



University of Zurich
Department of Informatics



Master Thesis July 31, 2010

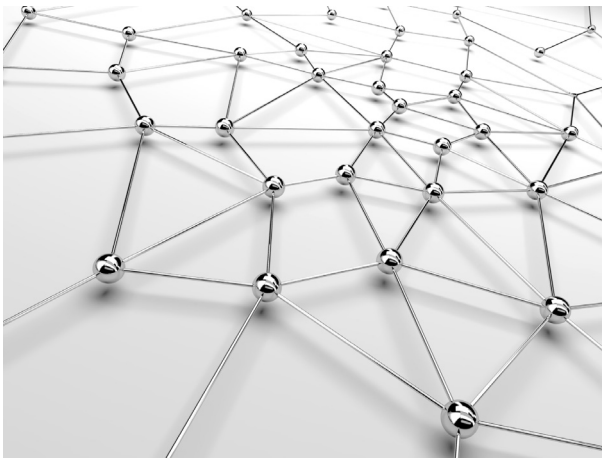
Markov Logic Inference on Signal/Collect

Stefan Schurgast

of Zürich ZH, Switzerland

Student-ID: 02-913-770

stefan@schurgast.com



Advisor: **Philip Stutz**

Prof. Abraham Bernstein, PhD

Department of Informatics

University of Zurich

<http://www.ifi.uzh.ch/ddis>

Acknowledgements

I sincerely thank Professor Abraham Bernstein, Ph.D., Head of the Dynamic and Distributed Information Systems (DDIS) group, Department of Informatics, University of Zurich for attracting me to this highly interesting topic in Artificial Intelligence and guidance through my thesis.

I am deeply grateful to my advisor, Philip Stutz, doctoral student and assistant in the DDIS group, Department of Informatics, University of Zurich for his important support, his constructive criticism and excellent advice throughout this work. Also, I want to thank him for introducing me to Scala and helping me out whenever problems occurred.

Special thanks also to Marc Körsgen for his review.

Last but not least, I owe my loving thanks to my wonderful wife Carmen Schurgast for her encouragement and understanding during the time I wrote this thesis.

Thank you all for the fun time here at the University of Zurich!

Stefan Schurgast,
Zurich, Switzerland

Abstract

Over the last several years, the vision of a Semantic Web has gained support from a vast of different fields of application. Meanwhile, there is a large number of datasets available with a tendency to interlink between each other, ready to be analyzed. But RDFS/OWL and SWRL, the standard languages for representing ontological knowledge and rules in RDF lack because of their limited expressiveness. Markov logic provides a good solution to this problem by putting weights on formulas, generalizing first-order logic with a probabilistic approach, allowing also contradictory rules.

By successfully implementing and evaluating the execution of loopy belief propagation on Markov networks using the Signal/Collect framework, an elegant, and yet highly efficient solution is ready to be provided for the use in further applications.

Zusammenfassung

Über die letzten Jahre erhielt die Vision eines Semantischen Webs breite Unterstützung aus einer Vielzahl von Anwendungsgebieten. Dabei entwickelt sich eine Tendenz, die Datensets untereinander immer mehr zu verlinken. Zur Analyse der Daten existieren zwar bereits Standardsprachen wie RDFS/OWL und SWRL, die ontologisches Wissen und Regeln repräsentieren zu vermögen, doch ihnen fehlt es oft an Ausdruckstärke. Markov Logik ist eine ideale Lösung zu diesem Problem. Sie generalisiert die Prädikatenlogik mit Ansätzen aus der Wahrscheinlichkeitstheorie, und ermöglicht so auch widersprüchliche Regeln in einer Wissensbasis zu vereinen.

Durch die erfolgreiche Implementierung und Evaluierung des Loopy Belief Algorithmus, der mittels Signal/Collect auf einem Markov Netzwerk ausgeführt wird, steht nun eine elegante und äusserst effiziente Lösung für die Anwendung in weiteren Systemen bereit.

Contents

Contents	ix
1 Introduction	1
1.1 Goal of this thesis	2
1.2 Structure	2
2 General Concepts	3
2.1 Logic	3
2.1.1 Fundamental concepts in logic	4
2.1.2 Propositional logic	4
2.1.3 First-order logic	7
2.2 Probabilistic Reasoning	9
2.2.1 Probability Theory	9
2.2.2 Probabilistic graphical models	11
2.2.3 Inference in probabilistic graphical models	13
2.2.4 A detailed look on Belief Propagation	16
2.3 Markov logic	19
2.3.1 The Markov Logic Language	19
2.3.2 Inference in Markov Logic	20
2.3.3 Markov Logic inference systems	21
2.4 Knowledge representation	25
2.4.1 The Semantic Web	25
2.4.2 Resource Description Framework RDF	26
2.4.3 RDF Schema RDFS	27
2.4.4 Web Ontology Language OWL	27
2.4.5 Rule formats	29
2.5 Signal/Collect	30

3	System Requirements	33
3.1	Functional requirements	33
3.2	Nonfunctional requirements	35
3.2.1	Attributes	35
3.2.2	Constraints	35
4	System Design and Implementation	37
4.1	Architecture	37
4.2	Markov Logic formulas	38
4.3	Network grounding	40
4.4	Running loopy belief propagation	42
5	Evaluation	47
5.1	Benchmark test with LISy and Alchemy	47
5.1.1	Test environment	48
5.1.2	Configuration	48
5.1.3	Results and conclusion	48
5.2	Scalability of LISy	50
5.2.1	Test environment	50
5.2.2	Configuration	51
5.2.3	Results and conclusion	51
6	Limitations	53
6.1	Limitations of the evaluation process	53
6.2	System limitations	54
7	Future Work	57
7.1	Features	57
7.2	Research	58
8	Conclusions	59
A	Appendix	61
A.1	Content CD-ROM	61
A.2	System usage	62
A.3	Data sets used	62
	List of Figures	63
	List of Tables	65
	List of Listings	67

Bibliography**69**

rather small compared to first-order logic or even Markov logic that generalizes first-order logic and combines it with Markov networks in order to add weights to formulas. This gives Markov logic the power to deal with uncertainty by using probability distributions, and the flexibility to allow even contradictory rules.

Although Markov logic has already been used in a wide range of AI applications, including link predication, entity resolution and information extraction (Domingos and Lowd, 2009), the infrastructure to access Semantic Web data with algorithms and perform them efficiently is missing. A new scalable programming model for typed graphs called Signal/Collect by Stutz et al. (2010) promises an improvement in this situation as it eases the development of message passing algorithms. As a number of inference algorithms like belief propagation use the idea of passing information (in this case beliefs) from one node over to another node, Signal/Collect seems well-suited for this kind of task.

1.1 Goal of this thesis

Given the above, this thesis presents on the one hand a solution to the missing infrastructure problem by providing the Logical Inference System (LISy), a system that enables developers to integrate efficient logic reasoning into other applications in a straight forward, and easy way. LISy provides the means to define formulas in first-order and Markov logic and ground them with RDF data from SPARQL endpoints, as well as an implementation of the loopy belief propagation algorithm based on the Signal/Collect framework as reasoning engine. On the other hand, this thesis demonstrates a proof of concept to run sophisticated algorithms efficiently on the Signal/Collect programming model and framework by applying an implementation of belief propagation to it.

1.2 Structure

In order to pursue these objectives, the remainder of this thesis is organized as follows: Chapter 2 discusses the general concepts on which this thesis is based on by giving an overview of some concepts of logic, probabilistic reasoning, Markov logic, knowledge representation in the Semantic web, as well as an introduction to the Signal/Collect programming model. Chapter 3 then shows the requirements that have been made for the system followed by the system design for LISy in chapter 4, describing how the three layers for formulas and groundings, network and algorithms are conceptually designed and implemented. The evaluation for the system then follows in chapter 5, showing a benchmark test with LISy and Alchemy, as well as a test to evaluate scalability. After pointing out the limitations of the system in chapter 6, the thesis closes with a discussion of future work in chapter 7 and the conclusion in chapter 8.

2

General Concepts

Before rushing into system details in the next chapters, the general concepts used in this thesis and for the later implementation of the system are pointed out first. The starting section 2.1 on logic shows an overview of the fundamental concepts in logic as well as some important logic representations including propositional logic and first-order logic. Since these concepts all expect crisp values (meaning true or false), probabilistic reasoning in the section 2.2 thereafter supplements this view by introducing probabilistic graphical models as well as reasoning algorithms. Section 2.3 then presents Markov logic as the combination of both worlds, generalizing first-order logic by applying Markov networks. Section 2.4 on knowledge representation will focus on data representation in the Semantic Web. Finally, section 2.5 explains the Signal/Collect programming model in order to apply it to LISy in the following chapters. If you are already familiar with one concept or the other, feel free to just skip the section.

2.1 Logic

Logic is a central field in a lot of disciplines like philosophy, mathematics, and computer science. Whately (1867) defines it as the "Art and Science of reasoning", but it has already been studied long before by several ancient civilizations (Britannica, 2007), including ancient India, China and Greece (Jowett, 1979). Nowadays, logic usually makes use of formal languages in order to ease valid inference and knowledge representation (Russell and Norvig, 2003, pages 794-795).

This section briefly covers fundamental concepts in logic in section 2.1.1, propositional logic in 2.1.2 and first-order logic in 2.1.3 in order to deal with Markov logic and inference algorithms later on.

2.1.1 Fundamental concepts in logic

As stated by Russell and Norvig (2003) and Genesereth and Nilsson (1987), a *knowledge base (KB)* is a set of (well formed) sentences, expressed according to the syntax of the representation language (as for example in first-order logic). The sentences in the KB are all implicitly conjoint, resulting in a single large sentence. In other words, the formulated sentences in the KB each define a known logical fact, together formulating a collection of facts and their relationships among each other.

Since collecting data especially makes sense when there is a way to query what is known thereafter, one main task of the KB is to infer, i.e. derive new sentences from existing. This process of drawing conclusions is either done by applying heuristics (based on logic, statistics, or others) to the existing sentences or by interpolating the next logical step in an intuited pattern. With logical *entailment*, being the relation between sentences that define if some sentence follows logically from another sentence, Inference can be defined as the process of finding a specific entailment in a KB (Russell and Norvig, 2003, pages 194-197).

The meaning of a sentence is defined by the semantics of the language. Therefore, a truth value defining the correctness of a sentence in a possible world is applied to every sentence. Usually this truth value is either true or false, but depending on the language may also be something in between. As an example, the sentence $FatherOf(x, y)$ is true in a world in first-order logic (see section 2.1.3) where x is *Hugo* and y is *Stefan*, whereas it is not if x is *AuntLisy*. The process of giving a meaning to variables is called *grounding*. In algorithms, this is usually made by establishing connections between the symbols and their meanings (Russell and Norvig, 2003, page 202-204).

In order to choose good inference algorithms, two properties are highly important. If and only if an inference algorithm derives only from its entailed sentences, it is called *sound* (i.e. its inference rules only follow what is valid with respect to the semantic and does not make things up by itself). The soundness property counts as an initial reason to find a logical system desirable. The other property that makes it provable is called *completeness*. An algorithm is complete, if it can derive any sentence that is entailed (Russell and Norvig, 2003, page 203).

In the following sections, a number of important languages, the simpler *propositional logic* and the more complex *first order logic*, are discussed in further detail.

2.1.2 Propositional logic

Propositional logic (or Boolean logic, named after George Boole) is basically defined by symbols, sentences, and connectives. Sentences are built recursively by using atomic sentences or by complex sentences. Atomic sentences can be a symbol or a truth value (true \top or false \perp). Complex sen-

tences are sentences connected with logical connectives (Russell and Norvig, 2003, page 204-205). They are listed in the following table 2.1 below sorted from highest the binding (precedence) ($'\neg'$) to the lowest ($'\Leftrightarrow'$) (Stärk, 2002, page 14).

Table 2.1: Logical connectives

logical connective	meaning
\neg	Negation (not)
\wedge	Conjunction (and)
\vee	Disjunction (or)
\Rightarrow	Implication
\Leftrightarrow	Biconditional

In figure 2.1, the grammar of sentences in propositional logic is shown in Backus-Naur Form [BNF] (Russell and Norvig, 2003, page 205) (Stärk, 2002, page 14).

Figure 2.1: Propositional logic BNF (Russell and Norvig, 2003) (Stärk, 2002)

```

Sentence      = AtomicSentence | ComplexSentence
AtomicSentence = Proposition | 'T' | 'F'
ComplexSentence = '(' Sentence ')' | '(' Sentence Connective Sentence ')'
Connective     = '&' | 'v' | '=>' | '=<=>'
Proposition    = Symbol

```

In order to evaluate the truth value of a sentence in a model, a meaning must be assigned to the connectives defined in table 2.1. These 2^n models can be represented by using truth tables as demonstrated in the example given by table 2.2 below (Russell and Norvig, 2003, 207).

Table 2.2: Truth tables for logical connectives (Russell and Norvig, 2003)

A	B	$\neg A$	$A \wedge B$	$A \vee B$	$A \Rightarrow B$	$A \Leftrightarrow B$
\perp	\perp	T	\perp	\perp	T	T
\perp	T	T	\perp	T	T	\perp
T	\perp	\perp	\perp	T	\perp	\perp
T	T	\perp	T	T	T	T

The semantics for the negation is defined as the opposite of the original sentence. A conjunction is true if all the conjoint sentences are true them selfs. In a disjunction, at least one sentence of the disjunction has to be true to make the sentence true. This means that the disjunction is not exclusive. In an implication, only the sentence where the conjunction of the premise and the

negated conclusion ($A \wedge \neg B$) remains unsatisfiable (known as *reductio ad absurdum*) the sentence evaluates to true. Logical equivalence means that two sentences are true in the same set of models. In other words, both sides of the equivalence must entail each other ($(A \Rightarrow B) \wedge (B \Rightarrow A)$) as defined in the deduction theorem. Standard logical equivalences are listed in the table 2.3 below (Russell and Norvig, 2003, pages 207-214) (Stärk, 2002, pages 18-48):

Table 2.3: Standard logical equivalences (Russell and Norvig, 2003)

description	standard equivalence
commutativity for the \wedge	$(A \wedge B) \equiv (B \wedge A)$
commutativity for the \vee	$(A \vee B) \equiv (B \vee A)$
associativity for the \wedge	$((A \wedge B) \wedge C) \equiv (A \wedge (B \wedge C))$
associativity for the \vee	$((A \vee B) \vee C) \equiv (A \vee (B \vee C))$
double-negation elimination	$\neg(\neg A) \equiv A$
contraposition	$(A \Rightarrow B) \equiv (\neg A \Rightarrow \neg B)$
implication elimination	$(A \Rightarrow B) \equiv (\neg A \vee B)$
biconditional elimination	$(A \Leftrightarrow B) \equiv ((A \Rightarrow B) \wedge (B \Rightarrow A))$
De Morgan	$\neg(A \wedge B) \equiv (\neg A \vee \neg B)$ and $\neg(A \vee B) \equiv (\neg A \wedge \neg B)$
distributivity of \wedge over \vee	$(A \wedge (B \vee C)) \equiv ((A \wedge B) \vee (A \wedge C))$
distributivity of \vee over \wedge	$(A \vee (B \wedge C)) \equiv ((A \vee B) \wedge (A \vee C))$

To achieve reasoning, inference rules are applied. As described with Modus Ponens ($\frac{A \Rightarrow B, A}{B}$), if A implies B and A is given, B can be inferred. Another implication rule, the And-Elimination, states, that from a conjunction any conjuncts can be inferred ($\frac{A \wedge B}{A}$) (Russell and Norvig, 2003, page 211).

For inference algorithms, it is useful to utilize the conjunctive normal form [CNF] because the resolution rule applies to disjunctions of literals. Since every sentence of propositional logic is equivalent to a conjunction of disjunctions of literals, a CNF is easily built by applying the inference rules as well as the standard equivalences as stated above (e.g. $(x_1 \vee \dots \vee x_n) \wedge \dots \wedge (x_o \vee \dots \vee x_z)$). The following procedure converts a sentence into CNF (Russell and Norvig, 2003, page 215):

1. Replace \Leftrightarrow by applying the biconditional elimination.
2. Replace \Rightarrow by applying the implication elimination.
3. Move \neg inwards by applying the double-negation elimination and De Morgan.
4. Apply distributive law until CNF is reached.

Reasoning algorithms using proof by contradiction can now easily show the unsatisfiability of a sentence because with one false conjunction, the entire sentence evaluates to false. Often reasoning algorithms work with Horn clauses (disjunctions with a most one positive literal) (Russell and Norvig, 2003, page 217), as does the programming language Prolog. But since Horn clauses lack

of expressiveness and are not very user friendly (i.e. in defining rules for a given problem), they will not be used in the implementation of LISy.

In many situations, propositional logic lacks expressiveness especially in larger domains. Since, for example, it is not possible to express that all married couples are happy, a more expressive language has to be found.

2.1.3 First-order logic

Russell and Norvig (2003) define first-order logic (or first-order predicate calculus) [FOL] as a formal logical system, which in contrast to propositional logic uses not only propositions, but also predicates and quantification. The following figure 2.2 describes how formulas are being created with the abstract syntax, provided in Backus-Naur Form [BNF] (Russell and Norvig, 2003, page 984).

Figure 2.2: First-order logic BNF (Russell and Norvig, 2003, page 247) (Stärk, 2002, page 83)

Formula	=	Atom '(' Term '≈' Term ')' '(' '¬' Formula ')' '(' Formula Connective Formula ')' Quantifier Variable Formula
Atom	=	'⊤' '⊥' Predicate '(' TermList ')'
Term	=	Variable Constant Function '(' TermList ')'
TermList	=	Term { ',' Term }
Connective	=	'∧' '∨' '⇒' '⇔'
Quantifier	=	'∀' '∃'
Variable	=	'x' 'y' ...
Constant	=	Identifier (e.g. Carmen, Stefan)
Predicate	=	Identifier (e.g. Married, IsHappy)
Function	=	Identifier (e.g. HusbandOf)

As put in writing by Domingos and Lowd (2009), in FOL there are four types of symbols that will be used for the implementation of formulas in LISy. *Constant* symbols declare individual objects in the domain of interest, and *variable* symbols range over the objects in the domain. *Predicate* symbols (e.g. Married) represent relations among objects in the domain or attributes of objects (e.g. IsHappy), and *function* symbols represent mappings from tuples of objects to objects. A term is any expression representing an object in the domain (e.g. Carmen, Married(x,y)) and an atom is a predicate symbol applied to a tuple of terms, or true (\top) and false (\perp).

In order to add meaning to a KB, all formulas have to be grounded with an *interpretation*. An interpretation defines which objects, functions, and relations in the domain are represented by which symbols. By replacing all variables in a term, atom, or predicate by constants, we receive a *ground term*, *ground atom*, or *ground predicate* respectively. A *possible world* assigns a *truth value*

to each possible ground atom in this interpretation (Domingos and Lowd, 2009, pages 9-10) (Russell and Norvig, 2003, page 249). An example for such a possible world is given by the grounded formula $Married(Stefan, Carmen) \Rightarrow IsHappy(Stefan)$ (which of course has the truth value for truthness and therefore is satisfiable).

In FOL, there exist two quantifiers, one for the universal \forall and one for the existential quantification \exists . They are listed in table 2.4 and are used to define the quantity of specimen in the domain of discourse that satisfy an open formula.

Table 2.4: FOL quantifiers

quantifier	meaning
\forall	Universal quantification (for all)
\exists	existential quantification (exists)

Their meaning can best be explained by an example. The universal quantification in the sentence $\forall x Married(x) \Rightarrow Happy(x)$ states, that every married object x is happy. So the universal quantification makes a statement about every object in the world. In contrast to the universal quantification, the existential quantification only makes a statement about some objects in the universe. For example $\exists x Play(x, Poker) \wedge Win(x, Money)$ states, that there exists at least one object x , that plays poker and wins money. From the logical point of view, the two quantifiers can be replaced by each other through negation, as shown in the following figure 2.3 below. Since the universal quantifier is a conjunction over the universe of objects, and the existential quantifier is a disjunction, they both follow De Morgan's rule too. To show that two expressions refer to the same object, the equality symbol $=$ respectively \neq can be used, e.g. $Husband(Carmen) = Stefan$ (Russell and Norvig, 2003, pages 249-253).

Figure 2.3: De Morgan rules for quantifiers

$$\begin{aligned}
 \forall x \neg Expression &\equiv \neg \exists x Expression \\
 \neg \forall x Expression &\equiv \exists x \neg Expression \\
 \forall x Expression &\equiv \neg \exists x \neg Expression \\
 \exists x Expression &\equiv \neg \forall x \neg Expression
 \end{aligned}$$

Inference in first-order logic can be done by reducing first-order logic to propositional logic, and then applying the inference rules defined in the previous section 2.1.2. To do so, the quantifiers must be replaced in the sentences. Universal quantifiers can be replaced by substituting the variables with concrete objects, i.e. by grounding the predicate. The existential quantifier can be omitted by introducing a Skolem constant (a constant that does not already exist in the knowledge base) and replacing the variable with it.

Although sound and complete inference can be achieved by using propositionalization and

unification techniques, the Turing problem about semidecidability still exists, meaning that an algorithm can only approve the existence of an entailment, but not the non-existence (Russell and Norvig, 2003, pages 272-278).

2.2 Probabilistic Reasoning

In the sections about logical languages described so far, propositions have always been either true, false or unknown. But in the real world, this is hardly ever the case. This is where uncertainty becomes a central aspect in reasoning. Consider the following rule in first-order logic $Smokes(x) \Rightarrow Cancer(x)$, where smoking causes cancer. In the real world, this may be true to a certain degree, since smoking need not cause cancer in every case. Our knowledge about this rule can therefore only provide a degree of belief (Domingos and Lowd, 2009). According to Russell and Norvig (2003), the reasons for that may be threefold:

1. Laziness: It may be too much work to completely model all aspects of the real world.
2. Theoretical ignorance: There may be no complete theory and thus too little information to model all aspects of the real world.
3. Practical ignorance: Although the entire theory may be available, the particular information about the modeled individual may be missing.

One approach in order to handle uncertainty within reasoning is by probability theory, which is introduced in the following section 2.2.1. After that, probabilistic graphical models which combine concepts from probability theory and graph theory in a specific data structure (Russell and Norvig, 2003) are introduced in section 2.2.2.

2.2.1 Probability Theory

In probability theory, the way of dealing with uncertainty is to assign a degree of belief, i.e. the probability which is a real number between 0 and 1 to the knowledge in the domain. The value 0 stands for a 0% chance and 1 for 100% degree of belief, i.e. the expectation that something is true (not to be mistaken with degree of truth, which is subject of fuzzy logic). If something, i.e. a proposition, is known with a degree of belief of 100%, this the known fact is evidence (Russell and Norvig, 2003).

Russell and Norvig (2003) declare the random variable as the basic element in probability theory, which has an initially unknown state. The domain of a variable is the values it can take on, e.g. the domain of *Hungry* might be $\langle true, false \rangle$. The domain of random variables may take three different types: boolean values $\langle true, false \rangle$, discrete values $\langle happy, sad \rangle$ or continuous values like $X = 5.2$ or $X \geq 2.1$. Also combinations of propositions are possible, e.g.

$Married = true \wedge Happy = true$ or with a slightly different syntax $Married \wedge Happy$. By adding probabilities to propositions, the degree of belief is expressed.

According to Russell and Norvig (2003), the unconditional or prior probability associated with a proposition is "the degree of belief accorded to it in the absence of any other information", e.g. $P(Mood = bored) = 0.2$. For different probabilities of the possible values, the probability can be written as a vector, e.g. for $P(Mood = bored) = 0.2, P(Mood = tired) = 0.3, P(Mood = excited) = 0.5$ there is a probability distribution $P(Mood) = \langle 0.2, 0.3, 0.5 \rangle$. Probability distributions like $P(Mood, Activity)$ are called joint probability distributions.

The conditional probability in $P(x|y)$ is the degree of belief of x given y , e.g. $P(Work = false|Mood = bored)$. This can also be transformed into terms of unconditional probability (Russell and Norvig, 2003):

$$P(x|y) = \frac{P(x \wedge y)}{P(y)} \quad (2.1)$$

$$P(x \wedge y) = P(x, y) = P(x|y)P(y) \quad (2.2)$$

Another way to define a conditional probability is by making use of the commutativity of the conjunction in $P(x \wedge y) = P(x|y)P(y)$ and $P(x \wedge y) = P(y|x)P(x)$. Equating the two formulas, the following more general equation 2.3 results, called the Bayes' rule (Russell and Norvig, 2003).

$$P(Y|X) = \frac{P(X|Y)P(Y)}{P(X)} \quad (2.3)$$

A joint probability distribution with the complete set of random variables is called a full joint probability distribution (Russell and Norvig, 2003). The resulting NxM table for the married/happiness example looks as in table 2.5, with all probabilities adding up to 1.

Table 2.5: Full joint probability distribution example

	married	\neg married
happy	0.4	0.3
\neg happy	0.25	0.05

The marginal probability is the distribution over a subset of variables, e.g. for the proposition *happy* it is $P(happy) = 0.4+0.3$. The process for this is called marginalization or summing out.

Although the full joint probability distribution is able to answer any question about the domain, the problem becomes exponentially large as the number of variables grows. Probabilistic graphical models, as discussed in the following section 2.2.2, provide a systematic way in order to represent probabilistic relationships explicitly (Russell and Norvig, 2003).

2.2.2 Probabilistic graphical models

Probabilistic graphical models provide a mechanism for exploiting structure in complex joint distributions in a graph based representation (Koller and Friedman, 2009). With the help of these representations, there exist effective methods to build and utilize complex distributions over a high-dimensional space.

The probabilistic graphical models in this thesis consist of nodes (also known as vertices) and edges (also called links or arches). Each node represents a random variable (or a group of random variables). The edges define probabilistic relationships between the variables. Thanks to the graphical representation, "the joint distributions over all of the random variables can be decomposed into a product of factors each depending only on a subset of the variables" (Bishop et al., 2006).

The following sections describe the two types of representations, Bayesian networks (see 2.2.2) and Markov networks (see 2.2.2), with their different perspectives.

Bayesian Networks

In Bayesian networks (also called directed graphs) the links of the graphs are arrows, indicating the directionality with a source and a target (Koller and Friedman, 2009).

As an example by Bishop et al. (2006), given a joint probability distribution $P(a, b, c)$ and applying the product rule will result in $P(a, b, c) = P(c|a, b)P(a, b)$. Applying the product rule a second time leads to equation 2.4.

$$P(a, b, c) = P(c|a, b)P(b|a)P(a) \quad (2.4)$$

The right hand side of the equation 2.4 can now be shown as a simple graph as in the example given in figure 2.4, in which each node represents the variable and the edges represent the factors (e.g. $P(c|a, b)$ represents the edges from a and b to c).

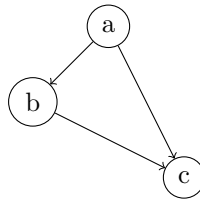


Figure 2.4: Bayes network

In the general case, the joint probability distribution of the random variables $X = X_1, \dots, X_n$ is given by equation 2.5. Bishop et al. (2006) and Russell and Norvig (2003) express in this equation the factorization property of the joint probability distribution.

$$P(X = x) = \prod_{k=1}^K P(x_k | \text{parent}_k) \quad (2.5)$$

This shows, that "each entry in the joint distribution is represented by the product of the appropriate elements of the conditional probability tables in the Bayesian network" (Russell and Norvig, 2003).

Although Bayesian networks are easier to understand, they do not allow cycles and the independence between variables is difficult to determine. In order to run inference algorithms on the network, the Bayesian networks are usually converted to Markov networks, as they are a special case of Bayesian networks (in which the partition function $Z = 1$) (Domingos and Lowd, 2009) (Bishop et al., 2006).

Markov networks

Pearl (1988) describes a *Markov network* (originally known as Markov random field (Kindermann et al., 1980)) as a "model for the joint distribution of a set of variables $X = (X_1, X_2, \dots, X_n) \in \chi$ ". It is represented by an undirected graph G with a node for each variable and a potential functions ϕ for each edge. The following figure 2.5 shows an example of a simple Markov network, which is a joint distribution for watered, withered and perennial plants.

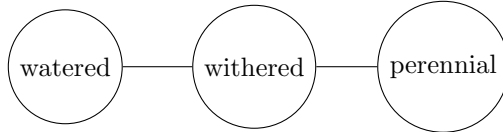


Figure 2.5: Example of Markov network

The joint distribution represented by the Markov network is given by equation 2.6, as stated by Domingos and Lowd (2009).

$$P(X = x) = \frac{1}{Z} \prod_k \phi_k(x_{\{k\}}) \quad (2.6)$$

Here, $x_{\{k\}}$ represents the state of the variable in k th clique. Z is the partition function, $Z = \sum_{x \in \chi} \prod_k \phi_k(x_{\{k\}})$. The two tables 2.6 and 2.7 summarize possible potential functions for the example given above. These potential functions denote the affinity between two values. The

higher the value, the more compatible the two values are. These values are not normalized and do not even have to be between 0 and 1 (Koller and Friedman, 2009).

Table 2.6: Potential function for example with random variables watered and withered plant

<i>watered</i>	<i>withered</i>	$\phi(\textit{watered}, \textit{withered})$
1	1	4.5
1	0	2.7
0	1	4.5
0	0	4.5

Table 2.7: Potential function for example with random variables withered and perennial

<i>withered</i>	<i>perennial</i>	$\phi(\textit{withered}, \textit{perennial})$
1	1	4.5
1	0	2.7
0	1	4.5
0	0	4.5

To find the joint probability distribution for the plant, that is watered, withered but not perennial, the calculation is $P(\textit{watered} = \textit{true}, \textit{withered} = \textit{true}, \textit{perennial} = \textit{false}) = \frac{1}{Z}(4.5 * 2.7) = \frac{12.15}{Z}$.

In order to reduce calculation effort, Markov networks are usually represented as log-linear models as described by Domingos and Lowd (2009) and Koller and Friedman (2009), in which each clique potential is replaced by an exponentiated weighted sum of features of the state (see equation 2.7).

$$P(X = x) = \frac{1}{Z} \exp\left(\sum_j w_j f_j(x)\right) \quad (2.7)$$

According to Roth (1996), inference in Markov networks is #P-complete. Therefore approximate inference algorithms like Markov chain Monte Carlo (MCMC) (e.g. Gibbs sampling) (Gilks et al., 1996) or belief propagation (Yedidia et al., 2005) are often used. The belief propagation algorithm will be discussed in this thesis later on.

2.2.3 Inference in probabilistic graphical models

In the previous sections, the main goal was to give the reader a complete representation of the respective joint probability distributions. In this section the question will be how to infer from this knowledge, as the implementation of an efficient inference algorithm will be the main challenge of

this thesis.

For many networks, exact inference performs efficiently. But because the computational and space complexity growth exponentially with the size of the network Koller and Friedman (2009), the exact methods may not deliver adequate results. Therefore, following the suggestion of Koller and Friedman (2009), the use of approximate algorithms become interesting as they may perform much better on large networks. The trick is to define simpler distributions as the most efficient approximation of the target distribution. Thus, the inference task becomes a constraint optimization problem. The most used method for solving this type of problems within graphical models is based on the use of Lagrange multipliers, which produces a set of equations that characterize the optima of the objective. Luckily, the set of equations are fixed-point equations that define each variables in terms of others and hence derive from the constrained energy optimization.

Since the constrained energy optimization can be viewed as passing messages over a graph object, all the corresponding inference approximations like sum-product algorithms and many more can be solved by the same procedure. In the section 2.5, the Signal/Collect framework, which will be utilized for the implementation, is described to simplify exactly this task even more.

Exact Algorithms

One example of an exact probabilistic inference is *enumerate-joint-ask* which enumerates the entries in a full joint distribution. It then adds up the probabilities from the joint table and normalizes the results (Russell and Norvig, 2003). Mathematically speaking, the formula with the query variable X , the evidence variable E (e being the observed values) and Y the unobserved variables is as follows:

$$P(X|e) = \alpha P(X, e) = \alpha \sum_y P(X, e, y) \quad (2.8)$$

In practice however, this algorithm does not scale well because in the worst case, almost all the hidden variables have to be summed out. With a growing number of Boolean variables, it requires exponential space and time $O(n2^n)$. By moving constant terms out of the summations and using depth-first recursion of the enumeration-ask algorithm (Russell and Norvig, 2003, see pages 505-506), the space complexity can be reduced to linear and the time complexity to $O(2^n)$. The problem is, that the products are computed once for each evidence variable. This can be avoided by making use of the variable elimination algorithm that saves the calculations for later use. For singly connected networks or polytrees, this reduces time and space complexity to linear. For multiply connected networks, this still takes exponential time and space $O(n^2)$ (Russell and Norvig, 2003).

The most commonly used exact inference algorithms are clustering algorithms (also known as

clique tree or join tree algorithms) (Russell and Norvig, 2003) (Wattthayu, 2009). Clustering algorithms join individual nodes into a cluster in order to receive a polytree as in the example given by Russell and Norvig (2003) in figure 2.6.

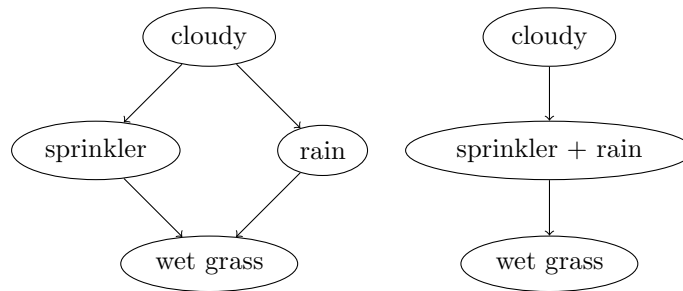


Figure 2.6: Example of a polytree (right) built from a multiply connected network (left)

With help of a constraint propagation algorithm on the polytree the query time can be reduced to $O(n)$. Although the problem remains NP-hard, because the construction of the network, variable elimination may still need exponential time and space, as stated by Russell and Norvig (2003).

One famous constraint propagation algorithm, named belief propagation, is a special case of the sum-product algorithm (see 2.2.4), introduced by Pearl (1982). By moralization and triangulation, the clique tree is constructed of a Bayesian network. Inference is computed by passing messages from each node of the tree to the other. The message sent is always based on the internal state as well as on the messages received (Wattthayu, 2009) (Jensen et al., 1990). For the purpose of this thesis that is dealing with Markov Logic (see section 2.3), multiply connected graphs are used in which belief propagation only delivers approximate results.

Approximate Algorithms

Before going into detail with belief propagation as discussed in the next section 2.2.4, another important class of approximate inference algorithms is briefly presented in order to give a complete overview on the subject. The prominent Markov logic system Alchemy use this category of algorithms as the default for inference in the Markov network.

Markov chain Monte Carlo (MCMC) is probably the most widely used category of approximate inference algorithm for Bayes networks as well as Markov networks (Gilks et al., 1996). MCMC algorithms use the idea of Markov chains in order to start from a sample and sample closer and closer to the desired target distribution.

Gibbs Sampling is a popular candidate of MCMC or more precisely random walk algorithms. Generally speaking, it generates a sample by forward sampling and then corrects itself by resam-

pling all of the unobserved variables and calculating the marginal probability distribution. The following code presents the Gibbs sampling algorithm (Koller and Friedman, 2009, pages 505-506).

```

1 function GibbsSample(
2    $X$  /* set of variables to be sampled */
3    $\phi$  /* set of factors defining factor distribution */
4    $P^{(0)}(X)$  /* initial state distribution */
5    $T$  /* number of time steps */
6 ) {
7   Sample  $x^{(0)}$  from  $P^{(0)}(X)$ 
8   for ( $t \leftarrow 1, \dots, T$ )
9      $x^{(t)} = x^{(t-1)}$ 
10    for ( $X_i \leftarrow X$ )
11      Sample  $x_i^{(t)}$  from  $P_\phi(X_i | x_{-i})$ 
12      Change  $X_i$  in  $x^{(t)}$ 
13   return  $x^{(0)}, \dots, x^{(T)}$ 
14 }
```

Listing 2.1: Gibbs Sample

In a Markov network, it randomly samples each variable given its Markov blanket¹) which in a node are the neighbors of the graph. By counting over the samples, the marginal probabilities are calculated (Domingos and Lowd, 2009).

As already stated for exact inference, belief propagation is another powerful and efficient algorithm if used in multiply connected networks like Markov networks. In the following section, a detailed insight to this algorithm is given.

2.2.4 A detailed look on Belief Propagation

Belief propagation is a highly efficient message-passing algorithm that performs exact inference on tree-structures Markov networks (Pearl, 1988). In loopy graphs, it may only return approximate results and may also not converge. In practice however, this convergence problem seldom occurs (Domingos and Lowd, 2009) (Bishop et al., 2006) (Yedidia et al., 2001). The more general category of belief propagation is called sum-product algorithms, which can most efficiently be implemented on factor graphs.

Factor graphs As pointed out earlier, Bayesian networks can be converted to Markov networks. In order to receive a graph that corresponds to the factorization as described in section 2.2.2, a factor graph is built representing the same directed or undirected graph as showed in figure 2.7 (example given by Bishop et al. (2006)). In addition to the variable nodes that are familiar from Bayesian and Markov networks, the factor graph also contains factor nodes.

¹Pearl (1988) defines a Markov blanket as the minimal set of nodes that are sufficient for forming it independent of the remaining network (Pearl, 1988, pages 120-121).

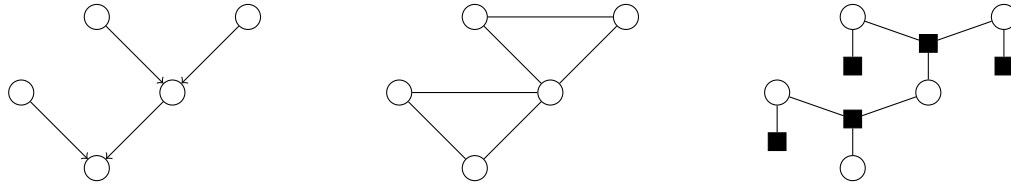


Figure 2.7: Conversion from directed polytree to undirected graph to factor graph

As an example, if there is an undirected graph with nodes say x_1, x_2, x_3 which is fully connected, a factor graph has a factor for the joint distribution $p(x) = f(x_1, x_2, x_3)$ or more specific factors $p(x) = f_a(x_1, x_2)f_b(x_1, x_3)f_c(x_2, x_3)$.

In this context, factor graphs are used to perform efficient inference algorithms for finding marginals and computation sharing. Having this in mind, the sum-product algorithm can be introduced.

The sum-product algorithm Bishop et al. (2006) explains, that calculating the marginal $p(x)$ for a specific variable node is done by summing the joint distribution over all variables except x leading to the following equation 2.9 where the variable x of the set of variables \mathbf{x} is omitted.

$$p(x) = \sum_{\mathbf{x} \setminus x} p(\mathbf{x}) \quad (2.9)$$

Substituting $p(\mathbf{x})$ using the factor expression $p(\mathbf{x}) = \prod_s f_s(\mathbf{x}_s)$ and interchanging summations and products, the joint distribution can be written as a product of the neighboring factor nodes $ne(x)$ with X_s being the set of all variables in the subtree connected to the variable x via the factor node f_s . $F_s(x, X_s)$ is the product of all factors that are associated with factor f_s .

$$p(\mathbf{x}) = \prod_{s \in ne(x)} F_s(x, X_s) \quad (2.10)$$

By substituting the two equations 2.9 and 2.10 to $p(x) = \prod_{s \in ne(x)} [\sum_{X_s} F_s(x, X_s)]$, resulting in $\prod_{s \in ne(x)} \mu_{f_s \rightarrow x}(x)$ where equation 2.11 defines the message from the factor node f_s to the variable node x , and $p(x)$ is the product of all incoming messages at variable node x .

$$\mu_{f_s \rightarrow x}(x) \equiv \sum_{X_s} F_s(x, X_s) \quad (2.11)$$

Because each factor $F_s(x, X_s)$ is itself described by a factor graph, it can again be factorized and substituted into the message from the factor to the variable, receiving the following equation (that gives the algorithm its name).

$$\mu_{f_s \rightarrow x}(x) = \sum_{x_1} \dots \sum_{x_M} f_s(x, x_1, \dots, x_M) \prod_{m \in ne(f_s) \setminus x} \mu_{x_m \rightarrow f_s}(x_m) \quad (2.12)$$

Now the message from variable nodes to factor nodes can be defined by equation 2.13 where $G_m(x_m, X_{sm})$ is a subfactor of $F_s(x, X_s)$.

$$\mu_{x_m \rightarrow f_s}(x_m) \equiv \sum_{X_{sm}} G_m(x_m, X_{sm}) \quad (2.13)$$

After having sent a message from a variable node to a factor node, it in turn sends a message itself to a variable node. So by substituting the factorization of $G_m(x_m, X_{sm}) = \prod_{l \in ne(x_m) \setminus f_s} F_l(x_m, X_{ml})$, the message from the variable node can be defined in terms of messages of the factor nodes.

$$\mu_{x_m \rightarrow f_s}(x_m) = \prod_{l \in ne(x_m) \setminus f_s} \left[\sum_{X_{ml}} F_l(x_m, X_{ml}) \right] = \prod_{l \in ne(x_m) \setminus f_s} \mu_{f_l \rightarrow x_m}(x_m) \quad (2.14)$$

This recursive calculation of the marginal for the variable node x is initialized with either a message value $\mu_{x \rightarrow f}(x) = 1$ for the variable node or a message value $\mu_{f \rightarrow x}(x) = f(x)$ for a factor node, depending on what kind of node the node is. The marginal for a distribution $p(\mathbf{x}_s)$ is finally given by equation 2.15.

$$p(\mathbf{x}_s) = f_s(\mathbf{x}_s) \prod_{i \in ne(f_s)} \mu_{x_i \rightarrow f_s}(x_i) \quad (2.15)$$

Since only in directed graphs the joint distribution is already correctly normalized, for undirected graphs this has to be done using an unknown normalization constant $1/Z$. To do so, sum-product is run in the unnormalized version of the joint probability distribution. In the end, the normalization constant is easily calculated by normalizing one of the marginals (Bishop et al., 2006, pages 399-411).

The idea behind belief propagation as introduced by Pearl (1982) was that it is only run on tree-like networks. Surprisingly, it also works well also on networks with loops and cycles where it is called loopy belief propagation and converges quickly (Murphy et al., 1999). However, in some cases it may also fail to converge (Yedidia et al., 2001).

Until now, all the reasoning methods used either a logic or probabilistic perspective. In the next section, Markov logic tries to unify both ideas.

2.3 Markov logic

This section deals with Markov Logic as a joint language between first-order logic and graphical models. In the first subsection, the general concept is described. After that, some inference systems that operate on Markov logic are presented.

2.3.1 The Markov Logic Language

Markov logic is an attempt to combine first-order logic and probability graphical models, in this case Markov networks, into a single probabilistic logic representation, called Markov logic networks [MLN]. For that purpose, formulas from first-order logic are taken and a weight (a real number) is assigned to every one of them (Richardson and Domingos, 2006, page 1). The definition of a Markov logic network (MLN) by Richardson and Domingos (2006) is "a set of pairs (F_i, w_i) , where F_i is a formula in first-order logic and w_i is a real number", like for example:

$$Friends(x, y) \wedge Friends(y, z) \Rightarrow Friends(x, z) \ 0.7 \quad (2.16)$$

$$Friends(x, y) \Leftrightarrow Smokes(x) \wedge Smokes(y) \ 1.1 \quad (2.17)$$

$$Smokes(x) \Rightarrow HasCancer(x) \ 1.5 \quad (2.18)$$

In these formulas 2.16, 2.17 and 2.18 friends of friends are friends with a weight of 0.7, friends have similar smoking habits with a weight of 1.1, and smoking may cause cancer with a weight of 1.5. In contrast to first-order logic, the intuition in Markov Logic is that states that violate a weighted formula are not completely impossible but only less probable. Also the weights behind the formulas are not necessarily between 0.0 and 1.0 as they would be intuitively when coming from probabilistic reasoning. Here, a weight describes the relevance of a formula. In the words of Richardson and Domingos (2006), "the higher the weight, the greater the difference in log probability between a world that satisfies the formula and one that does not, other things being equal".

In order to evaluate the truth value of a weighted formula in Markov logic, the Markov logic network needs to be grounded with constants as an interpretation of the used variables to a grounded Markov network. For every grounded predicate (e.g. $Smokes(Fritzli)$ and $HasCancer(Fritzli)$) and every grounded formula (e.g. $Smokes(Fritzli) \Rightarrow HasCancer(Fritzli)$) a node, that can either be true or false in the Markov network, exists. The graph in figure 2.8 shows the grounded Markov network for the example given by Domingos and Lowd (2009) with the formulas 2.17 and 2.18 and the constants Anna and Bob.

The probability distribution over possible worlds x is specified by the grounded Markov network. To handle the product in the joint distribution more conveniently, the Markov network can be stated as in the following equation 2.19 given by Domingos (2006). Here, F stands for the number

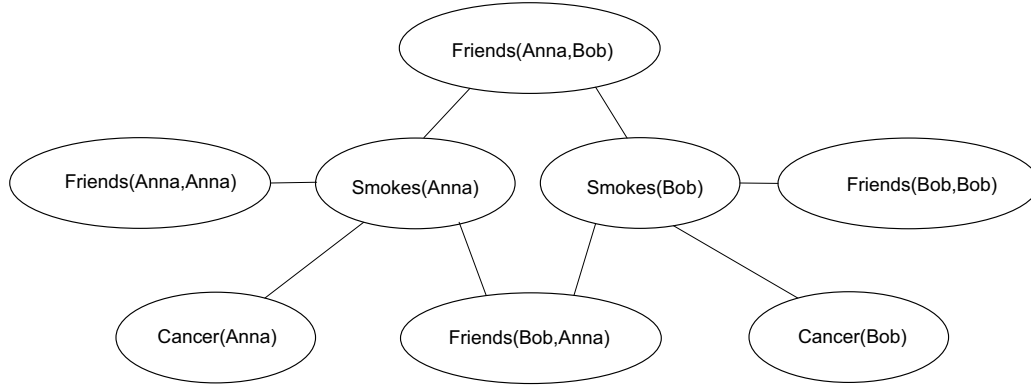


Figure 2.8: Ground Markov network for smokes example

of formulas in the MLN and $n_i(x)$ for the number of true groundings of F_i in x .

$$P(X = x) = \frac{1}{Z} \exp \left(\sum_{i=1}^F w_i n_i(x) \right) \quad (2.19)$$

In case of equal weights or if they tend to infinite, the MLN equals a traditional logic KB, generalizing first-order logic. Since contradictions between formulas are allowed, Domingos (2006) emphasizes, that Markov logic is extremely helpful in merging KBs. This problem is resolved by weighting the evidence on both sides.

Unfortunately as one can see, the ground Markov logic network for a domain with many constants can grow extremely large. However, the size can be significantly reduced by using typed constants and filtering the wrong types while grounding. But often, even this does not help. Some approaches to reduce this size are discussed later on in this thesis.

2.3.2 Inference in Markov Logic

Since Markov logic builds on Markov networks, the same algorithms can be used for inference. In the previous section 2.2.3, two prominent candidates for were presented, the MCMC algorithms like Gibbs sampling, or the class of sum-product algorithms including loopy belief propagation. These are also part of the implementation of the following inference systems.

2.3.3 Markov Logic inference systems

For Markov logic, there are already several applications implementing inference or learning. This section presents the two most popular inference engines as well as another system using them for weight learning in the Semantic Web. Alchemy, which is described first, is where all the Markov logic started. This system is later used in the benchmark evaluation of this thesis to measure the overall performance of LISy, the inference system presented in thesis.

Alchemy

Alchemy is an open-source Unix command-line system written in C++ that provides a collection of different Markov logic based algorithms for learning and inference. Its was developed by Kok et al. (2007) from the University of Washington. The available beta version covers discriminative weight learning (voted perceptron, conjugate gradient, newton's method), generative weight learning, structure learning, MAP/MPE inference and probabilistic inference (MC-SAT, Gibbs sampling, simulated tempering, belief propagation, lifted belief propagation).

In order to run inference or do learning, the Markov logic network is provided as predicates and functions with weights, types and constants in .mln files as in the example provided by Domingos and Lowd (2009) in listing 2.2.

```

1 // predicate declarations
2 Friends(person , person)
3 Smokes(person)
4 Cancer(person)
5
6 // If you smoke, you get cancer
7 1.5 Smokes(x) => Cancer(x)
8
9 // Friends have similar smoking habits
10 0.8 Friends(x,y) => (Smokes(x) <=> Smokes(y))

```

Listing 2.2: Example MLN file for friends and smokers domain

Ground atoms are defined in .db (see example by Domingos and Lowd (2009) in listing 2.3) files and can be defined as evidence (true, false) or unknown. The default is a closed world assumption, but also open world can be specified during inference.

```

1 Friends(Anna, Bob)
2 Friends(Bob, Anna)
3 Friends(Anna, Edward)
4 Friends(Edward, Anna)
5 Friends(Bob, Chris)
6 Friends(Chris, Bob)
7 Friends(Chris, Daniel)
8 Friends(Daniel, Chris)
9 Smokes(Anna)
10 Smokes(Bob)
11 Smokes(Edward)
12 Cancer(Anna)
13 Cancer(Edward)

```

Listing 2.3: Example .db file

To run inference on the system, the parameterized command is entered in the command line as for example in listing 2.4.

```

ALCHDIR/bin/infer -i smoking.mln -e smoking.db -r.smoking.results -q Smokes -ms -
maxSteps 20000

```

Listing 2.4: Run inference executable

The result in the output file then looks as in listing 2.5.

```

1 Smokes(Chris) 0.238926
2 Smokes(Daniel) 0.141286

```

Listing 2.5: Inference .result file

Alchemy also supports a lazy execution of most algorithms or a lifted version to avoid grounding the entire domain (Domingos and Lowd, 2009, pages 125-130). Thanks to the execution times it displays with every run, it can be easily used for benchmarking, as this will be done in the evaluation of this thesis.

PyMLN

PyMLN is a software package by Beetz and Jain (2010c) of the Intelligence Autonomous Systems (IAS) group TU München written in Python. It contains graphical tools for performing inference in MLNs and learning the parameters of MLNs, using the included PyMLN engine or the Alchemy system as underlying engine. It supports exact inference, MC-SAT as well as Gibbs sampling. Parameter learning works with several variants of maximum likelihood based on log-likelihood and pseudo-log-likelihood. A screen shot in figure 2.9 shows the MLN query tool of the PyMLN system, just to give an idea how other tools may look like.

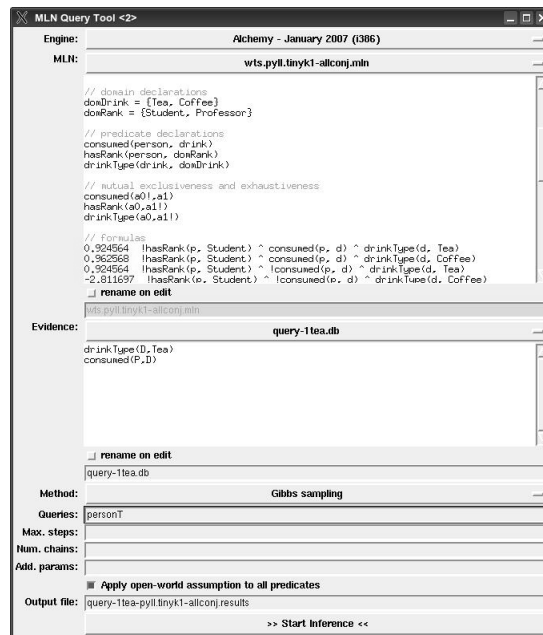


Figure 2.9: Screen shot of PyMLN query tool

The PyMLN package has been moved into the ProbCog Project by Beetz and Jain (2010a) where several graphical as well as inference and learning tools have been added, most of them written in Java. It now contains tools for Markov Logic that support MC-SAT, MaxWalkSAT, Toulbar2 Branch & Bound, as well as tools for Bayesian Logic Networks (BLN) with support for likelihood weighting, backward simulation, iterative join-graph propagation, SampleSearch, enumeration-ask. Also, graphical tools like a network editor in figure 2.10 are included (Beetz and Jain, 2010b).

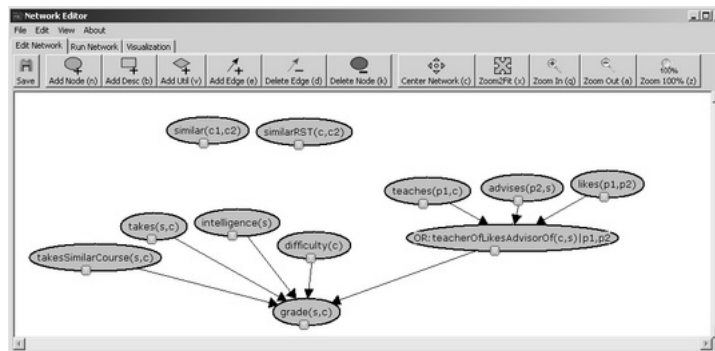


Figure 2.10: Screen shot of BNJ, a graphical editor for Bayesian and Bayesian logic networks

Incerto

Incerto is a probabilistic reasoner for the Semantic Web by de Oliveira and de Sousa Gomes (2009) and is written in Java. It uses the capabilities of Markov logic to learn and reason about uncertainty in OWL2 ontologies. As an underlying reasoning engine, both Alchemy and PyMLN are supported by Oliveira (2009).

The features declared on the Incerto website cover automatic weight learning of axioms uncertainty through analysis of ontology individuals, exact and approximate inference in weight-annotated ontologies and support for Alchemy and PyMLNs Markov logic engines. Further, Incerto supports OWL2, SWRL rules and first-order logic rules. Incerto comes with programmatic, graphical, and command line interfaces.

2.4 Knowledge representation

With the Internet as the largest representative of a knowledge base (KB) and the increasing possibilities in logical and probabilistic reasoning, the question how to handle the masses of data and how to make knowledge explicit and usable emerges. One of the key concepts to solve this issue is to make use of the Semantic Web.

2.4.1 The Semantic Web

The Semantic Web describes methods and technologies to allow machines to deal with the semantics of the World Wide Web (Berners-Lee et al., 2001), i.e. to infer meaning of objects. In order to describe concepts, terms and relationships within a knowledge domain, technologies like the Resource Description Framework (RDF), formats like RDF/XML or N3 as well as notations like the RDF Schema (RDFS), and most important for this thesis the Web Ontology language (OWL and OWL2) (Boris Motik, 2009, W3C), the Rule Interchange Format (RIF) (Axel Polleres, 2010, W3C) and the RDF query language SPARQL (Eric Prud'hommeaux, 2008, W3C) have been proposed by the W3C (Hitzler et al., 2007) (Herman, 2010, W3C). All of these are consistent, open standards, which is important for the success of this concept. Figure 2.11 captured from W3C website shows an overview the layered architecture of the Semantic Web (Berners-Lee et al., 2001).

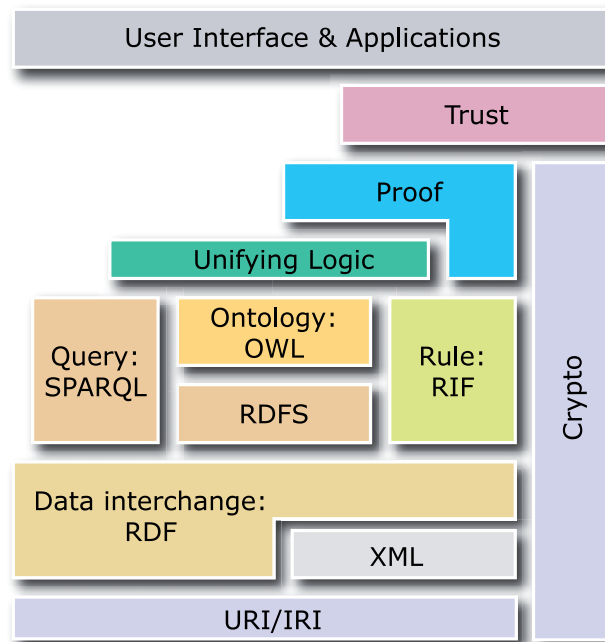


Figure 2.11: Layered architecture of the Semantic Web

In its lowest layer, Uniform Resource Identifiers (URI) represents the name or resource that is used, and consists of the Uniform Resource Locator (URL) and the Uniform Resource Name (URN) (URI Planning Interest Group, 2001).

2.4.2 Resource Description Framework RDF

In the second layer, RDF describes Information in a structured manner. The idea of RDF is to describe a directed graph from in which both nodes and edges (representing relations) use an URI each as an unique identifier.² This is best shown in an example as in figure 2.12.

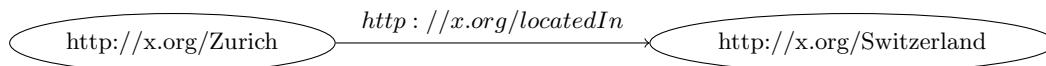


Figure 2.12: Example of a simple RDF graph that describes the location of Zurich

The three objects are called subject, predicate and object, describing an RDF triple. For serialization a number of different formats have been introduced like N3, N-triples, or RDF/XML (Hitzler et al., 2007), as for example in listing 2.6.

```

1 <?xml version="1.0" encoding="utf-8"?>
2 <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
3   xmlns:ex="http://x.org/">
4
5   <rdf:Description rdf:about="http://x.org/Zurich">
6     <ex:locatedIn>
7       <rdf:Description rdf:about="http://x.org/Switzerland">
8         </rdf:Description>
9       </ex:locatedIn>
10    </rdf:Description>
11 </rdf:RDF>
  
```

Listing 2.6: Example of an RDF serialization in RDF/XML

As seen above, document type and namespaces are declared first. Then, to describe statements, the tag `rdf:Description`, and for resources, the tag `rdf:about` is applied (Hitzler et al., 2007). To add values (literals) directly to RDF, datatypes are used, as for instance "Springer-Verlag" `xsd:string` as declared in the XML schema (Sperberg-McQueen and Thompson, 2000).

²Note that not every node needs to use an URI as there can also be blank nodes, or nodes with literals. But since these cases are not relevant for us at the moment, they will not be discussed here in further detail.

2.4.3 RDF Schema RDFS

In addition to the few defined properties in RDF, like `rdf:type`, there exists an extension to RDF called RDF Schema (RDFS) which can be found on top of the RDF layer. This additional vocabulary that is completely specified in RDF, can be used to declare the terminological knowledge like classes `rdf:class`, class relations `rdf:subClassOf`, properties `rdf:property` and property relations `rdf:subPropertyOf`. Other often used properties are literals `rdf:literal`, annotations `rdf:comment` and `rdf:label`, property restrictions `rdf:range` and `rdf:domain`. These RDFS properties can be used to build an ontology with individuals and class hierarchies, where an ontology is simply a knowledge base that models a domain (Hitzler et al., 2007), or, as defined by Gruber et al. (1993), "a formal specification of a shared conceptualization". An example of an ontology in RDFS is given by the following listing 2.7, in which Stefan and Carmen are individuals, belonging to the classes male respectively female.

```

1 <?xml version="1.0" encoding="utf-8"?>
2 <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
3   xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
4   xmlns:ex="http://x.org/">
5
6   <rdfs:Class rdf:about="http://x.org/person">
7     <rdfs:label xml:lang="en">person</rdfs:label>
8   </rdfs:Class>
9
10  <rdfs:Class rdf:about="http://x.org/male">
11    <rdfs:label xml:lang="en">male</rdfs:label>
12    <rdfs:subClassOf rdfs:resource="http://x.org/person">
13  </rdfs:Class>
14
15  <rdfs:Class rdf:about="http://x.org/female">
16    <rdfs:label xml:lang="en">female</rdfs:label>
17    <rdfs:subClassOf rdfs:resource="http://x.org/person">
18  </rdfs:Class>
19
20  <ex:male rdf:about="http://x.org/stefan">
21  <ex:female rdf:about="http://x.org/carmen">
22
23 </rdf:RDF>

```

Listing 2.7: Example of an Ontology in RDFS

2.4.4 Web Ontology Language OWL

Because of the limited expressiveness of RDFS, the W3C introduced OWL in 2004 and later in 2008 OWL2 (Boris Motik, 2009, W3C) as a standard ontology language. Meanwhile OWL has become quite popular with a large number of users and tools. OWL is a language with a large

expressiveness as well as efficient and scaling reasoning possibilities, and is based on predicate logic. To reduce complexity for the user, three sublanguages of OWL are defined: OWL Full, OWL DL and OWL Light (Hitzler et al., 2007).

1. OWL Full includes both OWL DL and OWL Lite as well as the entire RDFS. Therefore, it is highly expressive. The counter side is that the high expressiveness makes the language more complex. OWL Full is undecidable and also has some semantic aspects, that are problematic from the logical point of view.
2. OWL DL includes OWL Lite and is a sublanguage of OWL Full. In contrast to OWL Full, it is decidable and is therefore supported in many tools.
3. OWL Lite is a sublanguage of both OWL DL as well as OWL Full. It is decidable and less complex, but not as expressive as the other two.

Now with OWL, properties like the following examples can be expressed (Hitzler et al., 2007).

```

1 <owl:ObjectProperty rdf:about="membership">
2   <rdfs:domain rdf:resource="person"/>
3   <rdfs:range rdf:resource="organization"/>
4 </owl:ObjectProperty>
5
6 <owl:DatatypeProperty rdf:about="prename">
7   <rdfs:domain rdf:resource="person"/>
8   <rdfs:range rdf:resource="xsd:string"/>
9 </owl:DatatypeProperty>

```

Listing 2.8: Examples of OWL properties

The examples above only scratch the surface of the entire expressiveness of OWL. For instance, there are several classes constructors and relations like `owl:Class`, `owl:Thing`, `owl:disjointWith`, `owl:hasValue`, `owl:equivalentClass` and `owl:complementOf`, role properties like `owl:inverseOf` and `owl:sameAs` as well as datatypes Hitzler et al. (2007).

With the growing user community around OWL, new requirements have been incorporated into OWL2. There are three OWL2 profiles, each - as in OWL - a sublanguage in order to trade expressive power for the efficiency of reasoning (B. Motik, 2009):

1. OWL 2 EL is designed for the use with ontologies containing a large number of classes and properties. The expressive power is that of description logics EL (see (Baader et al., 2005)) that only provides existential quantifiers.
2. OWL 2 QL where QL stands for the ability to rewrite queries into a standard query language. It is used especially with large volumes of instance data.

3. OWL 2 RL stands for the ability to reason in standard Rule Language. OWL 2 RL has high expressive power and is built for reasoning applications.

In addition to the OWL 2 profiles, OWL 1 sublanguages can be expressed with OWL 2 (B. Motik, 2009).

A sophisticated graphical ontology editor and knowledge-base framework is Protégé by the (Mark Musen, 2010, Stanford University School of Medicine). It supports a variety of formats including RDF, RDFS and OWL as well as rules and reasoning.

2.4.5 Rule formats

In addition to representing primarily ontology data in OWL, the W3C has examined a significant number of rule formats like the Semantic Web Rule Language (SWRL) that uses horn clauses, RuleML and the brand new recommendation from June 2010, the Rule Interchange Format (RIF). Although it comes a little too late in order to be incorporated in the implementation part of this thesis, it would be a good idea to enable a follower version of LISy to read rules in RIF, or also in SWRL and RuleML.

2.5 Signal/Collect

Signal/Collect is a scalable programming model for running various algorithms on typed graphs developed by Stutz et al. (2010) of the DDIS Group at University of Zurich. In addition to the programming model, a Signal/Collect framework has been built, that eases the implementation of message passing algorithms substantially. As it is written in Scala, it runs on the Java Virtual Machine.

The principle of the model is that computations on the Semantic Web usually involve sending information between the resources (which depict vertices) over the properties (edges) of the graph. The target vertex then collects the messages, and may do some computation like, for instance, to update the state. This idea of signaling and collecting messages is illustrated in the following figure 2.13.

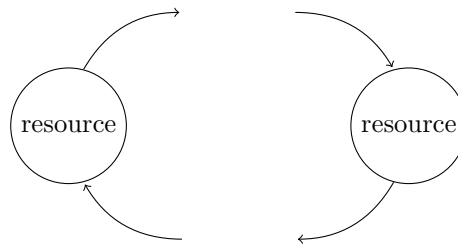


Figure 2.13: Illustration of the Signal/Collect model

Since many algorithms like page rank, RDFS subclass inference, and Bellman-Ford pass messages over their edges, it is perfectly suitable as a generic programming model. In this thesis, Signal/Collect is used to implement belief propagation in an efficient way.

The framework itself contains a collection of predefined features. For instance, it can construct the network by fetching data from a SPARQL endpoint. By defining a SPARQL query to retrieve the source and target predicates that fully specifies the edge, the Signal/Collect framework handles the construction of the network it runs on. The algorithm can then be executed in parallel by assigning a number of workers.

The compute graph, on which Signal/Collect builds, is defined as a tuple $G = (V, E)$ where V is the set of vertices, and E the set of edges in G . Furthermore, there are four core attributes defined on every vertex:

- **v.id:** unique id of the vertex
- **v.state:** current state representing computational intermediate results
- **v.signals:** most recent signals received from neighboring vertices as a map

- **v.uncollectedSignals:** list of arrived signals with signals all since the last collect operation was executed on the vertex

The edges are all directed, consisting of a source, a target and the following attributes:

- **e.sourceId:** source vertex id
- **e.targetId:** target vertex id
- **e.currentSignal:** last signal sent along the edge

To implement the algorithm of choice on Signal/Collect, the framework provides already pre-defined methods. Other methods like the two main tasks *signal* and *collect* need to be implemented separately. The standard methods **v.signal** and **v.collect**, as well as the method **v.initialState** are therefore overridden with custom code:

- **G.initialState: vertexId → state:** returns the initial value for a vertex for a given vertexId
- **G.signal: edge → signal:** returns the signal to be sent along a specified edge
- **G.collect: vertex → state:** returns the new vertex state for a specified vertex

To calculate and print the results, the method **v.processResults** needs to be implemented. Another predefined method that can be overridden is **v.scoreSignal**, which enables the developer to guide the algorithm where necessary, using a score. Like this, asynchronous algorithms can be easily developed.

After implementing the algorithm with the provided skeleton functions, Signal/Collect can be executed by calling **computeGraph.execute**.

Having all the conceptual utilities at hand, requirements for the Markov Logic inference system can be gathered.

3

System Requirements

As declared in the introduction of this thesis, the goal is to create infrastructure that allows to do inference on Markov Logic formulas by running belief propagation on the Signal/Collect framework. The resulting system will be called Logic Inference System, or LISy for short.

In order to meet the requirements, a specification of the system is presented hereafter, using an id for every requirement. In addition, every requirement receives one marker included in the id, depending on the priority. The two categories are either "must have" with marker **M** and "nice to have" with **N**. Hence the id looks something like **M01**.

3.1 Functional requirements

The *key functionality* of the system is to run the loopy belief propagation inference algorithm on Markov logic. With the grounding of Markov Logic formulas with data from an RDF database, a Markov Logic Network is built. The result of the loopy belief propagation algorithm should then deliver correct (approximate) results on the belief of the inference task.

Data requirements

- **M01:** The data for evidence and potential grounding of the Markov Logic formulas are pulled from an RDF database via a SPARQL endpoint. The predicate query therefore requires fetching all individuals that are `rdf:type` of any `rdfs:domain` or `rdfs:range`. Furthermore, duplicate values need to be filtered. For evidence query should retrieve all elements with any property and bind it to predicates.
- **N02:** Support for Alchemy-compatible parsing of `.mln` and `.db` files in order to use one or more for inference.

Logic requirements

- **M03:** The system supports the use of rules as formulas in Markov logic as defined in the EBNF of first-order logic with an additional weight.
- **M04:** Existential quantification is supported.
- **N05:** The equality predicate is supported.

Network requirements

- **M07:** Markov Logic Formulas can be compiled into a grounded Markov logic network, using the loaded data.

Inference requirements

- **M08:** Loopy belief propagation as an implementation of a inference algorithm can be run on a grounded Markov Logic Network using the Signal/Collect programming model.
- **M09:** The messages sent in belief propagation over the Markov logic network are as describes by Domingos and Lowd (2009) in figure 3.1. There is a message from each variable representing a grounded predicate to the corresponding factor representing the grounded formula where they are linked with an edge. In the next step, the messages are sent in the opposite direction.

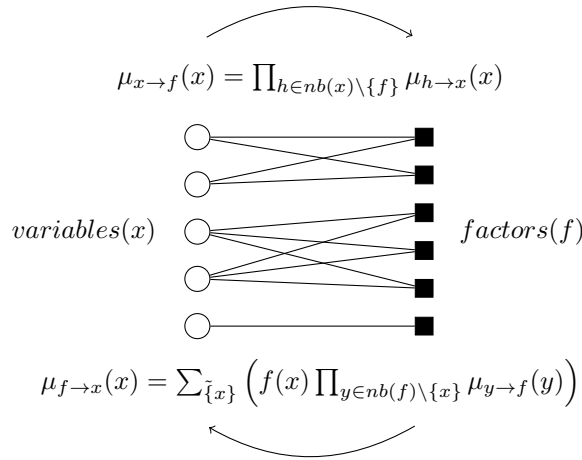


Figure 3.1: Belief propagation in a factor graph

- **M10:** The messages from variables to factors are initialized with the multiplicative identity element.
- **M11:** The factor f is calculated with an exponential function $f_i(x) = \exp(w_i g_i(x))$ per feature g_i .

- **M12:** Features over the same variables are not aggregated into a single factor.
- **M13:** Evidence from the database is incorporated by setting $f(\mathbf{x}) = 0$ for states \mathbf{x} that are incompatible with it.
- **N14:** RDFS sub-class inference is supported.
- **N15:** Special treatment to the !-operator (at least 1 and at most 1).

3.2 Nonfunctional requirements

This section includes all nonfunctional requirements referring to performance and other qualities, as well as constraints.

3.2.1 Attributes

Performance requirements

- **M16:** The inference algorithm can be run in parallel.

Special Qualities

- **M17:** Developers can easily integrate Markov Logic Formulas in the system.
- **N18:** The system should have the characteristics of a framework.

3.2.2 Constraints

Technology requirements

- **M19:** For the storage and query of triples, OpenRDF Sesame (Aduna, 2010) is used. Sesame runs on Apache Tomcat (Apache, 2010) which is a servlet container for Java Servlets and Java Server Pages (JSP). Furthermore, it provides a pure Java HTTP web server environment for Java code.
- **M20:** The query language used to retrieve the nodes and edges from the triple store is SPARQL.
- **M21:** The system runs on the Java Virtual Machine and is therefore written in either Scala or Java.
- **M22:** For the implementation, the Signal/Collect framework is used.

Development process requirements

- **M23:** During development, the revisions are regularly committed to the SVN server.
- **M24:** The classes and methods are documented using JavaDoc.

Based on the specified requirements, the system can be designed.

4

System Design and Implementation

This chapter builds on the requirements that were defined previously in chapter 3. In the following the system design decisions are made and the implementation of the Logic Inference System LISy is discussed in detail.

4.1 Architecture

LISy is built in a multi-tier approach using multiple layers as can be seen in 4.1 in order to separate concerns. It is built to conceptually match the layered architecture of the Semantic Web by building a network on top of single data elements. On this network, rules can be formulated and reasoned by them.

As the goal of LISy is to establish infrastructure for Markov logic applications, it is built itself as part of a multi-tier. LISy can be integrated in new applications by adding a business and application layer on top.

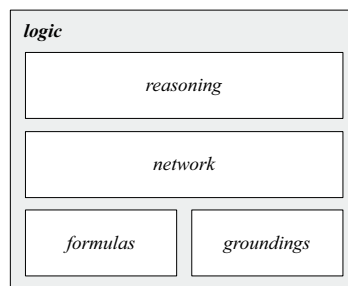


Figure 4.1: Logic layers

4.2 Markov Logic formulas

In order to ease the use of Markov Logic formulas for developers, formulas can be entered in Scala. This simplifies the later development of parsers, that import Markov Logic formulas from external files like the Alchemy .mln file format. Since Scala offers an elegant way of defining functions with all kinds of characters, the formulas can be written in the form given below and assigned to a variable of the type `Formula`:

```
val f1: Formula = "Smokes"~(x) -> "HasCancer"~(x) w 1.5
```

or even

```
val f2: Formula =  $\forall(x, \exists(y, \text{"Loves"~}(x,y) \wedge \text{"Loves"~}(y,x)))$  w 2
```

The strings followed by variables in brackets (`"Loves"~(x,y)`) are assigned automatically to a predicate and the weight after the `w` is assigned to the formula. The following symbols are used to create valid formulas: \forall or `forall`, \exists or `exists`, \neg or `not`, \wedge , \wedge or `And`, \vee , \vee or `Or`, \rightarrow , \rightarrow , or `Implies`, \leftrightarrow , \leftrightarrow or `Equivalent`, and finally `w`. Thus, all elements from first-order logic and Markov Logic are valid in this implementation.

By building the hierarchical structure of the formula consistently by recursion as shown figure 4.2, it can be kept for later reuse as for example the easy reconstruction of the string representation or the evaluation of the formula in first-order logic.

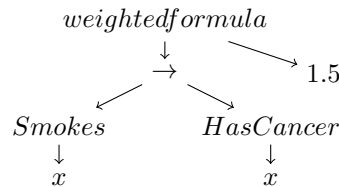


Figure 4.2: Example of a hierarchically defined formula.

The class structure can be similarly constructed, as presented in the class diagram of figure 4.3. The artificial seeming class `BinaryOperatorFormula` is used to ease the handling of the corresponding formulas and is therefore kept abstract, and hence can not be instantiated.

After having defined one or several formulas, they are added to the Markov logic network (MLN) by applying the `addFormula` method of an MLN object of type `MarkovLogicNetwork`.

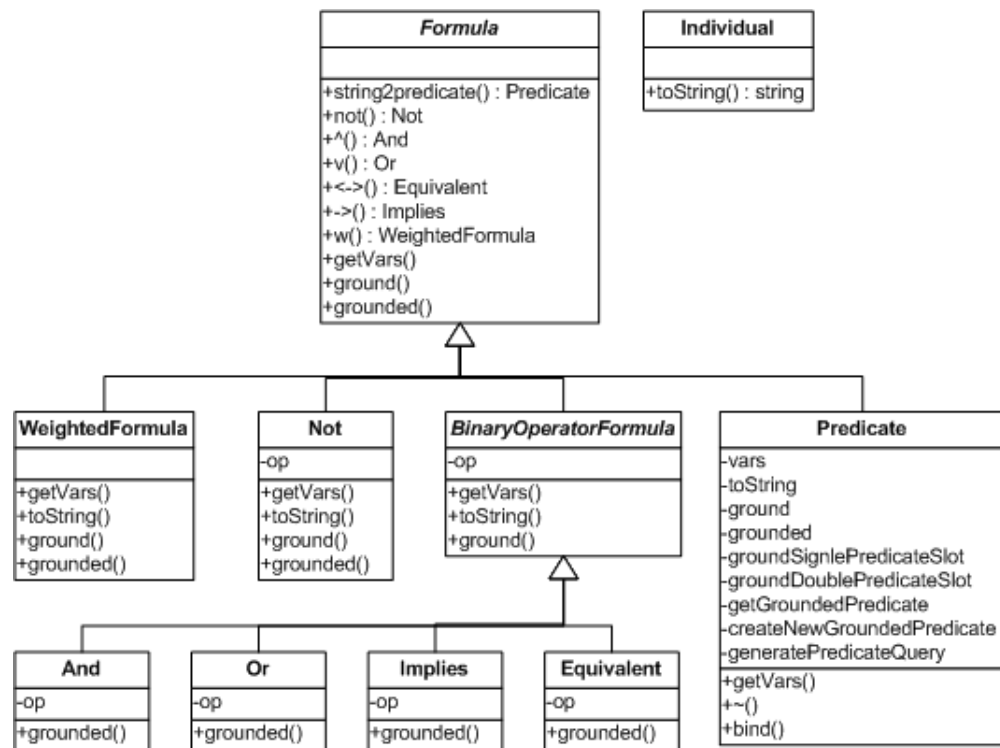


Figure 4.3: Class structure of Formula hierarchy in Markov Logic

4.3 Network grounding

Now with all formulas formulated, the MLN can be grounded with individuals of type **Individual** by running the `ground(db)` method of the MLN which in turn grounds all the formulas recursively as can be seen in figure 4.4, similar to the formula hierarchy. This configuration brings about, that the `rootFormula` attribute has to be set, in order to remember where the formula started.

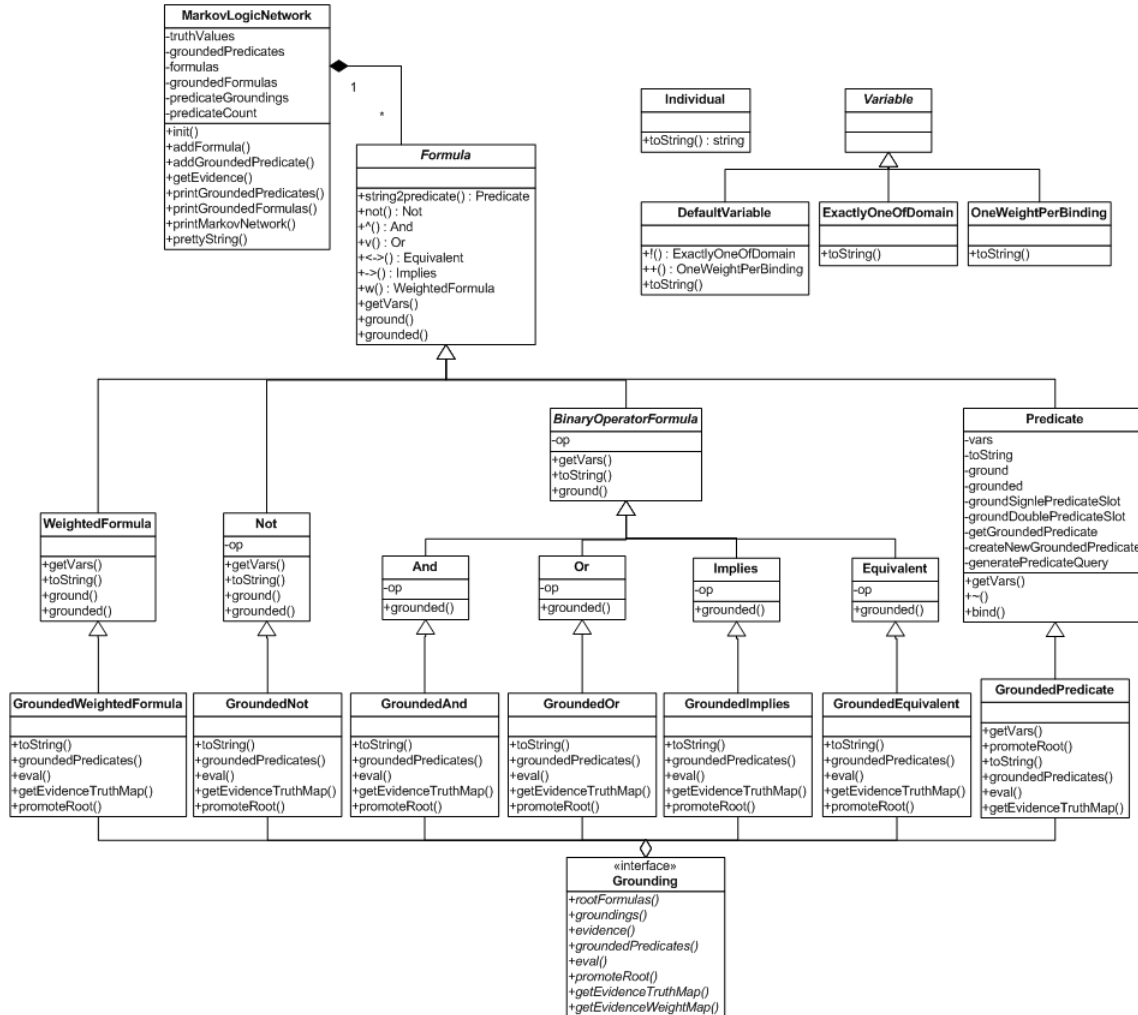


Figure 4.4: Implementation of grounded formulas

A grounded formula or predicate is a **Formula** extended with the trait **Grounding**. The concept therefore is described in the class diagram 4.5. It uses a mapping between variables and individuals, that are queried from the database given to the method by parameter.

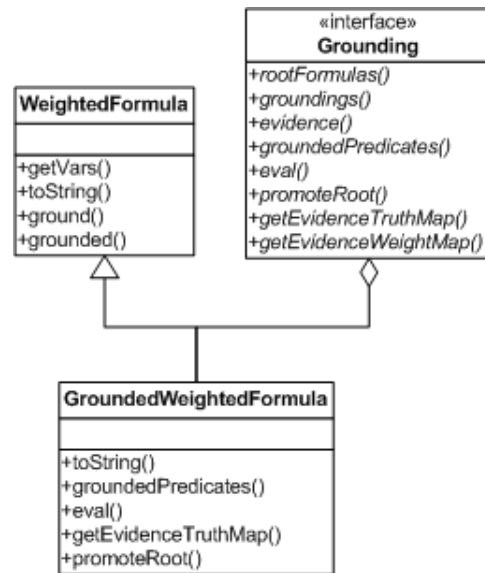


Figure 4.5: Grounded formulas built from formulas with grounding

Since the network is grounded with RDF triples from the Sesame triple store, the database is the abstract **SparqlAccessor** of the Signal/Collect framework for Scala by Stutz et al. (2010). It is instantiated as a **SesameSparql** object, that also comes with the SignalCollect framework, with the two parameters database URL and namespace each as string. The deployment diagram 4.6 below shows how the system is connected to OpenRDF Sesame implementation of a **SparqlAccessor**. Since **SparqlAccessor** is an abstract concept, any other triple store could be used as a data source, as for example Jena¹.

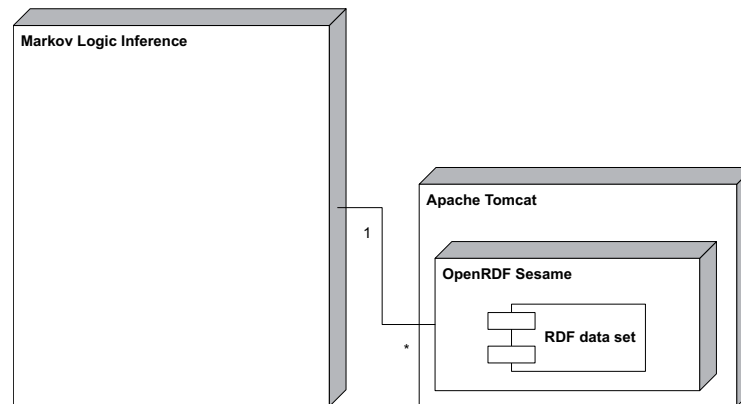


Figure 4.6: Deployment diagram

¹Jena is a semantic web framework for Java that provides RDF(S), OWL and SPARQL support. It is available unter <http://jena.sourceforge.net>.

On creation, the `GroundedPredicates` and the `Formula with Grounding` assign themselves to the MLN.

After the grounding of the network is complete, evidence in the MLN is marked. Evidence, in the terminology used in this thesis, is a known fact about a predicate in the MLN. The method that retrieves evidence is `getEvidence(db)`, which needs access to a `SparqlAccessor`, and then fetches the evidence by using a simple SPARQL query. It assigns a truth value of type `TruthValue`, which can either be true, false or unknown. For each of these, an object (`TRUE`, `FALSE`, `UNKNOWN`, see figure 4.7) exists, that eases comparison later on.

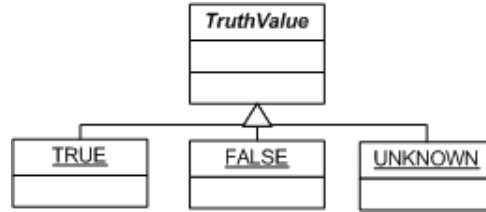


Figure 4.7: TruthValues for Evidence

Because usually in order to do inference on the network, both, grounding and assigning evidence has to be done, there exists a method `init(db)`, that covers all the preparation processes.

4.4 Running loopy belief propagation

With the complete grounded network including evidence, it is ready to run inference algorithms². In this case it will be loopy belief propagation. Since `Signal/Collect` offers all the needed functionality to do this kind of message passing algorithm, it makes sense to implement loopy belief propagation with this framework.

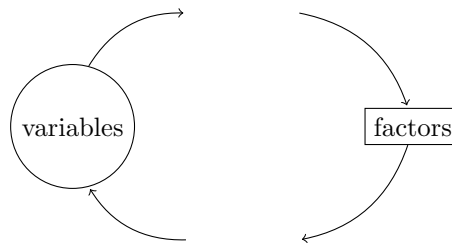


Figure 4.8: Two vertex types for belief propagation

Like specified in section 2.5, in order to run belief propagation on `Signal/Collect`, the `ComputeSubgraph`

²the fun stuff.

has to be defined by vertices and edges. When running loopy belief propagation on a factor graph, the vertices are twofold as shown in figure 4.8. Of course, variable nodes represent `GroundedPredicates` and factor nodes are of type `Formula with Grounding` (without grounded predicates), which can be any `Formula` from `GroundedNot` to `GroundedWeightedFormula` where the `rootFormula` attribute is set.

This results in separate implementations for variable and factor nodes in Signal/Collect, but with matching edges in order to form a graph without having edges pointing to nirvana. They are called `BeliefPropagationVariable` and `BeliefPropagationFactor`. Since in the framework vertices are defined over their edges, it is sufficient to only create them, which can be easily done by loading the `MarkovVariableEdges` and the `MarkovFactorEdges` as can be seen in figure 4.9.

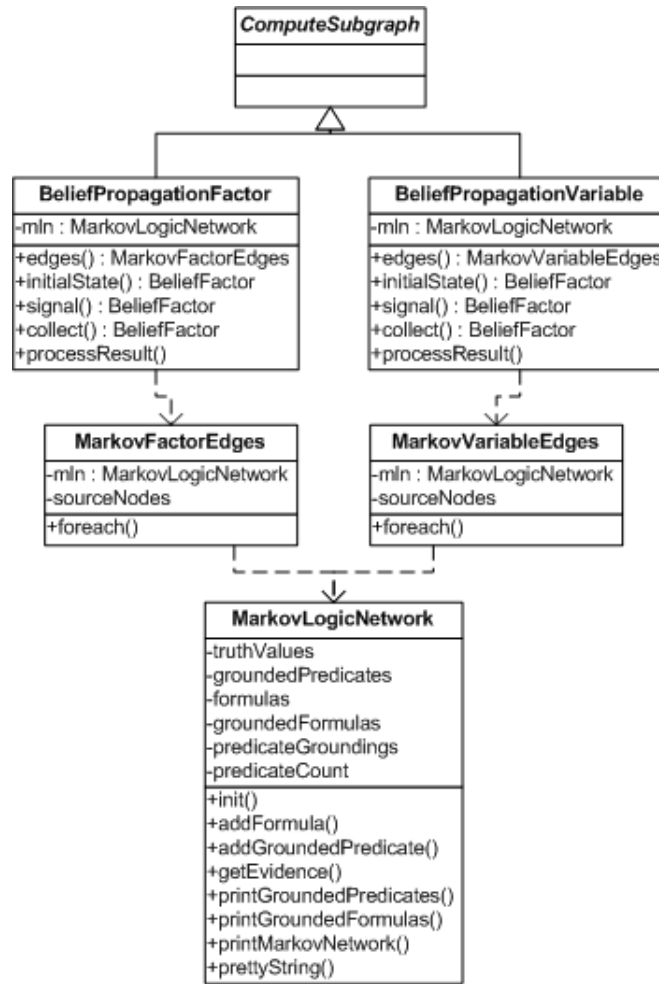


Figure 4.9: Belief propagation with Signal/Collect

Slightly more complicated is the design of joint distributions. For belief propagation, the truth table of every formula has to be created resulting in a function $g(x)$. To incorporate weight, the values of $g(x)$ are converted to real numbers between 0 (false) and 1 (true), then multiplied by the weight and finally exponentiated, resulting in $f(x)$.

Conveniently, trait **Grounding** offers a method `getEvidenceTruthMap` which can be seen as creating the truth table for the grounded formula. This method runs recursively, returning the truth map as type **TruthFactor**. To efficiently produce the truth map, the logical operators are defined in the type itself.

Also defined in trait **Grounding** is the method `getEvidenceWeightMap`. This method handles weighting of evidence, resulting in a weight map of type **BeliefFactor**. Similar to **TruthFactor**, it offers certain operations like the multiplication, addition as well as the inversion, which are used for collecting and signaling. Also for later use the method `marginalize` and `normalize` are provided. As a special instance of **BeliefFactor**, the **MultiplicativeIdentity** is provided. Figure 4.10 depict the different types of **Factors**.

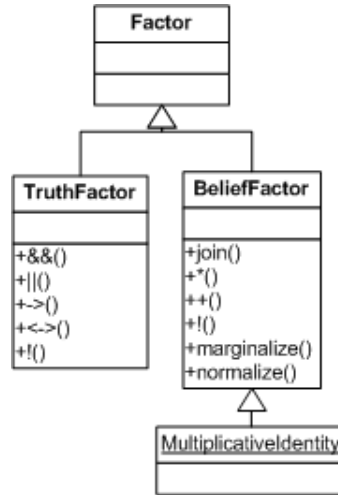


Figure 4.10: TruthFactor and BeliefFactor in Markov Logic Network

By using **Factors** as basic representation of the vertex state and for messages, loopy belief propagation can easily be applied to Signal/Collect. In the first node type, which is the **BeliefPropagationVariable**, the following methods need to be implemented:

- `initialState(vertexId:Formula with Grounding)`: the initial state is set to the multiplicative identity
- `edges`: the edges are loaded from the **MarkovVariableEdges** by passing the grounded MLN

as parameter

- `scoreSignal(v:Vertex)`: the vertex with the highest difference in state is handled first
- `signal(e:Edge)`: the normalized product of all signals are sent to the target (excluding the target contribution)
- `scoreCollect(v.Vertex)`: the vertex with the most uncollected messages is handled first
- `collect(v:Vertex)`: the signals of the vertex multiplied resulting in the new state
- `processResult(v:Vertex)`: the vertex state is normalized and printed for every grounded predicate

The other vertex type `BeliefPropagationFactor` is implemented similarly:

- `initialState(vertexId:Formula with Grounding)`: the initial state is set to the multiplicative identity
- `edges`: the edges are loaded from the `MarkovFactorEdges` by passing the grounded MLN as parameter
- `scoreSignal(v:Vertex)`: the vertex with the highest difference in state is handled first
- `signal(e:Edge)`: the signal is the marginalized product of the weight map times the joint distribution of all incoming message except the target contribution
- `scoreCollect(v.Vertex)`: the vertex with the most uncollected messages is handled first
- `collect(v:Vertex)`: the new state is the product of the weight map times the joint distribution of all the collected signals

Putting this all together, the algorithm can be executed by adding all the vertices to a new `ComputeGraph` and running the `execute` method. As a result, the inferred values for all the grounded predicates are listed.

Summing up, the system is built across three layers (see figure 4.1), conceptually matching the layered architecture of the Semantic Web. The first layer containing formulas and groundings represents the data, merging into a network in the second layer. On top of that, the algorithm can be applied.

Of course, this architecture leaves space to implement further algorithms on top of the Markov logic network later on, or even implement other network types like the Bayesian network.

5

Evaluation

Having implemented LISy through all its three layers from the formulas over the network to the belief propagation algorithm, the system can be evaluated. The first claim holds already: Namely the successful implementation of a complex algorithm (i.e. loopy belief propagation) on Signal/-Collect returning valid inferred values as a proof of concept. The fulfillment of the other objectives, which is the efficiency of the implementation as well as the infrastructure provided with LISy, follow beneath.

In the first section, a benchmark test is performed in order to compare the reasoning efficiency of the Logic Inference System (LISy) with Alchemy. In the second section, it is evaluated, how well the implementation scales on multiple processors.

The data sets utilized for this evaluation are built on the basis of the "friends smoke" example by Domingos and Lowd (2009). Since the interest lies on a large number of groundings as they are typical in Semantic Web applications, different datasets with a varying number of individuals are used. The largest dataset "largesmokes" has 300, the second "mediumsmokes" 100, the third "smallsmokes" 50 and "tinysmokes" only 5 individuals.

5.1 Benchmark test with LISy and Alchemy

In order to evaluate the overall performance of the system, LISy is compared with the most popular Markov logic inference system available, Alchemy, in a benchmark test. Therefore, a number of tests are run to compare the time for grounding (building and grounding the Markov logic network), initializing (loading) the network, the execution time of belief propagation as well as the overall run time.

5.1.1 Test environment

All of the following tests are run on a iMac OSX 10.6.4 with two 2.16 GHz Intel Core 2 Duo and 4 GB 667 MHz DDR2 SDRAM.

5.1.2 Configuration

The naming of the test indicates the number of workers assigned (e.g. A.2 for two workers), as well as if the test is run with or without scoring (e.g. AS.1 for scoring).

- **A:** Simple execution of BP for the trivial formula $smokes(x) \text{ w } 0.8$ and with 300 individuals for grounding, resulting in 300 grounded predicates and 300 grounded formulas. Test A.1 is run over 100 times on Lisy and 20 times on Alchemy.
- **B:** Execution of BP for the slightly more complex formula $smokes(x) \rightarrow cancer(x) \text{ w } 0.8$, resulting in 600 grounded predicates and 300 grounded formulas.
- **C:** In order to bloat up to the network for BP, C.1 uses already two variables in formula $smokes(x) \rightarrow cancer(y) \text{ w } 0.8$, resulting in 600 grounded predicates and 90'000 grounded formulas. The test is repeated only 10 times for both, Lisy and Alchemy.
- **D:** BP is now executed for the formula $friendOf(x, y) \rightarrow (smokes(x) \wedge smokes(y)) \text{ w } 0.8$. For the data set with 50 individuals, the network with 10'000 grounded predicates and 10'000 grounded formulas, and is run 10 times for both, Lisy and Alchemy.
- **E:** To achieve even more grounded predicates, the formula $friendOf(x, y) \rightarrow (smokes(x) \rightarrow hasCancer(y)) \text{ w } 0.8$ is evaluated with, again, 10 runs for both systems.
- **F:** Test F finally uses a third variable which results in an even higher number of grounded formulas. Here, the formula $(smokes(x) \wedge smokes(y)) \rightarrow hasCancer(z) \text{ w } 0.8$ is executed 10 times on both systems.

Of course, Alchemy was initialized with the belief propagation parameter `-bp`. In order to compare the same algorithm, lifting was not enabled in Alchemy.

5.1.3 Results and conclusion

The results of the benchmark tests A.1 to CS.2 are presented in the following figures 5.1 through 5.3, each showing a box plot of the tests with the minimum, lower quantile (Q1), median (Q2), upper quantile (Q3) and the maximum. The vertical axis shows the execution time in milliseconds for each test on the horizontal axis.

The trivial benchmark test A shows already that the main goal of reducing execution time was impressively successful indeed by running loopy belief propagation on the Signal/Collect. In

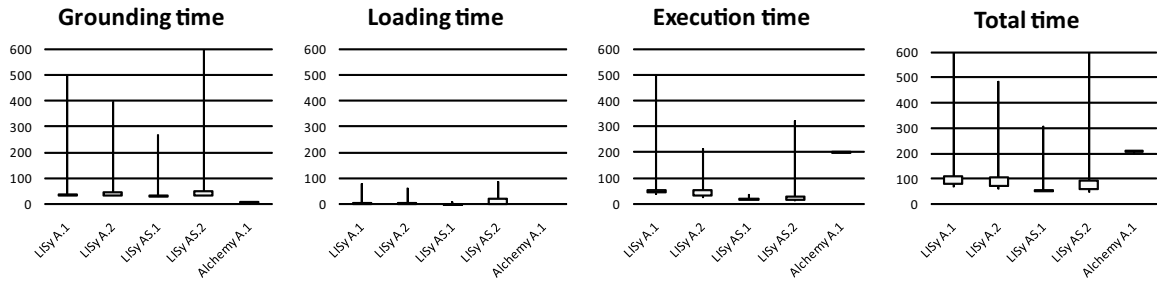


Figure 5.1: Results of tests A

comparison to running the test on Alchemy, LISy was 14.3 times faster in execution. Due to the short execution time of a couple of milliseconds, LISy only gains little by using more than one worker thread, and the example is too simple to really gain from scoring.

The grounding times for LISy, however, remain slightly longer because of the more complex retrieval of data from the Sesame RDF store. In contrast to Alchemy, which loads the data from its native *.mln and *.db files directly into its own data structure, LISy converts the grounded MLN into the edges that define the compute graph of Signal/Collect. Since defining the edges by a source and a target vertex is in the LISy implementation by using two for-loops, the loading time is extraordinarily high, resulting in a slightly longer total run time for LISy. But still, LISy beats Alchemy in its best run by being 4.2 times faster. The outliers visible in the box plot most certainly result from garbage collection as only the first couple of runs are effected.

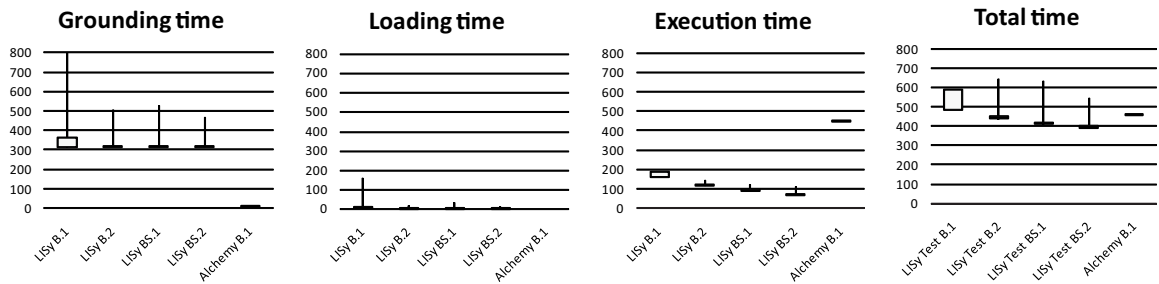


Figure 5.2: Results of tests B

Benchmark test B shows a similar picture with the fastest execution being 6.3 times faster with LISy than with Alchemy. But again, the grounding and loading of the network in LISy shoots down the head start a little, but still resulting in a slightly better run time in total.

It seems that the use of specialized scoring function in loopy belief propagation already shows a

gain in performance although it does not seem too significant, yet due to the fact, that the formula is still very simple. The main reason for the short execution time is that the algorithm already converges really fast on Signal/Collect due to the asynchronous execution as can be anticipated for the very low amount of signals sent (for 900 vertices, only 2400 signals are sent).

In test C, the problem becomes more complex as a second variable is introduced. This can clearly be seen with the long grounding times for the network, as well as the loading times of the edges of the compute graph in Signal/Collect. The problem is, that the Sesame server is already combining all the possible individuals for the later grounding. But this task could probably be done with lower effort if only the individuals and the domain restrictions were received, resulting in a lower transmission but higher loading time.

But although the initial grounding and loading time is very long, the execution converges again extremely fast, letting LISy win the race about execution time hands down. Alchemy was stopped by default after 1000 iterations with no convergence, where it is clear to see that scoring makes a huge difference in letting the algorithm converge quickly. As the intuitive objection may be, that Alchemy uses a lower threshold, this intuition can be proved wrong.

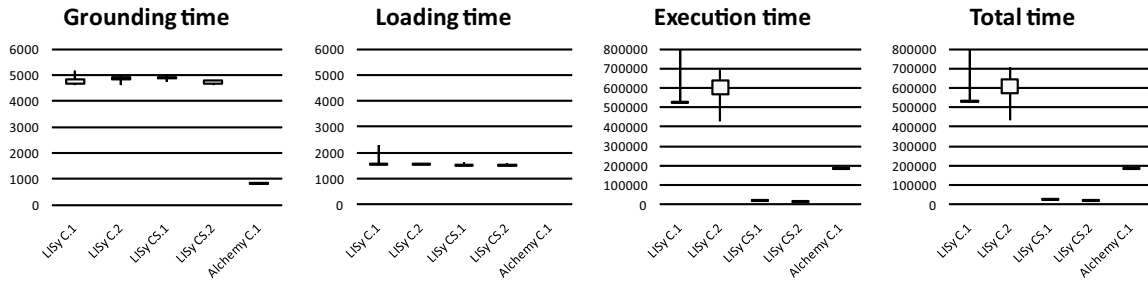


Figure 5.3: Results of tests C

5.2 Scalability of LISy

In order to evaluate the transparent scalability of the algorithm implementation, the test A to C from above have been run with a CPU core for each worker thread.

5.2.1 Test environment

The computer used for this has two quad-core X5570 CPU and 72 GB RAM. Since the network would never use 72 GB, the JVM was initialized with 42 GB in order to never reach the limit but

also not to slowdown the system.

5.2.2 Configuration

Each test was run again 100 times, measuring the time for network creation and load, for algorithm execution as well as the total time.

5.2.3 Results and conclusion

Unfortunately, the algorithm execution in LISy does not seem to scale linearly as we expected from the evaluation of Stutz et al. (2010). This fact can be seen by looking at figure 5.4, in which the number of threads are marked on the horizontal and the performance scale on the vertical axis. The measured speedup the average scalability of the test A to C made in the evaluation.

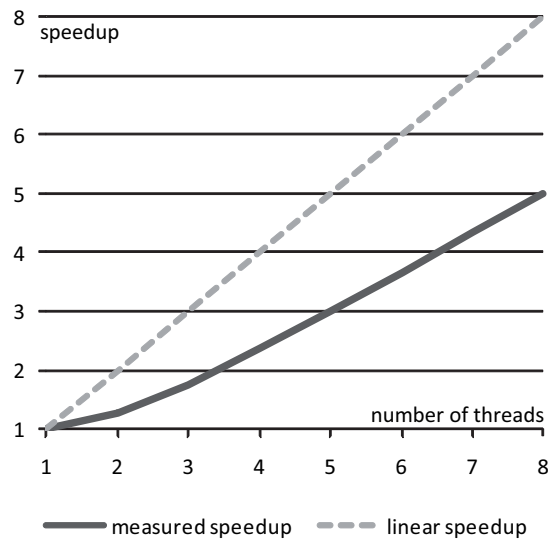


Figure 5.4: Speedup with additional threads

The reasons therefore have to do with the characteristics of the Markov network in the examples. While the graph used for Bellman-Ford by Stutz et al. (2010) consists of 1 million vertices and 94 million edges, the Markov networks in this example is less cross-linked. Therefore, asynchronous scoring does not affect the convergence as much as it did in the Bellman-Ford example. The different scalabilities of the algorithm can be seen in figure 5.5 below.

Overall, the results gained with the evaluation of LISy are very much satisfying. First, it could

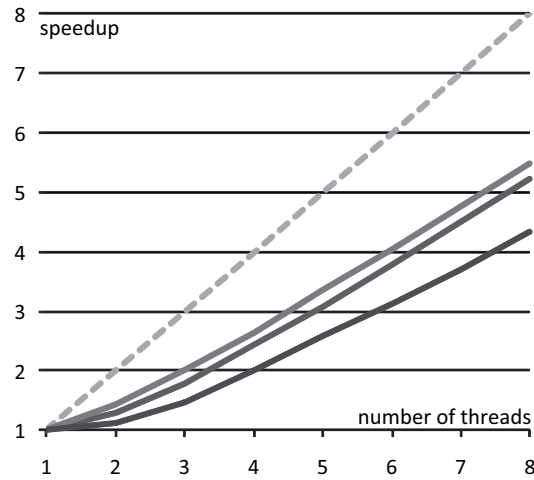


Figure 5.5: Different scalability per network

be shown that loopy belief propagation can be applied to the Signal/Collect programming model, proofing the concept. Second, the system performs very well despite that it needs to fetch the data from an RDF store, especially on the algorithm. LISy even outperforms the popular Markov logic reasoning and learning engine Alchemy when it comes to belief propagation due to its scoring capability and asynchronous message flow. Although the scalability is not exactly as good as expected, it is still fairly good none the less. Possible threats to validity as discussed in the next chapter 6.

6

Limitations

Whereas the project of this thesis was in many belongings a success, there usually are limitation that narrow the achievements. Some limitations in both, evaluation but also system design, are inevitable like physical limitations, but of course there is always something that can be done better or that will be subject to further research and development.

These limitations and learnings are subject of the following sections. First, some of the constraints on the evaluation process of the previous chapter are discussed, then further limitations one the system are revealed.

6.1 Limitations of the evaluation process

One of the most important limitations is the limited variety of datasets used for evaluating the system. The predicates used were only `friendsOf`, `smokes` and `hasCancer`. Of course, with these predicates, a complete network with different characteristics can be built, but testing the algorithms on real world data like with GeneID (University, 2010) or an interconnection between several datasets available would have given the evaluation a higher relevance and therefore higher credibility. Also, the graphs may scale differently because of different characteristics of applications in practice.

In coherence to this, the fact that only 600 individuals were grounded at the most for the creation of the Markov network, which is in terms of Semantic Web data a very limited number, the system could not be evaluated exactly for the application domain. Of course, with a million individuals for grounding, even the simplest formula would lead to combinatoric explosions. A social graph with friend-of-a-friend (FOAF) and XHTML Friends Network (XFN), however, as it may occur in applications using the Google Social Graph API (Google, 2010), the scenario is realistic. But a distributed grounding and inference environment is not already part of LISy, nor of the Signal/Collect framework (Stutz et al., 2010) LISy builds on.

Another limitation that is close to the one mentioned above is that only one type of RDF store and SPARQL endpoint was tested: Sesame. Here, the implementation and evaluation of other SPARQL binders may have shown a difference in grounding speed. Also, it would have given the system more flexibility to interconnect with different datasets. However, since running efficient belief propagation on RDF datasets were the main goal, the grounding itself was not considered as a very important issue.

Then, also coming from the database side, the evaluation could not be performed completely autonomous because the query on the data in order to retrieve the data needed for grounding either implied the Sesame server on the evaluation machine to use resources and therefore influence the measurements, or it could would suffer from latencies due to the network link between the Sesame server and the evaluation machine.

Another point is, that as benchmark applications, only Alchemy came to play. With an additional benchmark system like PyMLN, the evaluation would have gained more credibility.

Last but not least, while it is my personal opinion that LISy can now easily be used to develop applications with support for inference in Markov logic, this is not arisen by inquiries or by usability studies.

6.2 System limitations

An limitation of the underlying logical system of LISy is the missing support for existential quantifiers to this date. Of course, this does call the reasoning algorithm into question, but it limits the expressiveness of the formulas and hence the applications of the system drastically. More to this in future work.

Technical progress can also be made by allowing ontology inference. With the implementation as it is, groundings like *pregnant(Peter)* can be made, if they are not already eliminated by the ontology itself (e.g. allowing only individuals of the class women for the predicate rather than person).

Other requirements that have not been met are the implementation of a Alchemy file parser, as well as the at least and at most one operators. Another feature that was not part of the requirements, but will be one of the first feature requests is the ability to allow more than only retrieving RDFS and OWL classes and instances. Usually, datasets define their own formats, so the need for querying other relations is obvious.

Also, there is a bug that messes up the grounding process in specific cases. Although it is

natural that some features are not correctly implemented in the beginning, these bugs must be eliminated for non-restrictive use of the system. Usually, a bug tracking system is used for these cases.

Finally, a circumstance that limits the extensive use of the inference system in other application is that the Signal/Collect framework itself is still in its beginning and is subject to minor or maybe even major changes as new requirements arise. This leads to the suggestion to merge the code bases of both frameworks in order to assure consistent releases.

As usual, plenty of limitations flow smoothly into future work, which the next chapter is devoted to.

7

Future Work

Progress in science often picks up when evaluating a project with its limitations or when new implications and ideas arise from them. In the case of LISy, future research directions are manifold, since the subject ranges over a wide application domain.

7.1 Features

Some of the first things that are worth implementing in the near future are some missing features of LISy, as the universal and existential quantifiers. Since they are missing in the latest version, the light Markov logic in LISy lacks of expressiveness.

Another missing feature is taking RDFS class inference into account, as for example *fatherOf*(*x*). Since a lot of datasets use this kind of representation, the domain of application could be broadened.

There is also no clearly defined API available at this time, meaning well defined interfaces for the developer of other applications to use specific algorithms and networks. This would ease the usage of LISy heavily.

For the Signal/Collect framework, it would be a good to be able to distribute it easily, possibly letting the developer define, which parts of the network are distributed exactly. This could enable the system to run efficiently by e.g. only inferring on the Markov blanket and therefore not being bothered by exchanging the complete network.

One rather small but helpful feature is a collection of SPARQL connector besides the Sesame connector. This could interest more user groups for Signal/Collect.

The most important feature missing, however, is in my opinion a lifted version of belief propagation as proposed by Singla and Domingos (2008). As it would have been unfair to compare the standard belief propagation algorithm with a lifted version, it is not present in the evaluation. An informal test on the *mediumsmokes* dataset let LISy cut a rather poor figure. Especially with large datasets used for Semantic Web applications, a lifted version is inevitable in order to create a successful and popular reasoning framework.

7.2 Research

Since the scalability was not completely satisfactory, it is for future work to find out where weak links are. One assumption is, that it depends highly on the topology of the network, whether or not inference scales. The guess is, that highly interlinked networks scale much better than many local clusters. Therefore, an evaluation with entirely different data sets could show, what types of networks are suitable and which are not.

One big problem with LISy is the size of the grounding data. As this takes very long time, the data structure for the grounded network should be reorganized or replaced by something slimmer. Avoiding this problem has high priority in future work.

In order to give the evaluation even more credibility, it would be interesting to see how other inference engine besides Alchemy would score. One example is PyMLN (Beetz and Jain, 2010c), that comes with an own inference engine.

Since this is my personal opinion that LISy is well structured and easy to use in other applications that want to integrate Markov logic, this has not arisen by inquiries or any usability studies. In order to legitimate this claim, a usability study with Scala developers, each with the goal to implement a series of smaller tasks should be done.

Coming to the rule definition, there should be a way to use extended Semantic Web formats like the new W3C recommendation, the Rule Interchange Format RIF (Axel Polleres, 2010), or the Semantic Web Rule Language SWRL (Connor, 2009) for Markov logic.

It would be interesting to know, how good systems like LISy integrate in a RDF triple store as extensions. Because getting inferred data can also be viewed as getting data, this functionality would fit perfectly into a triple store (or database server).

8

Conclusions

This Master Thesis was about creating infrastructure for running inference for Semantic Web data. The goal was to create infrastructure that allows to do inference on Markov logic formulas by running loopy belief propagation on the Signal/Collect framework. This led to two main tasks:

The first was to show successfully that loopy belief propagation can be run on Markov logic networks using Signal/Collect in order to provide a solution to the missing infrastructure problem. By implementing the Logical Inference System LISy, an easy-to-use system is now available for developers, so that they can integrate Markov logic into other applications.

The second was to show, that this can be done efficiently and scalable. After running several evaluation tests, the main task which was the execution of the algorithm, performed very satisfactorily, as LISy won every benchmark against the popular Markov logic system, Alchemy. Especially the scoring functionality of Signal/Collect had a great impact on the execution times of LISy, letting it converge really fast while Alchemy often had to break without final convergence. Using Signal/Collect for loopy belief propagation seems to be a perfect approach since, according to the evaluation, LISy is capable of inferring also in networks with many individuals within excellent time.

The question of scalability could not be answered clearly in every aspect and is therefore still open for further evaluation. Depending highly on the network topology, the results achieved in the evaluation only showed still pleasing, but limited scalability for the used dataset.

Since the implementation of LISy on the Signal/Collect framework was a huge success, this certainly leads to further research on the subject. As a next step to run even faster inference, distributed lifted belief propagation could be the key.

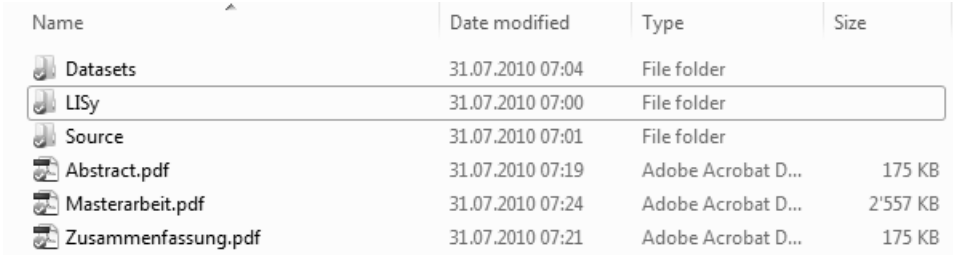
A

Appendix

A.1 Content CD-ROM

The enclosed CD-ROM contains the following data, as shown in figure A.1:

- The folder "Datasets" containing all datasets used for evaluation.
- The folder "LISy" contains the Logic Inference System as a JAR including the libraries used.
- The folder "Source" contains the Scala source files of LISy.
- The PDF file "Masterarbeit.pdf" contains the written part of this Master Thesis.
- The PDF file "Abstract.pdf" contains the abstract in English.
- The PDF file "Zusammenfassung.pdf" contains a German version of the abstract.









Name	Date modified	Type	Size
 Datasets	31.07.2010 07:04	File folder	
 LISy	31.07.2010 07:00	File folder	
 Source	31.07.2010 07:01	File folder	
 Abstract.pdf	31.07.2010 07:19	Adobe Acrobat D...	175 KB
 Masterarbeit.pdf	31.07.2010 07:24	Adobe Acrobat D...	2'557 KB
 Zusammenfassung.pdf	31.07.2010 07:21	Adobe Acrobat D...	175 KB

Figure A.1: Content of the Master Thesis CD-ROM

A.2 System usage

Since the Logic Inference System LISy is a Scala framework, it uses the Java Virtual Machine. To execute the JAR, copy the LISy folder to a writable directory and open a terminal in the corresponding path. Type `java -jar -Xmx3000m MarkovLogicInference.jar`.

In order to use LISy for other applications, be sure to install Scala ≥ 2.8 and a suitable IDE like Netbeans or Eclipse.

A.3 Data sets used

In order to evaluate the implemented system, the following datasets were used.

Alchemy:

- TestA.mln
- TestB.mln
- TestC.mln

LISy:

- largesmokes.rdf
- mediumsmokes.rdf
- smallsmokes.rdf
- tinysmokes.rdf

List of Figures

1.1	Extract of the RDF datasets available (Richard Cyganiak, 2009)	1
2.1	Propositional logic BNF (Russell and Norvig, 2003) (Stärk, 2002)	5
2.2	First-order logic BNF (Russell and Norvig, 2003, page 247) (Stärk, 2002, page 83)	7
2.3	De Morgan rules for quantifiers	8
2.4	Bayes network	11
2.5	Example of Markov network	12
2.6	Example of a polytree (right) built from a multiply connected network (left) . . .	15
2.7	Conversion from directed polytree to undirected graph to factor graph	17
2.8	Ground Markov network for smokes example	20
2.9	Screen shot of PyMLN query tool	23
2.10	Screen shot of BNJ, a graphical editor for Bayesian and Bayesian logic networks . .	23
2.11	Layered architecture of the Semantic Web	25
2.12	Example of a simple RDF graph that describes the location of Zurich	26
2.13	Illustration of the Signal/Collect model	30
3.1	Belief propagation in a factor graph	34
4.1	Logic layers	37
4.2	Example of a hierarchically defined formula.	38
4.3	Class structure of Formula hierarchy in Markov Logic	39
4.4	Implementation of grounded formulas	40
4.5	Grounded formulas built from formulas with grounding	41
4.6	Deployment diagram	41
4.7	TruthValues for Evidence	42
4.8	Two vertex types for belief propagation	42
4.9	Belief propagation with Signal/Collect	43
4.10	TruthFactor and BeliefFactor in Markov Logic Network	44
5.1	Results of tests A	49

5.2	Results of tests B	49
5.3	Results of tests C	50
5.4	Speedup with additional threads	51
5.5	Different scalability per network	52
A.1	Content of the Master Thesis CD-ROM	61

List of Tables

2.1	Logical connectives	5
2.2	Truth tables for logical connectives (Russell and Norvig, 2003)	5
2.3	Standard logical equivalences (Russell and Norvig, 2003)	6
2.4	FOL quantifiers	8
2.5	Full joint probability distribution example	10
2.6	Potential function for example with random variables watered and withered plant .	13
2.7	Potential function for example with random variables withered and perennial . . .	13

List of Listings

2.1	Gibbs Sample	16
2.2	Example MLN file for friends and smokers domain	21
2.3	Example .db file	22
2.4	Run inference executable	22
2.5	Inference .result file	22
2.6	Example of an RDF serialization in RDF/XML	26
2.7	Example of an Ontology in RDFS	27
2.8	Examples of OWL properties	28

Bibliography

- Aduna (2010). Openrdf sesame. <http://www.openrdf.org/>.
- Apache (2010). Apache tomcat. <http://tomcat.apache.org/>.
- Axel Polleres, Harold Boley, M. K. (2010). Rif basic logic dialect. <http://www.w3.org/TR/rif-bld/>.
- B. Motik, B. Cuenca Grau, I. H. Z. W. A. F. C. L. (2009). Owl 2 web ontology language: Profiles. <http://www.w3.org/TR/2009/REC-owl2-profiles-20091027/>.
- Baader, F., Brandt, S., and Lutz, C. (2005). Pushing the EL envelope. In *International Joint Conference on Artificial Intelligence*, volume 19, page 364. Citeseer.
- Beetz, M. and Jain, D. (2010a). Probcog: Probabilistic cognition for cognitive technical systems. <http://ias.in.tum.de/research-areas/knowledge-processing/probcog>.
- Beetz, M. and Jain, D. (2010b). Probcog wiki. <https://ias.in.tum.de/probcog-wiki/index.php>.
- Beetz, M. and Jain, D. (2010c). Pymlns: Markov logic networks in python. <http://www9-old.in.tum.de/people/jain/mlns/>.
- Berners-Lee, T., Hendler, J., Lassila, O., et al. (2001). The semantic web. *Scientific american*, 284(5):28–37.
- Bishop, C. et al. (2006). *Pattern recognition and machine learning*. Springer New York:.
- Boris Motik, Peter F. Patel-Schneider, B. P. (2009). Owl 2 web ontology language: Structural specification and functional-style syntax. <http://www.w3.org/TR/owl2-syntax/>.
- Britannica, E. (2007). *The New Encyclopaedia Britannica*. Encyclopaedia Britannica, Chicago.
- Connor, M. O. (2009). The Semantic Web Rule Language.
- Cyganiak, R. and Jentzsch, A. (2009). About the linking open data dataset cloud. <http://richard.cyganiak.de/2007/10/lod/>.

- de Oliveira, P. and de Sousa Gomes, P. (2009). Probabilistic reasoning in the semantic web using markov logic msc thesis.
- Domingos, P. (2006). Unifying logical and statistical ai.
- Domingos, P. and Lowd, D. (2009). *Markov Logic: An Interface Layer for Artificial Intelligence*. Morgan & Claypool Publishers.
- Eric Prud'hommeaux, A. S. (2008). Sparql query language for rdf. <http://www.w3.org/TR/rdf-sparql-query/>.
- Genesereth, M. R. and Nilsson, N. J. (1987). *Logical foundations of artificial intelligence*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.
- Gilks, W., Gilks, W., Richardson, S., and Spiegelhalter, D. (1996). *Markov chain Monte Carlo in practice*. Chapman & Hall/CRC.
- Google (2010). Social graph api. <http://code.google.com/apis/socialgraph/docs/edges.html/>.
- Gruber, T. et al. (1993). A translation approach to portable ontology specifications. *Knowledge acquisition*, 5:199–199.
- Herman, I. (2010). W3c semantic web activity. <http://www.w3.org/2001/sw/>.
- Hitzler, P., Krötzsch, M., Rudolph, S., and Sure, Y. (2007). *Semantic Web: Grundlagen*. Springer-Verlag New York Inc.
- Jensen, F., Olesen, K., and Andersen, S. (1990). An algebra of Bayesian belief universes for knowledge-based systems. *Networks*, 20(5):637–659.
- Jowett, B. (1979). *The Portable Plato*. Penguin Books, New York.
- Kindermann, R., Snell, J., and Society, A. M. (1980). *Markov random fields and their applications*. American Mathematical Society Providence, Rhode Island.
- Kok, S., Sumner, M., Richardson, M., Singla, P., Poon, H., Lowd, D., , and Domingos, P. (2007). The alchemy system for statistical relational ai. Technical report, Department of Computer Science and Engineering, University of Washington, Seattle, WA.
- Koller, D. and Friedman, N. (2009). *Probabilistic Graphical Models: Principles and Techniques*. The MIT Press.
- Mark Musen, e. a. (2010). Protégé. <http://protege.stanford.edu/>.
- Murphy, K., Weiss, Y., and Jordan, M. (1999). Loopy belief propagation for approximate inference: An empirical study. In *Proceedings of Uncertainty in AI*, pages 467–475. Citeseer.

- Oliveira, P. (2009). Incerto - A Probabilistic Reasoner for the Semantic Web based on Markov Logic. <http://code.google.com/p/incerto/>.
- Pearl, J. (1982). Reverend Bayes on inference engines: A distributed hierarchical approach. In *Proceedings of the AAAI National Conference on AI*, pages 133–136.
- Pearl, J. (1988). *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, San Francisco, CA.
- Richardson, M. and Domingos, P. (2006). Markov logic networks. *Machine Learning*, 62(1/2):107–136.
- Roth, D. (1996). On the hardness of approximate reasoning. *Artif. Intell.*, 82(1-2):273–302.
- Russell, S. and Norvig, P. (2003). Artificial intelligence: a modern approach.
- Singla, P. and Domingos, P. (2008). Lifted first-order belief propagation. *Association for the Advancement of Artificial Intelligence (AAAI)*.
- Sperberg-McQueen, C. M. and Thompson, H. (2000). Xml schema. <http://www.w3.org/XML/Schema>.
- Stärk, R. (2002). Logik, informatik, 1. semester.
- Stutz, P., Bernstein, A., and Cohen, W. (2010). Signal/collect: Graph algorithms for the (semantic) web.
- University, C. (2010). Bio2rdf. <http://bio2rdf.org/>.
- URI Planning Interest Group, W. (2001). Uris, urls, and urns: Clarifications and recommendations 1.0. <http://www.w3.org/TR/uri-clarification/>.
- Wattayau, W. (2009). Loopy Belief Propagation : Bayesian Networks for Multi-Criteria Decision Making (MCDM). *International Journal of Hybrid Information Technology*, 2(2):141–152.
- Whately, R. (1867). *Elements of logic*. Longmans, Green, Reader, and Dyer.
- Yedidia, J., Freeman, W., and Weiss, Y. (2001). Generalized belief propagation. *Advances in neural information processing systems*, pages 689–695.
- Yedidia, J., Freeman, W., and Weiss, Y. (2005). Constructing Free-Energy Approximations and Generalized Belief Propagation Algorithms. *IEEE Transactions on Information Theory*, 51(7):2282–2312.