Programming with Personalized PageRank: A Locally Groundable First-Order Probabilistic Logic

William Yang Wang
Language Technology Institute
Carnegie Mellon University
Pittsburgh, PA 15213
yww@cs.cmu.edu

Kathryn Mazaitis
Machine Learning Department
Carnegie Mellon University
Pittsburgh, PA 15213
krivard@cs.cmu.edu

William W. Cohen
Machine Learning Department
Carnegie Mellon University
Pittsburgh, PA 15213
wcohen@cs.cmu.edu

ABSTRACT

Many information-management tasks (including classification, retrieval, information extraction, and information integration) can be formalized as inference in an appropriate probabilistic first-order logic. However, most probabilistic first-order logics are not efficient enough for realistically-sized instances of these tasks. One key problem is that queries are typically answered by "grounding" the query—i.e., mapping it to a propositional representation, and then performing propositional inference—and with a large database of facts, groundings can be very large, making inference and learning computationally expensive. Here we present a first-order probabilistic language which is well-suited to approximate "local" grounding: in particular, every query Q can be approximately grounded with a small graph. The language is an extension of stochastic logic programs where inference is performed by a variant of personalized PageRank. Experimentally, we show that the approach performs well on an entity resolution task, a classification task, and a joint inference task; that the cost of inference is independent of database size; and that speedup in learning is possible by multi-threading.

Categories and Subject Descriptors

[**Information Systems Applications**]: Miscellaneous

Keywords

Probabilistic Prolog, personalized PageRank

1. INTRODUCTION

Many information-management tasks (including classification [18], retrieval [12], information extraction [23], and information integration [24, 7]) can be formalized as inference in an appropriate probabilistic first-order logic. However, most probabilistic first-order logics are not efficient enough to be used for the large-scale versions of these tasks. One key problem is that queries are typically answered by "grounding" the query—i.e., mapping it to a propositional representation, and then performing propositional inference—and for many logics, the size of the "grounding" can be extremely large for large databases. For instance, in probabilistic Datalog [12], a query is converted to a structure called an "event expression", which summarizes all possible proofs for the query against a database; in ProbLog [10] and MarkoViews [14] similar structures are created, encoded more compactly with binary decision diagrams (BDDs); in probabilistic similarity logic (PSL) an intentional probabilistic program, together with a database, is converted to constraints for a convex optimization problem; and in Markov Logic Networks (MLNs) [25], queries are converted to a (propositional) Markov network. As an illustration of the "grounding" process, Figure 1 shows a very simple MLN and its grounding (which here is query-independent.).

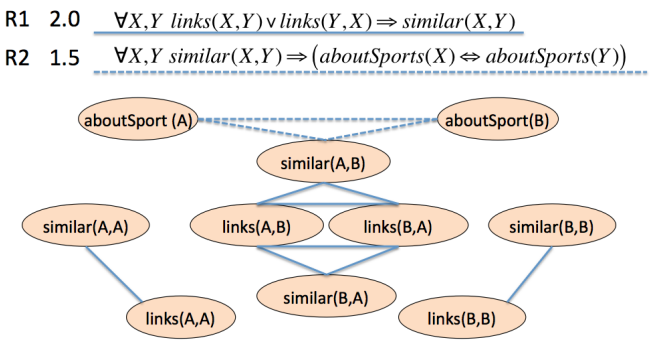

Figure 1: A Markov logic network program and its grounding. (Dotted lines are clique potentials associated with rule R2, solid lines with rule R1.)

The size of groundings is sometimes, but not always, difficult to analyze. For instance, for MLNs, a naive grounding is linear in the number of *possible* facts in the database—i.e., $O(n^k)$ where k is the maximal arity of a predicate and n the number of database constants. However, even a grounding of size linear in the number of facts in the database, $|DB|$, is impractically large for inference. Superficially, it would seem that groundings must inherently be $o(|DB|)$ for some programs: in the example, for instance, the probability of *aboutSport(x)* must depend to some extent on the entire hyperlink graph (if it is fully connected). However, it also seems intuitive that if we are interested in inferring information about a specific page—say, the probabil-

CIKM'13, Oct. 27–Nov. 1, 2013, San Francisco, CA, USA.

ACM 978-1-4503-2263-8/13/10 ...$15.00.
http://dx.doi.org/10.1145/2505515.2505573 .

ity of *aboutSport(d1)*—then the parts of the network only distantly connected to *d1* are likely to have a small influence. This suggests that an *approximate* grounding strategy might be feasible, in which a query such as *aboutSport(d1)* would be grounded by constructing a small subgraph of the full network, followed by inference on this small "locally grounded" subgraph. Likewise, consider learning (e.g., from a set of queries Q with their desired truth values). Learning might proceed by locally-grounding every query goal, allowing learning to also take less than $O(|DB|)$ time.

In this paper, we present a first-order probabilistic language which is well-suited to approximate "local" grounding. We present an extension to *stochastic logic programs* (SLP) [9] that is biased towards short derivations, and show that this is related to *personalized PageRank* (PPR) [22, 6] on a linearized version of the proof space. Based on the connection to PPR, we develop a proveably-correct approximate inference scheme, and an associated proveably-correct approximate grounding scheme: specifically, we show that it is possible to prove a query, or to build a graph which contains the information necessary for weight-learning, in time $O(\frac{1}{\alpha\epsilon})$, where α is a reset parameter associated with the bias towards short derivations, and ϵ is the worst-case approximation error across all intermediate stages of the proof. This means that both inference and learning can be approximated in time *independent of the size of the underlying database*—a surprising and important result.

The ability to locally ground queries has another important consequence: it is possible to decompose the problem of weight-learning to a number of moderate-size subtasks (in fact, tasks of size $O(\frac{1}{\alpha\epsilon})$ or less) which are weakly coupled. Based on this we outline a parallelization scheme, which in our initial implementation provides an order-of-magnitude speedup in learning time.

Below, we will first introduce our formalism, and then describe our weight-learning algorithm. We will then present experimental results on a prototypical inference task, and compare the scalability of our method to Markov logic networks. We finally discuss related work and conclude.

2. PROGRAMMING WITH PERSONALIZED PAGERANK (PROPPR)

2.1 Inference as Graph Search

We will now describe our "locally groundable" first-order probabilistic language, which we call ProPPR. Inference for ProPPR is based on a personalized PageRank process over the proof constructed by Prolog's Selective Linear Definite (SLD) resolution theorem-prover. To define the semantics we will use notation from logic programming [17]. Let LP be a program which contains a set of definite clauses $c_1, \ldots, c_n$, and consider a conjunctive query Q over the predicates appearing in LP. A traditional Prolog interpreter can be viewed as having the following actions. First, construct a "root vertex" v_0, which is a pair (Q, Q) and add it to an otherwise-empty graph $G'_{Q,LP}$. (For brevity, we drop the subscripts of G' where possible.) Then recursively add to G' new vertices and edges as follows: if u is a vertex of the form $(Q, (R_1, \ldots, R_k))$, and c is a clause in LP of the form $R' \leftarrow S'_1, \ldots, S'_\ell$, and R_1 and R' have a most general unifier $\theta = mgu(R_1, R')$, then add to G' a new edge $u \rightarrow v$ where $v = (Q\theta, (S'_1, \ldots, S'_\ell, R_2, \ldots, R_k)\theta)$. Let us call $Q\theta$ the *transformed query* and $(S'_1, \ldots, S'_\ell, R_2, \ldots, R_k)\theta$ the *associated subgoal list*. If a subgoal list is empty, we will denote it by $\Box$. Here $Q\theta$ denotes the result of applying the substitution θ to Q; for instance, if $Q = about(a, Z)$ and $\theta = \{Z = fashion\}$, then $Q\theta$ is $about(a, fashion)$.

Table 1: A simple program in ProPPR. See text for explanation.

```
about(X,Z) :- handLabeled(X,Z)          # base.
about(X,Z) :- sim(X,Y),about(Y,Z)       # prop.
sim(X,Y) :- links(X,Y)                  # sim,link.
sim(X,Y) :-
    hasWord(X,W),hasWord(Y,W),
    linkedBy(X,Y,W)                     # sim,word.
linkedBy(X,Y,W) :- true                 # by(W).
```

The graph G' is often large or infinite so it is not constructed explicitly. Instead Prolog performs a depth-first search on G' to find the first *solution vertex* v—i.e., a vertex with an empty subgoal list—and if one is found, returns the transformed query from v as an answer to Q. Table 1 and Figure 2 show a simple Prolog program and a proof graph for it.[1] Given the query Q = *about(a,Z)*, Prolog's depth-first search would return Q = *about(a,fashion)*.

Note that in this proof formulation, the nodes are *conjunctions* of literals, and the structure is, in general, a digraph (rather than a tree). Also note that the proof is encoded as a graph, not a hypergraph, even if the predicates in the LP are not binary: the edges represent a step in the proof that reduces one conjunction to another, not a binary relation between entities.

2.2 From SLPs to ProPPR

In *stochastic logic programs* (SLPs) [9], one defines a randomized procedure for traversing the graph G', which thus defines a probability distribution over vertices v, and hence (by selecting only solution vertices) a distribution over transformed queries (i.e. answers) $Q\theta$. The randomized procedure thus produces a distribution over possible answers, which can be tuned by learning to upweight desired (correct) answers and downweight others.

In past work, the randomized traversal of G' was defined by a probabilistic choice, at each node, of which clause to apply, based on a weight for each clause. We propose two extensions. First, we will introduce a new way of computing clause weights, which allows for a potentially richer parameterization of the traversal process. We will associate with each edge $u \rightarrow v$ in the graph a *feature vector* $\phi_{u\rightarrow v}$. This edge is produced indirectly, by associating with every clause $c \in LP$ a function $\Phi_c(\theta)$,[2] which produces the vector ϕ associated with an application of c using mgu θ. This feature vector[3] is computed during theorem-proving, and used to annotate the edge $u \rightarrow v$ in G' created by apply-

[1] The annotations after the hashmarks and the edge labels in the proof graph will be described below. For conciseness, only $R_1, \ldots, R_k$ is shown in each node $u = (Q, (R_1, \ldots, R_k))$.

[2] We use a set to denote a sparse vector with 0/1 weights.

[3] An example of the feature vector would be: if the last clause of the program in Table 1 was applied to $(Q, linkedBy(a, c, sprinter), about(c, Z))$ with mgu $\theta = \{X = a, Y = c, W = sprinter\}$ then $\Phi_c(\theta)$ would be $\{by(sprinter)\}$.

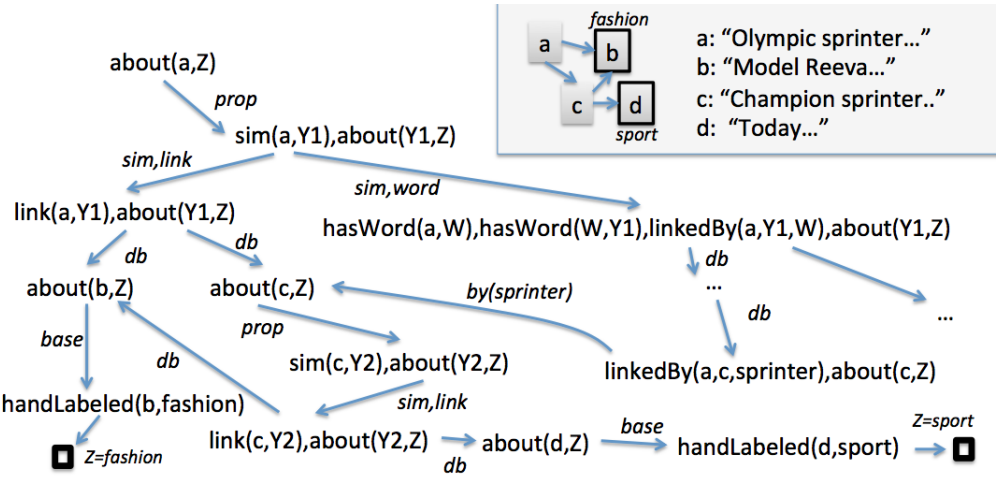

Figure 2: A partial proof graph for the query *about(a,Z)*. The upper right shows the link structure between documents a, b, c, and d, and some of the words in the documents. Restart links are not shown.

ing c with mgu θ. Finally, an edge $u \to v$ will be traversed with probability $\Pr(v|u) \propto f(\mathbf{w}, \phi_{u\to v})$ where $\mathbf{w}$ is a parameter vector and where $f(\mathbf{w}, \phi)$ is a weighting function—e.g., $f(\mathbf{w}, \phi) = \exp(\mathbf{w}_i \cdot \phi)$. This weighting function now determines the probability of a transition, in theorem-proving, from u to v: specifically, $\Pr_{\mathbf{w}}(v|u) \propto f(\mathbf{w}, \phi_{u\to v})$. Weights in $\mathbf{w}$ default to 1.0, and learning consists of tuning these.

The second and more fundamental extension is to add edges in G' from every solution vertex to itself, and also add an edge from every vertex to the start vertex v_0. We will call this augmented graph $G_{Q,LP}$ below (or just G if the subscripts are clear from context). These links make SLP's graph traversal a *personalized PageRank* (PPR) procedure, sometimes known as *random-walk-with-restart* [30]. These links are annotated by another feature vector function $\Phi_{\text{restart}}(R)$, which is applied to the leftmost literal R of the subgoal list for u to annotate the edge $u \to v_0$.

These links back to the start vertex bias the traversal of the proof graph to upweight the results of *short proofs*. To see this, note that if the restart probability $P(v_0|u) = \alpha$ for every node u, then the probability of reaching any node at depth d is bounded by $(1-\alpha)^d$.

To summarize, if u is a node of the search graph, $u = (Q\theta, (R_1, \ldots, R_k))$, then the transitions from u, and their respective probabilities, are defined as follows, where Z is an appropriate normalizing constant:

- If $v = (Q\theta\sigma, (S'_1, \ldots, S'_\ell, R2, \ldots, R_k)\theta\sigma)$ is a state derived by applying the clause c (with mgu σ), then
$$\Pr_{\mathbf{w}}(v|u) = \frac{1}{Z} f(\mathbf{w}, \Phi_c(\theta \circ \sigma))$$
- If $v = v_0 = (Q, Q)$ is the initial state in G, then
$$\Pr_{\mathbf{w}}(v|u) = \frac{1}{Z} f(\mathbf{w}, \Phi_{\text{restart}}(R_1\theta))$$
- If v is any other node, then $\Pr(v|u) = 0$.

Finally we must specify the functions Φ_c and Φ_{restart}. For clauses in LP, the feature-vector producing function $\Phi_c(\theta)$ for a clause is specified by annotating c as follows: every clause $c = (R \leftarrow S_1, \ldots, S_k)$ can be annotated with an additional conjunction of "feature literals" $F_1, \ldots, F_\ell$, which are written at the end of the clause after the special marker "#". The function $\Phi_c(\theta)$ then returns a vector $\phi = \{F_1\theta, \ldots, F_\ell\theta\}$, where every $F_i\theta$ must be ground.

The requirement[4] that edge features $F_i\theta$ are ground is the reason for introducing the apparently unnecessary predicate *linkedBy(X,Y,W)* into the program of Table 1: adding the feature literal *by(W)* to the second clause for *sim* would result in a non-ground feature *by(W)*, since W is a free variable when Φ_c is called. Notice also that the weight on the *by(W)* features are meaningful, even though there is only one clause in the definition of *linkedBy*, as the weight for applying this clause competes with the weight assigned to the restart edges.

It would be cumbersome to annotate every database fact, and difficult to learn weights for so many features. Thus, if c is the unit clause that corresponds to a database fact, then $\Phi_c(\theta)$ returns a default value $\phi = \{db\}$, where *db* is a special feature indicating that a database predicate was used.[5]

The function $\Phi_{\text{restart}}(R)$ depends on the functor and arity of R. If R is defined by clauses in LP, then $\Phi_{\text{restart}}(R)$ returns a unit vector $\phi = \{defRestart\}$. If R is a database predicate (e.g., *hasWord(doc1,W)*) then we follow a slightly different procedure, which is designed to ensure that the restart link has a reasonably large weight even with unit feature weights: we compute n, the number of possible bindings for R, and set $\phi[defRestart] = n \cdot \frac{\alpha}{1-\alpha}$, where α is a global parameter. This means that with unit weights, after normalization, the probability of following the restart link will be α.

Putting this all together with the standard iterative approach to computing personalized PageRank over a graph

[4]The requirement that the feature literals returned by $\Phi_c(\theta)$ must be ground in θ is not strictly necessary for correctness. However, in developing ProPPR programs we noted than non-ground features were usually not what the programmer intended.

[5]If a non-database clause c has no annotation, then the default vector is $\phi = \{id(c)\}$, where c is an identifier for the clause c.

[22], we arrive at the following inference algorithm for answering a query Q, using a weight vector $\mathbf{w}$. Below, we let $N_{v_0}(u)$ denote the *neighbors* of u—i.e., the set of nodes v where $\Pr(v|u) > 0$ (including the restart node $v = v_0$). We also let $\mathbf{W}$ be a matrix such that $\mathbf{W}[u, v] = \Pr_{\mathbf{w}}(v|u)$, and in our discussion, we use $\mathbf{ppr}(v_0)$ to denote the personalized PageRank vector for v_0.

1. Let $v_0 = (Q, Q)$ be the start node of the search graph. Let G be a graph containing just v_0. Let $\mathbf{v}^0 = \{v_0\}$.

2. For $t = 1, \ldots, T$ (i.e., until convergence):

   For each u with non-zero weight in $\mathbf{v}^{t-1}$, and each $v \in N_{u+0}(u)$, add $(u, v, \phi_{u\to v})$ to G with weight $\Pr_{\mathbf{w}}(v|u)$, and set $\mathbf{v}^t = \mathbf{W} \cdot \mathbf{v}^{t-1}$

3. At this point $\mathbf{v}^T \approx \mathbf{ppr}(v_0)$. Let S be the set of nodes $(Q\theta, \Box)$ that have empty subgoal lists and non-zero weight in $\mathbf{v}^T$, and let $Z = \sum_{u\in S} \mathbf{v}^T[u]$. The final probability for the literal $L = Q\theta$ is found by extracting these solution nodes S, and renormalizing:

$$\Pr_{\mathbf{w}}(L) \equiv \frac{1}{Z}\mathbf{v}^T[(L, \Box)]$$

For example, given the query $Q =$ *about(a,Z)* and the program of Table 1, this procedure would give assign a non-zero probability to the literals *about(a,sport)* and *about(a,fashion)*, concurrently building the graph of Figure 2.

2.3 Locally Grounding a Query

Note that this procedure both performs inference (by computing a distribution over literals $Q\theta$) and "grounds" the query, by constructing a graph G. ProPPR inference for this query can be re-done efficiently, by running an ordinary PPR process on G. This is useful for faster weight learning. Unfortunately, the grounding G can be very large: it need not include the entire database, but if T is the number of iterations until convergence for the sample program of Table 1 on the query $Q = about(d, Y)$, G will include a node for every page within T hyperlinks of d.

To construct a more compact local grounding graph G, we adapt an approximate personalized PageRank method called PageRank-Nibble [2]. This method has been used for the problem of *local partitioning*: in local partitioning, the goal is to find a small, low-conductance component $\hat{G}$ of a large graph G that contains a given node v.

The PageRank-Nibble-Prove algorithm is shown in Table 2. It maintains two vectors: $\mathbf{p}$, an approximation to the personalized PageRank vector associated with node v_0, and $\mathbf{r}$, a vector of "residual errors" in $\mathbf{p}$. Initially, $\mathbf{p} = \emptyset$ and $\mathbf{r} = \{v_0\}$. The algorithm repeatedly picks a node u with a large residual error $\mathbf{r}[u]$, and reduces this error by distributing a fraction α' of it to $\mathbf{p}[u]$, and the remaining fraction back to $\mathbf{r}[u]$ and $\mathbf{r}[v_1], \ldots, \mathbf{r}[v_n]$, where the v_i's are the neighbors of u. The order in which nodes u are picked does not matter for the analysis (in our implementation, we follow Prolog's usual depth-first search as much as possible.) Relative to PageRank-Nibble, the main differences are the the use of a lower-bound on α rather than a fixed restart weight and the construction of the graph $\hat{G}$.

Following the proof technique of Andersen et al. [2], it can be shown that after each push, $\mathbf{p} + \mathbf{r} = \mathbf{ppr}(v_0)$. It is also clear than when PageRank-Nibble terminates, then for any u, the error $\mathbf{ppr}(v_0)[u] - \mathbf{p}[u]$ is bounded by $\epsilon|N(u)|$: hence, in any graph where $N(u)$ is bounded, a good approximation can be obtained. It can also be shown [2] that the subgraph $\hat{G}$ is in some sense a "useful" subset of the full proof space: for an appropriate setting of ϵ, if there is a low-conductance subgraph G_* of the full graph that contains v_0, then G_* will be contained in $\hat{G}$: thus if there is a subgraph G_* containing v_0 that approximates the full graph well, PageRank-Nibble will find (a supergraph of) G_*.

Finally, we have the following efficiency bound:

THEOREM 1 (ANDERSEN,CHUNG,LANG). *Let u_i be the i-th node pushed by PageRank-Nibble-Prove. Then,* $\sum_i |N(u_i)| < \frac{1}{\alpha'\epsilon}$.

This can be proved by noting that initially $||\mathbf{r}||_1 = 1$, and also that $||\mathbf{r}||_1$ decreases by at least $\alpha'\epsilon|N(u_i)|$ on the i-th push. As a direct consequence we have the following:

COROLLARY 1. *The number of edges in the graph $\hat{G}$ produced by PageRank-Nibble-Prove is no more than $\frac{1}{\alpha'\epsilon}$.*

Importantly, the bound holds *independent of the size of the full database of facts.* The bound also holds regardless of the size or loopiness of the full proof graph, so this inference procedure will work for recursive logic programs.

To summarize, we have outlined an efficient approximate proof procedure, which is closely related to personalized PageRank. As a side-effect of inference for a query Q, this procedure will create a ground graph $\hat{G}_Q$ on which personalized PageRank can be run directly, without any (relatively expensive) manipulation of first-order theorem-proving constructs such as clauses or logical variables. As we will see, this "locally grounded" graph will be very useful in learning weights $\mathbf{w}$ to assign to the features of a ProPPR program.

As an illustration of the sorts of ProPPR programs that are possible, some small sample programs are shown in Figure 3. Clauses c_1 and c_2 are, together, a bag-of-words classifier: each proof of *predictedClass(D,Y)* adds some evidence for D having class Y, with the weight of this evidence depending on the weight given to c_2's use in establishing *related(w,y)*, where w and y are a specific word in D and y is a possible class label. In turn, c_2's weight depends on the weight assigned to the $r(w, y)$ feature by $\mathbf{w}$, relative to the weight of the restart link.[6] Adding c_3 and c_4 to this program implements label propagation, and adding c_5 and c_6 implements a sequential classifier.

These examples show that, in spite of its efficient inference procedure, and its limitation to only definite clauses, ProPPR appears to have much of the expressive power of MLNs [11], in that many useful heuristics can apparently be encoded.

2.4 Learning for ProPPR

As noted above, inference for a query Q in ProPPR is based on a personalized PageRank process over the graph associated with the SLD proof of a query goal G. More specifically, the edges $u \to v$ of the graph G are annotated with feature vectors $\phi_{u\to v}$, and from these feature vectors, weights are computed using a parameter vector $\mathbf{w}$, and finally normalized to form a probability distribution over the

[6]The existence of the restart link thus has another important role in this program, as it avoids a sort of "label bias problem" in which local decisions are difficult to adjust.

Table 2: The PageRank-Nibble-Prove algorithm for inference in ProPPR. α' is a lower-bound on $\Pr(v_0|u)$ for any node u to be added to the graph $\hat{G}$, and ϵ is the desired degree of approximation.

```
define PageRank-Nibble-Prove(Q):
    let v =PageRank-Nibble((Q,Q), α', ε)
    let S = {u : p[u] > u and u = (Qθ, □)}
    let Z = Σ_{u∈S} p[u]
    define Pr_w(L) ≡ (1/Z) v[(L, □)]
end

define PageRank-Nibble(v0, α', ε):
    let p = r = 0, let r[v0] = 1, and let Ĝ = ∅
    while ∃u : r(u)/|N(u)| > ε do: push(u)
    return p
end

define push(u, α'):
    comment: this modifies p, r, and Ĝ
    p[u] = p[u] + α' · r[u]
    r[u] = r[u] · (1 − α')
    for v ∈ N(u):
        add the edge (u, v, φ_{u→v}) to Ĝ
        if v = v0 then r[v] = r[v] + Pr(v|u) r[u]
        else r[v] = r[v] + (Pr(v|u) − α') r[u]
    endfor
end
```

neighbors of u. The "grounded" version of inference is thus a personalized PageRank process over a graph with feature-vector annotated edges.

In prior work, Backstrom and Leskovec [3] outlined a family of supervised learning procedures for this sort of annotated graph. In the simpler case of their learning procedure, an example is a triple (v_0, u, y) where v_0 is a query node, u is a node in in the personalized PageRank vector $\mathbf{p}_{v_0}$ for v_0, y is a target value, and a loss $\ell(v_0, u, y)$ is incurred if $\mathbf{p}_{v_0}[u] \neq y$. In the more complex case of "learning to rank", an example is a triple (v_0, u_+, u_-) where v_0 is a query node, u_+ and u_- are nodes in in the personalized PageRank vector $\mathbf{p}_{v_0}$ for v_0, and a loss is incurred unless $\mathbf{p}_{v_0}[u_+] \geq \mathbf{p}_{v_0}[u_-]$. The core of Backstrom and Leskovic's result is a method for computing the gradient of the loss on an example, given a differentiable feature-weighting function $f(\mathbf{w}, \phi)$ and a differentiable loss function ℓ. The gradient computation is broadly similar to the power-iteration method for computation of the personalized PageRank vector for v_0. Given the gradient, a number of optimization methods can be used to compute a local optimum.

Instead of directly using the above learning approach for ProPPR, we decompose the pairwise ranking loss into a standard positive-negative log loss function. The training data D is a set of triples $\{(Q^1, P^1, N^1), \ldots, (Q^m, P^m, N^m)\}$ where each Q^k is a query, $P^k = \langle Q\theta^1_+, \ldots, Q\theta^I_+ \rangle$ is a list of correct answers, and N^k is a list $\langle Q\theta^1_-, \ldots, Q\theta^J_- \rangle$ incorrect answers. We use a log loss with L_2 regularization of the parameter weights. Hence the final function to be optimized is

$$-\left(\sum_{k=1}^{I} \log \mathbf{p}_{v_0}[u^k_+] + \sum_{k=1}^{J} \log(1 - \mathbf{p}_{v_0}[u^k_-]) \right) + \mu ||\mathbf{w}||_2^2$$

To optimize this loss, we use stochastic gradient descent (SGD), rather than the quasi-Newton method of Backstrom and Leskovic. Weights are initialized to $1.0 + \delta$, where δ is randomly drawn from $[0, 0.01]$. We set the learning rate β of SGD to be $\beta = \frac{\eta}{\text{epoch}^2}$ where epoch is the current epoch in SGD, and η, the initial learning rate, defaults to 1.0.

We implemented SGD because it is fast and has been adapted to parallel learning tasks [32, 21]. Local grounding means that learning for ProPPR is quite well-suited to parallelization. The step of locally grounding each Q^i is "embarassingly" parallel, as every grounding can be done independently. To parallelize the weight-learning stage, we use multiple threads, each of which computes the gradient over a single grounding $\hat{G}_{Q^k}$, and all of which accesses a single shared parameter vector $\mathbf{w}$. Although the shared parameter vector is a potential bottleneck [31], it is not a severe one, as the gradient computation dominates the learning cost.[7]

3. EXPERIMENTS

3.1 Sample Tasks

To evaluate this method, we use data from several tasks. Because the semantics of ProPPR and other probabilistic logics are different, the tasks are evaluated by ranking the possible answers to a query, and scoring the ranking by a standard measure such as AUC; in other words, we are not attempting to evaluate the absolute probability scores produced by ProPPR, only the relative scores for a query.

Our first sample task is an entity resolution task previously studied as a test case for MLNs [27]. The program we use in the experiments is shown in Table 4: it is approximately the same as the MLN(B+T) approach from Singla and Domingos.[8] To evaluate accuracy, we use the CORA dataset, a collection of 1295 bibliography citations that refer to 132 distinct papers. We set the regularization coefficient μ to 0.001, the number of epochs to 5, and the learning rate parameter η to 1. An L_2-regularized standard log loss function was used in our objective function.

Our second task is a bag-of-words classification task, which was previously studied as a test case for both ProbLog [13] and MLNs [18]. In this experiment, we use the following ProPPR program:

```
class(X,Y) :- has(X,W), isLabel(Y), related(W,Y).
related(W,Y) :- true, # w(W,Y).
```

which is a bag-of-words classifier that is approximately the same as the ones used by Gutmann et al. [13], as well as

[7] This is not the case when learning a linear classifier, where gradient computations are much cheaper.

[8] The principle difference is that we do not include tests on the absence of words in a field in our clauses, and we drop the non-horn clauses from their program.

Table 3: Some more sample ProPPR programs. $LP = \{c_1, c_2\}$ is a bag-of-words classifier (see text). $LP = \{c_1, c_2, c_3, c_4\}$ is a recursive label-propagation scheme, in which predicted labels for one document are assigned to similar documents, with similarity being an (untrained) cosine distance-like measure. $LP = \{c_1, c_2, c_5, c_6\}$ is a sequential classifier for document sequences.

```
c1: predictedClass(Doc,Y) :-
        possibleClass(Y),
        hasWord(Doc,W),
        related(W,Y) # c1.
c2: related(W,Y) :- true,
    # relatedFeature(W,Y)

Database predicates:
hasWord(D,W): doc D contains word W
inDoc(W,D): doc D contains word W
previous(D1,D2): doc D2 precedes D1
possibleClass(Y): Y is a class label
```

```
c3: predictedClass(Doc,Y) :-
        similar(Doc,OtherDoc),
        predictedClass(OtherDoc,Y) # c3.
c4 : similar(Doc1,Doc2) :-
        hasWord(Doc1,W),
        inDoc(W,Doc2) # c4.

c5 : predictedClass(Doc,Y) :-
        previous(Doc,OtherDoc),
        predictedClass(OtherDoc,OtherY),
        transition(OtherY,Y) # c5.
c6: transition(Y1,Y2) :- true,
    # transitionFeature(Y1,Y2)
```

Lowd and Domingos[9] [18]. The dataset we use is the WebKb dataset, which includes a set of web pages from four computer science departments (Cornell, Wisconsin, Washington, and Texas). Each web page has one or multiple labels: *course, department, faculty, person, research project, staff, and student*. The task is to classify the given URL into the above categories. This dataset has a total of 4165 web pages. Using our ProPPR program, we learn a separate weight for each word for each label.

In addition to the entity resolution task and the bag-of-words classification task, we also investigate our approach for joint inference in a link (relation) prediction problem. In this experiment, our goal is to answer the following question: can we use ProPPR to perform joint inference to improve the performance of a link prediction task on a relational knowledge base? To do this, we use a subset of 19,527 beliefs from a knowledge base, which is extracted imperfectly from the web by NELL, a never-ending language learner [5]. The training set contains 12,331 queries, and the test set contains 1,185 queries. In contrast to a previous approach [16] for link prediction, we combine the top-ranked paths learned by PRA, another method for link prediction that we have applied to the NELL's KB [15], and transform these paths into ProPPR programs, then perform joint inference to predict the links between entities. The total number of translated rules is 797, and ϵ was set to 0.00001. One goal of this experiment is to evaluate the performance of ProPPR on larger logic programs, containing hundreds of rules. To do this, we build on a previous approach [16] called PRA for link prediction, which learns weighted sets of "paths" of relations. We compare two experimental settings: the rules (paths) that are non-recursive and recursive. A non-recursive rule only makes use of the information in the database, and therefore cannot be used for joint learning with other learned PRA rules. For example, the relation agentActsinLocation only has the following non-recursive rule:

```
agentActsinLocation :- fact_agentActsinLocation.
```

and this fact_agentActsinLocation predicate only uses the beliefs in the database, but not other PRA rules. Recursive rules, on the other hand, allow us to perform joint inference on all the learned PRA rules related to the given relation. Here is an excerpt of the recursive ProPPR program, translated from the learned PRA rules:

```
athletePlaySport(Athlete,Sport) :-
    fact_athletePlaySport(Athlete,Sport).

athletePlaySport(Athlete,Sport) :-
    onTeam(Athlete,Team), teamPlaysSport(Team,Sport)

teamPlaysSport(Team,Sport) :-
    member(Team,Conference),
    member(Team2,Conference),
    plays(Team2,Sport).

teamPlaysSport(Team,Sport) :-
    onTeam(Athlete,Team),
    athletePlaysSport(Athlete,Sport).
```

3.2 CORA Entity Resolution Results

We first consider the cost of the PageRank-Nibble-Prove inference/grounding technique. Table 5 shows the time required for inference (with uniform weights) for a set of 52 randomly chosen entity-resolution tasks from the CORA dataset, using a Python implementation of the theorem-prover. We report the time in seconds for all 52 tasks, as well as the mean average precision (MAP) of the scoring for each query. It is clear that PageRank-Nibble-Prove offers a substantial speedup on these problems with little loss in accuracy: on these problems, the same level of accuracy is achieved in less than a tenth of the time.

While the speedup in inference time is desirable, the more important advantages of the local grounding approach are

[9]Note that we do not use the negation rule and the link rule from Lowd and Domingos.

Table 4: ProPPR program used for entity resolution.

samebib(BC1,BC2) :- author(BC1,A1),sameauthor(A1,A2),authorinverse(A2,BC2)	# author.
samebib(BC1,BC2) :- title(BC1,A1),sametitle(A1,A2),titleinverse(A2,BC2)	# title.
samebib(BC1,BC2) :- venue(BC1,A1),samevenue(A1,A2),venueinverse(A2,BC2)	# venue.
samebib(BC1,BC2) :- samebib(BC1,BC3),samebib(BC3,BC2)	# tcbib.
sameauthor(A1,A2) :- haswordauthor(A1,W),haswordauthorinverse(W,A2),keyauthorword(W)	# authorword.
sameauthor(A1,A2) :- sameauthor(A1,A3),sameauthor(A3,A2)	# tcauthor.
sametitle(A1,A2) :- haswordtitle(A1,W),haswordtitleinverse(W,A2),keytitleword(W)	# titleword.
sametitle(A1,A2) :- sametitle(A1,A3),sametitle(A3,A2)	# tctitle.
samevenue(A1,A2) :- haswordvenue(A1,W),haswordvenueinverse(W,A2),keyvenueword(W)	# venueword.
samevenue(A1,A2) :- samevenue(A1,A3),samevenue(A3,A2)	# tcvenue.
keyauthorword(W) :- true	# authorWord(W).
keytitleword(W) :- true	# titleWord(W).
keyvenueword(W) :- true	# venueWord(W).

Table 5: Performance of the approximate PageRank-Nibble-Prove method on the Cora dataset, compared to the grounding by running personalized PageRank to convergence. In all cases $\alpha' = 0.1$.

ϵ	MAP	Time(sec)
0.0001	0.30	28
0.00005	0.40	39
0.00002	0.53	75
0.00001	0.54	116
0.000005	0.54	216
power iteration	0.54	819

Table 6: AUC results on CORA citation-matching.

	Cites	Authors	Venues	Titles
MLN(Fig 1)	0.513	0.532	0.602	0.544
MLN(S&D)	0.520	0.573	0.627	0.629
ProPPR(**w**=1)	0.680	0.836	0.860	**0.908**
ProPPR	**0.800**	**0.840**	**0.869**	0.900

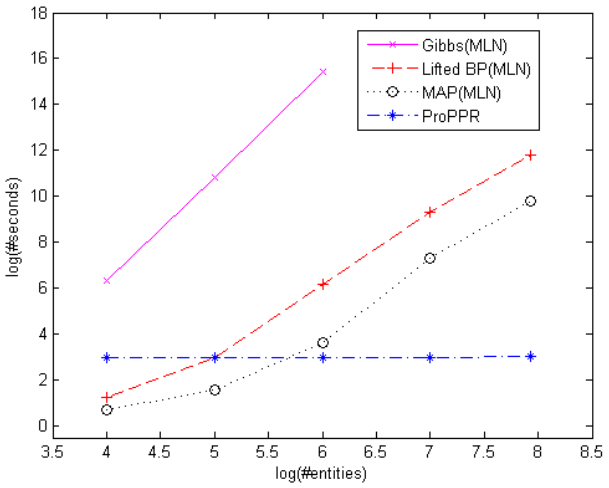

Figure 3: Run-time for inference on the Cora dataset using ProPPR (with a single thread) as a function of the number of entities in the database. The base of the log is 2.

that (1) grounding time, and hence inference, need not grow with the database size and (2) learning can be performed in parallel, by using multiple threads for parallel computations of gradients in SGD. Figure 3 illustrates the first of these points: the scalability of the PageRank-Nibble-Prove method as database size increases. For comparison, we also show the inference time for MLNs with three inference methods: Gibbs refers to Gibbs sampling, Lifted BP is the lifted belief propagation method, and MAP is the maximum a posteriori inference approach. In each case the performance task is inference over 16 test queries.

Note that ProPPR's runtime is constant, independent of the database size: it takes essentially the same time for $2^8 = 256$ entities as for $2^4 = 16$. In contrast, lifted belief propagation is around 1000 times slower on the larger database.

Figure 4 explores the speedup in learning (from grounded examples) due to multi-threading. The weight-learning is using a Java implementation of the algorithm which runs over ground graphs. The full CORA dataset was used in this experiment. As can be seen, the speedup that is obtained is nearly optimal, even with 16 threads running concurrently.

We finally consider the effectiveness of weight learning. We train on the first four sections of the CORA dataset, and report results on the fifth. Following Singla and Domingos [27] we report performance as area under the ROC curve (AUC). Table 6 shows AUC on the test set used by Singla and Domingos for several methods. The line for MLN(Fig 1) shows results obtained by an MLN version of the program of Figure 1. The line MLN(S&D) shows analogous results for the best-performing MLN from [27]. Compared to these methods, ProPPR does quite well even before training (with unit feature weights, **w**=1); the improvement here is likely due to the ProPPR's bias towards short proofs, and the tendency of the PPR method to put more weight on shared words that are rare (and hence have lower fanout in the graph walk.) Training ProPPR improves performance on three of the four tasks, and gives the most improvement on citation-matching, the most complex task.

The results in Table 6 all use the same data and evaluation procedure, and the MLNs were trained with the state-of-the-art Alchemy system using the recommended commands

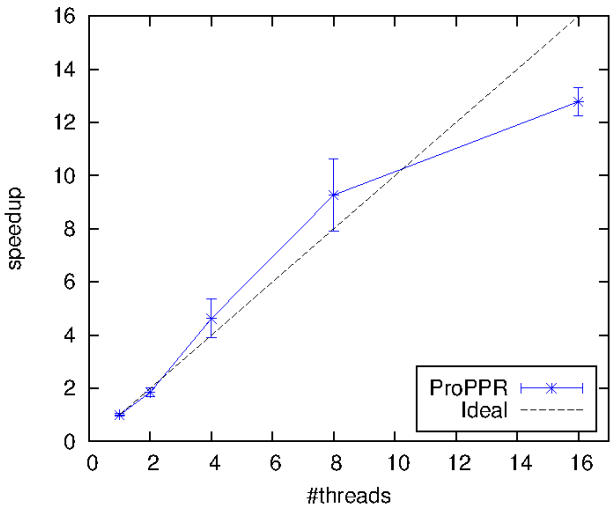

Figure 4: Performance of the parallel SGD method on CORA dataset. The x axis is the number of threads on a multicore machine, and the y axis is the speedup factor over a single-threaded implementation.

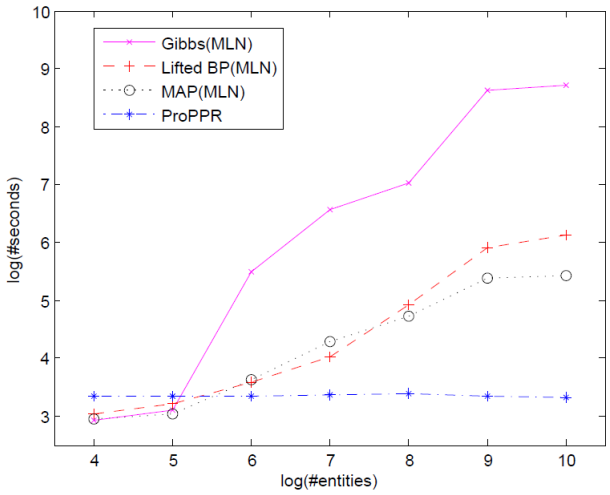

Figure 5: Run-time for inference on the WebKb dataset using ProPPR (with a single thread) as a function of the number of entities in the database. The base of the log is 2.

for this data (which is distributed with Alchemy[10]). However, we should note that the MLN results reproduced here are not identical to previous-reported ones [27]. Singla and Domingos used a number of complex heuristics that are difficult to reproduce—e.g., one of these was combining MLNs with a heuristic, TFIDF-based matching procedure based on canopies [19]. While the trained ProPPR model outperformed the reproduced MLN model in all prediction tasks, it outperforms the reported results from Singla and Domingos only on *venue*, and does less well than the reported results on *citation* and *author*[11].

3.3 WebKb Classification Results

Similar to the evaluation of the entity resolution task, here we focus on three evaluations: the cost of inference as a function of the database size, the accuracy in the classification task, and the speedup in the learning due to multi-threading.

We show the cost of inference as a function of database size on the WebKb dataset in the Figure 5. In this experiment, we fix the number of test queries, and vary the number of entities in the database. We see that the run time for ProPPR is independent of the size of the database: it takes the same amount of time for ProPPR to perform inference for $2^{10} = 1024$ entities as for $2^4 = 16$ entities. However, this is not the case for inference in the Markov logic network. We see that when the size of the database is small, all of the approaches have similar run time, but when there are 1024 entities in the database, the run time of each method diverges significantly. The result is consistent with those of the CORA experiments.

We also consider the accuracy of the ProPPR language on the Webkb dataset. We use exactly the same cross-validation experimental settings that previous work use [18, 13]: in each fold, for the four universities, we train on the three, and report result on the fourth. In Table 7, we show the detailed AUC results of each fold, as well as the averaged results[12]. First, we see that if we do not perform weight learning, the averaged result is equivalent to a random baseline. As reported by Gutmann et al. [13], the ProbLog approach obtains an AUC of 0.606 on the dataset. The voted perceptron algorithm (MLN VP, AUC $\approx$ 0.605) and the contrastive divergence algorithm (MLN CD, AUC $\approx$ 0.604) reported by Lowd and Domingos [18] are within the same range as ProbLog. When using the conjugate gradients approach, the MLN (CG) achieved an AUC of 0.730. When comparing to the trained version of ProPPR on the same dataset, we see that ProPPR obtains an AUC of 0.797, which outperforms all the results reported by ProbLog and MLN.

Finally, we consider the speedup in learning due to multi-threading on the WebKb dataset. Learning time averages about 950 seconds with a single thread, but this can be reduced to only two minutes if 16 threads are used. For comparison, Lowd and Domingos report that around 10,000 seconds were needed to obtain the best results were obtained for MLNs. The multi-threaded speed up performance on different sections of the WebKb dataset is shown in Table 8.

3.4 NELL Link Prediction Results

The accuracy results on the NELL link-prediction task are shown in Table 9. We observe that when not performing joint inference with the learned top-ranked paths, ProPPR obtains an AUC of 0.858. However, when using these rules for joint learning, we observe an AUC of 0.916 when using ProPPR. The total time for joint inference with 797 rules is 13 minutes.

4. RELATED WORK

[10]http://alchemy.cs.washington.edu

[11]Performance on *title* matching is not reported by Singla and Domingos.

[12]Note that both [18, 13] do not show the detailed breakdown of the results on WebKb dataset.

Table 7: AUC results on the WebKb classification task. Co.: Cornell. Wi.: Wisconsin. Wa.: Washington. Te.: Texas.

	Co.	Wi.	Wa.	Te.	Avg.
ProLog [13]	–	–	–	–	0.606
MLN (VP) [18]	–	–	–	–	0.605
MLN (CD) [18]	–	–	–	–	0.604
MLN (CG) [18]	–	–	–	–	0.730
ProPPR($\mathbf{w}$=1)	0.501	0.495	0.501	0.505	0.500
ProPPR	**0.785**	**0.779**	**0.795**	**0.828**	**0.797**

Table 8: Detailed Performance (seconds) of the parallel SGD method of ProPPR on the WebKb dataset. Co.: Cornell. Wi.: Wisconsin. Wa.: Washington. Te.: Texas.

#Threads	Co.	Wi.	Wa.	Te.	Avg.
1	1190.4	504.0	1085.9	1036.4	954.2
2	594.9	274.5	565.7	572.5	501.9
4	380.6	141.8	404.2	396.6	330.8
8	249.4	94.5	170.2	231.5	186.4
16	137.8	69.6	129.6	141.4	119.6

Although we have chosen here to compare mainly to MLNs [25, 27], ProPPR represents a rather different philosophy toward language design: rather than beginning with a highly-expressive but intractable logical core, we begin with a limited logical inference scheme and add to it a minimal set of extensions that allow probabilistic reasoning, while maintaining stable, efficient inference and learning. While ProPPR is less expressive than MLNs (for instance, it is limited to definite clause theories) it is also much more efficient. This philosophy is similar to that illustrated by probabilistic similarity logic (PSL) [4]; however, unlike ProPPR, PSL does not include a "local" grounding procedure, which leads to small inference problems, even for large databases. Our work also aligns with the lifted personalized PageRank [1] algorithm, which can be easily incorporated as an alternative inference algorithm in our language.

Technically, ProPPR is most similar to stochastic logic programs (SLPs) [9]. The key innovation is the integration of a restart into the random-walk process, which, as we have seen, leads to very different computational properties.

There has been some prior work on reducing the cost of grounding probabilistic logics: notably, Shavlik et al [26] describe a preprocessing algorithm called FROG that uses various heuristics to greatly reduce grounding size and inference cost, and Niu et al [20] describe a more efficient bottom-up grounding procedure that uses an RDBMS. Other methods that reduce grounding cost and memory usage include "lifted" inference methods (e.g., [29]) and "lazy" inference methods (e.g., [28]); in fact, the LazySAT inference scheme for Markov networks is broadly similar algorithmically to PageRank-Nibble-Prove, in that it incrementally extends a network in the course of theorem-proving. However, there is no theoretical analysis of the complexity of these methods, and experiments with FROG and LazySAT suggest that they still lead to a groundings that grow with DB size, albeit more slowly.

ProPPR is also closely related to the PRA, learning algorithm for link prediction [15], like ProPPR, PRA uses random walk processes to define a distribution, rather than some other forms of logical inference, such as belief propagation. In this respect PRA and ProPPR appear to be unique among probabilistic learning methods; however, this distinction may not be as great as it first appears, as it is known there are close connections between personalized PageRank and traditional probabilistic inference schemes [13]. PRA, however, is much more limited than ProPPR: PRA uses random-walk methods to approximate logical inference. The set of "inference rules" learned by PRA corresponds roughly to a logic program in a particular form—namely, the form

$$p(S,T) \leftarrow r_{1,1}(S,X_1),\ldots,r_{1,k_1}(X_{k_1-1},T).$$
$$p(S,T) \leftarrow r_{2,1}(S,X_1),\ldots,r_{2,k_2}(X_{k_2-1},T).$$

ProPPR allows much more general logic programs. However, unlike PRA, we do not consider the task of searching for new logic program clauses.

Table 9: AUC results on the NELL link prediction task.

	AUC
ProPPR(Non-recursive PRA rules)	0.858
ProPPR(Recursive PRA rules)	**0.916**

5. CONCLUSIONS

We described a new probabilistic first-order language which is designed with the goal of highly efficient inference and rapid learning. ProPPR takes Prolog's SLD theorem-proving, extends it with a probabilistic proof procedure, and then limits this procedure further, by including a "restart" step which biases the system to short proofs. This means that ProPPR has a simple polynomial-time proof procedure, based on the well-studied personalized PageRank (PPR) method.

Following prior work on PPR-like methods, we designed a local grounding procedure for ProPPR, based on local partitioning methods [2], which leads to an inference scheme that is an order of magnitude faster that the conventional power-iteration approach to computing PPR, takes time $O(\frac{1}{\epsilon\alpha'})$, independent of database size. This ability to "locally ground" a query also makes it possible to partition the weight learning task into many separate gradient computations, one for each training example, leading to a weight-learning method that can be easily parallelized. In our current implementation, an additional order-of-magnitude speedup in learning is made possible by parallelization. Experimentally, we showed that ProPPR performs well on an entity resolution task, a classification task, and a joint inference task. The cost of the inference is independent of the data size, and the speedup in learning is made possible due to multi-threading.

Acknowledgements

We are grateful to anonymous reviewers for useful comments. This work was sponsored in part by DARPA grant FA87501220342 to CMU and a Google Research Award.

[13]For instance, it is known that personalized PageRank can be used to approximate belief propagation on certain graphs [8].

6. REFERENCES

[1] Babak Ahmadi, Kristian Kersting, and Scott Sanner. Multi-evidence lifted message passing, with application to pagerank and the kalman filter. In *Proceedings of the Twenty-Second international joint conference on Artificial Intelligence*, 2011.

[2] Reid Andersen, Fan R. K. Chung, and Kevin J. Lang. Local partitioning for directed graphs using pagerank. *Internet Mathematics*, 5(1):3–22, 2008.

[3] Lars Backstrom and Jure Leskovec. Supervised random walks: predicting and recommending links in social networks. In *Proceedings of the fourth ACM international conference on Web search and data mining*, 2011.

[4] Matthias Brocheler, Lilyana Mihalkova, and Lise Getoor. Probabilistic similarity logic. In *Proceedings of the Conference on Uncertainty in Artificial Intelligence*, 2010.

[5] Andrew Carlson, Justin Betteridge, Bryan Kisiel, Burr Settles, Estevam R. Hruschka Jr., and Tom M. Mitchell. Toward an architecture for never-ending language learning. In Maria Fox and David Poole, editors, *AAAI*. AAAI Press, 2010.

[6] Soumen Chakrabarti. Dynamic personalized PageRank in entity-relation graphs. In *Proceedings of the 16th international conference on World Wide Web*, 2007.

[7] William W. Cohen. Data integration using similarity joins and a word-based information representation language. *ACM Transactions on Information Systems*, 18(3):288–321, July 2000.

[8] William W Cohen. *Graph Walks and Graphical Models*. Carnegie Mellon University, School of Computer Science, Machine Learning Department, 2010.

[9] James Cussens. Parameter estimation in stochastic logic programs. *Machine Learning*, 44(3):245–271, 2001.

[10] Luc De Raedt, Angelika Kimmig, and Hannu Toivonen. Problog: A probabilistic prolog and its application in link discovery. In *Proceedings of the 20th international joint conference on Artifical intelligence*, 2007.

[11] Pedro Domingos and Daniel Lowd. *Markov Logic: An Interface Layer for Artificial Intelligence*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2009.

[12] Norbert Fuhr. Probabilistic datalog—a logic for powerful retrieval methods. In *Proceedings of the 18th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 282–290. ACM, 1995.

[13] Bernd Gutmann, Angelika Kimmig, Kristian Kersting, and Luc De Raedt. Parameter estimation in problog from annotated queries. *CW Reports*, 2010.

[14] Abhay Jha and Dan Suciu. Probabilistic databases with markoviews. *Proceedings of the VLDB Endowment*, 5(11):1160–1171, 2012.

[15] Ni Lao and William W. Cohen. Relational retrieval using a combination of path-constrained random walks. *Machine Learning*, 81(1):53–67, 2010.

[16] Ni Lao, Tom M. Mitchell, and William W. Cohen. Random walk inference and learning in a large scale knowledge base. In *EMNLP*, pages 529–539. ACL, 2011.

[17] J. W. Lloyd. *Foundations of Logic Programming: Second Edition*. Springer-Verlag, 1987.

[18] Daniel Lowd and Pedro Domingos. Efficient weight learning for markov logic networks. In *Knowledge Discovery in Databases: PKDD 2007*, pages 200–211. Springer, 2007.

[19] Andrew McCallum, Kamal Nigam, and Lyle H. Ungar. Efficient clustering of high-dimensional data sets with application to reference matching. In *Knowledge Discovery and Data Mining*, pages 169–178, 2000.

[20] Feng Niu, Christopher Ré, AnHai Doan, and Jude Shavlik. Tuffy: Scaling up statistical inference in markov logic networks using an RDBMS. *Proceedings of the VLDB Endowment*, 4(6):373–384, 2011.

[21] Feng Niu, Benjamin Recht, Christopher Ré, and Stephen J Wright. Hogwild!: A lock-free approach to parallelizing stochastic gradient descent. *arXiv preprint arXiv:1106.5730*, 2011.

[22] Larry Page, Sergey Brin, R. Motwani, and T. Winograd. The PageRank citation ranking: Bringing order to the web. In *Technical Report, Computer Science department, Stanford University*, 1998.

[23] Hoifung Poon and Pedro Domingos. Joint inference in information extraction. In *Proceedings of the National Conference on Artificial Intelligence*, 2007.

[24] Hoifung Poon and Pedro Domingos. Joint unsupervised coreference resolution with markov logic. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, pages 650–659. Association for Computational Linguistics, 2008.

[25] Matthew Richardson and Pedro Domingos. Markov logic networks. *Mach. Learn.*, 62(1-2):107–136, 2006.

[26] Jude Shavlik and Sriraam Natarajan. Speeding up inference in markov logic networks by preprocessing to reduce the size of the resulting grounded network. In *Proceedings of the Twenty-first International Joint Conference on Artificial Intelligence (IJCAI-09)*, 2009.

[27] Parag Singla and Pedro Domingos. Entity resolution with markov logic. In *Data Mining, 2006. ICDM'06. Sixth International Conference on*, 2006.

[28] Parag Singla and Pedro Domingos. Memory-efficient inference in relational domains. In *Proceedings of the national conference on Artificial intelligence*, 2006.

[29] Parag Singla and Pedro Domingos. Lifted first-order belief propagation. In *Proceedings of the 23rd national conference on Artificial intelligence*, 2008.

[30] Hanghang Tong, Christos Faloutsos, and Jia-Yu Pan. Fast random walk with restart and its applications. In *ICDM*, pages 613–622. IEEE Computer Society, 2006.

[31] Martin Zinkevich, Alex Smola, and John Langford. Slow learners are fast. *Advances in Neural Information Processing Systems*, 22:2331–2339, 2009.

[32] Martin Zinkevich, Markus Weimer, Alex Smola, and Lihong Li. Parallelized stochastic gradient descent. *Advances in Neural Information Processing Systems*, 2010.