′}} | ∀(Cl, Literals) ∈ Refs_m} from worker m  
34: Update NewBeam, TC, BC  
35: end for  
36: else  
37: Send the set {{LL′′, {Cl′′}} | ∀(Cl, Literals) ∈ Refs_j} to master  
38: end if  
39: end while  
40: Beam ← NewBeam  
41: Steps ← Steps + 1  
42: until Steps > NI  
43: end for  
44: if MASTER then  
45: Th ← ∅, ThLL ← −∞  
46: repeat  
47: Remove the first couple (Cl, LL) from TC  
48: (LL′, Th′) ← EMBLEMIRS(I, Th ∪ {Cl}), n, ϵ, δ)  
49: if LL′ > ThLL then  
50: Th ← Th′, ThLL ← LL′  
51: end if  
52: until TC is empty  
53: Th ← Th ∪ (Cl, LL) ∈ Refs_1  
54: (LL, Th) ← EMBLEMIRS(I, Th, n, ϵ, δ)  
55: return Th  
56: end if  
57: end function