A Toolset for Propositional Probabilistic Logic

Paulo S. de S. Andrade[1] **, José C. F. da Rocha** [2] **,**
Danillo P. Couto [3] **, André da Costa Teves**[3] **, Fabio G. Cozman** [3]

[1] Instituto de Matemática e Estatística da Universidade de São Paulo
Cidade Universitária – São Paulo, SP – Brazil

[2]Laboratório de Engenharia de Software – Departamento de Informática
Universidade Estadual de Ponta Grossa (UEPG) – Ponta Grossa, PR – Brazil

[3]Escola Politécnica da Universidade de São Paulo
Cidade Universitária – São Paulo, SP – Brazil

p_s_s_a@yahoo.com.br, jrocha@uepg.br,
danillo.couto@poli.usp.br, andre.teves@poli.usp.br,
fgcozman@usp.br

Abstract. *Research in knowledge representation has explored several formalisms with logical and probabilistic features. In this paper we present three tools that support the development of probabilistic logic knowledge bases. The first tool is a generator of probabilistic propositional satisfiability (PSAT) problems that is useful in testing algorithms. The second tool is an consistency checker for PSAT that handles probabilities over CNFs. The third tool is a convenient editor for PSAT problems that simplifies the construction of knowledge bases.*

Resumo. *A pesquisa em representação de conhecimento têm explorado diversos formalismos envolvendo aspectos lógicos e probabilísticos. Neste artigo apresentamos três ferramentas que dão suporte ao desenvolvimento de bases de conhecimento lógico-probabilísticas. A primeira ferramenta é um gerador de problemas de satisfatibilidade probabilística proposicional (PSAT), que é útil para testar algoritmos. A segunda ferramenta é um verificador de consistência em PSAT que lida com probabilidades sobre CNFs. A terceira ferramenta é um editor de problemas PSAT que simplifica a construção de bases de conhecimento.*

1. Introduction

Research in probabilistic logic seeks to define formal languages that can handle logical sentences and probabilistic assessments [Nilson 1986b]. If successful, this effort can lead to a major computational framework for knowledge representation and reasoning. Logical languages are ubiquitous in AI, and probabilistic inference has had a central role in AI systems; thus the union of these methodologies holds great potential.

Research efforts in probabilistic logic date from Boole's work [Hailperin 1986]; the topic has resurfaced often [Andersen and Hooker 1994, Hansen et al. 1991, Nilson 1986b, Richardson and Domingos 2006], but despite this lasting interest in the subject, it is not easy at all to find software packages that support probabilistic

logic inference. In fact, this is hard even for the *propositional probabilistic satisfiability (PSAT)* problem. In our work on knowledge representation, we have noticed a particularly troublesome lack of computational support for PSAT in three areas:

- When testing algorithms, it is important to randomly generate examples of probabilistic logic knowledge bases; however there is no discussion of this topic in the literature, and no algorithms are available.
- There are no available implementations of the classic column generation algorithm for PSAT.
- There is no simple and convenient editor for PSAT problems.

In this paper we present tools that focus on each of these topics. We present a PSAT generator (Section 3), a consistency checker (Section 4), and an editor (Section 5). The consistency checker described in Section 4 contains an enhanced implementation of Hansen et al.'s algorithm [Hansen et al. 1991]; we enhance that classic algorithm so that it handles probabilities over conjunctive normal forms in a highly predictable and efficient manner. As the editor is capable of handling assessments over arbitrary sentences, the result is a system that can deal with general probabilistic logic bases.

2. Background

Propositional logic, given its relatively simple sintax and semantics, offers an useful starting point for knowledge representation [Russell and Norvig 1995]. The basic syntatic element in this logic is the concept of *propositional variable* or *atomic formula* that can assume one of two values, *true* or *false*. Atomic formulas are denoted by lower case letters as p, q, r and so on. A *literal* is either an atomic formula or its negation. Literals are indicated by indexed upper case letters as A_1, A_2, and so on. In this text compound formulas are indicated by the greek letters ϕ, ψ and θ with or without indexes. A disjunction of literals is a *clause*. A *Conjunctive Normal Form (CNF)* is a conjunction of clauses, denoted by $C_1 \wedge \ldots C_r$ where C_i is a clause.

The semantics of any expression in propositional logic depends on a mapping that establishes a correspondence between the variables in the formula to facts in a target domain. A *truth assignment* is a vector assigning either value true or false to each propositional variable of an expression (these assignments are often called *possible worlds* [Nilson 1986a]). If we have n propositional variables, there are 2^n truth assignments. A conjunctive formula is true if all its component formulas are true, otherwise it is false. A disjunctive formula is false only if all its component formulas are false, otherwise it is true. A negative formula is true (false) if its component formula is false (true).

A formula ϕ is *satisfiable* if it is true in some possible world ω; then ω is a *model* for ϕ ($\mathcal{M}(\phi, \omega)$). If ϕ has no model it is unsatisfiable. A formula ψ *entails* a formula θ ($\psi \models \theta$) if every model for ψ is a model for θ. An *inference* $\psi \vdash \theta$ determines whether a premise ψ entails a conclusion θ. The *satisfiability problem (SAT)* is: given a CNF ψ with m clauses $C_1, \ldots C_m$, is ψ satisfiable? This question has a strong relationship to logic entailment and logic inference because $\psi \models \theta$ iff $\psi \wedge \neg\theta$ is unsatisfiable. That is, it is possible to approach inference as SAT problem. A particular kind of SAT problem is the *k-SAT*, a SAT which every clause has k literals.

An important limitation of propositional logic, from a point of view of knowledge representation, is its inability to deal with uncertainty. As stressed by Neapolitan

[Neapolitan 1990]: "We also must acknowledge that in some cases the truth of certain premisses may be suggestive of the truth of a conclusion, but not imply it conclusively." To overcome this difficulty, propositional probabilistic logic extends propositional logic by attaching probability assessments to formulas. In this context, $P(\phi)$ denotes the measure of all assignments satisfying ϕ; that is, $P(\phi) = \sum_{\omega:\mathcal{M}(\phi,\omega)} P(w)$.

The counterpart of SAT in probabilistic logic is the probabilistic satisfiability problem (*PSAT*) [Georgakapoulos et al. 1988]. The PSAT structure is similar to SAT but it poses the following question: is there a probability distribution satifying a set of m assessments that assign probability interval to $P(\phi_i)$ for a set of formulas $\{\phi_i\}_{i=1}^{m}$ over n propositions.

If the assessments are such that no probability distribution p over truth assignments can be specified, the assessments are *inconsistent*. The *consistency problem* of probabilistic satisfiability is: given a set of assessments, determine whether they are inconsistent or not. The *inference problem* of probabilistic satisfiability is: given a set of assessments and a formula ϕ, obtain the infimum of $P(\phi)$ — that is, the infimum value α such that the constraint $P(\phi) = \alpha$ and the assessments are consistent. The infimum is denoted by $\underline{P}(\phi)$ and called the *lower probability* of ϕ. This infimum is attained because probabilitisc assessments on the logical formulas and constraints $\sum_{\omega} P(\omega) = 1$, $P(\omega) \geq 0$ define a bounded polyhedron in the space of probability measures over truth assignments. Like SAT, PSAT is a NP-complete [Georgakapoulos et al. 1988].

3. Generating PSATs

The test of PSAT algorithms requires the ability to generate PSAT problems with specific features. Here the goal should be to cover the space of PSAT problems as evenly as possible; to generate satisfiable and non-satisfiable problems; to be able to vary parameters such as number of clauses, number of variables. We start by noting that it is not hard to generate inconsistent PSAT problems; in fact, in our initial experiments we noted that just assigning random values in $[0, 1]$ to randomly generated sets of clauses[1] is very likely to generate an inconsistent PSAT. In fact, our preliminary failed efforts to generate consistent PSATs led us to consider the algorithm presented in this section. The algorithm was designed to guarantee that every generated instance is *consistent*, and that the generation procedure is reasonably fast. The algorithm is implemented in the package PSATGen (available from the second author).

The basic idea we use is to attach probabilities to logical sentences by computing probabilities in Bayesian networks. A Bayesian network consists of a directed acyclic graph (DAG) where nodes represent random variables, and arcs denote conditional dependences [Pearl 1988]. Let $\mathbf{X}$ be the set of n random variables associated with nodes in a Bayesian network; then $X_i \in \mathbf{X}$ is a variable, and $D(X_i)$ and $\mathrm{pa}(X_i)$ are X_i's descendants and parents in the DAG, respectively. This formalism assumes that X_i is conditionally independent of $\mathbf{X} \setminus D(X_i)$ when the joint state of variables in $\mathrm{pa}(X_i)$ is known. If X_i is a root nodes it stores the marginal distribution of X_i; if that is not the case, X_i stores a local distribution function $p(X_i|\mathrm{pa}(X_i))$ that defines a collection of conditional distributions

[1]Clauses, and more generally k-SATs, can be easily generated, for example with *makewff* (downloadable from www.cs.rochester.edu/u/kautz/walksat/). We use this software in PSATGen whenever we need to generate a k-SAT with n variables and m clauses.

$p(X_i|\text{pa}(X_i) = pa_1) \ldots p(X_i|\text{pa}(X_i) = pa_r)$, one for each joint instantiation of $\text{pa}(X_i)$. The structure of a Bayesian network encodes a joint probability distribution on $\mathbf{X}$:

$$p(X) = \prod_i^n p(X_i|\text{pa}(X_i)) \,.$$

Marginal and conditional probabilities of every variable can be computed by belief updating algorithms.

PSATGen computes the probability $p(C_j)$ of a clause C_j, with literals $A_1, \ldots A_k$ defined on atoms $p_1, \ldots p_k$ of a CNF ϕ, using a three step procedure. First, it produces a Bayesian network $\mathcal{N}$ with n $(n \geq k)$ variables with categories *true* and *false* and adds a new binary variable named Y_l in $\mathcal{N}$ for each negative literal A_l in C_j. The new variable is made a child of X_l and its conditional probabilities are set to $P(Y_l = true|X_l = true) = 0$, $P(Y_l = false|X_l = true) = 1$, $P(Y_l = true|X_l = false) = 1$ and $P(Y_l = false|X_l = false) = 0$. These assigments encode the negation operator between Y_l and X_l [Pearl 1988]. Second, a new binary variable Z_j is added to $\mathcal{N}$. This variable represents the clause C_j and for each A_l in C_j it is made a child of the variable X_l if A_l is positive, otherwise it is make a child of Y_l . The local distributions of Z_j are defined as follows. Let pa_z to denote a joint instantiation of Z's parents. If the truth table of pa_z is evaluated as true then $P(Z = true|pa_z) = 1$ and $P(Z = false|pa_z) = 0$; if not $P(Z = true|pa_z) = 0$ and $P(Z = false|pa_z) = 1$. Finally, the PSATGen runs the belief updating algorithm on Z to compute the marginal $P(Z = true)$ and associates this value to C_j.

Algorithm 1 The PSATGen algorithm

Input: Let n, k, s and c be the numbers of variables, the number of literals in an clause, the max number of conjunctive sentences in a PSAT and the number of clauses per sentence; let z be the number of PSATs that must to be generated; additionally, let w, e and d be the induced width, the max number of edges and the max degree of an node in the Bayesian networks generated by BNGenerator.

1: Generate z Bayesian networks with n variables, max-induced-width w, max-degree d and with no more than e edges.
2: Set $i := 1$.
3: **repeat**
4: $m = c * r$.
5: Generate a k-SAT with n variables and m clauses indicated by $C_1, ..., C_m$.
6: Let $\mathcal{N}$ be the ith network generated with BNGenerator.
7: $a := 1$.
8: **for** $j := 1$ to s **do**
9: Randomly generate an integer $l \in \{1..c\}$.
10: define ϕ as the CNF formed by the clauses $C_a \wedge \ldots C_{a+b-1}$ from the k-SAT.
11: Represent ϕ in $\mathcal{N}$ by adding nodes as describe previously.
12: Compute the marginal $P(\phi)$ in $\mathcal{N}$ as described before.
13: Associate ϕ to $P(\phi)$.
14: **end for**
15: Save the resulting PSAT.
16: **until** $i > s$.

The probability of a CNF ϕ with clauses $C_1, \ldots, C_m$ is calculated similarly. First, the clauses of ϕ are inserted in a given Bayesian network $\mathcal{N}$ as previously described. Then a new binary variable W is added to $\mathcal{N}$. The local distributions of W are generated as follows. Let pa_w be a joint instantiation of variables $Z_1 \ldots, Z_m$. As before, if pa_w is evaluated as true it causes $P(W = true|pa_w) = 1$ and $P(W = false|pa_w) = 0$; if not $P(W = true|pa_w) = 0$ and $P(W = false|pa_w) = 1$. This assessement encodes the conjunction operator among the clauses in ϕ. The value of $P(W = true)$ is obtained by running belief updating.[2]

To generate randomly connected Bayesian networks, we use the BNGenerator package [Ide and Cozman 2002]. This package generates samples of Bayesian networks with control on several parameters: induced-width, node degree, number of edges. In particular, induced-width controls the complexity of the generated networks [Dechter 1996, Zhang and Poole 1996]. As the BNGenerator package guarantees that samples are generated with (asymptotically) uniform distribution, the PSATGen package does not introduce biases in the complexity of generated PSATs.

4. Checking PSATs consistency

In this section we consider the following statement of the PSAT problem. Consider given a set $\mathcal{S}$ of m logical sentences, all of them in CNF, where $\mathcal{S} = \{S_1, \ldots, S_m\}$, and where S_i consists of clauses in a set $\mathcal{C}$ of q clauses, where $\mathcal{C} = \{C_1, C_2, \ldots, C_q\}$, defined over n logical variables $\mathcal{X} = \{x_1, x_2, \ldots, x_n\}$; and given a vector $\pi \in \Re^m$ of probabilities. We then have 2^n possible worlds, referred to as w_j. We define a matrix $A_{(m+1)\times 2^n}$, with row a_0 a vector containing only ones, and all other entries $a_{i,j}$ equal to 1 if S_i is true in w_j and 0 otherwise. We also define a vector $d = [1, \pi^T]$. The decision form of PSAT is:

$$\begin{array}{ll} \min & 0p \\ \text{subject to} & \\ & Ap = d \\ & p \geq 0 \end{array} \qquad (1)$$

where the constraint formed by a_0 and d_0, and the constraint $p \geq 0$ express the fact that p must be a probability distribution. That is, to decide whether π is consistent, we must verify whether Expression (1) has a feasible solution.

The method of *column generation* [Hansen et al. 1991] circumvents the need to process the exponential number of columns of A, by requiring the iterative solution of two problems associated with Expression (1), as in the *revised simplex* algorithm:

- the *Restricted Master Problem (RMP)* is a linear program where we have a smaller number of columns than in Expression (1), and where we produce the dual optimal values for all variables fixed as basic.
- the *Subproblem (SP)* is an optimization on binary variables, where we start from the dual optimal values produced by the RMP, and we produces the column that must enter the basis of the RMP.

[2]This procedure is similar to Dechter and Smyth's method to compute Boolean queries in Bayesian networks [Dechter and Smyth 2000]. In our implementation, Bayesian networks are handled through the JavaBayes package, downloadable from www.cs.cmu.edu/~javabayes/Home.

The RMP and the SP are iterated over and over, until we reach either an RMP solution with value zero (we stop with consistency), or an SP with no negative reduced costs (we stop with inconsistency). We denote by B the set of indices of columns in the current basis of the RMP; N is the set of indices of columns not in the current basis of the RMP.

To solve Expression (1), it is sufficient to implement the phase 0 of the revised simplex method, thus minimizing the sum of artificial variables added to constraints. The SP to be solved, assuming that B and N are previously fixed, is:

$$\min_{j \in N} -c_B A_B^{-1} A^j = \min_{x \in \{0,1\}} -u_0 - \sum_{i=1}^{m} u_i S_i \tag{2}$$

where $u = c_B A_B^{-1}$ is the current vector or *dual variables* resulting from the solution of the RMP, and c_B refers to the coefficients of the objective function for the RMP.

At this point we have a nonlinear problem, specified by Expression (2). We wish to reduce this problem to an linear integer program. The method suggested by Hansen et al. [Hansen et al. 1991] is to first transform the sentences S_i to nonlinear expressions using the following equivalences:

$$x_i \vee x_j \equiv x_i + x_j - x_i \times x_j, \qquad x_i \wedge x_j \equiv x_i \times x_j, \qquad \neg x_i \equiv 1 - x_i. \tag{3}$$

The resulting nonlinear program is then reduced to a linear integer program. However, these equivalences are computationally very cumbersome: they produce a nonlinear program with a number of terms that cannot be predicted from Expression (2) alone.

We now propose a different reduction of Expression (2) to linear integer programming. We first include new structural variables y_i with associated zero coefficient in the objective function, so as to deal with the negation of clauses — as a negated clause generates a single multilinear term. These new variables assist in the calculation of values for variables z_i (those evaluate to S_i, as z_i manipulates the conjunction of variables $\neg y_i$ in S_i). Thus the resulting SP, using this procedure and the second and third equivalences in Expression (3), is:

$$\begin{array}{l} \min_{x \in \{0,1\}} -u_0 - \sum_{i=1}^{m} u_i S_i + \sum_{i=1}^{q} 0 \prod_{j \in C_i^-} x_j \prod_{j \in C_i^+} \neg x_j = \\ \min_{x \in \{0,1\}} -u_0 - \sum_{i=1}^{m} u_i \prod_{j \in S_i} \neg y_j + \sum_{i=1}^{q} 0 y_i = \\ \min_{x \in \{0,1\}} -u_0 - \sum_{i=1}^{m} u_i z_i + \sum_{i=1}^{q} 0 y_i + \sum_{i=1}^{n} 0 x_i \end{array} \tag{4}$$

where C_i^- and C_i^+ define the sets of indices of negated and non-negated variables that appear in clause C_i.

These operations aims at producing a *linearization* by introducing y_i and z_i, such that every feasible solution for them takes on the value of the corresponding multilinear term associated with them. In Expression (4), if for each multilinear term we apply a linearization scheme as follows:

$$r \prod_{j \in J} b_j, \tag{5}$$

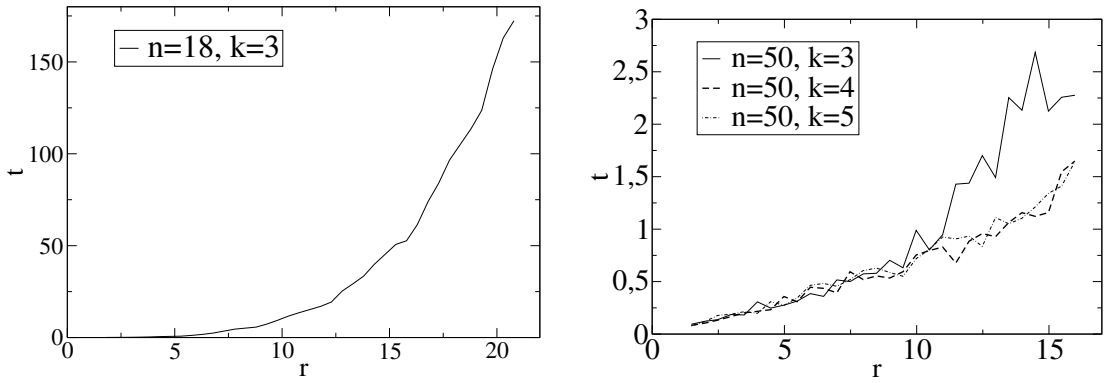

Figure 1. Left: Time (seconds) to check satisfiability of consistent PSATs. Right: Time (seconds) to check satisfiability of inconsistent PSATs (note that these problems required less effort).

where $r \in \Re$ and $b_j \in \{0, 1\}$ for $j \in J$, then Expression (5) is equivalent to:

$$\begin{array}{l} rt \\ \text{subject to} \\ \qquad t \leq b_j, j \in J \\ \qquad t \geq \sum_{j \in J} b_j - |J| + 1 \\ \qquad t \geq 0 \end{array} \tag{6}$$

where $|J|$ denotes the number of elements of J. Note that if $\neg b_j$ appears in Expression (5), then it is replaced by $(1 - b_j)$ in Expression (6).

At the end of the linearization procedure, the resulting SP to be exactly solved has $(n + q + m)$ variables and $(q + m + \sum_{i=1}^{q} |C_i| + \sum_{i=1}^{m} |S_i|)$ constraints. There is also the constraint that variables x are binary, and the constraint that all other variables are non-negative. We note that, in any given iteration of column generation, the call to an SP with vector u generates the column that is to be used in the RMP by collecting the variables z (when the optimal value of the SP is negative).

This method has been implemented in a software package called Geracol (available from the first author). The package generates the RMP and SP, produces the linear and linear integer programs, and then makes calls to the CPLEX solver from ILOG. It is possible to replace the CPLEX solver by a free solver, even though the performance of CPLEX tends to be substantially better than existing options.

To illustrate the Geracol package, we present a summary of several runs on consistent PSAT problems in Figure 1. The figure also shows tests with larger but inconsistent PSAT problems — in fact we have noted that inconsistent problems are generally easier than consistent ones. This is further evidence that our PSATGen generator is necessary to avoid tests that are excessively easy on consistency checker. We also note that in our preliminary tests we have not seen the phenomenon of "phase transition" that is known to occur in SAT. We leave for the future a complete investigation of this possibility.

5. The PPL Editor

The construction of a probabilistic logic knowledge base is not a simple task. Formulas must be inserted; assessments associated with them; consistency must be checked, and

revisions must be made continuously. Currently the only system that allows interactive development of a probabilistic logic base, to the best of our knowledge, is the *Check Coherence (CkC)* package. CkC is distributed for noncommercial use, for Windows platforms only, at www.dipmat.unipg.it/˜upkd/paid/software.html. The package deals with PSAT and allows conditioning on events of zero probability, a possibility we avoid in this paper. CkC asks the user to enter each formula and assessment in a sequence of steps, using an graphical interface to guide the process. While the CkC package is useful and very general in its operation, we find that the manipulation of formulas is excessively rigid and a bit difficult at times.[3]

We have thus decided to investigate a different strategy to edit probabilistic logic bases. Our idea was to start from a well known prototyping language, and add features to this language so that it can serve as a convenient, simple and easy-to-learn editor of probabilistic logic bases. We wanted to create a tool that could be easily extended by others; that could be freely distributed; and that could run in a variety of operating systems. After a comparative analysis of several prototyping languages currently available, we settled on the Python language (www.python.org), as it has a clean syntax, a free implementation and an associated development system. The remainder of this section describes a package we have written in Python and that allows easy development of probabilistic logic bases. The package is called PPL (for Propositional Probabilistic Logic), and can be obtained from the third and fourth authors.

In the PPL package, the user types in arbitrary propositional formulas, using an intuitive syntax (described in the system manual). The user interacts with the package using the friendly Python editor (the IDLE system), and the user can benefit from all Python facilities such as memory control and string processing. The package can call functions that translate formulas into CNF if so desired. The user can attach either probabilities or probability intervals to formulas, and check consistency at any point in time. To check consistency, the package executes calls to the consistency checker described in the previous section.

Consider the following typical interaction with the system, where formulas are defined, translated into CNF, associated with probabilities and probability intervals, and then consistency is verified:

```
>>> import PPL
>>> s1 = 'a <=> (b—c)'
>>> s1
'a <=> (b—c)'
>>> s2 = PPL.toCNF(s1)
>>> s2
'((˜b | a) & (˜c | a) & (b | c | ˜a))'
>>> PPL.p(s1, 0.5)
>>> s3 = 'd | (e & f) | g'
>>> s3
```

[3]We note that CkC contains an inference engine; thus the CkC package is similar to the editor we describe in this section *plus* the consistency checker discussed in the previous section. We also note that inferences with CkC is a slow process: in our tests, we observed that it usually takes hours to obtain inferences for relatively small problems (say 20 variables/50 assessments).

```
'd | (e & f) | g'
>>> s4 = PPL.toCNF(s3)
>>> s4
'((e | d | g) & (f | d | g))'
>>> PPL.p(s3, 0.3, 0.8)
>>> PPL.checkCoherence()
Coherent!
```

6. Conclusion

In this paper we have presented three contributions:

- An algorithm for random generation of *consistent* PSAT problems (we stress that inconsistent PSAT problems can be easily generated, by producing a random SAT and then assigning probabilities to clauses randomly).
- A consistency checker that implements the classic column generation algorithm for PSAT consistency verification, enhanced with a novel method that handles assessments over CNFs.
- An editor for PSAT that is extensible, portable, easy-to-use, and freely distributed.

These tools are important in practice as researchers gradually address knowledge bases that mix logical sentences and probabilistic assessments. Given the lack of software packages that support this task, we expect the tools presented here to be useful, however modest they may be in their current stage.

Acknowledgements

This work has been supported by FAPESP grant 2004/09568-0; the third author is supported by FAPESP grant 06/58252-1; the fourth author is supported by FAPESP grant 06/58251-5; the fifth author is partially supported by CNPq grant 3000183/98-4.

References

Andersen, K. A. and Hooker, J. N. (1994). Bayesian logic. *Decision Support Systems*, 11:191–210.

Dechter, R. and Smyth, P. (2000). Processing Boolean queries over belief networks. Technical Report www1.ics.uci.edu/ dechter/publications, ICS.

Georgakapoulos, G., Kavvadias, D., and Papadimitriou, C. (1988). Probabilistic satisfiability. *Journal of Complexity*, 4:1–11.

Goodrich, M. T. and Tamassia, R. (2002). *Algorithm Design: Foundations, Analysis and Internet Examples*. John Willey and Sons.

Hailperin, T. (1986). *Boole's Logic and Probability*. North-Holland, 2nd edition.

Hansen, P. and Jaumard, B. (2000). Probabilistic Satisfiability, Les Cahiers du GERAD, G-96-31, Montreal.

Hansen, P., Jaumard, B., and de Aragão, M. (1991). Column generation methods for probabilistic logic. ORSA Journal on Computing 3:135–148.

Pearl, J. (1988). *Intelligent Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, 1st edition.

Ide, J. S. and Cozman, F. G. (2002). Random generation of Bayesian networks. In *Proc. of the XVI Braziliam Symposium on Artifical Intelligence*, pages 99–99. Springer-Verlag.

Nilson, N. (1986a). Probabilistic logic. *Artificial Intelligence*, 28:71–87.

Nilson, N. J. (1986b). Probabilistic logic. *Artificial Intelligence*, 28:71–87.

Papadimitriou, C. H. and Steiglitz, K. (1982). *Combinatorial Optimization: Algorithms and Complexity*. Prentice Hall.

Dechter, R. (1996). Bucket elimination: A unifying approach for probabilistic inference algorithms. In *Proc. of 10th Annual Conference on Uncertainty in Artificial Intelligence*, pages 211–219. Morgan Kaufmann.

Neapolitan, R. E. (1990). *Probabilistic Reasoning in Expert Systems*. Prentice Hall, 1st edition.

Richardson, M. and Domingos, P. (2006). Markov logic networks. *Machine Learning*, 62(1-2):107–136.

Russell, S. and Norvig, P. (1995). *Artificial Intelligence: A Modern Approach*. Prentice Hall, 1st edition.

Zhang, N. L. and Poole, D. (1996). Exploiting causal independence in Bayesian network inference. *Journal of Artificial Intelligence Research*, 5:301–328.