Memory-Efficient Inference in Relational Domains

Parag Singla **Pedro Domingos**
Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195-2350, U.S.A.
{*parag, pedrod*}*@cs.washington.edu*

Abstract

Propositionalization of a first-order theory followed by satisfiability testing has proved to be a remarkably efficient approach to inference in relational domains such as planning (Kautz & Selman 1996) and verification (Jackson 2000). More recently, weighted satisfiability solvers have been used successfully for MPE inference in statistical relational learners (Singla & Domingos 2005). However, fully instantiating a finite first-order theory requires memory on the order of the number of constants raised to the arity of the clauses, which significantly limits the size of domains it can be applied to. In this paper we propose LazySAT, a variation of the WalkSAT solver that avoids this blowup by taking advantage of the extreme sparseness that is typical of relational domains (i.e., only a small fraction of ground atoms are true, and most clauses are trivially satisfied). Experiments on entity resolution and planning problems show that LazySAT reduces memory usage by orders of magnitude compared to WalkSAT, while taking comparable time to run and producing the same solutions.

Introduction

Most problems of interest to AI are relational in nature: they involve explicit consideration of multiple objects and relations among them, not just unpacked propositions. Inference in these domains has traditionally been performed using lifted techniques like resolution. Unfortunately, this type of inference does not, in general, scale well for fully automated theorem proving. In the last decade, however, propositionalization followed by satisfiability testing has emerged as a highly efficient alternative. While the basic idea is not new, it has been given fresh impetus by the development of very fast solvers like WalkSAT (Selman, Kautz, & Cohen 1996) and zChaff (Moskewicz *et al.* 2001). Despite its successes, the applicability of this approach to complex relational problems is still severely limited by at least one key factor: the exponential memory cost of propositionalization. To apply a satisfiability solver, we need to create a Boolean variable for every possible grounding of every predicate in the domain, and a propositional clause for every grounding of every first-order clause. If n is the number of objects in the domain and r is the highest clause arity, this requires memory on the order of n^r. Clearly, even domains of moderate size are potentially problematic, and large ones are completely infeasible.

The emergence in the last few years of the field of statistical relational learning (SRL) has added new urgency to this problem (Dietterich, Getoor, & Murphy 2004). SRL supports probabilistic inference over relational domains, and often requires it to be repeatedly performed during learning. While some promising steps have been taken in developing lifted inference techniques for this setting, they are still quite far from being applicable to problems of realistic size (Poole 2003; de S. Braz, Amir, & Roth 2005). Recently, Richardson and Domingos (2006) and Singla and Domingos (2005) showed how weighted satisfiability solvers can be used for MPE/MAP inference in a broad class of SRL models (those expressible in Markov logic). However, this approach is limited by the same memory explosion as more traditional satisfiability applications.

This paper introduces an approach to satisfiability testing that largely overcomes this problem. It is based on a property that seems to characterize almost all relational domains: their extreme sparseness. The vast majority of predicate groundings are false, and as a result the vast majority of clauses (all clauses that have at least one false precondition) are trivially satisfied. For example, in the domain of scientific research, most groundings of the predicate $\texttt{Author}(\texttt{person}, \texttt{paper})$ are false, and most groundings of the clause $\texttt{Author}(\texttt{person1}, \texttt{paper}) \wedge \texttt{Author}(\texttt{person2}, \texttt{paper}) \Rightarrow \texttt{Coauthor}(\texttt{person1}, \texttt{person2})$ are satisfied. Our approach is embodied in LazySAT, a variant of WalkSAT that reduces memory while producing the same results. In LazySAT, the memory cost does not scale with the number of possible clause groundings, but only with the number of groundings that are potentially unsatisfied at some point in the search. Clauses that are never considered for flipping literals are never grounded. Experiments on entity resolution and planning problems show that this can yield very large memory reductions, and these reductions increase with domain size. For domains whose full instantiations fit in memory, running time is comparable; as problems become larger, full instantiation for WalkSAT becomes impossible.

We begin by briefly reviewing some necessary background. We then describe LazySAT in detail, and report on our experiments.

Relational Inference Using Satisfiability

Satisfiability

A *knowledge base (KB)* in propositional logic is a set of formulas over Boolean variables. Every KB can be converted to *conjunctive normal form (CNF)*: a conjunction of clauses, each clause being a disjunction of literals, each literal being a variable or its negation. *Satisfiability* is the problem of finding an assignment of truth values to the variables that satisfies all the clauses (i.e., makes them true) or determining that none exists. It is the prototypical NP-complete problem. The last decade and a half has seen tremendous progress in the development of highly efficient satisfiability solvers. One of the most efficient approaches is stochastic local search, exemplified by the WalkSAT solver (Selman, Kautz, & Cohen 1996). Starting from a random initial state, WalkSAT repeatedly flips (changes the truth value of) a variable in a random unsatisfied clause. With probability p, WalkSAT chooses the variable that minimizes a cost function (such as the number of currently satisfied clauses that become unsatisfied, or the total number of unsatisfied clauses; see Gent & Walsh (1993) for discussion), and with probability $1 - p$ it chooses a random variable. WalkSAT keeps going even if it finds a local maximum, and after n flips restarts from a new random state. The whole procedure is repeated m times. WalkSAT can solve random problems with hundreds of thousands of variables in a fraction of a second, and hard ones in minutes. However, it cannot distinguish between an unsatisfiable CNF and one that takes too long to solve.

The MaxWalkSAT (Kautz, Selman, & Jiang 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. (Systematic solvers have also been extended to weighted satisfiability, but tend to work poorly.) Park (2005) showed how the problem of finding the most likely state of a Bayesian network given some evidence can be efficiently solved by reduction to weighted satisfiability. 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. Algorithm 1 gives pseudo-code for WalkSAT. DeltaCost(*v*) computes the change in the sum of weights of unsatisfied clauses that results from flipping variable *v* in the current solution. Uniform(0,1) returns a uniform deviate from the interval $[0, 1]$.

Algorithm 1 WalkSAT(*weighted_clauses*, *max_flips*, *max_tries*, *target*, *p*)

```
vars ← variables in weighted_clauses
for i ← 1 to max_tries do
    soln ← a random truth assignment to vars
    cost ← sum of weights of unsatisfied clauses in soln
    for i ← 1 to max_flips do
        if cost ≤ target then
            return "Success, solution is", soln
        end if
        c ← a randomly chosen unsatisfied clause
        if Uniform(0,1) < p then
            v_f ← a randomly chosen variable from c
        else
            for each variable v in c do
                compute DeltaCost(v)
            end for
            v_f ← v with lowest DeltaCost(v)
        end if
        soln ← soln with v_f flipped
        cost ← cost + DeltaCost(v_f)
    end for
end for
return "Failure, best assignment is", best soln found
```

Relational Domains and Propositionalization

First-order logic (FOL) allows us to explicity represent a domain's relational structure (Genesereth & Nilsson 1987). Objects are represented by constants, and the relations among them by predicates. This paper focuses on function-free FOL with the domain closure assumption (i.e., the only objects in the domain are those represented by the constants). A predicate or formula is *grounded* by replacing all its variables by constants. *Propositionalization* is the process of replacing a first-order KB by an equivalent propositional one. In finite domains, this can be done by replacing each universally (existentially) quantified formula with a conjunction (disjunction) of all its groundings. A first-order KB is satisfiable iff the equivalent propositional KB is satisfiable. Hence, inference over a first-order KB can be performed by propositionalization followed by satisfiability testing.

Learning statistical models of relational domains is currently a very active research area (Dietterich, Getoor, & Murphy 2004). One of the most powerful representations used in this area is Markov logic (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 predicate 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. Finding the most probable state of a grounded MLN given some evidence is thus an instance of weighted satisfiability, and the LazySAT algorithm we develop in this paper can be used to scale MLN inference to much larger domains than standard WalkSAT. Although we do not pursue it in this paper, the ideas in LazySAT are also directly applicable to scaling up the computation of marginal and conditional probabilities in MLNs using Markov chain Monte Carlo algorithms.

Memory-Efficient Inference

The LazySAT algorithm reduces the memory required by satisfiability testing in relational domains by taking advantage of their sparseness. Because the great majority of

ground atoms are false, most clauses are satisfied throughout the search, and never need to be considered. LazySAT grounds clauses lazily, at each step in the search adding only the clauses that could become unsatisfied. In contrast, WalkSAT grounds all possible clauses at the outset, consuming time and memory exponential in their arity.

Algorithm 2 gives pseudo-code for LazySAT, highlighting the places where it differs from WalkSAT. LazySAT inputs an MLN (or a pure first-order KB, in which case all clauses are assigned weight 1) and a database (DB). A database is a set of ground atoms. (For example, in planning problems the database is the set of ground atoms describing the initial and goal states. In probabilistic inference, the database is the evidence we condition on.) An *evidence atom* is either a ground atom in the database, or a ground atom that is false by the closed world assumption (i.e., it is a grounding of an evidence predicate, and does not appear in the database). The truth values of evidence atoms are fixed throughout the search, and ground clauses are simplified by removing the evidence atoms. LazySAT maintains a set of *active atoms* and a set of *active clauses*. A clause is active if it can be made unsatisfied by flipping zero or more of its active atoms. (Thus, by definition, an unsatisfied clause is always active.) An atom is active if it is in the initial set of active atoms, or if it was flipped at some point in the search. The initial active atoms are all those appearing in clauses that are unsatisfied if only the atoms in the database are true, and all others are false. We use dynamic arrays to store the active clauses and the atoms in them. The unsatisfied clauses are obtained by simply going through each possible grounding of all the first-order clauses and materializing the groundings that are unsatisfied; search is pruned as soon the partial grounding of a clause is satisfied. Given the initial active atoms, the definition of active clause requires that some clauses become active, and these are found using a similar process (with the difference that, instead of checking whether a ground clause is unsatisfied, we check whether it should be active).[1] Each run of LazySAT is initialized by assigning random truth values to the active atoms. This differs from WalkSAT, which assigns random values to all atoms. However, the LazySAT initialization is a valid WalkSAT initialization, and we have verified experimentally that the two give very similar results. Given the same initialization, the two algorithms will produce exactly the same results.

At each step in the search, the variable that is flipped is activated, as are any clauses that by definition should become active as a result. When evaluating the effect on cost of flipping a variable v, if v is active then all of the relevant clauses are already active, and DeltaCost(v) can be computed as in WalkSAT. If v is inactive, DeltaCost(v) needs to be computed using the knowledge base. (In principle, repeated calculation of this kind can be avoided by activating a variable (and its clauses) when it is first encountered, but we found the time savings from this to be outweighed by the added memory cost.) This is done by retrieving from the KB all first-order clauses containing the predicate that v is a grounding of, and grounding each such clause with the constants in v and all possible groundings of the remaining variables. As before, we prune search as soon as a partial grounding is satisfied, and add the appropriate multiple of the clause weight to DeltaCost(v). (A similar process is used to activate clauses.) While this process is costlier than using pre-grounded clauses, it is amortized over many tests of active variables. In typical satisfiability problems, a small core of "problem" clauses is repeatedly tested, and when this is the case LazySAT will be quite efficient.

Algorithm 2 LazySAT(*weighted_KB*, *DB*, *max_flips*, *max_tries*, *target*, *p*)

```
for i ← 1 to max_tries do
    active_atoms ← atoms in clauses not satisfied by DB
    active_clauses ← clauses activated by active_atoms
    soln ← a random truth assignment to active_atoms
    cost ← sum of weights of unsatisfied clauses in soln
    for i ← 1 to max_flips do
        if cost ≤ target then
            return "Success, solution is", soln
        end if
        c ← a randomly chosen unsatisfied clause
        if Uniform(0,1) < p then
            v_f ← a randomly chosen variable from c
        else
            for each variable v in c do
                compute DeltaCost(v), using weighted_KB if
                v ∉ active_atoms
            end for
            v_f ← v with lowest DeltaCost(v)
        end if
        if v_f ∉ active_atoms then
            add v_f to active_atoms
            add clauses activated by v_f to active_clauses
        end if
        soln ← soln with v_f flipped
        cost ← cost + DeltaCost(v_f)
    end for
end for
return "Failure, best assignment is", best soln found
```

At each step, LazySAT flips the same variable that WalkSAT would, and hence the result of the search is the same. The memory cost of LazySAT is on the order of the maximum number of clauses active at the end of a run of flips. (The memory required to store the active atoms is dominated by the memory required to store the active clauses, since each active atom appears in at least one active clause.) In the current version of LazySAT, clauses and atoms are never deactivated, but this could easily be changed to save memory without affecting the output by periodically reinitializing the active sets (using the current truth values of all non-evidence atoms, instead of all false). In our experiments this was not necessary, but it could be quite useful in very long searches.

[1] Although for simplicity the pseudo-code does not show this, the initial set of active clauses and active atoms can be saved and reused for each restart, saving time.

Experiments

We performed experiments on two entity resolution domains and a planning domain to compare the memory usage and running time of LazySAT and WalkSAT. We implemented LazySAT as an extension of the Alchemy system (Kok *et al.* 2005), and used Kautz et al.'s (1997) implementation of MaxWalkSAT, included in Alchemy. When propositionalizing a problem for WalkSAT, we did not ground clauses that are always true given the evidence, and this saved a significant amount of memory. Since the two algorithms can be guaranteed to produce the same results by using the same random initialization for both, we do not report solution quality. In all the experiments we ran WalkSAT and LazySAT for a million flips, with no restarts. The experiments were run on a cluster of nodes, each node having 3.46 GB of RAM and two processors running at 3 GHz. All results reported are averages over five random problem instances.

Entity Resolution

In many domains, the entities of interest are not uniquely identified, and we need to determine which observations correspond to the same entity. For example, when merging databases we need to determine which records are duplicates. This problem is of crucial importance to many large scientific projects, businesses, and government agencies, and has received increasing attention in the AI community in recent years. We used two publicly available citation databases in our experiments: McCallum's Cora database as segmented by Bilenko and Mooney (2003) (available at http://www.cs.utexas.edu/users/ml/riddle/data/cora.tar.gz); and BibServ.org, which combines CiteSeer, DBLP, and user-donated databases. Cora contains 1295 citations, extracted from the original Cora database of over 50,000, and BibServ contains approximately half a million citations. We used the user-donated subset of BibServ, with 21,805 citations. The inference task was to de-duplicate citations, authors and venues (i.e., to determine which pairs of citations refer to the same underlying paper, and similarly for author fields and venue fields). We used the Markov logic network constructed by Singla and Domingos (2005), ignoring clauses with negative weight (which we are currently extending MaxWalkSAT to handle). This contains 33 first-order clauses stating regularities such as: if two fields have high TF-IDF similarity, they are (probably) the same; if two records are the same, their fields are the same, and vice-versa; etc. Crucially, we added the transitivity rule with a very high weight: $\forall x, y, z\ x = y \wedge y = z \Rightarrow x = z$. This rule is used in an *ad hoc* way in most entity resolution systems, and greatly complicates inference. The highest clause arity, after conditioning on evidence, was three. We learned clause weights on Cora using Singla and Domingos' (2005) algorithm, and used these weights on both Cora and BibServ. (We could not learn weights on BibServ because the data is not labeled.)

We used the cleaned version of Cora as described in Singla and Domingos (2005). We varied the number of records from 50 to 500 in intervals of 50, generating five random subsets of the data for each number of records. We ensured that each real cluster in the data was either completely included in a subset or completely excluded, with the exception of the last one to be added, which had to be truncated to ensure the required number of records. Figure 1 (left) shows how the total number of clauses grounded by LazySAT and WalkSAT varies with the number of records. The RAM usage in bytes correlates closely with the number of groundings (e.g., for 250 records WalkSAT uses 2.1 GB, and LazySAT 288 MB). The memory reduction obtained by LazySAT increases rapidly with the number of records. At 250 records, it is about an order of magnitude. Beyond this point, WalkSAT runs out of memory. To extrapolate beyond it, we fitted the function $f(x) = ax^b$ to both curves, obtaining $a = 1.19$, $b = 2.97$ (with $R^2 = 0.99$) for WalkSAT and $a = 6.02$, $b = 2.34$ (with $R^2 = 0.98$) for LazySAT. Using these, on the full Cora database LazySAT would reduce memory usage by a factor of over 300.

Figure 1 (right) compares the speed of the two algorithms in average flips per second (i.e., total number of flips over total running time). LazySAT is somewhat faster than WalkSAT for low numbers of records, but by 250 records this advantage has disappeared. Recall that total running time has two main components: initialization, where LazySAT has the advantage, and variable flipping, where WalkSAT does. The relative contribution of the two depends on the problem size and total number of flips. WalkSAT flips variables at a constant rate throughout the search. LazySAT initially flips them slower, on average, but as more clauses become active its flipping rate increases and converges to WalkSAT's.

On BibServ, we formed random subsets of size 50 to 500 records using the same approach as in Cora, except that instead of using the (unknown) real clusters we formed clusters using the canopy approach of McCallum et al. (2000). Figure 2 (left) shows the number of clause groundings as a function of the number of records. RAM usage behaved similarly, being 1.9 GB for WalkSAT and 78 MB for LazySAT at 250 records. As before, WalkSAT runs out of memory at 250 records. LazySAT's memory requirements increase at a much lower rate on this database, giving it a very large advantage over WalkSAT. Fitting the function $f(x) = ax^b$ to the two curves, we obtained $a = 1.02$, $b = 2.98$ (with $R^2 = 0.99$) for WalkSAT and $a = 28.15$, $b = 1.75$ (with $R^2 = 0.98$) for LazySAT. Extrapolating these, on the full BibServ database LazySAT would reduce memory compared to WalkSAT by a factor of over 400,000. The greater advantage of LazySAT on BibServ is directly attributable to its larger size and consequent greater sparseness, and we expect the advantage to be even greater for larger databases. (To see why larger size leads to greater sparseness, consider for example the predicate $\texttt{Author}(\texttt{person}, \texttt{paper})$. When the number of papers and authors increases, its number of true groundings increases only approximately proportionally to the number of papers, while the number of possible groundings increases proportionally to its product by the number of authors. On a database with thousands of authors, the resulting difference can be quite large.)

Figure 2 (right) compares the speeds of the two algorithms. They are similar, with LazySAT being slightly faster

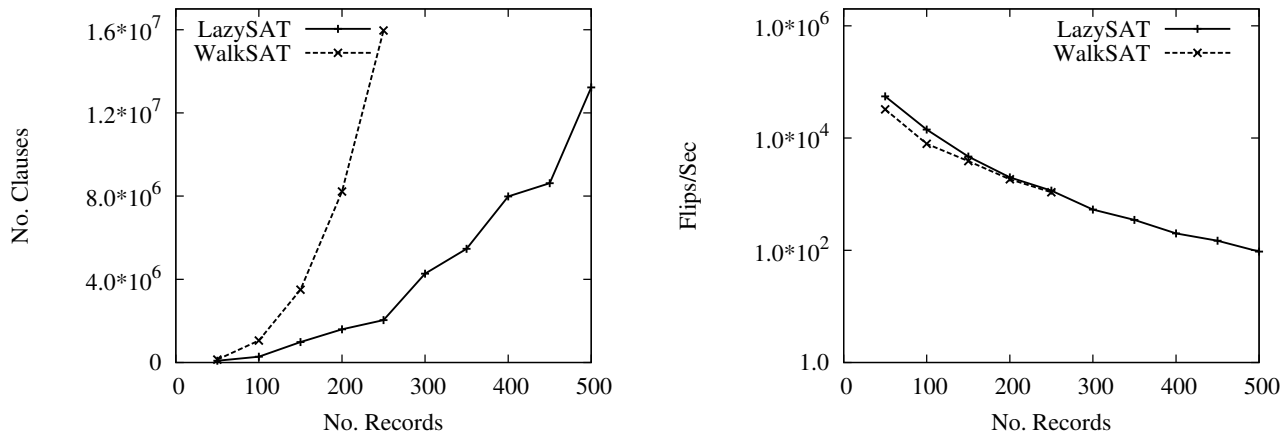

Figure 1: Experimental results on Cora: memory (left) and speed (right) as a function of the number of records.

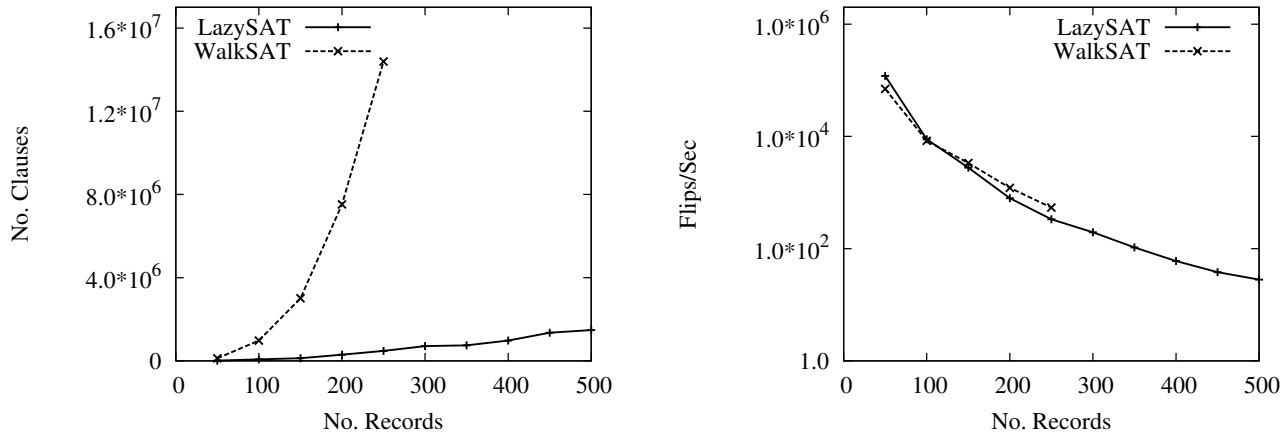

Figure 2: Experimental results on BibServ: memory (left) and speed (right) as a function of the number of records.

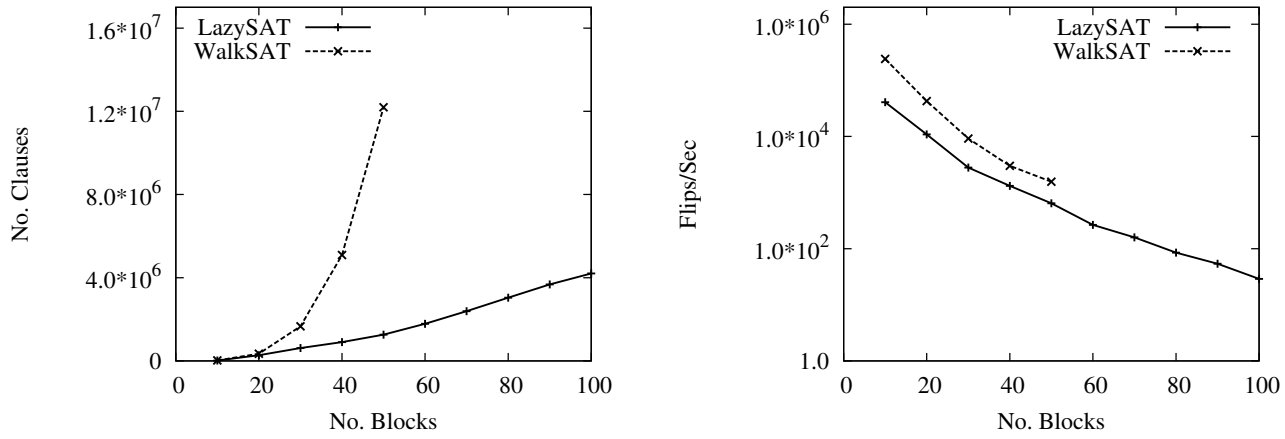

Figure 3: Experimental results on the blocks world domain: memory (left) and speed (right) as a function of the number of blocks.

below 100 objects and WalkSAT slightly faster above.

Planning

To compare LazySAT and WalkSAT on a pure (unweighted) satisfiability problem, we experimented on the classical blocks world planning domain, where the goal is to find a sequence of block moves that transforms the initial stacking of blocks into the goal stacking. Kautz & Selman (1996) showed how to encode planning problems as instances of satisfiability, by writing formulas that specify action definitions, frame axioms, etc., and using a database of ground atoms to specify the initial state and goal state. We used the formulas they wrote for the blocks world domain (publicly available at http://www.cs.washington.edu/homes/kautz/satplan/blackbox/satplan_dist_2001.tar.Z). The maximum clause arity was four. All the clauses were assigned unit weight. Given a number of blocks, we generated random problem instances as follows. We set the number of stacks in both the initial and goal state to the square root of the number of blocks (truncated to the nearest integer). The stacks were then populated by randomly assigning blocks to them. The number of allowed moves was liberally set to the length of the trivial solution (unstacking and restacking all the blocks). We added rules to allow null moves, making it possible for the algorithms to find shorter plans. We varied the number of blocks from 10 to 100 in increments of 10, and generated five random instances for each. Figure 3 (left) shows how the number of clauses grounded by the two algorithms varies with the number of blocks. LazySAT once again obtains very large reductions. RAM usage closely paralleled the number of clauses, being 1.6 GB for WalkSAT and 203 MB for LazySAT at 50 blocks. WalkSAT is only able to go up 50 blocks, and LazySAT's gain at this size is about an order of magnitude. Fitting the function $f(x) = ax^b$ to the two curves, we obtained $a = 2.79$, $b = 3.90$ (with $R^2 = 0.99$) for WalkSAT, and $a = 1869.58$, $b = 1.68$ (with $R^2 = 0.99$) for LazySAT. The difference in the asymptotic behavior is even more prominent here than in the entity resolution domains.

Figure 3 (right) compares the speed of the two algorithms. WalkSAT is somewhat faster, but the difference decreases gradually with the number of blocks.

In all domains, LazySAT reduced memory by an order of magnitude or more at the point that WalkSAT exceeded available RAM. Most importantly, LazySAT makes it feasible to solve much larger problems than before.

Conclusion

Satisfiability testing is very effective for inference in relational domains, but is limited by the exponential memory cost of propositionalization. The LazySAT algorithm overcomes this problem by exploiting the sparseness of relational domains. Experiments on entity resolution and planning problems show that it greatly reduces memory requirements compared to WalkSAT, without sacrificing speed or solution quality.

Directions for future work include applying the ideas in LazySAT to other SAT solvers and to MCMC, combining it with Richardson and Domingos' (2006) KBMC algorithm for bounding the ground network, and extending it to degrade gracefully when the number of weighted clauses exceeds the available memory.

Acknowledgments

We are grateful to Henry Kautz for helpful discussions. This research was partly supported by DARPA grant FA8750-05-2-0283 (managed by AFRL), DARPA contract NBCH-D030010, NSF grant IIS-0534881, and ONR grant N00014-05-1-0313. 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 United States Government.

References

Bilenko, M., and Mooney, R. 2003. Adaptive duplicate detection using learnable string similarity measures. In *Proc. KDD-03*, 39–48.

de S. Braz, R.; Amir, E.; and Roth, D. 2005. Lifted first-order probabilistic inference. In *Proc. IJCAI-05*, 1319–1324.

Dietterich, T.; Getoor, L.; and Murphy, K., eds. 2004. *Proc. ICML-04 Workshop on SRL and its Connections to Other Fields*.

Genesereth, M. R., and Nilsson, N. J. 1987. *Logical Foundations of Artificial Intelligence*. San Mateo, CA: Morgan Kaufmann.

Gent, I. P., and Walsh, T. 1993. Towards an understanding of hill-climbing procedures for SAT. In *Proc. AAAI-93*, 28–33.

Jackson, D. 2000. Automating first-order relational logic. In *Proc. ACM SIGSOFT Symposium on Foundations of Software Engineering*, 130–139.

Kautz, H., and Selman, B. 1996. Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proc. AAAI-96*, 1194–1201.

Kautz, H.; Selman, B.; and Jiang, Y. 1997. A general stochastic approach to solving problems with hard and soft constraints. In Gu, D.; Du, J.; and Pardalos, P., eds., *The Satisfiability Problem: Theory and Applications*. New York, NY: AMS. 573–586.

Kok, S.; Singla, P.; Richardson, M.; and Domingos, P. 2005. The Alchemy system for statistical relational AI. http://www.cs.washington.edu/ai/alchemy.

McCallum, A.; Nigam, K.; and Ungar, L. 2000. Efficient clustering of high-dimensional data sets with application to reference matching. In *Proc. KDD-00*, 169–178.

Moskewicz, M.; Madigan, C.; Zhao, Y.; Zhang, L.; and Malik, S. 2001. Chaff: Engineering an efficient SAT solver. In *Proc. DAC-01*, 530–535.

Park, J. D. 2005. Using weighted MAX-SAT engines to solve MPE. In *Proc. AAAI-05*, 682–687.

Poole, D. 2003. First-order probabilistic inference. In *Proc. IJCAI-03*, 985–991.

Richardson, M., and Domingos, P. 2006. Markov logic networks. *Machine Learning* 62:107–136.

Selman, B.; Kautz, H.; and Cohen, B. 1996. Local search strategies for satisfiability testing. In Johnson, D. S., and Trick, M. A., eds., *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge*. Washington, DC: AMS. 521–532.

Singla, P., and Domingos, P. 2005. Discriminative training of Markov Logic Networks. In *Proc. AAAI-05*, 868–873.