A General Method for Reducing the Complexity of Relational Inference And its Application to MCMC

Hoifung Poon **Pedro Domingos** **Marc Sumner**
Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195-2350, U.S.A.
{*hoifung, pedrod, marcs*}@*cs.washington.edu*

Abstract

Many real-world problems are characterized by complex relational structure, which can be succinctly represented in first-order logic. However, many relational inference algorithms proceed by first fully instantiating the first-order theory and then working at the propositional level. The applicability of such approaches is severely limited by the exponential time and memory cost of propositionalization. Singla and Domingos (2006) addressed this by developing a "lazy" version of the WalkSAT algorithm, which grounds atoms and clauses only as needed. In this paper we generalize their ideas to a much broader class of algorithms, including other types of SAT solvers and probabilistic inference methods like MCMC. Lazy inference is potentially applicable whenever variables and functions have default values (i.e., a value that is much more frequent than the others). In relational domains, the default is false for atoms and true for clauses. We illustrate our framework by applying it to MC-SAT, a state-of-the-art MCMC algorithm. Experiments on a number of real-world domains show that lazy inference reduces both space and time by several orders of magnitude, making probabilistic relational inference applicable in previously infeasible domains.

Introduction

In many real-world problems, the variables and functions in question have "default" values that occur much more frequently than others. Algorithms that exploit this can greatly reduce the space and time complexity of computation (e.g., sparse matrix computation). In relational domains, the default is false for atoms and true for clauses. For example, with 1000 persons, the predicate $\texttt{Coauthor}(\texttt{person}, \texttt{person})$ yields 1,000,000 groundings, and the clause of transitivity for coauthorship $\texttt{Coauthor}(\texttt{p1}, \texttt{p2}) \wedge \texttt{Coauthor}(\texttt{p2}, \texttt{p3}) \Rightarrow \texttt{Coauthor}(\texttt{p1}, \texttt{p3})$ yields 1,000,000,000 groundings; however, most Coauthor atoms are false, which trivially satisfies most transitivity clauses. Therefore, although the number of potential groundings can be very large, most of their values need not be computed explicitly.

Recently, there has been increasing interest in statistical relational learning (Getoor & Taskar 2007) and structured prediction (Bakir *et al.* 2007). However, most existing approaches fail to leverage the sparsity of relational domains. Instead, many approaches avoid the combinatorial blow-up by either not modeling relational uncertainty, or doing so in a very limited way. For example, probabilistic relational models (PRMs) (Getoor *et al.* 2001) only model uncertainty over binary relations given attributes, not arbitrary dependencies between relations (e.g., transitivity). Other approaches such as Markov logic (Richardson & Domingos 2006) do not sacrifice modeling capability, but their inference algorithms (e.g., MC-SAT) typically require complete grounding of atoms/clauses, which severely limits their applicability. While there are some promising developments in lifted inference (Braz *et al.* 2006), they are an extension of variable elimination and their scalability is limited. Milch & Russell (2006) proposed a form of MCMC inference for relational domains that can avoid completely grounding the state, but their approach requires users to provide domain-specific proposal distributions and its scalability is unknown.

The LazySAT algorithm (Singla & Domingos 2006) is a notable exception that takes advantage of relational sparsity and greatly reduces memory usage. It is the lazy version of WalkSAT (Selman *et al.* 1996), but the ideas go beyond satisfiability testing. In this paper, we present a general method for lazy inference that applies to any algorithm with the following characteristics: the algorithm works by repeatedly selecting a function or variable and recomputing its value, and the value of a variable/function depends only on the values of a subset of the other variables/functions. These include a diverse set of algorithms for a variety of applications: WalkSAT, MaxWalkSAT, SampleSAT, DPLL, iterated conditional modes, simulated annealing and MCMC algorithms that update one variable at a time (Gibbs sampling, single-component Metropolis-Hastings and simulated tempering, etc.), MC-SAT, belief propagation, algorithms for maximum expected utility, etc. We illustrate our method by applying it to MC-SAT (Poon & Domingos 2006), a state-of-the-art MCMC algorithm. Experiments on three real-world domains show that Lazy-MC-SAT, the resulting algorithm, reduces both space and time by several orders of magnitude.

We begin by briefly reviewing some necessary background. We then describe our general method for lazy inference and illustrate how it applies to various algorithms including MC-SAT. Finally, we present our experimental results and conclude.

Copyright © 2008, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

Background

A *first-order knowledge base (KB)* is a set of sentences or formulas in first-order logic (Genesereth & Nilsson 1987). A *term* is any expression representing an object in the domain. An *atomic formula* or *atom* is a predicate symbol applied to a tuple of terms. Formulas are recursively constructed from atomic formulas using logical connectives and quantifiers. A KB in *clausal form* is a conjunction of *clauses*, a clause being a disjunction of literals, a literal being an atom or its negation. A *unit clause* is a clause with one literal. A literal is *pure* if it is always negated or always non-negated in the clauses that contain it. A *ground term* is a term containing no variables. A *ground atom* or *ground predicate* is an atom all of whose arguments are ground terms. In finite domains, first-order KBs can be *propositionalized* by replacing each universally (existentially) quantified formula with a conjunction (disjunction) of all its groundings.

A central problem in logic is that of determining if a KB (usually in clausal form) is *satisfiable*, i.e., if there is an assignment of truth values to ground atoms that makes the KB true. One approach to this problem is stochastic local search, exemplified by WalkSAT (Selman *et al.* 1996). Starting from a random initial state, WalkSAT repeatedly flips (changes the truth value of) an atom in a random unsatisfied clause. With probability p, WalkSAT chooses the atom that minimizes the number of unsatisfied clauses, and with probability $1-p$ it chooses a random one. WalkSAT is able to solve hard instances of satisfiability with hundreds of thousands of variables in minutes. The MaxWalkSAT (Kautz *et al.* 1997) algorithm extends WalkSAT to the weighted satisfiability problem, where each clause has a weight and the goal is to maximize the sum of the weights of satisfied clauses. WalkSAT is essentially the special case of MaxWalkSAT obtained by giving all clauses the same weight. For simplicity, in this paper we will just treat them as one algorithm, called WalkSAT, with the sum of the weights of *unsatisfied* clauses as the cost function that we seek to minimize. Further, instead of directly minimizing the cost, we will minimize DeltaCost(v), the change in cost resulting from flipping v in the current solution, which is equivalent.

Markov logic is a probabilistic extension of first-order logic that makes it possible to compactly specify probability distributions over complex relational domains (Richardson & Domingos 2006). A *Markov logic network (MLN)* is a set of weighted first-order clauses. Together with a set of constants, it defines a Markov network with one node per ground atom and one feature per ground clause. The weight of a feature is the weight of the first-order clause that originated it. The probability of a state x in such a network is given by $P(x) = (1/Z)\exp\left(\sum_i w_i f_i(x)\right)$, where Z is a normalization constant, w_i is the weight of the ith clause, $f_i = 1$ if the ith clause is true, and $f_i = 0$ otherwise. MPE inference (finding the most probable state given evidence) can be done using a weighted satisfiability solver (e.g., LazySAT); conditional probabilities can be computed using Markov chain Monte Carlo (e.g., MC-SAT).

A General Method for Lazy Inference

In a domain where most variables assume the default value, it is wasteful to allocate memory for all variables and functions in advance. The basic idea of lazy inference is to allocate memory only for a small subset of "active" variables and functions, and activate more if necessary as inference proceeds. In addition to saving memory, this can reduce inference time as well, for we do not allocate memory and compute values for functions that are never used.

Definition 1 Let X be the set of variables and D be their domain.[1] The *default value* $d^* \in D$ is the most frequent value of the variables. An *evidence variable* is a variable whose value is given and fixed. A *function* $f = f(z_1, z_2, \cdots, z_k)$ inputs z_i's, which are either variables or functions, and outputs some value in the range of f.

In the rest of this paper, we focus on relational domains. Variables are ground atoms, which take binary values (i.e., $D = \{true, false\}$). The default value for variables is false (i.e., $d^* = false$). Examples of functions are clauses and DeltaCost. Like variables, functions may also have default values (e.g., true for clauses). The inputs to a relational inference algorithm are a weighted KB and a set of evidence atoms (DB). Eager algorithms work by first carrying out propositionalization and then calling a propositional algorithm. In lazy inference, we directly work on the KB and DB. The following concepts are crucial to lazy inference.

Definition 2 A variable v is *active* iff v is set to a non-default value at some point, and x is *inactive* if the value of x has always been d^*. A function f is *activated* by a variable v if either v is an input of f, or v activates a function g that is an input of f.

Basic Lazy Inference

Let $\mathcal{A}$ be the eager algorithm that we want to make lazy. We make three assumptions about $\mathcal{A}$:

1. $\mathcal{A}$ updates one variable at a time. (If not, the extension is straightforward.)
2. The values of variables in $\mathcal{A}$ are properly encapsulated so that they can be accessed by the rest of the algorithm only via two methods: ReadVar(x) (which returns the value of x) and WriteVar(x, v) (which sets the value of x to v). This is reasonable given the conventions in software development, and if not, it is easy to implement.
3. $\mathcal{A}$ always sets values of the variables before calling a function that depends on those variables, as it should be.

To develop the lazy version of $\mathcal{A}$, first, we identify the variables (usually all of them) and functions to make lazy. We then modify the value-accessing methods and replace the propositionalization step with lazy initialization as follows. The rest of the algorithm remains the same.

ReadVar(x): If x is in memory, Lazy-$\mathcal{A}$ returns its value as $\mathcal{A}$; otherwise, it returns d^*.

[1]For simplicity we assume that all variables have the same domain. The extension to different domains is straightforward.

WriteVar(x, v): If x is in memory, Lazy-$\mathcal{A}$ updates its value as $\mathcal{A}$. If not, and if $v = d^*$, no action is taken; otherwise, Lazy-$\mathcal{A}$ activates (allocates memory for) x and the functions activated by x, and then sets the value.

Initialization: Lazy-$\mathcal{A}$ starts by allocating memory for the lazy functions that output non-default values when all variables assume the default values. It then calls `WriteVar()` to set values for evidence variables, which activates those evidence variables with non-default values and the functions they activate. Such variables become the initial active variables and their values are fixed throughout the inference.

For example, for WalkSAT, the functions to be made lazy are the clauses, and Lazy-WalkSAT initializes by activating true evidence atoms and initial unsatisfied clauses (i.e., they are unsatisfied when the true evidence atoms are set to true and all other atoms are set to false). While computing DeltaCost(v), if v is active, the relevant clauses are already in memory; otherwise, they will be activated when v is set to true (a necessary step before computing the cost change when v is set to true). Lazy-$\mathcal{A}$ carries out the same inference steps as $\mathcal{A}$ and produces the same result. It never allocates memory for more variables/functions than $\mathcal{A}$, but each access incurs slightly more overhead (in checking whether a variable or function is in memory). In the worst case, most variables are updated, and Lazy-$\mathcal{A}$ produces little savings. However, if the updates are sparse, as is the case for most algorithms in relational domains, Lazy-$\mathcal{A}$ can greatly reduce memory and time because it activates and computes the values for many fewer variables and functions.

This basic version of lazy inference is generally applicable, but it does not exploit the characteristics of the algorithm and may yield little savings in some cases. Next, we describe three refinements that apply to many algorithms. We then discuss implementation issues.

Refinements

Function activation: To find the functions activated by a variable v, the basic version includes all functions that depend on v by traversing the dependency graph of variables/functions, starting from v. This can be done very efficiently, but may include functions whose values remain the default even if v changes its value. A more intelligent approach traverses the graph and only returns functions whose values become non-default if v changes its value. In general, this requires evaluating the functions, which can be expensive, but for some functions (e.g., clauses), it can be done efficiently as described at the end of this section. Which approach to use depends on the function.

Variable recycling: In the basic version, a variable is activated once it is set to a non-default value, and so are the functions activated by the variable. This is necessary if the variable keeps the non-default value, but could be wasteful in algorithms where many such updates are temporary (e.g., in simulated annealing and MCMC, we may temporarily set a variable to true to compute the probability for flipping). To save memory, if an inactive variable considered for flipping is not flipped in the end, we can discard it along with the functions activated by it. However, this will increase inference time if the variable is considered again later, so we should only apply this if we want to trade off time for memory, or if the variable is unlikely to be considered again.

Smart randomization: Many local-search and MCMC algorithms use randomization to set initial values for variables, or to choose the next variable to update. This is usually done without taking the sparsity of relational domains into account. For example, WalkSAT assigns random initial values to all variables; in a sparse domain where most atoms should be false, this is wasteful, because we will need to flip many atoms back to false. In lazy inference, we can randomize in a better way by focusing more on the active atoms and those "nearby", since they are more likely to become true. Formally, a variable w is a *1-neighbor* of a variable v if w is an input of a function f activated by v; w is a k*-neighbor* of v ($k > 1$) if w is a $(k-1)$-neighbor of v, or if w is a 1-neighbor of a $(k-1)$-neighbor of v. For initialization, we only randomize the k-neighbors of initial active variables; for variable selection, we also favor such variables.[2] k can be used to trade off efficiency and solution quality. With large k, this is just the same as before. The smaller k is, the more time and memory we can save from reducing the number of activation. However, we also run the risk of missing some relevant variables. If the domain is sparse, as typical relational domains are, the risk is negligible, and empirically we have observed that $k = 1$ greatly improves efficiency without sacrificing solution quality.

Implementation Details

We describe two issues in implementation and show how we address them. The first is how to obtain a random ground atom. In lazy inference, most atoms are not in memory, so we pick an atom in two steps: first pick the predicate with probability proportional to its number of groundings, then ground each argument of the predicate to a random constant of its type. Another key operation in lazy inference is finding the clauses activated by an atom. We use a general inverted-index scheme to facilitate this. For each evidence predicate that appears as a negative (positive) literal in a clause, we create an index to map the constants in each argument to the true (false) groundings. We illustrate how this works by considering the clause $\neg\mathtt{R}(\mathtt{x},\mathtt{y}) \vee \neg\mathtt{S}(\mathtt{z},\mathtt{y}) \vee \mathtt{T}(\mathtt{x},\mathtt{z})$, where $\mathtt{R}(\mathtt{A},\mathtt{B})$ is the atom in question and $\mathtt{S}$ and $\mathtt{T}$ are evidence predicates. To find the groundings of this clause activated by $\mathtt{R}(\mathtt{A},\mathtt{B})$, we ground $\mathtt{z}$ by taking the intersection of the true groundings of $\mathtt{S}(\mathtt{z},\mathtt{B})$ and false groundings of $\mathtt{T}(\mathtt{A},\mathtt{z})$, which are available from the indices.

Lazy Inference Algorithms

Lazy WalkSAT

As evident from the previous section, the Lazy-WalkSAT algorithm (LazySAT for short) is easily derived using our

[2]This does not preclude activation of remote relevant variables, which can happen after their clauses are active, or by random selection (e.g., in simulated annealing).

Algorithm 1 MC-SAT(*clauses, weights, num_samples*)

$x^{(0)} \leftarrow$ Satisfy(hard *clauses*)
for $i \leftarrow 1$ to *num_samples* **do**
  $M \leftarrow \emptyset$
  for all $c_k \in$ *clauses* satisfied by $x^{(i-1)}$ **do**
    With probability $1 - e^{-w_k}$ add c_k to M
  end for
  Sample $x^{(i)} \sim \mathcal{U}_{SAT(M)}$
end for

method by making clauses lazy and applying smart randomization with $k = 1$ to initialization. Singla & Domingos (2006) showed that LazySAT greatly reduces memory usage without sacrificing speed or solution quality.

Lazy DPLL

Basic DPLL (Davis *et al.* 1962) is a backtracking-based algorithm for satisfiability testing. During propositionalization, DPLL activates all clauses. If there is a unit clause, it sets its atom to satisfy the clause and then calls itself; if there is a pure literal, it sets it to satisfy the clauses it is in and then calls itself; otherwise, it picks an atom and sets it to true and calls itself; if this does not lead to a satisfying solution, it backtracks, sets the atom to false, and calls itself again.

In Lazy-DPLL, we make the clauses lazy and initialize by activating the true evidence atoms and the initially unsatisfied clauses. When an inactive atom v is set to false, no action is taken; when v is set to true, Lazy-DPLL also activates the clauses activated by v. Compared to DPLL, Lazy-DPLL activates many fewer clauses and reduces both memory and time. Note that modern implementations of DPLL also use various heuristics (e.g., MOMS), and an interesting problem for future research is adapting them for lazy inference.

Lazy Gibbs Sampling

Gibbs sampling initializes with a random state and repeatedly samples each atom according to its conditional probability given other atoms' values. For each query atom, it keeps a counter of how many times the atom is set to true. In Lazy-Gibbs, we make clauses and counters lazy and initialize by activating true evidence atoms and initial unsatisfied clauses. When an inactive atom v is set to false, no action is taken; when v is set to true, Lazy-Gibbs also activates the counter for v and the clauses activated by v. Both variable recycling and smart randomization are applicable here.

There are two ways to pick the next atom to sample in Gibbs sampling: cycling through all atoms in turn, or randomly picking one. The random approach usually performs better, and the atoms need not be picked uniformly. Lazy inference yields little savings for the cycling approach. For the random approach, however, it activates many fewer atoms, clauses, and counters, and reduces both memory and time.

Lazy MC-SAT

Relational domains often contain strong dependencies (e.g., transitivity), which result in slow convergence for MCMC algorithms; in the limit of deterministic dependencies, these methods are trapped in a single region and never converge to the correct answers. MC-SAT (Poon & Domingos 2006) is a *slice sampling* (Damien *et al.* 1999) algorithm that overcomes this problem by introducing auxiliary variables to capture the dependencies. Algorithm 1 gives pseudo-code for MC-SAT. It initializes with a solution to all hard clauses found by WalkSAT. At each iteration, it generates a set of clauses M by sampling from the currently satisfied clauses according to their weights. It then calls SampleSAT (Wei *et al.* 2004) to select the next state by sampling near-uniformly the solutions to M. SampleSAT initializes with a random state, and at each iteration, performs a simulated annealing step with probability p and a WalkSAT step with probability $1-p$. To improve efficiency, MC-SAT runs unit propagation (i.e., repeatedly fixes atoms in unit clauses and simplifies the remaining clauses) in M before calling SampleSAT. MC-SAT is orders of magnitude faster than previous MCMC algorithms such as Gibbs sampling and simulated tempering.

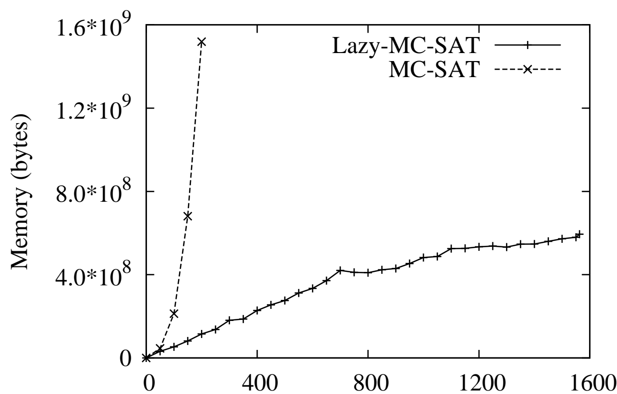
At first sight, MC-SAT does not seem like a suitable candidate for lazy inference. However, deriving Lazy-MC-SAT becomes straightforward using our method. Here, the functions to be made lazy are clauses, clauses' memberships in M, and whether an atom is fixed by unit propagation. Effectively, Lazy-MC-SAT initializes by calling LazySAT to find a solution to all hard clauses. At each iteration, it computes M-memberships for and runs unit propagation among active clauses only. It then calls Lazy-SampleSAT, the lazy version of SampleSAT. Lazy-SampleSAT initializes using smart randomization with $k = 1$. When an inactive atom v is set to false, no action is taken. When it is set to true, Lazy-SampleSAT activates v and the clauses activated by v; it then computes M-memberships for these clauses,[3] and runs unit propagation among them; if v is fixed to false by unit propagation, Lazy-SampleSAT sets v back to false and applies variable recycling. It also applies variable recycling to simulated annealing steps. Notice that, when an M-membership or fixed-atom flag is set, this is remembered until the current run of Lazy-SampleSAT ends, to avoid inconsistencies.

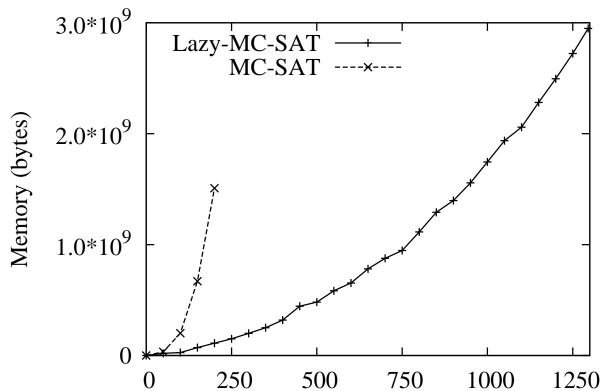
Experiments

Domains

Information extraction (IE) is the problem of extracting database records from text or semi-structured sources. Two key components of IE are *segmentation* (locating candidate database fields) and *entity resolution* (identifying duplicate records and fields). We used the CiteSeer and Cora datasets from Poon & Domingos (2007), which contain 1593 and 1295 citations, respectively. The goal is to compute marginal probabilities of fields and equalities. For both datasets, we used the learned MLNs from Poon & Domingos (2007), and added the transitivity rule: $\forall \mathtt{x}, \mathtt{y}, \mathtt{z}\ \mathtt{x} = \mathtt{y} \wedge \mathtt{y} = \mathtt{z} \Rightarrow \mathtt{x} = \mathtt{z}$. This rule is used in an *ad hoc* fashion in most entity resolution systems, and greatly complicates inference. Moreover, its arity of three leads to an explosion in memory and time, and as a result, MC-SAT cannot handle this rule given the full datasets. In CiteSeer, the total num-

[3] These clauses must be previously satisfied for otherwise they would have been activated before.





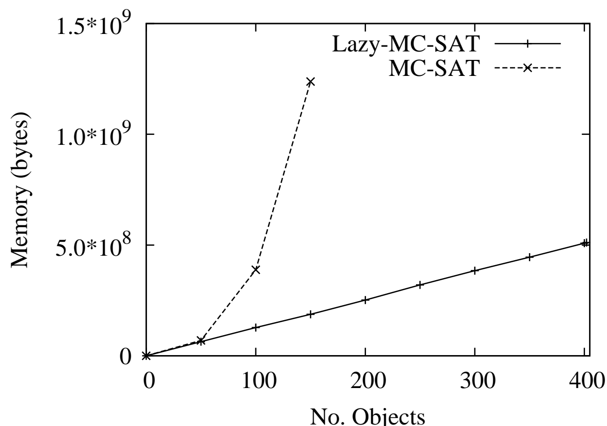
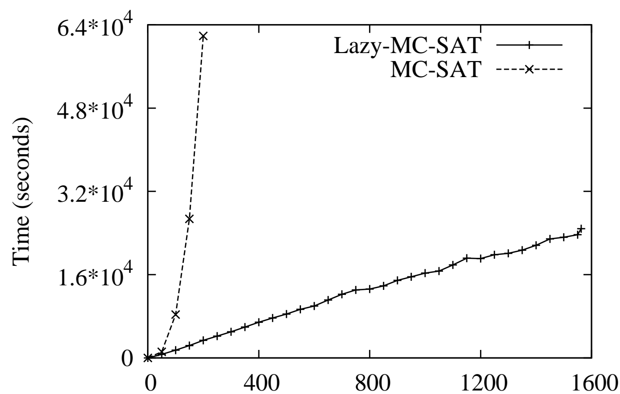
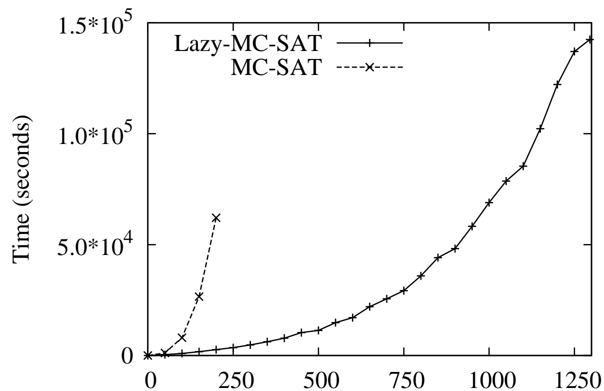


Figure 1: Memory usage vs. number of objects in CiteSeer (top), Cora (middle), and UW-CSE (bottom).





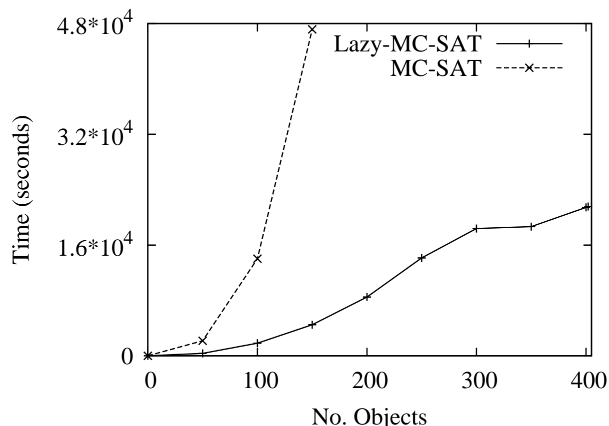


Figure 2: Inference time vs. number of objects in CiteSeer (top), Cora (middle), and UW-CSE (bottom).

bers of ground atoms and clauses are about 1 million and 3.2 billion, respectively; in Cora, 0.9 million and 1.8 billion.

Link prediction is the problem of identifying binary relations among objects. This has numerous applications in social network analysis, biology, etc. We used the UW-CSE dataset and the hand-coded KB from Richardson & Domingos (2006). The published dataset contains information about 402 faculty and students in UW-CSE. The goal is to compute marginal probabilities of advising relations. The highest clause arity is three, for clauses such as $\neg\texttt{TempAdvisedBy}(\texttt{s},\texttt{p}) \vee \neg\texttt{AdvisedBy}(\texttt{s},\texttt{q})$ (which says that a student cannot have both temporary and formal advisors at the same time, a true statement at UW-CSE). The total numbers of ground atoms and clauses are about 0.3 million and 0.1 billion, respectively.

Methodology

We implemented Lazy-MC-SAT as an extension of the Alchemy system (Kok *et al.* 2007), and used the existing MC-SAT implementation. We used the default values in Alchemy for all parameters. Ideally, we would run MC-SAT and Lazy-MC-SAT until convergence. However, it is very difficult to diagnose convergence. Instead, we ran the algorithms for both 1000 steps and 10,000 steps, and observed similar solution quality. We report the results for 10,000 steps. Each experiment was conducted on a cluster node with 3 GB of RAM and two processors running at 2.8 GHz.

Since the two algorithms produce the same results, we do

Table 1: Polynomial exponents and extrapolation.

Memory	CiteSeer	Cora	UW-CSE
Eager	2.51	2.74	2.60
Lazy	0.85	1.66	0.99
Savings	$2.0 \cdot 10^8$	5041	31
Time	CiteSeer	Cora	UW-CSE
Eager	2.84	2.88	2.79
Lazy	1.00	1.89	1.98
Speed-Up	$1.9 \cdot 10^9$	5072	26

not report solution quality. In all three datasets, we varied the number of objects from 50 up to the full size in increments of 50, generated three random subsets for each number of objects except the full size,[4] and report the averages. In UW-CSE, we excluded a few rules with existential quantifiers, for Alchemy will convert them into large disjunctions. This is not a major limitation of lazy inference, for most statistical relational representations do not have existentials, and we can handle existentials lazily by partially grounding the clauses (i.e, we ground a partial disjunction using only active atoms, and extend it when more atoms are activated). We will implement this extension in the future.

Results

The results are shown in Figures 1 and 2. The memory and time savings obtained by Lazy-MC-SAT grow rapidly with the number of objects. Lazy-MC-SAT is able to handle all full datasets, whereas MC-SAT runs out of memory with 250 objects in CiteSeer and Cora, and with 200 in UW-CSE. To extrapolate beyond them, we fitted the function $f(x) = ax^b$ to all curves, and extrapolated it to the sizes of the complete CiteSeer database (over 4.5 million), the complete Cora database (over 50,000) and UW-CSE (402). Table 1 shows the exponent b's and the estimated factors of memory and time reduction. The memory and time used by Lazy-MC-SAT grow linearly in CiteSeer, and sub-quadratically in Cora and UW-CSE, whereas that by MC-SAT grow closer to cubically. Lazy-MC-SAT's speed-up results mainly from the lazy computation of M-memberships. It also grounds many fewer atoms/clauses, and makes fewer flips. For example, for a Cora dataset with 200 citations, MC-SAT spends 17 hours in total, among which 14 hours are spent in computing memberships in M, 6 minutes in grounding atoms/clauses (8.1 million groundings), and 2.5 hours in SampleSAT (1.8 million flips). In comparison, Lazy-MC-SAT spends 45 minutes in total, among which 6 minutes in computing memberships in M, 7 seconds in grounding atoms/clauses (0.2 million groundings), and 38 minutes in SampleSAT (0.4 million flips). Overall, the reduction of memory and time by Lazy-MC-SAT is very large, e.g., for the complete CiteSeer database, Lazy-MC-SAT reduces memory by eight orders of magnitude, and reduces time by nine orders of magnitude.

[4] In all domains, we preserved the natural clusters (corefering citations in CiteSeer and Cora, people in the same research area in UW-CSE) as much as possible.

Conclusion

We proposed a general method for lazy inference and applied it to various algorithms. Experiments on real-world domains show that Lazy-MC-SAT, one of the resulting algorithms, reduces memory and time by orders of magnitude.

Directions for future work include applying lazy inference to other algorithms, combining lazy inference with lifted inference, applying it to learning, etc.

Acknowledgements

We thank the anonymous reviewers for their comments. This research was funded by DARPA contracts NBCH-D030010/02-000225, FA8750-07-D-0185, and HR0011-07-C-0060, DARPA grant FA8750-05-2-0283, NSF grant IIS-0534881, and ONR grant N-00014-05-1-0313. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of DARPA, NSF, ONR, or the U.S. Government.

References

Bakir, G.; Hofmann, T.; Schölkopf, B.; Smola, A.; Taskar, B. and Vishwanathan, S. (eds.) 2007. *Predicting Structured Data*. MIT Press.

Braz, R.; Amir, E. and Roth, D. 2006. MPE and partial inversion in lifted probabilistic variable elimination. In *Proc. AAAI-06*.

Damien, P.; Wakefield, J. and Walker, S. 1999. Gibbs sampling for Bayesian non-conjugate and hierarchical models by auxiliary variables. *Journal of the Royal Statistical Society* B 61(2).

Davis, M.; Logemann, G. and Loveland, D. 1962. A machine program for theorem proving. *Communications of the ACM* 5(7).

Genesereth, M. and Nilsson, N. 1987. *Logical Foundations of Artificial Intelligence*. Morgan Kaufmann.

Getoor, L.; Friedman, N.; Koller, D. and Taskar, B. 2001. Learning probabilistic models of relational structure. *In Proc. ICML-01*.

Getoor, L. & Taskar, B., eds. 2007. *Introduction to Statistical Relational Learning*. MIT Press.

Kautz, H.; Selman, B.; and Jiang, Y. 1997. A general stochastic approach to solving problems with hard and soft constraints. In *The Satisfiability Problem: Theory and Applications*. AMS.

Kok, S.; Singla, P.; Richardson, M.; Domingos, P.; Sumner, M. and Poon, H. 2007. The Alchemy system for statistical relational AI. http://alchemy.cs.washington.edu/.

Milch, B. and Russell, S. 2006. General-purpose MCMC inference over relational structures. In *Proc. UAI-06*.

Poon, H. and Domingos, P. 2006. Sound and efficient inference with probabilistic and deterministic dependencies. In *Proc. AAAI-06*.

Poon, H. and Domingos, P. 2007. Joint inference in information extraction. In *Proc. AAAI-07*.

Richardson, M. and Domingos, P. 2006. Markov logic networks. *Machine Learning* 62(1-2).

Selman, B.; Kautz, H.; and Cohen, B. 1996. Local search strategies for satisfiability testing. In *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge*. AMS.

Singla, P. and Domingos, P. 2006. Memory-efficient inference in relational domains. In *Proc. AAAI-06*.

Wei, W.; Erenrich, J. and Selman, B. 2004. Towards efficient sampling: Exploiting random walk strategies. In *Proc. AAAI-04*.