Learning and Exploiting Relational Structure for Efficient Inference

Aniruddh Nath

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington

2015

Reading Committee:

Pedro Domingos, Chair

Luke Zettlemoyer

Michael Ernst

Program Authorized to Offer Degree:
Computer Science & Engineering

University of Washington

Abstract

Learning and Exploiting Relational Structure for Efficient Inference

Aniruddh Nath

Chair of the Supervisory Committee:
Professor Pedro Domingos
Computer Science & Engineering

One of the central challenges of statistical relational learning is the tradeoff between expressiveness and computational tractability. Representations such as Markov logic can capture rich joint probabilistic models over a set of related objects. However, inference in Markov logic and similar languages is #P-complete. Most existing tractable statistical relational representations are very limited in expressiveness.

This dissertation explores two strategies for dealing with intractability while preserving expressiveness. The first strategy is to exploit the approximate symmetries frequently found in relational domains to perform approximate lifted inference. We provide error bounds for two approaches for approximate lifted belief propagation. We also describe propositional and lifted inference algorithms for repeated inference in statistical relational models. We describe a general approach for expected utility maximization in relational domains, making use of these algorithms.

The second strategy we explore is learning rich relational representations directly from data. First, we propose a method for learning multiple hierarchical relational clusterings, unifying several previous approaches to relational clustering. Second, we describe a tractable high-treewidth statistical relational representation based on Sum-Product Networks, and propose a learning algorithm for this language. Finally, we apply state-of-the-art tractable learning methods to the problem of software fault localization.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

Thanks to my advisor, Pedro Domingos, whose guidance made this work possible. I am also grateful to my other committee members, Luke Zettlemoyer, Michael Ernst, and Ali Shojaie, for their feedback on this dissertation.

I owe thanks to many members of the UW CSE community, for attending all those practice talks, reading all those paper drafts, and sharing many lunches on the Ave. Particular thanks to Rob Gens, Chloé Kiddon, Abe Friesen, Hoifung Poon, Daniel Lowd, Stanley Kok, Parag Singla, Jesse Davis, Vibhav Gogate, Xu Miao, Marc Sumner, Xiao Ling, Sai Zhang, and Austin Webb.

I am grateful to my parents, Gita and Vishwambhar Nath, and my brother Abhinav, for all their support and wisdom.

Chapter 1

INTRODUCTION

Probabilistic graphical models are at the heart of many state-of-the-art algorithms in computer vision, natural language processing, computational biology, and many other fields. Their popularity is due to their ability to compactly represent a rich joint probability distribution over a set of random variables. However, this expressiveness comes at the cost of computational tractability; inference in arbitrary graphical models is #P-complete [120]. This intractability is compounded in statistical relational languages such as Markov Logic Networks (MLNs) [118, 40], which model a set of instances jointly. To cope with this intractability, practitioners usually either (a) resort to approximate inference algorithms, or (b) restrict the modeling language to a subset that permits tractable inference (at great cost to expressiveness).

This dissertation explores two strategies for dealing with intractability in statistical relational models, without excessively restricting the expressiveness of the modeling language. The first strategy is to exploit the known relational structure of the domain to perform lifted inference, reasoning over sets of indistinguishable variables, instead of individual random variables. Lifting is possible for a variety of inference tasks, including marginal inference, MAP inference, and expected utility maximization. Chapter 3 formalizes the expected utility maximization problem in the statistical relational setting. Utility maximization is an instance of the more general problem of *repeated inference*, which also arises in Marginal MAP, weight and structure learning, etc. Since small changes to the evidence typically only affect a small region of the network, repeatedly performing inference from scratch can be massively redundant. We propose *Expanding Frontier Belief Propagation* (EFBP), an efficient propositional algorithm for probabilistic inference with incremental changes to the evidence. EFBP is an extension of loopy belief propagation (BP) where each run of inference reuses results from the previous ones, instead of starting from scratch with the new

evidence; messages are only propagated in regions of the network affected by the changes. We provide theoretical guarantees bounding the difference in beliefs generated by EFBP and standard BP, and apply EFBP to the problem of expected utility maximization in relational models. Experiments on viral marketing and combinatorial auction problems show that EFBP can converge much faster than BP without significantly affecting the quality of the solutions.

One of the limitations of lifted inference is that constructing the lifted network (i.e. identifying symmetries) can itself be quite costly. In addition, the exact lifted network is often very close in size to the fully propositionalized model. Approximate lifted inference algorithms overcome these problems by grouping together similar but distinguishable objects and, treating them as if they were identical. *Early stopping* terminates the execution of the lifted network construction at an early stage, resulting in a coarser network. *Noise-tolerant hypercubes* allow for marginal errors in the representation of the lifted network itself. Both algorithms can significantly speed up the process of lifted network construction as well as result in much smaller models. Chapter 4 provides a theoretical analysis of both algorithms, allowing the user to adjust coarseness of the approximation depending on the accuracy required, and bound the resulting error.

Chapter 5 unifies the research directions explored in the previous two chapters, proposing an approximate lifted inference algorithm for repeated inference in relational domains. In the repeated inference setting, constructing even an approximate lifted network from scratch for each new inference can be extremely wasteful, because the evidence typically changes little from one inference to the next. We propose an efficient algorithm for updating the structure of an existing lifted network with incremental changes to the evidence. This allows us to construct the lifted network once for the initial inference problem, and amortize the cost over the subsequent problems. Experiments on video segmentation and viral marketing problems show that the algorithm greatly reduces the cost of inference without affecting the quality of the solutions.

The second strategy for dealing with intractability in statistical relational models is to learn rich relational representations that support efficient inference and learning algorithms. This dissertation proposes two such approaches. Chapter 6 describes a method for learning *Multiple Hierarchical Relational Clusterings* (MHRC) from a relational database. While this model can be used to pre-

dict missing data, the clustering cannot be learned on one database and applied to another, over a different set of objects. To address this limitation, chapter 7 proposes *Relational Sum-Product Networks* (RSPNs), a new tractable first-order probabilistic architecture. RSPNs build on Sum-Product Networks (SPNs [113]), a recently-proposed deep architecture that guarantees tractable inference, even on certain high-treewidth models. SPNs are a propositional architecture, treating the instances as independent and identically distributed. RSPNs generalize SPNs by modeling a set of instances jointly, allowing them to influence each other's probability distributions, as well as modeling probabilities of relations between objects. We also present LearnRSPN, the first algorithm for learning high-treewidth tractable statistical relational models. LearnRSPN is a recursive top-down structure learning algorithm for RSPNs, based on Gens and Domingos' LearnSPN algorithm for propositional SPN learning. We evaluate the algorithm on three datasets; the RSPN learning algorithm outperforms Markov Logic Networks in both running time and predictive accuracy.

Chapter 8 describes an application of an RSPN-like model (*Tractable Fault Localization Models*, TFLMs) to the problem of software fault localization. While most previous statistical debugging methods generalize over many executions of a single program, TFLMs are trained on a corpus of previously seen buggy programs, and learn to identify recurring patterns of bugs. Widely-used fault localization techniques such as TARANTULA [68] evaluate the suspiciousness of each line in isolation; in contrast, a TFLM defines a joint probability distribution over buggy indicator variables for each line. Further, TFLMs can incorporate additional sources of information, including coverage-based features such as TARANTULA. We evaluate the fault localization performance of TFLMs that include TARANTULA scores as features in the probabilistic model. Our study shows that the learned TFLMs isolate bugs more effectively than previous statistical methods or using TARANTULA directly, and does so at near-interactive speeds.

Each of these two strategies for dealing with intractability has certain advantages over the other. Traditional statistical relational languages such as Markov logic allow probability to be combined with logical rules in an extremely intuitive manner. Human experts can create rules based on their knowledge of the domain, without additionally needing expertise in probabilistic models. The weights can be learned from data, and the rules can optionally be supplemented or refined using

structure learning algorithms. This approach can yield rich, intuitive probabilistic models of complex relational domains. However, the key limitation is the intractability of probabilistic inference, which is a subroutine of both structure and weight learning in MLNs and similar languages. Inference is also used to answer queries using the learned models. Therefore, the design of more efficient inference algorithms is essential for making these highly expressive statistical relational languages usable in this manner. This is the underlying motivation behind the first part of this dissertation (chapters 3, 4, 5).

In contrast, the learned models considered in the second half of the dissertation (chapters 6, 7, 8) are less amenable to being specified by hand by a domain expert. Instead, they are learned directly from data. The lack of a human-provided starting model gives us greater freedom to design representations and learning algorithms that best model the data, with less consideration to the ease of manually specifying or understanding the model (though these are still desirable qualities in the language). In addition, computational tractability can be encoded into the design of the language. MLNs and similar systems suffer from the drawback that a domain expert can unintentionally create a simple, intuitive model that is well beyond the reach of even state-of-the-art approximate inference algorithms. However, this cannot happen with RSPNs (chapter 7) and TFLMs (chapter 8), both of which build tractability into the language.

One drawback to the approach we consider in the second half of the dissertation is that MHRC and RSPNs both rely on structure learning algorithms, which require a relatively large amount of training data, compared to starting with a hand-specified structure and simply learning the weights. For the smaller datasets required for our debugging application (chapter 8), we instead use a hand-specified structure for an RSPN-like language. Specifying an RSPN structure by hand requires knowledge of the problem domain and as well as an understanding of the computational issues that might arise from a candidate structure; this makes representations like MHRC and RSPNs a less suitable choice than MLNs for domain experts who have a small amount of available training data.

Chapter 2

BACKGROUND

2.1 Probabilistic Graphical Models

Probabilistic graphical models (PGMs) [106, 81] compactly represent the joint distribution of a set of variables $\mathbf{X} = (X_1, \ldots, X_n) \in \mathcal{X}$ as a product of factors [106]:

$$P(\mathbf{X}{=}\mathbf{x}) = \frac{1}{Z} \prod_k f_k(\mathbf{x}_k)$$

where each factor f_k is a non-negative function of a subset of the variables $\mathbf{x}_k$, and Z is a normalization constant. Under appropriate restrictions, the model is a *Bayesian network* and $Z = 1$. A *Markov network* or *Markov random field* can have arbitrary factors. Graphical models can also be represented in *log-linear form*:

$$P(\mathbf{X}{=}\mathbf{x}) = \frac{1}{Z} \exp\left(\sum_i w_i g_i(\mathbf{x})\right)$$

where the *features* $g_i(\mathbf{x})$ are arbitrary functions of the state. A *factor graph* [82] is a bipartite undirected graph with a node for each variable and factor in the model. Variables are connected to the factors they appear in.

PGMs are extremely flexible, and can represent complex probabilistic models in an intuitive way. By making certain conditional independence assumptions, they can capture joint probability distributions over a large number of variables without suffering from an explosion in the number of parameters. These qualities have made this approach quite popular, and PGMs are at the heart of many state-of-the-art algorithms in computer vision, natural language processing, computational biology, social network modeling, and many other fields.

A key inference task in graphical models is computing the marginal probabilities of some variables (the query) given the values of some others (the evidence). Unfortunately, this is a #P-

complete problem [120]; even (bounded) approximate inference is intractable for general PGMs. Therefore, in practice, users of graphical models must usually resort to one of the following:

1. use approximate inference algorithms based on sampling, belief propagation, etc; or

2. restrict the model to a subset of graphical models on which inference is tractable.

A popular approximate inference approach is loopy belief propagation, which works by passing messages from variable nodes to factor nodes and vice versa. The message from a variable to a factor is:

$$\mu_{xf}(a) = \prod_{h \in nb(x) \setminus \{f\}} \mu_{hx}(a)$$

where $nb(x)$ is the set of factors it appears in. (Evidence variables send 1 for the evidence value, and 0 for others.) The message from a factor to a variable is:

$$\mu_{fx}(a) = \sum_{\mathbf{x}_a} \left(f(\mathbf{x}_a) \prod_{y \in nb(f) \setminus \{x\}} \mu_{yf}(y_{\mathbf{x}_a}) \right)$$

where $nb(f)$ are the arguments of f; $\mathbf{x}_a$ is an assignment of values to the variables in $nb(f)$, with x set to a; $y_{\mathbf{x}_a}$ is the value of y in $\mathbf{x}_a$. The (unnormalized) marginal of variable x is given by:

$$M_x(a) = \prod_{h \in nb(x)} \mu_{hx}(a)$$

Many message-passing schedules are possible; the most widely used one is *flooding*, where all nodes send messages at each step. In this case, the messages and marginals in iteration $i+1$ are as follows:

$$\mu_{xf,i+1}(a) = \prod_{h \in nb(x) \setminus \{f\}} \mu_{hx,i}(a)$$

$$\mu_{fx,i}(a) = \sum_{\mathbf{x}_a} \left(f(\mathbf{x}_a) \prod_{y \in nb(f) \setminus \{x\}} \mu_{yf,i}(y_{\mathbf{x}_a}) \right)$$

$$M_{x,i}(a) = \prod_{h \in nb(x)} \mu_{hx,i}(a)$$

In general, belief propagation is not guaranteed to converge, and it may converge to an incorrect result, but in practice it often approximates the true probabilities well.

A related task is to infer the most probable values of all variables. The *max-product* algorithm does this by replacing summation with maximization in the factor-to-variable messages.

Belief propagation can equivalently be formulated as a message-passing algorithm directly on the graph of variables (as opposed to the factor graph). For a pairwise graphical model, the message from node t to node s is:

$$\mu_{ts}(x_s) = \sum_{x_t} f_{ts}(x_t, x_s) f_t(x_t) \prod_{u \in nb(t) \setminus \{s\}} \mu_{ut}(x_t)$$

where $nb(t)$ is the set of neighbors of t. The (unnormalized) belief at node t is given by:

$$M_x(x_t) = f_t(x_t) \prod_{u \in nb(t)} \mu_{ut}(x_t)$$

2.2 Tractable Models

There is a large body of research on approximate inference algorithms for graphical models. However, this approach poses several problems. The approximation quality is often hard to judge, since these algorithms typically do not have approximation bounds. Further, these algorithms often have many parameters and options to tune; successfully applying approximate inference algorithms to a new problem often requires deep understanding of both the problem domain and the inference algorithm. Unreliable, inaccurate inference algorithms negate some of the advantage gained from the expressiveness of PGMs.

Tractable probabilistic models sacrifice some of the expressiveness of general PGMs in return for the ability to perform exact inference in polynomial time. Although early tractable probabilistic models were quite restrictive, recent work in this field has revealed that tractable models can be more expressive than previously believed. Current tractable models often outperform traditional graphical models due to the ease of exact inference [113, 50, 51]; in practice, the improvement due to accurate inference often outweighs the reduction in expressiveness. Inference in tractable models is push-button, with no parameters to tune; the only human expertise is in specifying or

learning the model. This section discusses some recent developments in tractable probabilistic models.

2.2.1 Low Treewidth Graphical Models

For PGMs with tree structure, exact probabilistic inference is possible in polynomial time using dynamic programming algorithms (belief propagation [106] or special cases). Arguably the simplest class of useful graphical models are Naïve Bayes Mixture Models (NBMM) [88]. A Naïve Bayes model has a single, latent class variable, and the remaining variables in the distribution are independent conditioned on the class. NBMMs with exact inference were shown to be more accurate than intractable Bayesian networks using then state-of-the-art learning and inference algorithms.

Hidden Markov Models (HMMs) are an extremely popular special case of tree-structured graphical models, and have been successfully applied to problems in speech recognition, computational biology, natural language processing etc.

When the structure of the graphical model contains loops, belief propagation is no longer exact. Exact inference can be performed by constructing a *junction tree* (a tree of clusters of variables), and performing message passing on it. The *width* of a junction tree is one less than the size of its largest cluster. The *treewidth* of a graph is the width of the thinnest (lowest width) junction tree that can be constructed from the graph. Inference on a PGM using the junction tree algorithm is exponential in the PGM's treewidth (in both time and space). Therefore, probabilistic inference is tractable for low-treewidth PGMs. Motivated by this insight, several methods have been proposed for learning low-treewidth PGMs, known as *thin junction trees* [9, 20].

2.2.2 Tractable High Treewidth Models

The junction tree algorithm and other exact inference algorithms for PGMs run in time exponential in the treewidth of the graph. Therefore, conventional wisdom in the probabilistic inference community was that low-treewidth models were tractable, and high-treewidth models were intractable. However, not all high-treewidth probabilistic models require exponential inference time. Certain

representations and inference algorithms that take advantage of context-specific independence can support polynomial-time exact inference, even if inference on the equivalent PGM is exponential.

Much of the recent progress on this class of models builds on the field of *knowledge compilation* [26]. Knowledge compilation is an inference strategy for PGMs that works by compiling the graphical model into an intermediate representation, such as Arithmetic Circuits (ACs) [25] or AND/OR graphs [32]. These circuits are typically directed acyclic graphs that represent probability computations on the given PGM.

Inference on these intermediate structures is linear in the size of the circuit. Naïvely compiling a PGM into an AC yields a circuit whose size is exponential in the treewidth of the PGM; inference through knowledge compilation techniques is therefore in general no more efficient than the classic junction tree algorithm. The advantage of the knowledge compilation strategy is that the compilation cost can be amortized over many queries, since the structure of the compiled circuit is independent of the choice of query and evidence variables. A PGM only needs to undergo the expensive knowledge compilation process once; if the resulting circuit is compact, the model can answer a large number of queries in a fast, predictable manner.

Interestingly, the intermediate representations used by knowledge compilation methods can represent some context-specific independence assumptions that cannot be captured by the traditional PGM graph representation. This allows certain probability distributions to be represented compactly in the intermediate distributions, even though the corresponding PGM is high treewidth. This insight has motivated a line of research that views ACs and related representations as a probabilistic representation in their own right, rather than merely a compilation structure for PGM inference. This class of algorithms learns compact ACs (or similar structures) directly from data, without regard for the treewidth of the corresponding PGM. This line of work includes multi-linear representations [121], LEARNAC [89], ACMN [90] and Sum-Product Networks (SPNs) [113]. Most recent approaches to learning high-treewidth probabilistic models have been described in terms of SPNs [50, 51, 108, 119], which we describe in more detail in the following section.

The emerging line of research by Niepert et al. [101, 102] investigates another class of tractable probabilistic models, built on assumptions of exchangeability, instead of conditional independence.

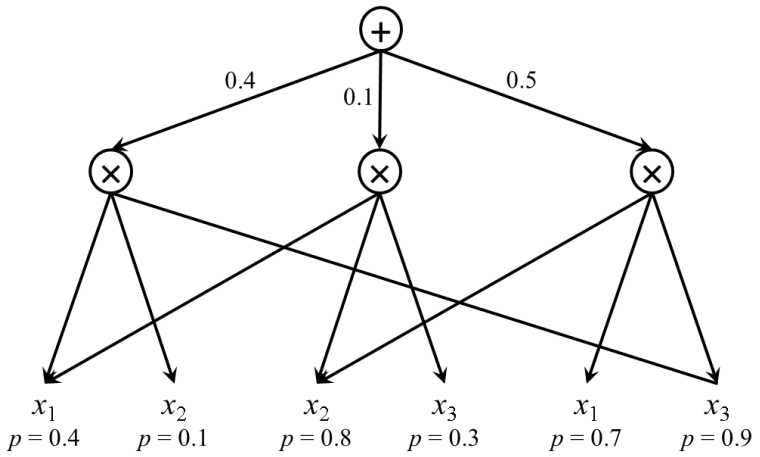


Figure 2.1: Example SPN over the variables x_1, x_2 and x_3. All leaves are Bernoulli distributions, with the given parameters. The weights of the sum node are indicated next to the corresponding edges.

Sum-Product Networks

A sum-product network (SPN) [113] is a rooted directed acyclic graph with univariate distributions at the leaves; the internal nodes are (weighted) sums and (unweighted) products. In this work, we use the simplified definition of Gens and Domingos [51]:

Definition 1. A sum-product network (SPN) is defined as follows.

1. A tractable univariate distribution is an SPN.

2. A product of SPNs with disjoint scopes[1] is an SPN.

3. A weighted sum of SPNs with the same scope is an SPN, provided all weights are positive.

4. Nothing else is an SPN.

[1]The scope of an SPN is the set of variables that appear in it.

A univariate distribution is tractable iff its partition function and mode can be computed in $O(1)$ time. Intuitively, an SPN S can be thought of as an alternating set of mixtures (sums) and decompositions (products) of the leaf variables (Fig. 2.1). If the values at the leaf nodes are set to the partition functions of the corresponding univariate distributions, then the value at the root is the partition function (i.e. the sum of the unnormalized probabilities of all possible assignments to the leaf variables). This allows the partition function to be computed in time linear in the size of the SPN. We denote the partition function of S by $S(*)$.

Similarly, if some of the leaves are known, setting their values to their probabilities (according to the corresponding univariates) yields the unnormalized probability of the evidence (denoted $S(e)$). This can be divided by the partition function to obtain the normalized probability ($S(e)/S(*)$). MPE inference is also linear in the size of the SPN, replacing sum nodes by max nodes, which corresponds to viewing these nodes as hidden variables that we also maximize over.

The first learning algorithms for sum-product networks used a fixed network structure, and only optimized the weights [113, 4, 50]. The network structure is domain-dependent; applying SPNs to a new problem required manually designing a suitable network structure.

More recently, several algorithms have been proposed that learn both the weights and the network structure, allowing SPNs to be applied out-of-the-box to new domains. Dennis and Ventura [37] suggested an algorithm that builds an SPN based on a hierarchical clustering of variables. Gens and Domingos [51] construct SPNs top-down, recursively partitioning instances and variables. Peharz et al. [108] construct SPNs bottom-up, greedily merging SPNs into models of larger scope. These algorithms have been shown to perform well on a variety of domains, making more accurate predictions than conventional graphical models, while guaranteeing tractable inference.

2.3 Statistical Relational Learning

Intelligent agents must be able to handle the complexity and uncertainty of the real world. First-order logic is useful for the first, and probabilistic graphical models for the second. Combining the two has been the focus of much recent research in the emerging field of *Statistical Relational Learning* (SRL) [52]. A variety of languages have been proposed that combine the desirable fea-

tures of both these representations. The first inference algorithms for these languages worked by propositionalizing all atoms and clauses, and then running standard probabilistic inference algorithms on the ground model.

More recently, several algorithms have been proposed for *lifted* inference, which deals with groups of indistinguishable variables, rather than individual ground atoms. Approaches for identifying sets of indistinguishable variables include *lifted network construction* (LNC) [131], *shattering* [29], and *finding lifting partitions* using graph automorphism [19]. Since the first-order representation is often much smaller than the fully ground model, lifted inference is potentially much more efficient than propositionalized inference. The first lifted probabilistic inference algorithm was *first-order variable elimination* (FOVE), proposed by Poole [111], and extended by de Salvo Braz, Amir and Roth [29] and Milch et al. [94]. Singla and Domingos [131] proposed the first lifted approximate inference algorithm, a first-order version of *loopy belief propagation* (BP) [149]. Interest in lifted inference has grown rapidly in recent years [71, 74, 75, 57, 15, 135, 71, 22, 93, 72, 53, 139, 137, 54, 141, 19].

Markov logic [118] is a probabilistic extension of first-order logic. Formulas in first-order logic are constructed from logical connectives, predicates, constants, variables and functions. A *grounding* of a predicate (or *ground atom*) is a replacement of all its arguments by constants (or, more generally, ground terms). A grounding of a formula is a replacement of all its variables by constants. A *possible world* is an assignment of truth values to all possible groundings of predicates.

A *Markov logic network (MLN)* is a set of weighted first-order formulas. Together with a set of constants, it defines a Markov network with one node per ground atom and one feature per ground formula. The weight of a feature is the weight of the first-order formula that originated it. The probability distribution over possible worlds $\mathbf{x}$ specified by the MLN and constants is thus

$$P(\mathbf{x}) = \frac{1}{Z} \exp\left(\sum_i w_i n_i(\mathbf{x})\right)$$

where w_i is the weight of the ith formula and $n_i(\mathbf{x})$ its number of true groundings in $\mathbf{x}$.

Inference in an MLN can be carried out by creating the corresponding ground Markov network

and applying standard BP to it. A more efficient alternative is *lifted belief propagation*, which avoids grounding the network as much as possible [131]. Lifted BP (LBP) constructs a *lifted network* composed of *supernodes* and *superfeatures*, and and applies BP to it (*LBP = LNC + BP*). It is guaranteed to give the exact same results as running BP on the ground network. A supernode is a set of atoms that send and receive exactly the same messages throughout BP, and a superfeature is a set of ground clauses that send and receive the same messages. A supernode and a superfeature are connected iff some atom in the supernode occurs in some ground clause in the superfeature.

The lifted network is constructed by starting with an extremely coarse network, and then refining it essentially by simulating BP and keeping track of which nodes send the same messages (Algorithm 1). The count $n(s, F)$ is the number of identical messages atom s would receive in message passing from the features in F. Belief propagation must be altered slightly to take these counts into account. Since all the atoms in a supernode have the same counts, we can set $n(X, F) = n(s, F)$, where s is an arbitrary atom in X. The message from F to X remains the same as before:

$$\mu_{FX}(a) = \sum_{\mathbf{X}_a} \left(f(\mathbf{X}_a) \prod_{Y \in nb(F) \setminus \{X\}} \mu_{YF,i}(y_{\mathbf{X}_a}) \right)$$

The message from X to F becomes

$$\mu_{XF}(a) = \mu_{FX,i}(a)^{n(X,F)-1} \prod_{H \in nb(X) \setminus \{F\}} \mu_{HX,i}(a)^{n(X,H)}$$

The marginal becomes

$$M_X(a) = \prod_{H \in nb(X)} \mu_{HX,i}(x)^{n(X,H)}$$

An important question remains: how to represent supernodes and superfeatures. Although this does not affect the space or time cost of inference on the lifted network, it can greatly affect the cost of constructing the lifted network. The choice of particular representation can also pave the way for an approximate construction of the lifted network (thereby saving computational cost). In general, finding the most compact representation for supernodes and superfeatures is an intractable problem.

Singla and Domingos [131] considered two representations. The simplest option is to represent each supernode or superfeature extensionally as a set of tuples (i.e., a relation), in which case joins and projections reduce to standard database operations. However, in this case the cost of constructing the lifted network is similar to the cost of constructing the full ground network, and can easily become the bottleneck. Another option is to use a more compact resolution-like representation, as done by Poole [111] and de Salvo Braz et al. [29, 30]. This representation allows for equality/inequality constraints to be represented explicitly. Taghipour et al. [134] propose a framework for representing arbitrary constraints in lifted variable elimination. They propose a constraint language similar to Singla, Nath and Domingos' [132] hypercube representation for lifted BP (section 4.5).

2.3.1 Tractable Models for SRL

Even with the benefit of lifted inference, MLNs and most other statistical relational models are intractable. Like propositional graphical models, statistical relational models can be trivially restricted to the low-treewidth case, but this comes at great cost to the representational power of the model.

Tractable Markov Logic (TML) [42, 143] is a subset of Markov Logic that guarantees polynomial-time inference, even in certain cases where the ground propositional model would be high-treewidth. TML is expressive enough to capture several cases of interest, including junction trees, non-recursive PCFGs and hierarchical mixture models. A TML knowledge base is a generative model that decomposes the domain into parts, with each part drawn probabilistically from a class hierarchy. Each part is further probabilistically decomposed into subparts (according to its class). The key limitation of TML is that the knowledge base fully specifies the set of possible objects in the domain, and their class and part structure. A TML knowledge base cannot generalize across mega-examples of varying size or structure.

Algorithm 1 LNC(MLN $\mathbf{M}$, constants $\mathbf{C}$, evidence $\mathbf{E}$)

for all predicates P **do**
  for all truth values v **do**
    Form a supernode with all groundings of P
      with truth value v
  end for
end for
repeat
  for all clauses C involving $P_1, \ldots, P_k$ **do**
    for all tuples of supernodes $(X_1, \ldots, X_k)$,
    where X_i is a P_i supernode **do**
      Form a superfeature by joining $X_1, \ldots, X_k$
    end for
  end for
  for all supernodes X of predicate P **do**
    for all superfeatures F it is connected to **do**
      for all tuples s in X **do**
        $n(s, F) \leftarrow$ num of F's tuples that project to s
      end for
    end for
    Form a new supernode from each set of tuples in X with
      same $n(s, F)$ counts for all F
  end for
until convergence
return Lifted network $\mathbf{L}$, containing all current supernodes
    and superfeatures

Chapter 3

RELATIONAL DECISION THEORY

3.1 Introduction

As discussed in section 2.3, Markov logic and other SRL languages combine the strengths of probabilistic and logical representations. However, there is little work to date on extending these languages to the decision-theoretic setting, which is needed to allow agents to intelligently choose actions. The one major exception is relational reinforcement learning and first-order Markov decision processes (e.g., [43, 140, 125]). However, the representations and algorithms in these approaches are geared to the problem of sequential decision-making, and many decision-theoretic problems are not sequential. In particular, relational domains often lead to very large and complex decision problems for which no effective general solution is currently available (e.g., influence maximization in social networks, combinatorial auctions with uncertain supplies, etc.).

The goal of this chapter is thus to provide a general decision-theoretic extension of first-order probabilistic representations and their inference algorithms. We extend Markov logic networks to represent decision-theoretic problems. The task of maximizing expected utility in this formulation can be cast as a repeated inference problem in probabilistic graphical models. We describe an efficient approximate algorithm for repeated inference, based on lifted belief propagation [131]. We provide theoretical guarantees for our algorithm. Experiments in viral marketing and combinatorial auction domains show that (a) these decision-making problems can be elegantly formulated in our language, and (b) our inference algorithm is much more efficient than a direct application of belief propagation.

3.2 Background

An *influence diagram* or *decision network* is a graphical representation of a decision problem [63]. It consists of a Bayesian network augmented with two types of nodes: *decision* or *action* nodes and *utility* nodes. The action nodes represent the agent's choices; factors involving these nodes and *state* nodes in the Bayesian network represent the (probabilistic) effect of the actions on the world. *Utility* nodes represent the agent's utility function, and are connected to the state nodes that directly influence utility. We can also define a *Markov decision network* as a decision network with a Markov network instead of a Bayesian network.

The fundamental inference problem in decision networks is finding the assignment of values to the action nodes that maximizes the agent's expected utility, possibly conditioned on some evidence. If $\mathbf{a}$ is a choice of actions, $\mathbf{e}$ is the evidence, $\mathbf{x}$ is a state, and $U(\mathbf{x}|\mathbf{a},\mathbf{e})$ is the utility of $\mathbf{x}$ given $\mathbf{a}$ and $\mathbf{e}$, then the *MEU problem* is to compute

$$\text{argmax}_{\mathbf{a}} E[U(\mathbf{x}|\mathbf{a},\mathbf{e})] = \text{argmax}_{\mathbf{a}} \sum_{\mathbf{x}} P(\mathbf{x}|\mathbf{a},\mathbf{e}) U(\mathbf{x}|\mathbf{a},\mathbf{e})$$

3.3 Markov Logic Decision Networks

Decision theory can be incorporated into Markov logic simply by allowing formulas to have utilities as well as weights. This puts the expressiveness of first-order logic at our disposal for defining utility functions, at the cost of very little additional complexity in the language. Let an *action predicate* be a predicate whose groundings correspond to possible actions (choices, decisions) by the agent, and a *state predicate* be any predicate in a standard MLN. Formally:

Definition 2. A *Markov logic decision network (MLDN)* L is a set of triples (F_i, w_i, u_i), where F_i is a formula in first-order logic and w_i and u_i are real numbers. Together with a finite set of constants $C = \{c_1, c_2, \ldots, c_{|C|}\}$, it defines a Markov decision network $M_{L,C}$ as follows:

1. $M_{L,C}$ contains one binary node for each possible grounding of each state and action predicate appearing in L. The value of the node is 1 if the ground atom is true, and 0 otherwise.

2. $M_{L,C}$ contains one feature for each possible grounding of each formula F_i in L for which $w_i \neq 0$. The value of this feature is 1 if the ground formula is true, and 0 otherwise. The weight of the feature is the w_i associated with F_i in L.

3. $M_{L,C}$ contains one utility node for each possible grounding of each formula F_i in L for which $u_i \neq 0$. The value of the node is the utility u_i associated with F_i in L if F_i is true, and 0 otherwise.

We refer to groundings of action predicates as *action atoms*, and groundings of state predicates as *state atoms*. An assignment of truth values to all action atoms is an *action choice*. An assignment of truth values to all state atoms is a *state of the world* or *possible world*. The utility of world $\mathbf{x}$ given action choice $\mathbf{a}$ and evidence $\mathbf{e}$ is $U(\mathbf{x}|\mathbf{a},\mathbf{e}) = \sum_i u_i n_i(\mathbf{x},\mathbf{a},\mathbf{e})$, where n_i is the number of true groundings of F_i. The expected utility of action choice $\mathbf{a}$ given evidence $\mathbf{e}$ is:

$$\begin{aligned} E[U(\mathbf{x}|\mathbf{a},\mathbf{e})] &= \sum_{\mathbf{x}} P(\mathbf{x}|\mathbf{a},\mathbf{e}) \sum_i u_i n_i(\mathbf{x},\mathbf{a},\mathbf{e}) \\ &= \sum_i u_i E[n_i] \end{aligned}$$

The MEU problem in MLDNs is finding the action choice that maximizes expected utility, and is obviously intractable. The following sections present an efficient approximate algorithm for solving it.

A wide range of decision problems can be elegantly formulated as MLDNs, including both classical planning and Markov decision processes (MDPs). To represent an MDP as an MLDN, we can define a constant for each state, action and time step, and the predicates $\texttt{State}(\texttt{s!},\texttt{t})$ and $\texttt{Action}(\texttt{a!},\texttt{t})$, with the obvious meaning. (The ! notation indicates that, for each t, exactly one grounding of $\texttt{State}(\texttt{s},\texttt{t})$ is true.) The transition function is then represented by the formula $\texttt{State}(+\texttt{s},\texttt{t}) \wedge \texttt{Action}(+\texttt{a},\texttt{t}) \Rightarrow \texttt{State}(+\texttt{s}',\texttt{t}+1)$, with a separate weight for each $(\texttt{s},\texttt{a},\texttt{s}')$ triple. (Formulas with + signs before certain variables represent sets of identical formulas with separate weights, one for each combination of groundings of the variables with + signs.) The reward function is defined by the unit clause $\texttt{State}(*\texttt{s},\texttt{t})$, with a utility for each

state (using $*$ to represent per-grounding utilities). Policies can be represented by formulas of the form $\texttt{State}(+\texttt{s}, \texttt{t}) \Rightarrow \texttt{Action}(+\texttt{a}, \texttt{t})$. Infinite-horizon MDPs can be represented using infinite MLNs [130]. Partially-observable MDPs are represented by adding the observation model: $\texttt{State}(+\texttt{s}, \texttt{t}) \Rightarrow \texttt{Observation}(+\texttt{o}, \texttt{t})$.

Since classical planning languages are variants of first-order logic, translating problems formulated in these languages into MLDNs is straightforward. For simplicity, suppose the problem has been expressed in satisfiability form [69]. It suffices then to translate the (first-order) CNF into a deterministic MLN by assigning infinite weight to all clauses, and to assign a positive utility to the formula defining the goal states. MLDNs now offer a path to extend classical planning with uncertain actions, complex utilities, etc., by assigning finite weights and utilities to formulas. (For example, an action with uncertain effects can be represented by assigning a finite weight to the axiom that defines them.) This can be used to represent first-order MDPs in a manner analogous to Boutilier et al. [14].

3.4 Repeated Inference

Most work on approximate inference in probabilistic graphical models focuses on computing the marginal probabilities of a set of variables, or determining their most probable state, given some fixed evidence and a fixed model. However, many interesting problems require repeated inference, with changing evidence or a changing model:

- **Utility maximization** (discussed above).

- **Maximum a posteriori (MAP) inference** [105], i.e., computing the most likely state of a set of query variables $\mathbf{Q}$ given partial evidence $\mathbf{e}$ of the variables in the complement of $\mathbf{Q}$. We can treat the variables in $\mathbf{Q}$ as changing evidence, and compute the likelihood for each assignment of values to $\mathbf{Q}$.

- **Online inference**, where the values of evidence variables are repeatedly updated (possibly in real time).

- **Interactive inference**, where the choice of evidence variables, their values, and the choice of query can change arbitrarily as dictated by the user.

- **Parameter and structure learning** [60, 35] involve repeatedly modifying the model and recomputing the likelihood.

- **Dynamic inference** [96] involves sequential problems, where the query at one time step depends on evidence and hidden variables at that step and previous ones.

The obvious way to solve these problems is simply to repeatedly perform inference from scratch each time the evidence (or model) changes. However, if the problem remains largely unchanged between two successive search iterations, it may be the case that most of the beliefs are not significantly altered by the changes. In these situations, it would be beneficial to reuse as much of the computation as possible between successive runs of inference.

Delcher et al. [34] presented an exact online inference algorithm for tree-structured Bayesian networks. Acar et al. [2] refer to the problem of repeated inference on variations of a model as *adaptive inference*. They described an exact algorithm that updates marginals as the dependencies in the model are updated. However, we are not aware of any general-purpose *approximate* inference algorithms that avoid redundant computation as the evidence changes. In this chapter, we propose *expanding frontier belief propagation* (EFBP), an approximate inference algorithm that only updates the beliefs of variables significantly affected by the changed evidence. EFBP can be straightforwardly extended to handle changes to the model. We provide guarantees on the performance of EFBP relative to BP. These are based on the fact that, if the potentials in a model are bounded, the difference in marginal probabilities calculated by EFBP and traditional loopy BP can be bounded as well.

We apply EFBP to the problem of expected utility maximization in MLDNs. Utility maximization can be cast as repeated calculation of the marginals in a graphical model with changing evidence. Under certain conditions, it can be shown that the same actions are chosen whether BP or EFBP is used to calculate the marginals. As seen in our experiments, EFBP can be orders of

magnitude faster than BP, without significantly affecting the quality of the solutions.

3.5 Expanding Frontier Belief Propagation

Let $\mathbf{G}$ be a graphical model on the variables in $\mathbf{X}$. $\mathbf{E} \subseteq \mathbf{X}$ is the *evidence set*. We are given a sequence $\vec{\mathbf{e}} = (\mathbf{e}_1, \ldots, \mathbf{e}_{|\vec{\mathbf{e}}|})$, each element of which is an assignment of values to the variables in $\mathbf{E}$. For each element $\mathbf{e}_k$ of $\vec{\mathbf{e}}$, we wish to infer the marginal probabilities of the variables in $\mathbf{X} \backslash \mathbf{E}$, given that $\mathbf{E} = \mathbf{e}_k$.

Changing the values of a small number of evidence nodes is unlikely to significantly change the probabilities of most state nodes in a large network. EFBP takes advantage of this by only updating regions of the network affected by the new evidence. The algorithm starts by computing the marginal probabilities of non-evidence nodes given the initial evidence values using standard BP (e.g., by flooding). Then, each time the evidence is changed, it maintains a set Δ of nodes affected by the changed evidence variables, initialized to contain only those changed nodes. In each iteration of BP, only the nodes in Δ send messages. Neighbors of nodes in Δ are added to Δ if the messages they receive differ by more than a threshold γ from the final messages they received when they last participated in BP. (The neighbors of a node are the nodes that appear in some factor with it, i.e., its Markov blanket.) In the worst case, the whole network may be added to Δ, and we revert to BP. In most domains, however, the effect of a change usually dies down quickly, and only influences a small region of the network. In such situations, EFBP can converge much faster than BP.

Pseudocode for EFBP is shown in Algorithm 2. $\hat{\mu}_{ts}^{k,i}$ is the message from node t to node s in iteration i of inference run k. $\hat{\mu}_{ts}^{k,c_k}$ is the final message sent from t to s in inference run k. (If s was not in Δ in inference run k, then $\hat{\mu}_{ts}^{k,c_k} = \hat{\mu}_{ts}^{k-1,c_{k-1}}$.) To handle changes to the model from one iteration to the next, we simply add to Δ variables directly affected by that change (e.g., the child variable of a new edge when learning a Bayesian network).

EFBP is based on a similar principle to residual belief propagation (RBP) [44]: focus effort on the regions of the graph that are furthest from convergence. However, while RBP is used to schedule message updates in a single run of belief propagation, the purpose of EFBP is to minimize

Algorithm 2 EFBP(variables $\mathbf{X}$, graphical model $\mathbf{G}$, evidence $\vec{\mathbf{e}}$, threshold γ)

for all t, s: $\hat{\mu}_{ts}^{1,1} \leftarrow 1$

Perform standard BP on $\mathbf{X}$ and $\mathbf{G}$ with evidence $\mathbf{e}_1$.

for $k \leftarrow 2$ to $|\vec{\mathbf{e}}|$ **do**

for all t, s: $\hat{\mu}_{ts}^{k,1} \leftarrow \hat{\mu}_{ts}^{k-1,c_{k-1}}$

$\Delta \leftarrow$ set of evidence nodes that change

between $\mathbf{e}_k$ and $\mathbf{e}_{k-1}$

$converged \leftarrow$ `False`

$i \leftarrow 1$

while $converged =$ `False` **do**

Send messages from all nodes in Δ,

given evidence $\mathbf{e}_k$

$converged \leftarrow$ `True`

for all nodes s that receive messages **do**

if $|\hat{\mu}_{ts}^{k,i} - \hat{\mu}_{ts}^{k-1,c_{k-1}}| > \gamma$ for any $t \in nb(s)$ **then**

Insert s into Δ

end if

if $s \in \Delta$ and $|\hat{\mu}_{ts}^{k,i} - \hat{\mu}_{ts}^{k,i-1}| > \gamma$

for any $t \in nb(s)$ **then**

$converged \leftarrow$ `False`

end if

end for

$i \leftarrow i + 1$

end while

end for

redundant computation when repeatedly running BP on the same network, with varying settings of some of the variables. One could run EFBP using RBP to schedule messages, to speed up convergence.

If the potentials are bounded, EFBP's marginal probability estimates provide bounds on BP's. We can show this by viewing the difference between the beliefs generated by EFBP and those generated by BP as multiplicative error in the messages passed by EFBP:

$$\hat{\mu}_{ts}^{k,i}(x_s) = \tilde{\mu}_{ts}^{k,i}(x_s)\tilde{e}_{ts}^{i}(x_s)$$

where $\hat{\mu}_{ts}^{k,i}(x_s)$ is the message sent by EFBP in iteration i of inference run k (i.e., nodes not in Δ effectively resend the same messages as in $i-1$); $\tilde{\mu}_{ts}^{k,i}(x_s)$ is the message sent if all nodes recalculate their messages in iteration i, as in BP; and $\tilde{e}_{ts}^{i}(x_s)$ is the multiplicative error introduced in iteration i of EFBP.

This allows us to bound the difference in marginal probabilities calculated by BP and EFBP. For simplicitly, we prove this for the special case of binary nodes. The proof uses a measure called the *dynamic range* of a function, defined as follows [64]:

$$d(f) = \sup_{x,y} \sqrt{f(x)/f(y)}$$

Theorem 1. For binary node x_t, the probability estimated by BP at convergence (p_t) can be bounded as follows in terms of the probability estimated by EFBP ($\hat{p}_t$) after n iterations:

$$\begin{aligned} p_t &\geq \frac{1}{(\zeta_t^n)^2[(1/\hat{p}_t)-1]+1} = lb(p_t) \\ p_t &\leq \frac{1}{(1/\zeta_t^n)^2[(1/\hat{p}_t)-1]+1} = ub(p_t) \end{aligned}$$

where $\log \zeta_t^n = \sum_{u \in nb(t)} \log \nu_{ut}^n$, $\nu_{ts}^1 = \delta(\tilde{e}_{ts}^1)d(f_{ts})^2$, and ν_{ut}^i is given by:

$$\begin{aligned} \log \nu_{ts}^{i+1} &= \log \frac{d(f_{ts})^2 \varepsilon_{ts}^i + 1}{d(f_{ts})^2 + \varepsilon_{ts}^i} + \log \delta(\tilde{e}_{ts}^i) \\ \log \varepsilon_{ts}^i &= \sum_{u \in nb(t) \setminus s} \log \nu_{ut}^i \end{aligned}$$

$$\delta(\tilde{e}_{ts}^i) \quad = \quad \sup_{x_s,y_s} \sqrt{\frac{1+\gamma/\left(\tilde{\mu}_{ts}^{k,i}(x_s)\right)}{1-\gamma/\left(\tilde{\mu}_{ts}^{k,i}(y_s)\right)}}$$

Proof.

Since EFBP recalculates messages when $|\hat{\mu}_{ts}^{k,i} - \tilde{\mu}_{ts}^{k,i}| > \gamma$,

$$\begin{aligned} \tilde{\mu}_{ts}^{k,i}(x_s) - \gamma \;&\leq\; \tilde{\mu}_{ts}^{k,i}(x_s)\tilde{e}_{ts}^i(x_s) \;&\leq \tilde{\mu}_{ts}^{k,i}(x_s) + \gamma \\ 1 - \gamma/\tilde{\mu}_{ts}^{k,i}(x_s) \;&\leq\; \tilde{e}_{ts}^i(x_s) \;&\leq 1 + \gamma/\tilde{\mu}_{ts}^{k,i}(x_s) \end{aligned}$$

This allows us to bound the dynamic range of $\tilde{e}_{ts}^i$:

$$d(\tilde{e}_{ts}^i) \leq \sup_{x_s,y_s} \sqrt{\frac{1+\gamma/\left(\tilde{\mu}_{ts}^{k,i}(x_s)\right)}{1-\gamma/\left(\tilde{\mu}_{ts}^{k,i}(y_s)\right)}} = \delta(\tilde{e}_{ts}^i)$$

Theorem 15 of Ihler et al. [64] implies that, for any fixed point beliefs $\{M_t\}$ found by belief propagation, after $n \geq 1$ iterations of EFBP resulting in beliefs $\{\hat{M}_t^n\}$ we have

$$\log d(M_t/\hat{M}_t^n) \leq \sum_{u \in nb(t)} \log \nu_{ut}^n = \log \zeta_t^n$$

It follows that $d(M_t/\hat{M}_t^n) \leq \zeta_t^n$, and therefore $\frac{M_t(1)/\hat{M}_t^n(1)}{M_t(0)/\hat{M}_t^n(0)} \leq (\zeta_t^n)^2$ and $(1-\hat{p}_t)/\hat{p}_t \leq (\zeta_t^n)^2(1-p_t)/p_t$, where p_t and $\hat{p}_t$ are obtained by normalizing M_t and $\hat{M}_t$. The upper bound follows, and the lower bound can be obtained similarly. □

Intuitively, ν_{ut}^i can be thought of as a measure of the accumulated error in the incoming message from u to t in iteration i. It can be computed iteratively using a message-passing algorithm similar to BP.

3.6 Maximizing Expected Utility

The MEU problem in influence diagrams and Markov decision networks can be cast as a series of marginal probability computations with changing evidence. If $u_i(\mathbf{x}_i)$ and $p_i(\mathbf{x}_i|\mathbf{a},\mathbf{e})$ are respectively the utility and marginal probability of utility node i's parents $\mathbf{X}_i$ taking values $\mathbf{x}_i$, then the

expected utility of an action choice is:

$$\begin{aligned} E[U(\mathbf{a}|\mathbf{e})] &= \sum_{\mathbf{x}} P(\mathbf{x}|\mathbf{a},\mathbf{e})U(\mathbf{x}|\mathbf{a},\mathbf{e}) \\ &= \sum_{\mathbf{x}} P(\mathbf{x}|\mathbf{a},\mathbf{e}) \sum_{i} \sum_{\mathbf{x}_i} \mathbb{1}\{\mathbf{X}_i = \mathbf{x}_i\} u_i(\mathbf{x}_i) \\ &= \sum_{i} \sum_{\mathbf{x}_i} u_i(\mathbf{x}_i) \sum_{\mathbf{x}} \mathbb{1}\{\mathbf{X}_i = \mathbf{x}_i\} P(\mathbf{x}|\mathbf{a},\mathbf{e}) \\ &= \sum_{i} \sum_{\mathbf{x}_i} u_i(\mathbf{x}_i) p_i(\mathbf{x}_i|\mathbf{a},\mathbf{e}) \end{aligned}$$

Given a choice of actions, we can treat action nodes as evidence. To calculate the expected utility, we simply need to calculate the marginal probabilities of all values of all utility nodes. Different action choices can be thought of as different evidence.

In principle, the MEU action choice can be found by searching exhaustively over all action choices, computing the expected utility of each using any inference method. However, this will be infeasible in all but the smallest domains. An obvious alternative, particularly in very large domains, is greedy search: starting from a random action choice, consider flipping each action in turn, do so if it increases expected utility, and stop when a complete cycle produces no improvements. This is guaranteed to converge to a local optimum of the expected utility, and is the method we use in our experiments.

Lemma 1. The expected utility $E_{bp}[U]$ of an action choice estimated by BP can be bounded as follows:

$$\begin{aligned} E_{bp}[U] \geq \sum_{i} \sum_{\mathbf{x}_i} u_i(\mathbf{x}_i) \Big[& \mathbb{1}\{u_i(\mathbf{x}_i) > 0\} lb(p_i(\mathbf{x}_i|\mathbf{a},\mathbf{e})) \\ & + \mathbb{1}\{u_i(\mathbf{x}_i) < 0\} ub(p_i(\mathbf{x}_i|\mathbf{a},\mathbf{e})) \Big] \\ E_{bp}[U] \leq \sum_{i} \sum_{\mathbf{x}_i} u_i(\mathbf{x}_i) \Big[& \mathbb{1}\{u_i(\mathbf{x}_i) > 0\} ub(p_i(\mathbf{x}_i|\mathbf{a},\mathbf{e})) \\ & + \mathbb{1}\{u_i(\mathbf{x}_i) < 0\} lb(p_i(\mathbf{x}_i|\mathbf{a},\mathbf{e})) \Big] \end{aligned}$$

Proof. The lemma follows immediately from Theorem 1 and the definition of expected utility. □

Based on this lemma, we can design a variant of EFBP that is guaranteed to produce the same action choice as BP when used with greedy search. Let $\mathbf{A}_i$ denote the set of action choices considered in the ith step of greedy search. If EFBP picks action choice $\mathbf{a} \in \mathbf{A}_i$, BP is guaranteed to make the same choice if the following condition holds:

$$lb(E[U(\mathbf{a})]) > \max_{\mathbf{a}' \in \mathbf{A}_i} ub(E[U(\mathbf{a}')]) \tag{3.1}$$

When the condition does not hold, we revert the messages of all nodes to their values after the initial BP run, insert all changed actions into Δ, and rerun EFBP. If condition 3.1 still does not hold, we halve γ and repeat. (If γ becomes smaller than the BP convergence threshold, we run standard BP.) We call this algorithm EFBP$*$.

Note that using the above bound for search requires a slight modification to the calculation of ν_{ts}^1, since EFBP and BP result in different message initializations for the first BP iteration at every search step. If BP initializes all messages to 1 before every search step, the error function $\tilde{e}_{ts}^1(x_s) = \hat{\mu}_{ts}^{k,1}(x_s)$. If BP initializes its messages with their values from end of the previous search iteration, then ν_{ts}^1 can also be similarly initialized. The calculations of ν_{ts}^i for $i \neq 1$ are not altered.

Let Greedy(I) denote greedy search using inference algorithm I to compute expected utilities.

Theorem 2. Greedy(EFBP$*$) outputs the same action choice as Greedy(BP).

Proof. Condition 3.1 guarantees that EFBP makes the same action choice as BP. Since EFBP$*$ guarantees that the condition holds in every iteration of the final run of EFBP, it guarantees that the action choice produced is the same. □

In practice, EFBP$*$ is unlikely to provide large speedups over BP, since the bound in Theorem 1 must be repeatedly recalculated even for nodes outside Δ. Empirically, running EFBP with a fixed threshold not much higher than BP's convergence threshold seems to yield an action choice of very similar utility to BP's in a fraction of the time.

3.7 Experiments

Our experiments had two goals: to find out how well MLDNs can model a variety of decision problems in relational, uncertain domains, and to evaluate the performance of EFBP. We used two domains: viral marketing and combinatorial auctions. We implemented MLDNs and EFBP as extensions of the Alchemy system [80]. The experiments were run on a cluster of 8-processor machines with 2GB of RAM per processor, running at 2.33 GHz. We used a convergence threshold of 10^{-4} for flooding, including the initial BP run for EFBP, and a threshold of $\gamma = 10^{-3}$ for the remaining iterations of EFBP. We evaluated the utility of the actions chosen by EFBP using a second run of full BP, with a threshold of 10^{-4}. As is usually done, both algorithms were run for a few (10) iterations after the convergence criterion was met.

3.7.1 Viral Marketing

Viral marketing is based on the premise that members of a social network influence each other's purchasing decisions. The goal is then to select the best set of people to market to, such that the overall profit is maximized by propagation of influence through the network. Originally formalized by Domingos and Richardson [41], this problem has since received much attention, including both empirical and theoretical results.

A standard dataset in this area is the Epinions web of trust [117]. Epinions.com is a knowledge-sharing Web site that allows users to post and read reviews of products. The "web of trust" is formed by allowing users to maintain a list of peers whose opinions they trust. We used this network, containing 75,888 users and over 500,000 directed edges, in our experiments. With over 75,000 action nodes, this is a very large decision problem, and no general-purpose MEU algorithms have previously been applied to it (only domain-specific implementations).

We defined an MLDN very similar to Domingos and Richardson's [41] model using the state predicates $\texttt{Buys}(\texttt{x})$ and $\texttt{Trusts}(\texttt{x}_1, \texttt{x}_2)$, and the action predicate $\texttt{MarketTo}(\texttt{x})$. The utility function is represented by the unit clauses $\texttt{Buys}(\texttt{x})$ (with positive utility, representing profits from sales) and $\texttt{MarketTo}(\texttt{x})$ (with negative utility, representing the cost of marketing). The topology of the social

network is specified by an evidence database of $\texttt{Trusts}(\texttt{x}_1, \texttt{x}_2)$ atoms. Information about who has already bought the product can also be incorporated, in the form of $\texttt{Buys}(\texttt{x})$ evidence atoms.

The core of the model consists of two formulas:

$$\texttt{Buys}(+\texttt{x}_1) \wedge \texttt{Trusts}(+\texttt{x}_2, \texttt{x}_1) \Rightarrow \texttt{Buys}(\texttt{x}_2) \tag{3.2}$$

$$\texttt{MarketTo}(+\texttt{x}) \Rightarrow \texttt{Buys}(\texttt{x}) \tag{3.3}$$

In addition, the model includes the unit clause $\texttt{Buys}(\texttt{x})$ with a negative weight, representing the fact that most users do not buy most products. The weight of Formula 3.2 for user pair $(\texttt{x}_1, \texttt{x}_2)$ represents how strongly $\texttt{x}_1$ influences $\texttt{x}_2$, and the weight of Formula 3.3 represents how strongly users are influenced by marketing. These parameters can be estimated from data, as in [41]. For our experiments, however, we varied the weights of the formulas, to see how this affected the behavior of the algorithms.

We ran greedy search with BP and EFBP directly on this model. All results are averages of five runs. We compared BP and EFBP, without lifting. We set the cost of marketing to each user to -1, and the profit from each purchase to 20. The weight of the $\texttt{Buys}(\texttt{x})$ formula was -2. The initial action choice was to market to no one. All results are averages of six runs. Fixing the weight of formula 3.3 at 0.8 for all users, inference times varied as shown in Figure 3.1a as we varied the weight of Formula 3.2. Running utility maximization with BP until convergence was not feasible; instead, we extrapolated from the number of actions considered during the first 24 hours, assuming that search with BP would consider the same number of actions as search with EFBP. Figure 3.1b plots the result of a similar experiment, with influence weight fixed at 0.6 and varying the weight of Formula 3.3. In both cases, EFBP was consistently orders of magnitude faster than BP.

We can also compare the utility obtained by BP and EFBP given a fixed running time of 24 hours. EFBP consistently achieves about 40% higher utility than BP when varying the influence weight, with higher advantage for lower weights. For a marketing weight of 1.0, EFBP achieves about 30% higher utility than BP; this advantage decreases gradually to zero as the weight is reduced (reflecting the fact that fewer and fewer customers buy).

Richardson and Domingos [117] tested various specially designed algorithms on the same net-

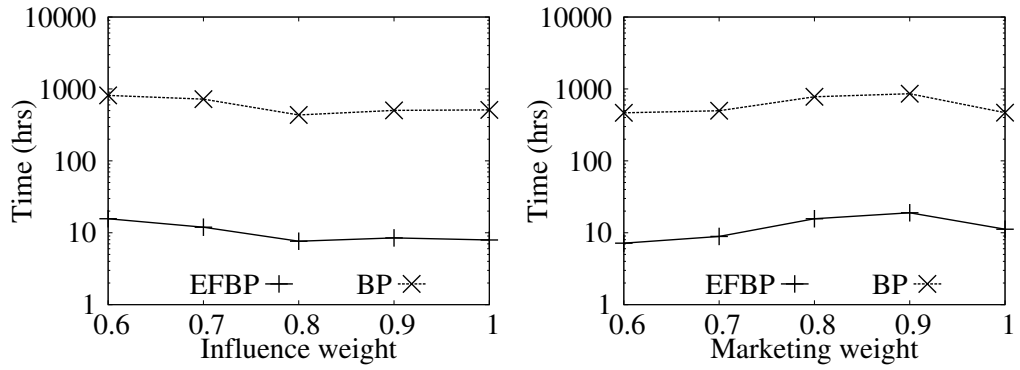


Figure 3.1: Convergence times for viral marketing.

work. They estimated that inference using the model and algorithm of Domingos and Richardson [41], which are the most directly comparable to ours, would have taken hundreds of hours to converge (100 per pass). (This was reduced to 10-15 minutes using additional problem-specific approximations, and a much simpler linear model that did not require search was even faster.) Although these convergence times cannot be directly compared with our own (due to the different hardware, parameters, etc. used), it is noteworthy that EFBP converges faster than the most general of their methods.

Unlike previous hand-coded models, our MLDN can be easily extended to incorporate customer and product attributes, purchase history information, multiple types of relationships, products, actors in the network, marketing actions, etc. Doing so is a direction for future work.

3.7.2 Probabilistic Combinatorial Auctions

Combinatorial auctions can be used to solve a wide variety of resource allocation problems [24]. An auction consists of a set of bids, each of which is a set of requested products and an offered reward. The seller determines which bids to assign each product to. When all requested products are assigned to a bid, the bid is said to be satisfied, and the seller collects the reward. The seller's goal is to choose an assignment of products to bids that maximizes the sum of the rewards of

satisfied bids; this is an NP-hard problem. While there has been much research on combinatorial auctions, it generally assumes that the world is deterministic (with a few exceptions, e.g. Golovin [56]). In practice, however, both the supply and demand for products are subject to many sources of uncertainty. For example, supply of one product may make supply of another less likely because they compete for resources.

We model uncertainty in combinatorial auctions by allowing product assignments to fail, resulting in the loss of reward from the corresponding bids. The following MLDN describes our formulation of the problem.

$$\forall\, \texttt{product} : \texttt{InBid}(\texttt{product}, *\texttt{bid}) \Rightarrow \texttt{Assign}(\texttt{product}, \texttt{bid}) \wedge \texttt{Succeed}(\texttt{product}) \quad (3.4)$$

$$\texttt{Competes}(+\texttt{product1}, +\texttt{product2}) \Rightarrow (\neg\texttt{Succeed}(\texttt{product1}) \vee \neg\texttt{Succeed}(\texttt{product2})) \quad (3.5)$$

$\texttt{InBid}(\texttt{product}, \texttt{bid})$ and $\texttt{Competes}(\texttt{product}, \texttt{product})$ are evidence predicates.
$\texttt{Assign}(\texttt{product}, \texttt{bid})$ is an action predicate. Formula 3.4 is a utility formula; each grounding represents a single bid. Formula 3.5 represents the supply uncertainty. It models competition between two products; the success of one increases the failure probability of the other. This formula creates a network of dependencies among product failures.

We generated bids according to the decay distribution described by Sandholm [124], with $\alpha = 0.75$, and randomly generated 1000 true $\texttt{Competes}()$ atoms from a uniform distribution over product pairs. Figure 3.2 plots the inference time for a 1000-product, 1000-bid auction, varying the weight of Formula 3.5. The results are averages of ten runs. As in the viral marketing experiment, EFBP converges much faster than BP. Since in this case both algorithms can be run until convergence, the expected utilities of the chosen product assignments are extremely similar.

Our MLDN can be easily extended to incorporate more interesting bidding languages and more complex probabilistic dependencies. For instance, each bid could be an arbitrary logical formula, rather than a conjunction. Or it could be a probability distribution depending on several products, representing a form of demand uncertainty. MLDNs could also be used to introduce other kinds of supply and demand uncertainty to combinatorial auction problems.

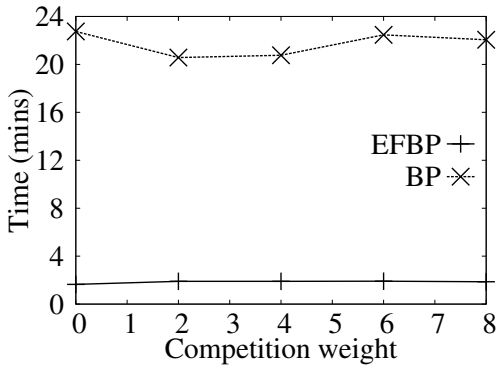


Figure 3.2: Convergence time for combinatorial auctions.

3.8 Conclusions & Future Work

In this chapter, we presented *Markov logic decision networks*, a language that represents first-order probabilistic decision-theoretic problems by adding utility weights to Markov logic formulas. We then developed EFBP, an efficient algorithm for inference in MLDNs (and other repeated inference tasks), and provided theoretical guarantees for it. Applications to viral marketing and combinatorial auctions provided evidence of the flexibility of MLDNs and the efficiency of EFBP.

Directions for future work include: applying MLDNs to other domains, including planning ones; combining EFBP with algorithms for inference in first-order MDPs, and with domain-specific algorithms (e.g., local search algorithms for solving combinatorial auctions); extending BP and EFBP to better handle hard constraints; developing further algorithms for MEU inference in MLDNs; using MLDNs for utility-guided learning, including relational reinforcement learning; applying EFBP to other problems besides utility maximization (e.g. online inference, learning, etc.).

Chapter 4

ERROR BOUNDS FOR APPROXIMATE LIFTED BELIEF PROPAGATION

4.1 Introduction

Lifted inference algorithms (section 2.3) can greatly reduce the cost of inference in statistical relational models. These approaches work by identifying sets of indistinguishable variables, resulting in a *lifted network* that is potentially much smaller than the full ground network. The cost of inference for these algorithms depends on the size of the lifted representation.

However, lifted network construction (LNC) can itself be quite expensive. The cost of LNC is highly sensitive to the representation of the lifted network. Another problem with standard lifted inference algorithms is that they often yield a lifted network very close in size to the fully propositionalized network. In these situations, the expensive process of lifted network construction yields little or no speedup. Both these problems can be averted through approximate lifted inference, which groups together variables that behave similarly, but are not completely identical. Approximate lifting can yield a much smaller model, and therefore a greater speedup. Approximate lifting can also reduce the cost of lifted network construction.

In this chapter, we describe two methods for approximate lifting, originally developed by Singla et al. [129, 132]. First, we discuss *early stopping*, which runs LNC only up to a fixed number of iterations, leading to the construction of a coarser network. Second, we describe a compact *hypercube representation* for the lifted network. This forms the basis for approximate lifting by allowing for a threshold noise in hypercubes while constructing the lifted network. We describe a new bound on the error for a given level of approximation, based on an extension of Ihler et al.'s [64] error bounds for loopy belief propagation in pairwise graphical models. Like Singla and Domingos [131], we use Markov logic as the representation language, but our methods are applicable to many

other first-order probabilistic representations.

4.2 Related Work

De Salvo Braz et al. [31] proposed a form of approximate lifting that combines lifted belief propagation with box propagation [95]. To our knowledge, this algorithm has not been implemented and evaluated, and a detailed description has not yet been published. Sen et al. [127] proposed a lifting algorithm based on the notion of bisimulation, and an approximate variant of it using mini-buckets [33]. Their technique is very effective on some domains, but it is not clear how scalable it is on more general domains, such as those considered by Singla, Nath and Domingos [132] (including Cora entity resolution task where it fails to provide a speed-up).

Kersting et al. [72] proposed an algorithm called *informed* lifted belief propagation (iLBP), which interleaves the LNC steps and the BP steps and avoids creating the exact lifted network when BP converges in fewer iterations than LNC. iLBP can be further sped up by combining it with the approximation schemes and compact representations presented in this chapter; this is a direction for future work.

Gogate and Domingos [53] introduce a sampling version of their lifted weighted model counting approach which does approximate inference, but only over exact lifted models. Van den Broeck et al. [137] present a technique for approximate lifting by first relaxing some of the constraints and then compensating for the relaxations. The set of relaxed constraints determines the quality of the approximation. Their approach provides a spectrum of solutions with lifted BP (fastest) on one extreme and lifted exact inference (most accurate) on the other. They report experiments over relatively small-sized domains, and it is not clear how well their approach would scale. On the other hand, our lifted approximations further relax the lifted network constructed by BP to make lifting scalable to much larger real world domains of practical interest.

Van den Broeck and Darwiche [138] explore the possibility of lifted inference by approximating evidence using a low rank Boolean matrix factorization. The experimental evaluation is presented over a limited set of domains. Our noisy hypercube approximation can be seen as an instance of over-symmetric evidence approximation [138] which we show works well in practice

for a variety of real world domains.

Ahmadi et al. [3] present an approach for breaking the network into tree-structured pieces over which (lifted) inference can be performed independently and results combined later. Our early stopping can be viewed as a form of piecewise decomposition up to depth d, although this decomposition may not in general result in a tree. Further, in our case, the decomposition is only used for lifting (inference still retains all the connections) unlike the piecewise inference of Ahmadi et al. [3] where certain connections might be lost leading to larger deviations from the original network. Exploring the exact connection between the two approaches and making an empirical comparison is an item for future work.

4.3 Message Errors on Factor Graphs

Ihler et al. [64] derived a bound on the effects of message errors on loopy belief propagation. In this chapter, we make use of Ihler et al.'s methods to analyze approximate lifted belief propagation, providing a bound on the error introduced by the approximation. However, Ihler et al.'s result only applies to pairwise Markov networks, i.e. graphical models in which each factor is connected to at most two nodes. While any graphical model can be converted into an equivalent pairwise model, doing so can increase the state dimension of the nodes, and introduce deterministic dependencies, both of which are known to make belief propagation perform poorly [112]. In this section, we extend Ihler et al.'s bound to arbitrary factor graphs, without changing the state dimension or introducing any new dependencies.

In section 2.1, we defined μ_{xf} and μ_{fx}, the BP messages, and M_x, the marginal of a variable. We define a similar quantity for factors:

$$M_{fx,i}(\mathbf{x}) = \prod_{y \in nb(f) \setminus \{x\}} \mu_{yf,i}(y_{\mathbf{x}})$$

Let $\hat{\mu}_{xf}$, $\hat{\mu}_{fx}$, $\hat{M}_x$ and $\hat{M}_{fx}$ be our approximations of these quantities. These approximations

can be viewed as multiplicative errors on the 'true' quantities at some fixed point of BP:

$$\begin{aligned}
\hat{\mu}_{xf,i}(x) &= \mu_{xf,i}(x)e_{xf,i}(x) \\
\hat{\mu}_{fx,i}(x) &= \mu_{fx,i}(x)e_{fx,i}(x) \\
\hat{M}_{x,i}(x) &= M_{x,i}(x)E_{x,i}(x) \\
\hat{M}_{fx,i}(x) &= M_{fx,i}(x)E_{fx,i}(x)
\end{aligned}$$

(Here, e and E are the multiplicative error functions.)

If the potentials are finite, we can bound the growth of the dynamic range of the error with respect to the operations of BP, using logic very similar to [64].

Theorem 3.

$$\begin{aligned}
\log d(e_{xf,i+1}) &\leq \sum_{h \in nb(x)\setminus\{f\}} \log d(e_{hx,i}) \\
\log d(E_{x,i+1}) &\leq \sum_{h \in nb(x)} \log d(e_{hx,i})
\end{aligned}$$

Proof. Both equations can be proved by the same argument as Theorem 6 of [64]. □

Theorem 4.

$$d(e_{fx,i+1}) \leq \frac{d(f)^2 d(E_{fx,i}) + 1}{d(f)^2 + d(E_{xf,i})}$$

Proof.

$$\begin{aligned}
d(e_{fx,i+1}) &= d(\hat{\mu}_{fx}/\mu_{fx}) \\
&= \max_{a,b} \frac{\sum_{\mathbf{x}_a} f(\mathbf{x}_a) M_{fx}(\mathbf{x}_a) E_{fx}(\mathbf{x}_a)}{\sum_{\mathbf{x}_a} f(\mathbf{x}_a) M_{fx}(\mathbf{x}_a)} \\
&\quad \cdot \frac{\sum_{\mathbf{x}_b} f(\mathbf{x}_b) M_{fx}(\mathbf{x}_b)}{\sum_{\mathbf{x}_b} f(\mathbf{x}_b) M_{fx}(\mathbf{x}_b) E_{fx}(\mathbf{x}_b)}
\end{aligned}$$

The result follows from the same argument as Appendix A of Ihler et al. [64]. □

4.4 Early Stopping

The LNC algorithm described in section 2.3 is guaranteed to create the minimal lifted network [131]. Running BP on this network is equivalent to BP on the fully propositionalized network, but potentially much faster. However, the minimal exact lifted network is often very close in size to the propositionalized network. Running LNC to convergence may also be prohibitively expensive. Both of these problems can be alleviated with approximate lifting. The simplest way to make LNC approximate is to terminate LNC after some fixed number of iterations k, thus creating k lifted networks at increasing levels of fineness [99]. No new supernodes are created in the final iteration. Instead, we simply average the superfeature counts $n(X, F)$ for supernode X over all atoms s in X.

Each resulting supernode contains nodes that send and receive the same messages during the first k steps of BP. By stopping LNC after k iterations, we are in effect pretending that nodes with identical behavior up to this point will continue to behave identically. This is a reasonable approximation, since for two nodes to be placed in the same supernode, all evidence and network topology within a distance of $k-1$ links must be identical. If the nodes would have been separated in exact LNC, this would have been a result of some difference at a distance greater than $k-1$. In BP, the effects of evidence typically die down within a short distance. Therefore, two nodes with similar neighborhoods would be expected to behave similarly for the duration of BP.

4.4.1 Error Bound for Early Stopping

The error induced by stopping after k iterations can be bounded by calculating the additional error introduced at each BP step as a result of the approximation. To do this, we perform one additional iteration of LNC (for a total of $k+1$), and compute the difference between the messages calculated at the level k and level $k+1$ at each BP step (after initializing the messages at level $k+1$ from those at level k). To see why this captures the new error in step i of BP, consider a fully ground network initialized with the messages at level k of the lifted network. Run one step of BP on the ground network, passing messages from variables to factors and vice versa. The resulting messages on

the ground network are identical to those yielded by initializing the messages at level $k+1$ the same way, and running a single step of BP. In other words, computations on level $k+1$ tell us what the correct messages in the next step should be, assuming the current messages are correct. (In general, the current messages are actually incorrect, but the errors have already been accounted for.) This allows us to compute the error introduced in the current iteration, and incorporate it into the bound.

Let $sn_k(x)$ and $sf_k(f)$ respectively be the supernode containing x and the superfactor containing f at level k. We view the difference between the messages at level k and level $k+1$ as additional multiplicative error introduced in each BP step in the variable-factor messages at level k:

$$\hat{\mu}_{xf,i}(x_s) = \tilde{\mu}_{xf,i}(x_s)\tilde{e}_{xf,i}(x_s)$$

where $\hat{\mu}_{xf,i}(x_s)$ is the message from $sn_k(x)$ to $sf_k(f)$ in BP step i; $\tilde{\mu}_{xf,i}(x_s)$ is the message from $sn_{k+1}(x)$ to $sf_{k+1}(f)$ in step i, after initializing $sn_{k+1}(x)$ from $sn_k(x)$ in step $i-1$; and $\tilde{e}_{xf,i}(x_s)$ is the multiplicative error introduced in the message from x to f at level k in step i.

This allows us to bound the difference in marginal probabilities calculated by ground BP and lifted BP at level k.

Theorem 5. If ground BP converges, then for node x, the probability estimated by ground BP at convergence (p_x) can be bounded as follows in terms of the probability estimated by level k lifted BP ($\hat{p}_x$) after n BP steps:

$$\begin{aligned} p_x &\geq \frac{1}{(\zeta_{x,n})^2[(1/\hat{p}_x)-1]+1} = lb(p_x) \\ p_x &\leq \frac{1}{(1/\zeta_{x,n})^2[(1/\hat{p}_x)-1]+1} = ub(p_x) \end{aligned}$$

$$
\begin{aligned}
\textit{where} \quad \log \zeta_{x,n} &= \sum_{f \in nb(x)} \log \nu_{fx,n} \\
\nu_{fx,1} &= d(f)^2 \\
\log \nu_{xf,i+1} &= \sum_{h \in nb(x) \setminus \{f\}} \log \nu_{hx,i} + \log \delta_{xf,i+1} \\
\log \nu_{fx,i} &= \log \frac{d(f)^2 \varepsilon_{fx,i} + 1}{d(f)^2 + \varepsilon_{fx,i}} \\
\log \varepsilon_{fx,i} &= \sum_{y \in nb(f) \setminus \{x\}} \log \nu_{yf,i} \\
\delta_{xf,i} &= \sup_{x_s, y_s} \sqrt{\frac{\hat{\mu}_{xf,i}(x_s)}{\tilde{\mu}_{xf,i}(x_s)} \frac{\tilde{\mu}_{xf,i}(y_s)}{\hat{\mu}_{xf,i}(y_s)}} \\
d(f) &= \sup_{x,y} \sqrt{f(x)/f(y)}
\end{aligned}
$$

Proof. $d(f)$ is the *dynamic range* of function f, as defined in Ihler et al. [64]. Since we can calculate $\hat{\mu}_{xf,i}(x_s)$ and $\tilde{\mu}_{xf,i}(x_s)$, the dynamic range of the error function can be calculated as follows:

$$
d(\tilde{e}_{xf,i}) = \sup_{x_s, y_s} \sqrt{\frac{\hat{\mu}_{xf,i}(x_s)}{\tilde{\mu}_{xf,i}(x_s)} \frac{\tilde{\mu}_{xf,i}(y_s)}{\hat{\mu}_{xf,i}(y_s)}} = \delta_{xf,i}
$$

Using the same logic as Theorem 15 of Ihler et al. [64], for any fixed point beliefs $\{M_x\}$ found by ground BP, after $n \geq 1$ steps of BP at level k resulting in beliefs $\{\hat{M}_{x,n}\}$

$$
\log d(M_x / \hat{M}_{x,n}) \leq \sum_{f \in nb(x)} \log \nu_{fx,n} = \log \zeta_{x,n}
$$

It follows that $d(M_t / \hat{M}_{x,n}) \leq \zeta_{x,n}$, and therefore:

$$
\begin{aligned}
& \frac{M_x(1)/\hat{M}_{x,n}(1)}{M_x(0)/\hat{M}_{x,n}(0)} \leq (\zeta_{x,n})^2 \\
\text{and} \qquad & (1 - \hat{p}_x)/\hat{p}_x \leq (\zeta_{x,n})^2 (1 - p_x)/p_x
\end{aligned}
$$

where p_x and $\hat{p}_x$ are obtained by normalizing M_x and $\hat{M}_{x,n}$. The upper bound follows, and the lower bound can be obtained similarly. □

Intuitively, $\nu_{ut,i}$ can be thought of as a measure of the accumulated error in the message from u to t in BP step i. It can be computed iteratively using a message-passing algorithm similar to lifted BP on the level $k+1$ graph.

4.5 Noise-Tolerant Hypercubes

Another approach to approximate lifting is to allow for marginal error in the representation of each supernode (superfeature) in the lifted network. We use a hypercube based representation for the lifted network. This forms the basis for approximate lifting by giving a framework for representing noise during the lifted network construction phase.

Hypercube Representation

A *hypercube* is a vector of sets of literals, $[S_1, S_2, \ldots, S_k]$; the corresponding set of tuples is the Cartesian product $S_1 \times S_2 \times \ldots \times S_k$. A supernode or superfeature can be represented by a union of disjoint hypercubes. Hypercube representation can be exponentially more compact than both the extensional and resolution-like representations. Hypercube (or similar) representation can also be used effectively in lifted exact inference algorithms such as FOVE [134]. In general, there can be more than one minimal decomposition, and finding the best one is an NP-hard problem. (In two dimensions, it is equivalent to the minimum biclique partition problem, Amilhastre et al. [5].) In the construction of hypercubes, first, a bounding hypercube is constructed, i.e., one which includes all the tuples in the set. This is a crude approximation to the set of tuples which need to be represented. The hypercube is then recursively sub-divided so as to split apart tuples from non-tuples, using a heuristic splitting criterion (see Algorithm 3). Other criteria such as information gain can also be used, as is commonly done in decision trees. The process is continued recursively until each hypercube either contains all valid tuples or none (in which case it is discarded). This process does not guarantee a minimal splitting, since hypercubes from two different branches could potentially be merged. Therefore, once the final set of hypercubes is obtained, we recursively merge the resulting hypercubes in a bottom-up manner to get a minimal set. (Running the bottom approach directly on individual set of tuples can be inefficient.)

Hypercube Join: When joining two supernodes, we join each possible hypercube pair (each element of the pair coming from the respective supernode). Joining a pair of hypercubes simply corresponds to taking an intersection of the common variables and keeping the remaining ones as

Algorithm 3 FormHypercubes(Tuple set $\mathbf{T}$)

Form bounding hypercube C_{bound} from $\mathbf{T}$

$\mathbf{H} \leftarrow \{C_{bound}\}$

while $\mathbf{H}$ contains some impure hypercube C **do**

Calculate r_C, the fraction of true tuples $\in C$

for all variables v in C **do**

Decompose C into $\mathbf{C} = (C_1, \ldots, C_k)$ byvalue of v (one hypercube per value)

Calculate $(r_{C_1}, \ldots, r_{C_k})$

$C^+ \leftarrow$ merge all $C_i \in \mathbf{C}$ with $r_{C_i} > r_C$

$C^- \leftarrow$ merge all $C_i \in \mathbf{C}$ with $r_{C_i} \leq r_C$

Calculate r_{C^+}

end for

Split C into (C^+, C^-) with highest r_{C^+}

Remove C from $\mathbf{H}$; insert C^+ and C^-

end while

while $\mathbf{H}$ contains some mergeable pair (C_1, C_2) **do**

Replace C_1 and C_2 with merged hypercube C_{new}

end while

is. Instead of joining each possible pair of hypercubes, an index can be maintained which tells which pairs will have a non-zero intersection.

Hypercube Project: The project operation now projects superfeature hypercubes onto the arguments the superfeature shares with each of its predicates. Each supernode hypercube maintains a separate set of counts. This presents a problem because different superfeatures may now project onto hypercubes which are neither disjoint nor identical. Therefore, the hypercubes resulting from the project operation have to be split into finer hypercubes such that each pair of resulting hypercubes is either identical with each other or disjoint. This can be done by choosing each intersecting (non-disjoint) pair of hypercubes in turn and splitting them into the intersection and the remaining components. The process continues until each pair of hypercubes is disjoint.

Noise Tolerance

During the top-down construction of hypercubes, instead of refining the hypercubes until the end, we stop the process on the way allowing for at most a certain maximum amount of noise in each hypercube. The goal is to obtain the largest possible hypercubes while staying within the noise tolerance threshold. Different measures of noise tolerance can be used. One such measure which is simple to use and does well in practice is the number of tuples not sharing the majority truth value in the hypercube. Depending on the application, other measures can be used.

This scheme allows for a more compact representation of supernodes and superfeatures, potentially saving both memory and time. This approximation also allows the construction of larger supernodes, by virtue of grouping together certain ground predicates which were earlier in different supernodes. These gains come at the cost of some loss in accuracy due to the approximations introduced by noise tolerance. However, in experiments on a variety of domains, it has been shown that the loss in accuracy is offset by the gains in both time and memory [132].

4.5.1 Error Bound for Noisy Hypercubes

The methods of [64] can be extended to bound the error introduced by noise-tolerant hypercube formation, using logic similar to Theorem 5 (the error bound for early stopping).

Note that the noisy hypercube approximation is equivalent to flipping the values of certain nodes: true evidence nodes may become false or unknown, or vice versa. For the flipped nodes, the bound is vacuous: the error in the probability may be at most 1. However, we can bound the change in probability on the remaining nodes in the network, by bounding the change in the outgoing messages from the factors.

We can place an upper bound on the errors by assuming that the flipped nodes will maximally alter the outgoing messages from the factors adjacent to them. Since each factor f corresponds to some MLN formula with weight w_f, $f(\mathbf{x})$ for all states is between 1 and e^{w_f}. As a result, the normalized outgoing messages $\mu_{fx}(a)$ to node x are between 1 and e^{w_f} for all a, both before and after the introduction of noise. The dynamic range of the error function can be bounded as follows:

$$d(e_{fx}) \leq \max\left(\sqrt{e^{2w_f}}, \sqrt{e^{-2w_f}}\right)$$

Thus, Theorem 5 can be modified as follows for the noisy hypercube case:

Theorem 6. If ground BP converges, then for node x, the probability estimated by ground BP at convergence (p_x) can be bounded as follows in terms of the probability $\hat{p}_x$ estimated by lifted BP after n BP steps with some set $X_{flipped}$ of the nodes flipped to a different evidence value.

$$\begin{aligned} p_x &\geq \frac{1}{(\zeta_{x,n})^2[(1/\hat{p}_x) - 1] + 1} = lb(p_x) \\ p_x &\leq \frac{1}{(1/\zeta_{x,n})^2[(1/\hat{p}_x) - 1] + 1} = ub(p_x) \end{aligned}$$

where $\log \zeta_{x,n} = \sum_{f \in nb(x)} \log \nu_{fx,n}$,

$$\log \nu_{xf,i+1} = \sum_{h \in nb(x) \setminus \{f\}} \log \nu_{hx,i}$$

For factors f adjacent to some node $x \in X_{flipped}$,

$$\nu_{fx,i} = \max\left(\sqrt{e^{2w_f}}, \sqrt{e^{-2w_f}}\right)$$

For all other factors f,

$$\begin{aligned} \log \nu_{fx,i} &= \log \frac{d(f)^2 \varepsilon_{fx,i} + 1}{d(f)^2 + \varepsilon_{fx,i}} \\ \text{and} \quad \nu_{fx,1} &= d(f)^2 \\ \log \varepsilon_{fx,i} &= \sum_{y \in nb(f) \setminus \{x\}} \log \nu_{yf,i} \\ d(f) &= \sup_{x,y} \sqrt{f(x)/f(y)} \end{aligned}$$

Note that (unlike the early stopping bound), calculating the error bound for noise-tolerant hypercube formation requires running a message-passing algorithm similar to BP on the exact lifted network.

4.6 Conclusions

This chapter extended Ihler et al.'s [64] BP error bounds to arbitrary factor graphs, and used this result to bound the effects of approximation on two lifted inference approaches. Singla et al. [132] have previously shown that these approximate lifted BP approaches work well empirically; our analysis provides them with a theoretical foundation as well.

Chapter 5

EFFICIENT LIFTING FOR ONLINE PROBABILISTIC INFERENCE

5.1 Introduction

As discussed in section 2.3, lifted inference algorithms perform probabilistic inference more efficiently by reasoning on a *lifted model*, i.e. a representation where sets of indistinguishable variables are grouped together. Since the lifted representation can much smaller than the fully ground model, lifted inference is potentially much more efficient than propositionalized inference. However, constructing the lifted model can itself be quite costly; sometimes more so than the actual probabilistic inference. This is particularly problematic in online applications, where the lifted network must be constructed for each new set of observations. This can be extremely wasteful, since the evidence typically changes little from one set of observations to the next. The same is true in many other problems that require repeated inference, such as utility maximization, MAP inference (finding the most probable state of a subset of the variables while summing out others), interactive inference, parameter and structure learning, etc.

In this chapter, we propose ΔLNC, an efficient algorithm for updating the structure of an existing lifted network with incremental changes to the evidence. This allows us to construct the lifted network once for the initial inference problem, and amortize the cost over the subsequent problems. ΔLNC can also be used to efficiently update an approximate lifted network. Experiments on video segmentation and viral marketing problems show that approximate ΔLNC provides great speedups over both ground belief propagation and standard LNC, without significantly affecting the quality of the solutions.

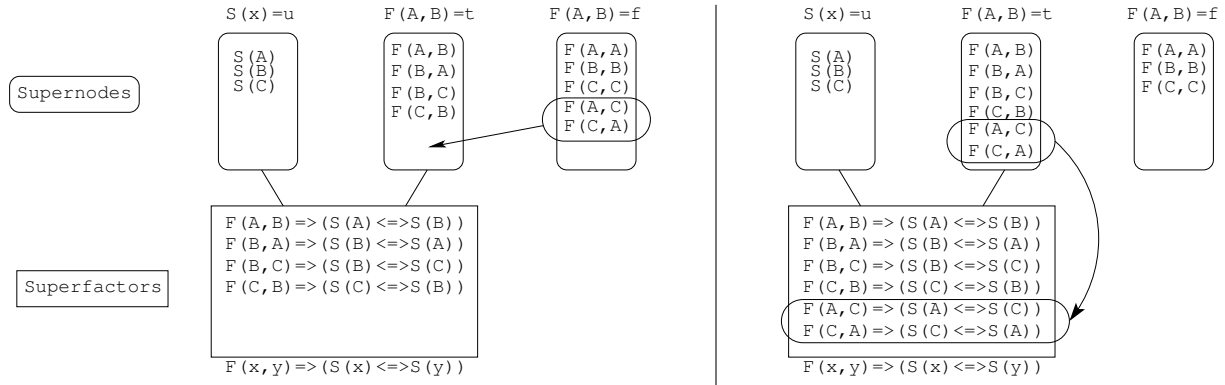
5.2 Lifted Online Inference

Even with the benefit of the approximation methods described in the previous chapter, LNC can be expensive. This is particularly problematic in applications involving several queries on a single network, with different values for evidence variables. We refer to this setting as *online inference*. Various algorithms exist for online inference and related problems on graphical models (e.g. [2, 34]), but to our knowledge, no algorithms exist for lifted online probabilistic inference.

In the online setting, it can be extremely wasteful to construct the lifted network from scratch for each new set of observations. Particularly if the changes are small, the structure of the minimal network may not change much from one set of observations to the next. To take advantage of this, we propose an algorithm called ΔLNC. ΔLNC efficiently updates an existing lifted network, given a set of changes to the values of evidence variables.

Given a set Δ of changed atoms, ΔLNC works by retracing the steps these nodes would have taken over the course of LNC. As the network is updated, we must also keep track of which other atoms were indirectly affected by the changes, and retrace their paths as well. At the start of ΔLNC, each changed node x is placed in the initial supernode corresponding to its new value ($true$, $false$, $unknown$), and removed from its previous starting supernode. The next step is to update the superfeatures connected to the starting supernodes. The tuples containing x must be removed from the superfeatures connected to x's original starting supernode. These tuples are then added to matching superfeatures connected to the newly assigned supernode, creating a new superfeature if no match is found.

At the next step, we must also consider the nodes that would have received messages from the factors that changed in the first iteration of BP. These nodes are added to Δ. Now, for each node $x \in \Delta$, we start by removing it from its level two supernode, and then locate a new supernode to insert it into. This is done by calculating the number of tuple projections to x from each of the superfeatures connected to its level one supernode. If the projection counts match any of the children of x's initial supernode, x is added to that child. If not, a new child supernode is created for x.



(a) Initial lifted network

(b) Updated lifted network

Figure 5.1: The first iteration of ΔLNC on an MLN with binary predicates $S(x)$ and $F(x)$, and formula $F(x, y) => (S(x) <=> S(y))$. Trivially false superfeatures are omitted. In the above example, all $S(x)$ atoms are $unknown$, and two $F(x, y)$ atoms are changing from $false$ to $true$.

ΔLNC thus closely mirrors standard LNC, alternately refining the supernodes and the superfeatures. The only difference is that the finer supernodes and superfeatures do not need to be constructed from scratch; only the changed tuples need to be moved. Let $sn_k(x)$ be the supernode containing node x at level k. Pseudocode for ΔLNC is given in Algorithm 4. (Like Algorithm 1, this pseudocode assumes for clarity that each predicate occurs at most once in each clause.) See Figure 5.1 for an example.

Note that the lifted network produced by running k iterations of ΔLNC is identical to that produced by running standard LNC. The error bounds described in the previous chapter apply to approximate ΔLNC as well. ΔLNC can also be run to convergence to update an exact lifted network.

5.3 Experiments

We evaluated the performance of ΔLNC on two problems: video segmentation and viral marketing. The experiments were run on 2.33 GHz processors with 2 GB of RAM. The purpose of the experiment was to compare the speed and accuracy of ΔLNC to those of standard $beliefpropagation$.

5.3.1 Video Segmentation

BP is commonly used to solve a variety of vision problems on still images, but it can be very expensive to run BP on videos (e.g. [142]). ΔLNC can be used to make BP on videos much more scalable. We illustrate this by running ΔLNC on a simple binary video segmentation problem. Starting from a complete segmentation of the first frame into background and foreground, we wish to segment the remaining frames. For each frame, pixels whose colors change by less than some threshold keep their segmentation labels from the previous frame as biases for their new labels. We then infer new segmentation labels for all pixels.

The model is defined by an MLN with predicates $\texttt{KnownLabel}(\texttt{x}, \texttt{l})$, where $\texttt{l} \in \{\texttt{Background}, \texttt{Foreground}, \texttt{Unknown}\}$; $\texttt{PredictedLabel}(\texttt{x}, \texttt{l})$; $\texttt{Color-Difference}(\texttt{x}, \texttt{y}, \texttt{d})$. The formulas are:

$$\texttt{KnownLabel}(\texttt{x}, \texttt{l}) \Rightarrow \texttt{PredictedLabel}(\texttt{x}, \texttt{l}) \tag{5.1}$$

$$\texttt{ColorDifference}(\texttt{x}, \texttt{y}, \texttt{d}) \quad \Rightarrow \quad (\texttt{PredictedLabel}(\texttt{y}, \texttt{l}) \Leftrightarrow \texttt{PredictedLabel}(\texttt{x}, \texttt{l})) \tag{5.2}$$

We also introduced deterministic constraints that $\texttt{KnownLabel}(\texttt{x}, \texttt{l})$ and $\texttt{PredictedLabel}(\texttt{x}, \texttt{l})$ can only be $true$ for a single $\texttt{l}$, and $\texttt{PredictedLabel}(\texttt{x}, \texttt{Unknown})$ must be $false$. In our experiments, $\texttt{ColorDifference}$ was calculated as the RGB Euclidean distance, and discretized into six levels. $\texttt{KnownLabel}(\texttt{x}, \texttt{Unknown})$ was set to $true$ if the difference from one frame to the next was over 30.0. The weight of Formula 5.1 was 1.0, and the weight of Formula 5.2 varied inversely with d, up to 2.0 for the smallest difference level.

We ran ΔLNC with four refinement levels (LBP-4), and compared it to ground belief propagation (BP). 1000 steps of max-product BP were performed, to infer the globally most probable labels. We predicted labels for 10 frames of two videos: a space shuttle takeoff [97], and a cricket demonstration [48]. The first frames were segmented by hand. The results are in Table 5.1. Figure 5.2 shows the segmentations produced in the last frames. In both experiments, ΔLNC provided dramatic speedups over full ground BP, and produced a similar (but not identical) segmentation. On average, ΔLNC was about 28X faster than the initial LNC on the shuttle video, and 48X faster on the cricket video.

Table 5.1: Results of video stream experiments.

Shuttle takeoff	BP	LBP-4
Initial number of nodes	76800	2992
Initial number of factors	306080	13564
Initial LNC time (s)	-	200.46
Average ΔLNC time (s)	-	6.95
Average BP time (s)	394.46	30.91
Total time (s)	**3900.92**	**558.3**
Cricket	BP	LBP-4
Initial number of nodes	76800	4520
Initial number of factors	306080	19965
Initial LNC time (s)	-	870.69
Average ΔLNC time (s)	-	18.29
Average BP time (s)	398.55	78.99
Total time (s)	**3976.40**	**1619.07**

5.3.2 Viral Marketing

We also evaluated ΔLNC on the viral marketing MLDN described in section 3.7.1. We used a greedy search strategy, changing the value of one action at a time and recalculating the marginals using belief propagation. If the change does not prove to be beneficial, it is reversed.

On a lifted network, greedy search involves finding the optimal number of atoms to flip within each `MarketTo(x)` supernode. In an exact lifted network, all atoms within each supernode would be identical, and the choice of which specific atoms we flip is not relevant. This is not precisely true for approximate lifted networks, but it is a useful approximation to make, since it reduces the number of flips that need to be considered. To find the greedy optimum within a supernode, we simply need to start with all its atoms set to $true$ or all its atoms set to $false$, and flip one atom at

(a) Last frame (b) BP labeling (c) LBP-4 labeling (d) Last frame (e) BP labeling (f) LBP-4 labeling

Figure 5.2: Output labelings for the final frames of both videos.

Table 5.2: Results of viral marketing experiment.

	BP	LBP-2	LBP-1
Initial number of nodes	75888	37380	891
Initial number of factors	508960	484895	201387
Initial LNC time (s)	-	478.59	336.60
Average ΔLNC time (s)	-	0.82	0.68
Average BP time (s)	20.79	20.09	13.41
Total flips	4038	4003	5863
Best utility found	102110	106965	137927
Improvement in utility	**668**	**5523**	**36485**

a time until we make a non-beneficial change (which we then reverse). The decision of whether to start with all atoms set to $true$ or $false$ is also made greedily.

We compared three algorithms: fully ground belief propagation (BP), lifted BP with LNC terminated after one refinement (LBP-1), and lifted BP with two refinements (LBP-2). Formula 3.2 had a weight of 0.6, and Formula 3.3 had a weight of 0.8. $\texttt{Buys}(\texttt{x})$ had a weight of -2 and a utility of 20. $\texttt{MarketTo}(\texttt{x})$ had a utility of -1. BP was run for 25 iterations. The starting action choice was to market to no one. The entire MEU inference was given 24 hours to complete, and all algorithms reported the highest utility found within that time. The final solution quality was evaluated with BP on the ground network, with 100 iterations.

The results of the experiment are shown in Table 5.2. LBP-1 was the only one of the three algorithms to converge to a local optimum within the time limit. ΔLNC achieved approximately a 500X speedup over the initial LNC. ΔLNC also significantly lowers the BP running time, due to the smaller factor graph. LBP yields a much higher utility improvement than ground BP in a comparable number of flips, since each non-beneficial flip on a lifted network tells us that we need not consider flipping other atoms in the same supernode. LBP-1 is competitive in performance with the most directly comparable algorithm of Richardson and Domingos [117].

5.4 Conclusions & Future Work

In this chapter, we proposed ΔLNC, an algorithm for efficiently updating an exact or approximate lifted network, given changes to the evidence. ΔLNC works by retracing the path the changed atoms would have taken during LNC, modifying the network as necessary. Experiments on video segmentation and viral marketing problems show that ΔLNC provides very large speedups over standard LNC, making it applicable to a wider range of problems.

There are several directions for future work. ΔLNC can be applied to a variety of problems, including learning and MAP inference. On video problems, it can be made even more scalable by combining it with efficient BP algorithms designed specifically for vision (e.g. [46]). Using BP for more complicated problems will also require more sophisticated models of inter-frame dependencies (e.g. [65]). The ability to efficiently update a lifted network may also allow us to lift other probabilistic inference algorithms besides BP.

Algorithm 4 ΔLNC(changed nodes Δ, lifted networks $(\mathbf{L}_1, \ldots, \mathbf{L}_k)$)

for all nodes $x \in \Delta$ **do**
  Remove tuples containing x from $sn_1(x)$ and
    superfeatures connected to it
  Move x to $X_{new} \in \mathbf{L}_1$ of appropriate value v
  for all features t containing x **do**
    if there exists superfeature F connected to X_{new}
      from the same clause as t, such that each atom in
      t is in a supernode connected to F **then**
      Insert t into F
    else
      Create new superfeature F, and insert t
    end if
  end for
end for
for all $2 \leq i \leq k$ **do**
  Insert $\bigcup_{x \in \Delta} nb(nb(x))$ into Δ
  for all nodes $x \in \Delta$ **do**
    Remove tuples containing x from superfeatures
      connected to $sn_i(x)$
    if $sn_{i-1}(x)$ has child X in L_i with projection counts
      $n(X, F) = n(x, F)$ for all $F \in nb(X)$ **then**
      Move x from the previous $sn_i(x)$ to X
    else
      Move x to a newly created child of $sn_{i-1}(x)$
    end if
    for all features t containing x **do**
      if there exists superfeature F connected to the new
        $sn_i(x)$ from the same clause as t, such that each
        atom in t is in a supernode connected to F **then**
        Insert t into F
      else
        Create new superfeature F, and insert t
      end if
    end for
  end for
end for
return updated $(\mathbf{L}_1, \ldots, \mathbf{L}_k)$

Chapter 6

LEARNING MULTIPLE HIERARCHICAL CLUSTERINGS

6.1 Introduction

In recent years, relational clustering has been successfully used to learn simple models of complex domains, both for human interpretability and to make predictions about unobserved relations. Most existing approaches to relational clustering learn flat clusterings, grouping together similar objects in a relational database. However, in complex relational domains, a flat clustering is often an excessively simplistic model of the data. Different properties of the same objects may be best explained by different partitions of the objects. Some attributes and relations may be best modeled with a coarser or finer clustering than others. Multiple clustering and hierarchical clustering have been previously studied in isolation, by Kok and Domingos [77] and Roy et al. [122] respectively. In this chapter, we present the first unified model for multiple hierarchical relational clusterings. Each atom in the database is predicted by the hierarchical clustering that best explains it. Within each clustering, each prediction takes into account information at multiple levels of the hierarchy, using shrinkage to learn more robust parameters.

This line of work was inspired by Kemp et al.'s [70] *Infinite Relational Model (IRM)*, which jointly clusters objects and predicates in a relational database. Given the cluster assignments, the truth value of an atom can be predicted from the *cluster combination* the atom belongs to (i.e., the vector of cluster assignments of the predicate and objects associated with the atom).

Building on this idea, Kok and Domingos' [77] *Multiple Relational Clustering (MRC)* learned several cross-cutting clusterings of a domain, rather than trying to explain a database from a single clustering. The rationale is that different clusterings may be better for predicting different subsets of the atoms; it may not be possible to learn a single clustering that satisfactorily explains the entire database. Kok and Domingos empirically demonstrated that learning multiple clusterings

can greatly improve prediction quality.

One limitation of IRM and MRC is that both models learn flat clusterings rather than hierarchical models. Hierarchical models can be useful both for human interpretability and for improving predictive performance. Roy et al. [122] proposed a generative model for *annotated hierarchies* of relational data, where objects are hierarchically clustered into a single tree, and each predicate is associated with the tree-consistent partition of objects that best explains it. We refer to their system as the *Annotated Hierarchy Model (AHM)*. The motivation for this system was twofold: (i) to learn compact, interpretable structures suitable for data analysis, and (ii) to provide a model of human learning. However, its capabilities as a predictive model were not investigated.

The rest of this chapter is organized as follows. In section 6.2, we describe a unified model for multiple hierarchical relational clustering. In section 6.3, we describe an efficient algorithm for learning the weights and structure of the model. Section 6.4 describes a preliminary empirical evaluation of the single-hierarchy version of the algorithm. Finally, section 6.5 lists directions for future work.

6.2 Model

We start with a standard Naive Bayes mixture model, and then describe a series of extensions capturing relational, hierarchical and multiple clustering models.

6.2.1 Naive Bayes Mixture Model

A Naive Bayes mixture model assumes that an object's attributes $(X_1, \ldots, X_a)$ are independent given the object's class, C. The joint probability is simply:

$$P(C, X_1, \ldots, X_a) = P(C) \prod_{i=1}^{a} P(X_i|C) \tag{6.1}$$

6.2.2 Hierarchical Clustering Model

Hierachical models allow more robust predictions through the use of *shrinkage* [92], a statistical technique that smoothes parameter estimates of data-sparse feature towards their more data-rich

ancestors in the hierarchy.

We define $(\mathcal{D}^{(1)}, \ldots, \mathcal{D}^{(h)})$, where $\mathcal{D}^{(i)}$ is the set of cluster symbols $\{c_1^{(i)}, \ldots, c_k^{(i)}\}$ in the ith level of the hierarchy. Each cluster $c_j^{(i)}$ with $1 \leq i < h$ has a *parent* cluster $\rho(c_j^{(i)})$ in layer $\mathcal{D}^{(i+1)}$.

The model contains one cluster assignment variable $C^{(i)}$ for each level i in the hierarchy. $\mathcal{D}^{(i)}$ is the set of possible values for $C^{(i)}$. If $C^{(i)} = c$, then $C^{(i+1)} = \rho(c)$; this ensures that the cluster assignments are consistent with the hierarchy structure.

The cluster assignments are generated top-down; $C^{(h)}$ is generated from a multinomial distribution over the symbols in $\mathcal{D}^{(h)}$. For the finer layers, cluster memberships are generated from a multinomial distribution over a set of siblings, i.e. clusters that share the same parent. $P(c)$ is the multinomial parameter corresponding to cluster c.

The attributes are generated conditioned on the vector of cluster assignments for the object. The distribution for each attribute is a log-linear model, with one feature for each level in the hierarchy:

$$P(X_j | C^{(1)}, \ldots, C^{(h)}) = \frac{1}{1 + \exp\left(-\sum_{i=1}^{h} w_{C^{(i)}}^{(j)}\right)}$$

The full joint probability distribution is:

$$\begin{aligned} P(\mathbf{C}, \mathbf{X}) = & \prod_{i=1}^{h} P(C^{(i)}) \\ & \prod_{X_j \in \mathbf{X}^+} \frac{1}{1 + \exp\left(-\sum_{i=1}^{h} w_{C^{(i)}}^{(j)}\right)} \\ & \prod_{X_j \in \mathbf{X}^-} \frac{1}{1 + \exp\left(\sum_{i=1}^{h} w_{C^{(i)}}^{(j)}\right)} \end{aligned} \tag{6.2}$$

Where $\mathbf{C} = (C^{(1)}, \ldots, C^{(h)})$; $\mathbf{X} = (X_1, \ldots, X_a)$; $\mathbf{X}^+$ and $\mathbf{X}^-$ are respectively the sets of true and false attributes in $\mathbf{X}$.

6.2.3 Relational Clustering Model

Relational models such as the *stochastic blockmodel* [61, 104] jointly cluster a set of object and predicate symbols $\{o_1, \ldots, o_n\}$ based on their relations. The traditional blockmodel formulation

draws the cluster memberships from a multinomial distribution with a fixed number of components, k. Let $\{c_1, \ldots, c_k\}$ be the cluster symbols. Let $\mathbf{C} = (C_1, \ldots, C_n)$ be the vector of cluster membership variables for the n object and predicate symbols (i.e., $C_i = c_j$ means that item o_i is assigned to cluster c_j). The probability of cluster assignment $\mathbf{C}$ is:

$$P(\mathbf{C}) = \prod_{i=1}^{n} P(C_i) = \prod_{j=1}^{k} P(c_k)^{|c_k|}$$

(Here, $P(c_k)$ is the probability of component c_k in the multinomial, and $|c_k|$ is the number of items assigned to component c_k.)

Each atom is then generated conditioned on the vector of cluster assignments of the items in the relation:

$$X = o_r(o_1, \ldots, o_p) \sim \text{Bernoulli}(\eta(C_r, C_1, \ldots, C_p))$$

We refer to a vector of cluster assignments as a *cluster combination*. Let M_X be the cluster combination that explains atom X as described above, and let η_{M_X} be the parameter of the corresponding Bernoulli distribution. $\mathbf{X}$ is the set of atoms in the domain; $\mathbf{X}^+$ and $\mathbf{X}^-$ are respectively the sets of true and false atoms in $\mathbf{X}$. The full joint probability distribution is:

$$P(\mathbf{C}, \mathbf{X}) = \prod_{j=1}^{k} P(c_k)^{|c_k|} \prod_{X \in \mathbf{X}^+} \eta_{M_X} \prod_{X \in \mathbf{X}^-} (1 - \eta_{M_X}) \tag{6.3}$$

Note that the number of components need not be fixed in advance; IRM generates the clusters using a Chinese Restaurant Process [110], allowing the data to dictate the number of components.

6.2.4 Hierarchical Relational Clustering Model

In this section, we describe a hierarchical relational clustering model that generalizes stochastic blockmodels much like the model in section 6.2.2 extends Naive Bayes mixture models. The hierarchical model contains a vector of cluster assignments for each level i in the hierarchy: $\mathbf{C}^{(i)} = (C_1^{(i)}, \ldots, C_n^{(i)})$. The full hierarchical clustering is defined by the matrix $\mathbf{C} = (\mathbf{C}^{(1)}, \ldots, \mathbf{C}^{(h)})$, each row of which captures the cluster assignments in one level of the hierarchy.

As in section 6.2.2, each cluster $c_j^{(i)}$ has a *parent* in the layer above, and each cluster membership is generated from a multinomial distribution over a set of siblings.

As in section 6.2.3, the atoms are generated conditioned on the cluster memberships of their predicate and object symbols. For each atom $X = o_r(o_1, \ldots, o_p)$, let cluster combination $M_X^{(i)} = (C_r^{(i)}, C_1^{(i)}, \ldots, C_p^{(i)})$ be the vector of cluster assignments of the symbols in X, in level i of the hierarchy. X is generated from a log-linear model with one feature for each cluster combination that explains it:

$$P(X|\mathbf{C}) = \frac{1}{1 + \exp\left(-\sum_{i=1}^{h} w_{M_X^{(i)}}\right)}$$

w_M is the weight of the feature corresponding to cluster combination M. The model can be straightforwardly extended to include cross-level features, but we assume in this work that all clusters in a cluster combination come from the same level in the hierarchy.

The full joint probability distribution is:

$$\begin{aligned} P(\mathbf{C}, \mathbf{X}) = & \left(\prod_{i=1}^{h} \prod_{c_k \in \mathcal{D}^{(i)}} P(c_k)^{|c_k|} \right) \\ & \prod_{X \in \mathbf{X}^+} \frac{1}{1 + \exp\left(-\sum_{i=1}^{h} w_{M_X^{(i)}}\right)} \\ & \prod_{X \in \mathbf{X}^-} \frac{1}{1 + \exp\left(\sum_{i=1}^{h} w_{M_X^{(i)}}\right)} \end{aligned} \tag{6.4}$$

There are several differences between our hierarchical model (HRC) and Roy et al.'s [122] AHM. Their system does not incorporate any form of predicate clustering, which is important in domains with a large number of predicates. AHM learns the single tree-consistent partition of objects that best explains each predicate, while our model makes the prediction jointly over the entire hierarchy. A form of shrinkage can be achieved within the AHM framework, by marginalizing over tree-consistent partitions. However, HRC achieves shrinkage even when we just use the MAP hypothesis, which is advantageous both for computational reasons and for human interpretability. Another difference is that HRC allows a node to have an arbitrary number of children, allowing us to represent more compact, interpretable trees (sometimes referred to as *rose trees* [13]).

6.2.5 *Multiple Hierarchical Relational Clustering Model*

There are two alternative approaches to unifying hierarchical clustering with multiple clustering: the model could consist of multiple hierarchical clusterings, or a hierarchy of multiple-clustering models. The model we describe in this section takes the former approach.

A *hierarchical clustering* $\mathbf{C}^{(\cdot,j)} = (\mathbf{C}^{(1,j)}, \ldots \mathbf{C}^{(h_j,j)})$ is a matrix of cluster assignment variables (as in section 6.2.4). $\mathbf{C}^{(i,j)} = (C_1^{(i,j)}, \ldots, C_n^{(i,j)})$ represents the cluster assignments of the n items in level i of the jth hierarchical clustering. h_j is the number of levels in the jth clustering. The complete set of cluster assignments in the model is defined by the 3-dimensional tensor $\mathbf{C} = \{\mathbf{C}^{(\cdot,1)}, \ldots, \mathbf{C}^{(\cdot,m)}\}$, which represents m hierarchical clusterings.

To allow different clusterings to model different subsets of the objects and predicates, the model contains a matrix of indicator variables O; when $O_i^{(\cdot,j)} = 1$, we say that clustering j *contains* item o_i. For notational convenience, we also define $N_X^{(\cdot,j)}$; this indicator variable has a value of 1 iff $O_i^{(\cdot,j)} = 1$ for all of X's arguments o_i. When this is the case, we say that clustering j *contains* atom X.

The cluster memberships in hierarchical clusterings are generated from multinomials, as in sections 6.2.2 and 6.2.4. As in the previous section, the atoms are generated conditioned on the cluster assignments using a log-linear model with one feature per cluster combination. The only difference is that we now need to include features from all of the hierarchical clusterings that contain each atom:

$$
\begin{aligned}
P(\mathbf{C}, \mathbf{X}) = & \left(\prod_{j=1}^{m} \prod_{i=1}^{h_j} \prod_{c_k \in \mathcal{D}^{(i,j)}} P(c_k)^{|c_k|} \right) \\
& \prod_{X \in \mathbf{X}^+} \frac{1}{1 + \exp\left(- \sum_{j=1}^{m} N_X^{(\cdot,j)} \sum_{i=1}^{h_j} w_{M_X^{(i,j)}} \right)} \\
& \prod_{X \in \mathbf{X}^-} \frac{1}{1 + \exp\left(\sum_{j=1}^{m} N_X^{(\cdot,j)} \sum_{i=1}^{h_j} w_{M_X^{(i,j)}} \right)}
\end{aligned}
\tag{6.5}
$$

Here, $\mathcal{D}^{(i,j)}$ is the set of cluster symbols in level i of clustering j. Cluster combination $M_X^{(i,j)}$ is the vector of cluster assignments of the arguments of atom X in level i of clustering j.

6.3 Learning

Given this model and an evidence database, the learning problem is to find the MAP (*maximum a posteriori*) cluster assignment and the optimal parameter values. The resulting model can be used directly to predict missing data, or it may be used as input for subsequent processing, for instance using coarse-to-fine algorithms [73].

The MAP inference problem has two subtasks, each of which introduces some new challenges in our setting: **weight learning** (learning optimal values of the parameters, given a candidate structure) and **structure search** (finding the highest-scoring cluster assignment tensor, $\mathbf{C}$). We describe our approach to each of these problems below.

6.3.1 Weight Learning

Let us first assume that cluster assignments are known. The learning problem then reduces to finding the optimal parameters for the model, i.e., the cluster weights ($P(c)$) and the weights of the predictive features ($w_{M_X}^{(i,j)}$). The $P(c)$'s are simply multinomial parameters, and can be trivially computed in closed form.

Computing the atom prediction weights is more challenging. In other relational clustering models such as IRM, MRC and AHM, each atom is predicted by a single cluster combination, making the weight-learning problem trivial. For instance, in MRC (a special case of our model), the weight of a cluster combination is simply the log-odds of a random atom being true. In our setting, however, a single atom may be influenced by features at multiple levels in the hierarchy.

A further complication is that we regularize the predictive weights to prevent overfitting; we use both ℓ_0 and ℓ_1 penalties. The ℓ_0 term is simply a penalty in log-space for each non-zero predictive feature we introduce; this has the effect of discouraging unnecessarily fine-grained predictive features. The ℓ_1 term penalizes the sum of the absolute values of the weights. Weight learning for log-linear models with ℓ_1 regularization typically requires the use of iterative optimization algorithms (e.g. [6]). Since the weights need to be relearned for each structure evaluated during the search, these algorithms may be prohibitively expensive.

Instead of running a full ℓ_1 optimizer at each step of the search, we learn the weights approximately, in a coarse-to-fine manner that takes advantage of the hierarchical structure. This can be done if each atom is explained by exactly one hierarchical clustering (as is the case for the search algorithm discussed in the next section). Note that each cluster combination M on level i is a subset of some cluster combination $M' = \rho(M)$ on level $i+1$; we refer to the latter as the *parent* cluster combination. Notice that if $M = (c_1, \ldots, c_p)$, then $\rho(M) = (\rho(c_1), \ldots, \rho(c_p))$. We compute the weights one level at a time, starting at the coarsest level of the hierarchy. The model learned at each level is a refinement of the level above, adding new, more specific features if the improvement to likelihood outweighs the cost incurred by the ℓ_0 and ℓ_1 penalties.

Consider a cluster combination M, containing true atoms $\mathbf{X}_M^+$ and false atoms $\mathbf{X}_M^-$. When we compute w_M, we have already computed the weights of M's ancestors in the hierarchy (i.e., the cluster combinations of which M is a subset). Let s_M be the sum of the weights of M's ancestors. At this point, all atoms in M occur in the same set of non-zero features (i.e., M and its ancestors); this may change as we learn the weights of finer features in the lower levels of the tree. At this stage of the computation, the likelihood of the atoms in M can be written as follows:

$$L_M = \left(\frac{1}{1+e^{-(s_M+w_M)}}\right)^{|\mathbf{X}_M^+|} \left(\frac{1}{1+e^{(s_M+w_M)}}\right)^{|\mathbf{X}_M^-|}$$

s_M is fixed, having been previously computed. The goal is to learn the optimal value of w_M:

$$w_M^* = \underset{w_M}{\textbf{arg max}}\ \log L_M - \alpha_0 \mathbb{1}\{w_M \neq 0\} - \alpha_1 |w_M|$$

α_0 and α_1 are respectively the ℓ_0 and ℓ_1 penalty parameters. We also add smoothing parameters β^+ and β^- to the true and false atom counts; we omit them from the above equations for simplicity.

The above computation can be done in closed form, despite the discontinuity introduced by the ℓ_1 term. The expression $(s_M + w_M)$ can be interpreted as a 'smoothed' value for M's feature, shrunk towards a coarser feature in the hierarchy. Note that without the regularization terms, the optimal value of w_M would simply be $(\log({}^t/_f) - s_M)$, effectively ignoring the coarser features and learning a flat model. The higher we set the regularization terms, the less M changes the predictions of its ancestors.

6.3.2 Structure Search

The search for the optimal cluster assignments proceeds in two stages. First, we search for the best flat multiple clustering. This initial clustering can be done with any multiple clustering algorithm; MRC is a natural choice, since its model is a special case of ours. Like MRC, we require for tractability that each atom occur in exactly one clustering (though objects may still occur in multiple clusterings). This partitions the database into several smaller databases, each of which poses a separate hierarchical clustering problem. Algorithm 5 provides an overview of this procedure. MULTICLUSTER$(\mathbf{X}, ml)$ may be any search algorithm that returns a set of non-overlapping flat clusterings; we refer the reader to Kok and Domingos [77] for an example of how this can be done. The novel part of the algorithm is HIERARCHICALCLUSTER$(\mathbf{X}, ml)$, which replaces each of the flat clusterings with a hierarchical clustering. We devote the rest of this section to the discussion of this subroutine.

We construct the hierarchy layer by layer, starting with each object and predicate in a singleton cluster, and building progressively coarser clusters one level at a time. This allows us to treat the hierarchical clustering problem as a series of flat clustering problems, where the individual items being clustered at each layer are the clusters from the level below.

Algorithm 6 describes this procedure in more detail. FLATCLUSTER$(\mathbf{C}^{(i)})$ is a subroutine that takes a set of clusters as input and partitions them, returning a coarser set of clusters. We can use any clustering procedure that exactly or approximately optimizes the posterior probability. In our experiments, we use a simple greedy search that individually moves each item to the best cluster, iterating until convergence. Items can also be moved to newly created singleton clusters. (The ℓ_0 penalty described in the previous section discourages the creation of new clusters.) It is straightforward to incorporate other search operations, such as greedy merges and splits, random moves, etc. We also restrict the algorithm to only merge predicate symbols with other predicates of the same arity.

The coarse-to-fine weight learning scheme in section 6.3.1 runs in time linear in the number of features. However, since we have one feature per cluster combination, this can still be too

Algorithm 5 MHRC-LEARN($\mathbf{X}$, ml)

input: $\mathbf{X}$, a relational database

ml, max number of levels in the hierarchy

output: $\mathbf{C}$, a multiple hierarchical clustering

$\mathbf{C} \leftarrow \emptyset$

$\mathbf{D} \leftarrow$MULTICLUSTER($\mathbf{X}$)

for all flat clusterings $\mathbf{D}^{(\cdot,j)} \in \mathbf{D}$ **do**

$\mathbf{X}_{D_j} \leftarrow$ atoms contained by $\mathbf{D}^{(\cdot,j)}$

$\mathbf{C}^{(\cdot,j)} \leftarrow$HIERARCHICALCLUSTER($\mathbf{D}^{(1,j)}, ml$)

end for

return $\mathbf{C}$

expensive to run at each step of the search. The weight learning can be further sped up by only updating the weights significantly affected by the change in structure. With the search procedure described in Algorithm 6, the only features affected by a change are the descendents of the altered top-level cluster combinations. Furthermore, we only update the *immediate* descendents of the altered features. This is an approximation; in principle, the weights of the entire subtree could be affected. In practice, however, the weights of non-immediate descendents are not significantly affected by changes in the top-level structure.

For example, consider a top-level cluster combination M_0, its child M_1, and M_1's child M_2. If one of M_0's clusters is altered, then M_0's true and false atom counts may change, necessitating an update for M_0's weight, w_0. This changes s_1, the sum of the weights of M_1's ancestors ($s_1 = w_0$). We update w_1 accordingly.

Now, note that $s_2 = w_0 + w_1 = s_1 + w_1$. Recall that $s_1 + w_1$ is a smoothed version of M_1's parameter. M_1's counts are not affected by the changes to M_0; the feature explains the same set of atoms before and after the change. Therefore, unless the regularization parameters are extremely large, changes to M_0 will not cause big changes to $s_2 = w_1 + s_1$. Consequently, w_2 is relatively unaffected.

Algorithm 6 HIERARCHICALCLUSTER($\mathbf{X}$, ml)

input: $\mathbf{X}$, a relational database

ml, max number of levels in the hierarchy

output: $\mathbf{C}$, a hierarchical clustering

for all object and predicate symbols o_k in $\mathbf{X}$ **do**

$C_k \leftarrow \{o_k\}$

end for

$curLevel \leftarrow (C_1, \ldots, C_n)$

for $i \leftarrow 1$ to ml **do**

$topLevel \leftarrow$ FLATCLUSTER($curLevel$)

if $topLevel$ is identical to $curLevel$ **then**

break

end if

$\mathbf{C}^{(i)} \leftarrow topLevel$

$curLevel \leftarrow topLevel$

end for

return $(\mathbf{C}^{(1)}, \ldots, \mathbf{C}^{(h)})$

6.4 Experiments

Note that HIERARCHICALCLUSTERING($\mathbf{X}, ml$) can be run as a standalone algorithm on the full database, optimizing the model in section 6.2.4. For our initial experiments, we implemented this version of the algorithm, learning a single hierarchy. We compared two versions of the algorithm: HRC1, which constructs a single level clustering ($ml = 1$); HRC10 learns a hierarchy with up to 10 levels ($ml = 10$). We evaluated these algorithms on the Animals (ANML), Nations (NATS), Kinship (KINS) and UMLS datasets from Kemp et al. [70]. Table 6.1 describes the size and composition of these domains. We performed 10-fold cross validation, and evaluated the algorithms in a transductive setting: the test atoms were set to 'unknown', and their values were predicted from

	Objects	Features	Relations	Total atoms	True atoms	HRC1		HRC10		IRM		MRC	
						AUC	CLL	AUC	CLL	AUC	CLL	AUC	CLL
ANML	50	85	0	4250	1562	0.81	-0.46	0.80	-0.49	0.79	-0.43	0.80	-0.43
NATS	14	111	56	12,530	2565	0.68	-0.41	0.69	-0.43	0.75	-0.31	0.75	-0.31
KINS	104	0	26	281,216	10,686	0.74	-0.06	0.75	-0.06	0.68	-0.06	0.85	-0.05
UMLS	135	0	46	838,350	6529	0.70	-0.014	0.70	-0.014	0.80	-0.011	0.97	-0.004

Table 6.1: Experiments.

the known atoms.

We used two evaluation criteria: **AUC** (area under the precision-recall curve of the query atoms) and **CLL** (average conditional log-likelihood of query atoms).

We compared our algorithm to IRM and MRC. IRM and MRC's results are taken directly from Kok and Domingos [77]. For HRC, we use $\alpha_0 = 0.01$. $\alpha_1 = 0.1$ on ANIMALS, and 0.01 on the other domains. We set $\beta^+ + \beta^- = 0.1$, with the ratio between them reflecting the ratio of true to false evidence atoms for the corresponding predicate (this smoothing scheme is similar to the one used by Kok and Domingos [78]).

Despite the simpler search strategy we use, even the flat version of HRC is generally competitive with IRM's more sophisticated search. Learning multiple levels does not prove to be useful on the propositional ANML domain, but slightly improves AUC on the three relational domains (at the cost of a slightly lower CLL on NATS). Although the improvements are small, the difference in AUC and CLL is statistically significant on KINS and UMLS, the two largest domains (as measured by a sign test with $p = 0.05$). We suspect that the hierarchy will provide greater benefit on larger, more complex domains with more missing data. MRC proves to be the most successful system overall, suggesting that the full multiple clustering version of the algorithm may perform better than using a single hierarchy.

6.5 Conclusions & Future Work

In this chapter, we described a unified model for relational clustering, incorporating both hierarchies and multiple clustering, and we proposed an efficient learning algorithm for the model. The empirical usefulness of multiple clusterings has been previously demonstrated by Kok and Domingos [77]; in this work, we performed a preliminary evaluation of hierarchical relational clustering. Our initial experiments show that our approach is competitive with other relational clustering algorithms; however, more experiments are needed on larger, more complex domains.

The most immediate direction for future work is to evaluate the full version of the algorithm, learning multiple hierarchical clusterings. A further direction is to apply hierarchical relational clustering to learn the structure of Tractable Markov Logic Networks (TML) [42].

One key limitation of both MHRC and TML is their inability to generalize across different mega-examples, involving different sets of objects. The following chapter describes a new representation that retains much of the richness of MHRC, while allowing generalization over mega-examples.

Chapter 7

LEARNING RELATIONAL SUM-PRODUCT NETWORKS

7.1 Introduction

Graphical probabilistic models compactly represent a joint probability distribution among a set of variables. Unfortunately, inference in graphical models is intractable. In practice, using graphical models for most real-world applications requires either using approximate algorithms, or restricting oneself to a subset of graphical models on which inference is tractable. A common restriction that ensures tractability is to use models with low treewidth [9, 20]. However, as seen in section 2.2.2, not all tractable models have low treewidth. Sum-Product Networks [113] and similar representations compactly capture some high-treewidth distributions, while still guaranteeing efficient exact inference.

Besides intractability, one other key shortcoming of most widely used graphical models is their reliance on the i.i.d. assumption. In many real-world applications, instances are not truly independent, and can be better modeled if their interactions are taken into account. This is one of the key insights of the Statistical Relational Learning (SRL) community [52]. SRL techniques have been applied to a wide variety of tasks, including collective classification, link prediction, natural language processing, etc. However, most SRL methods build on graphical models (e.g. Markov logic [118]), and suffer from the same computational difficulties; these are compounded by the additional problem of modeling interactions between instances.

In this chapter, our goal is to combine these two lines of research: tractable probabilistic models and relational learning. One line of related work uses Naïve Bayes models in structured domains ([47]; [83]; [28]). Although tractable, Naïve Bayes models are quite limited in expressiveness. PRISM is a probabilistic logic that supports efficient inference, but only under a very restrictive set of assumptions [126]. PSL [17] supports efficient inference, but uses fuzzy logic-based seman-

tics instead of standard probabilistic semantics. TML [42] is a subset of Markov logic on which efficient inference can be guaranteed. TML is surprisingly expressive, subsuming most previous tractable models. However, a TML knowledge base determines the set of possible objects in the domain, and the relational structure among them. This limits the applicability of TML to learning; a TML knowledge base cannot be learned on one set of objects and applied to another mega-example with different size or structure. The newer TPKB language [143] suffers from the same limitation, as does the MHRC model described in chapter 6.

To address this, we propose *Relational Sum-Product Networks* (RSPNs), a new tractable relational probabilistic architecture. RSPNs generalize SPNs by modeling a set of instances jointly, allowing them to influence each other's probability distributions, as well as modeling the probabilities of relations between objects. An RSPN can be trained on a set of mega-examples, and applied to a previously unseen mega-example with different structure (given a part decomposition as input). We also introduce *LearnRSPN*, the first algorithm for learning tractable statistical relational models. Intractable inference has historically been a major obstacle to the wider adoption of statistical relational methods; the development of learning and inference algorithms for tractable relational models could go a long way towards making SRL more widely applicable.

7.2 Relational Sum-Product Networks

7.2.1 Exchangeable Distribution Templates

Before we define RSPNs, we first define the notion of an *Exchangeable Distribution Template* (EDT).

Definition 3. Consider a finite set of variables $\{X_1, \ldots, X_n\}$ with joint probability distribution P. Let $S(n)$ be the set of all permutations on $\{1, \ldots, n\}$. $\{X_1, \ldots, X_n\}$ is a *finite exchangeable set* with respect to P if and only if $P(X_1, \ldots, X_n) = P(X_{\pi(1)}, \ldots, X_{\pi(n)})$ for all $\pi \in S(n)$. [38].

Note that finite exchangeability does not require independence: a set of variables can be exchangeable despite having strong dependencies. (For example, consider binary variables $X_1, \ldots, X_n$, with a uniform distribution over value assignments with an even number of non-zero variables.)

Definition 4. An *Exchangeable Distribution Template* (EDT) is a function that takes a set of variables $\{X_1, \ldots, X_n\}$ as input (n is unknown a priori), and returns a joint probability distribution P with respect to which $\{X_1, \ldots, X_n\}$ is exchangeable. We refer to the probability distribution P returned by the EDT for a given set of variables as an *instantiation* of that EDT.

Example 1. The simplest family of EDTs simply returns a product of identical univariate distributions over each of $X_1, \ldots, X_n$. For example, if the variables are binary, then an EDT might model them as a product of Bernoulli distributions with some probability p.

Example 2. Consider an EDT over a set of binary variables $X_1, \ldots, X_n$ (with n unknown a priori), returning the following distribution: $P(X_1, \ldots, X_n) \propto \frac{\lambda^k}{k!} e^{-\lambda}$, where $k = \sum_{X_i} \mathbf{1}[X_i]$. λ is a parameter of the EDT. This is an EDT with a Poisson distribution over the number of variables in the set with value 1. (The probabilities must be renormalized, since the set of variables is finite.) Note that this EDT does not assume independence among variables.

Intuitively, EDTs can be thought of as probability distributions that depend only on aggregate statistics, and not on the values of individual variables in the set.

Relational Sum-Product Networks (RSPNs) jointly model the attributes and relations among a set of objects. RSPNs inherit TML's notion of parts and classes. As in TML, each part of a class also belongs to some class. Unlike TML, an RSPN class's parts may be *unique* or *exchangeable*. An object's unique parts are those that play a special role, e.g. the commander of a platoon, the queen of a bee colony, or the hub of a social network. The exchangeable parts are those that behave interchangeably: soldiers in a platoon, worker bees in a colony, spokes in a network, and so on.

Definition 5. A *definition* for class C in an RSPN consists of:

- A set of *attributes*: unary predicates A applicable to individuals of C.
- A vector $U_C = (P_1, \ldots, P_n)$ specifying the classes of *unique parts*.
- A vector $E_C = (P_1, \ldots, P_n)$ of classes of *exchangeable parts*

- A set of *relations* between parts: predicates of the form $R_1(P_1, P_2)$ or $R_2(C, P_1)$, where P_1 and P_2 are either unique or exchangeable part classes of C. Predicates may be of any arity.

- A *class SPN* whose leaves fall into three categories:

  - $L_A^{(C)}$ is a univariate distribution over attribute A of C;
  - $L_R^{(C)}$ is an EDT over binary (or higher-order) predicate R involving C and/or its part types (e.g. formulas of the form $R_1(P_1, P_2)$ or $R_2(C, P_1)$, where P_1 and P_2 are part classes of C);
  - $L_P^{(C)}$ is a sub-SPN for part class P (i.e. a valid class SPN for class P).

  All attributes, relations and parts of class C must be included in the class SPN. The class SPNs can have arbitrary internal structure.

Each P_k in U_C and E_C is an RSPN class. In principle, a class may occur multiple times in each part vector. In this case, each part may be uniquely identified by its index in the corresponding vector. However, to simplify this discussion, we assume that each class occurs at most once in (U_C, E_C).

Example 3. The following is a partial class specification for a simple political domain. A 'Region' consists of an arbitrary number of nations, and relationships between nations are modeled at this level. A 'Nation' has a unique government and an arbitrary number of people. National properties such as 'High GDP' are modeled here. The 'Supports' relation can capture a distribution over the number of people in the nation who support the government.

```
class Region:
  exchangeable part Nation
  relation Adjacent(Nation, Nation)
  relation Conflict(Nation, Nation)

class Nation:
```

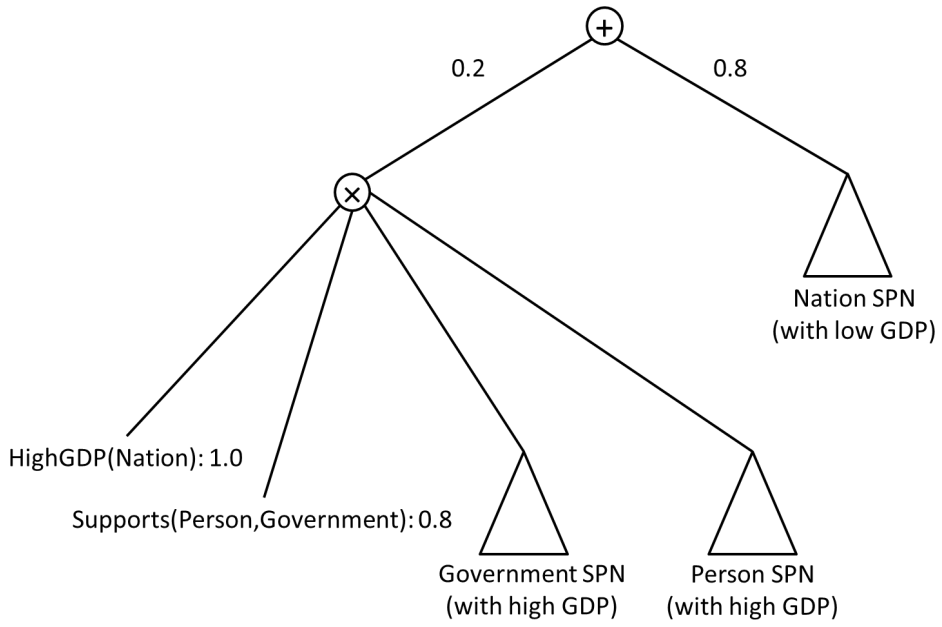


Figure 7.1: Partial SPN for the 'Nation' class (example 3). The sum node at the root represents a mixture model over two possible SPNs for 'Nation': one with high GDP (left), and the other with low GDP (right; omitted). The 'Supports' predicate is modeled by an EDT of the form described in example 1.

```
unique part Government
exchangeable part Person
attribute HighGDP
relation Supports(Person, Government)
```

See fig. 7.1 for an example class SPN.

7.2.2 Grounding an RSPN

Like MLNs, RSPNs are templates for propositional models. To generate a ground SPN from an RSPN, we take as input a *part decomposition*:

Definition 6. For RSPN class C, a *C-rooted part decomposition* consists of:

- An object O of class C (‘*root*’);
- Exactly one P-rooted part decomposition for each unique part class P in U_C (‘*unique child*’);
- A (possibly empty) set of P-rooted decompositions for each exchangeable part P in E_C (‘*exchangeable children*’).

The decomposition must be acyclic, i.e. an object may not be its own child or descendant. (This ensures that the ground SPN is acyclic even when the RSPN class structure contains cycles.)

To *ground* a class SPN is to instantiate the template for a specific set of objects. Given a class C and a part decomposition D rooted at object O, grounding C's SPN yields a propositional SPN whose leaf distributions are over attributes and relations involving the objects in D. This is done recursively as follows:

- For leaves of the form $L_A^{(C)}$: replace the univariate distribution over predicate A in the class SPN with a univariate distribution over $A(O)$ in the ground SPN.
- For leaves of the form $L_R^{(C)}$: replace the EDT over $R(X, Y)$ in the class SPN with an instantiation of that EDT over the groundings of R.

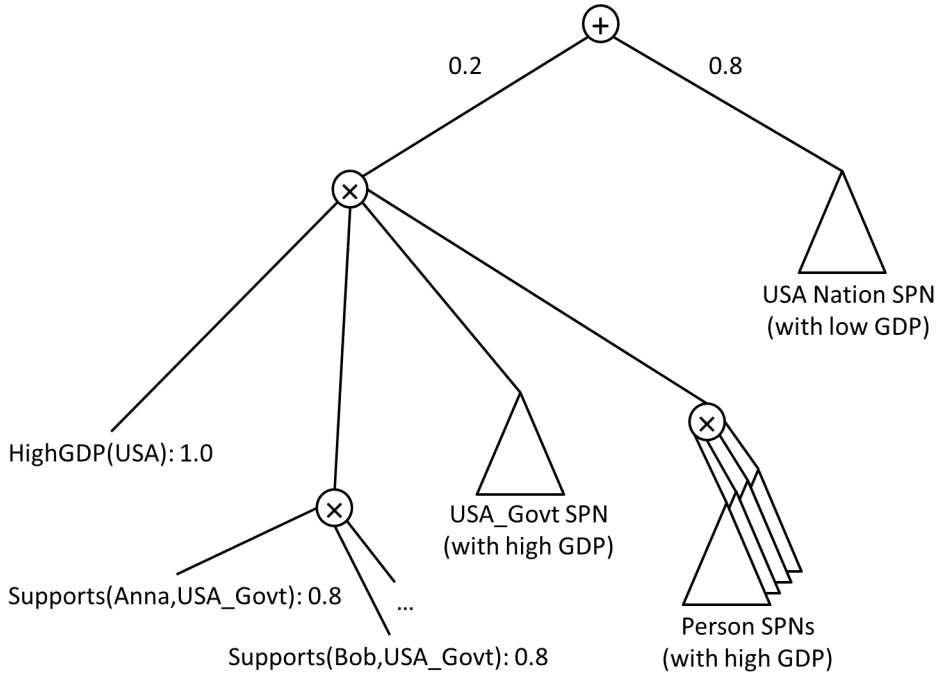


Figure 7.2: Example grounding of the ‘Nation’ class, with SPN from fig. 7.1.

- For leaves of the form $L_P^{(C)}$: recursively ground P's class SPN over each type-P child of object O. Replace the leaf with a product over the resulting ground SPNs.

Note that each attribute/relation/part may correspond to more than one leaf in the SPN (in different children of a sum node), potentially modeled by a different univariate distribution/EDT/class SPN. See fig. 7.2 for an example ground SPN.

7.2.3 Representational Power

The main limitation of the RSPN representation is that individual relational atoms are not modeled directly, but through aggregations. In this respect, it is similar to Probabilistic Relational Models (PRMs; [49])—though RSPNs guarantee tractable inference, unlike PRMs. Aggregations are extremely useful for capturing relational dependencies [98]. For example, a person's smoking habits may depend on the number of friends she has who smoke, and not the smoking habits of each individual friend. Nevertheless, aggregations are not well-suited to capturing relational patterns that depend on specific paths of influence, such as Ising models.

It is important to note than RSPNs (and SPNs) are not simply tree-structured graphical models. The graph of an RSPN is not a conditional dependency graph, but a graphical representation of the computation of the partition function. A ground RSPN can be converted into an equivalent graphical model, but the resulting model may be high-treewidth, and computationally intractable as such.

In effect, RSPNs are a way to compactly represent context-specific independences in relational domains: different children of a sum node may have different variable decompositions in their product nodes. These context-specific independences are what give RSPNs more expressiveness than low-treewidth tractable models.

7.3 Learning RSPNs

The learning task for RSPNs is to determine the structure and parameters of all the class SPNs in the domain. In this work, we assume that the part relationships among classes are known, i.e. the

user determines what types of unique and exchangeable parts are allowed for each class. The input for the learning algorithm is a set of part decompositions, and an evidence database specifying the values of the attributes and relations among the objects in those decompositions.

Our learning algorithm is based on the top-down LearnSPN algorithm [51]. LearnSPN(T, V) is a propositional SPN learning algorithm, and takes as input a set of training instances T and variables V. The algorithm attempts to decompose V into independent subsets $V_1, \ldots, V_k$ (using pairwise statistical independence tests); if such a decomposition exists, LearnSPN recurses over each set (calling LearnSPN(T, V_1),. . .,LearnSPN(T, V_k)), and returns a product node over the recursively learned sub-SPNs. If V does not decompose, LearnSPN instead clusters the instances T, recursively learns a sub-SPN over each subset $T_1, \ldots, T_k$, and returns a sum-node over the sub-SPNs, weighted by the mixture proportions. Under certain assumptions, LearnSPN can be seen as a greedy search maximizing the likelihood of the learned SPN.

Given an RSPN class C, LearnSPN could be used directly to learn an SPN over C's attributes. However, C's exchangeable parts pose a problem for LearnSPN: the number of leaf variables in the ground SPN can differ from one training instance to another, and between training instances and test instances. To address this, we propose the LearnRSPN algorithm (Alg. 7), a relational extension of LearnSPN. (For the purpose of this discussion, parts are assumed to be exchangeable; attributes of unique parts can be handled the same way as attributes of the parent part, and represented as separate leaves in the class SPN.)

LearnRSPN is a top-down algorithm similar to LearnSPN; it attempts to find independent subsets from among the object's set of attributes, relations and parts; if multiple subsets exist, the algorithm learns a sub-SPN over each subset, and returns a product over the sub-SPNs. If independent subsets cannot be found, LearnRSPN instead clusters the instances, returning a sum node over the components, weighted by the mixture proportions.

LearnRSPN exploits the fact that predicates involving exchangeable parts are grounded into finite sets of exchangeable variables. Instead of treating each ground atom as a separate leaf in the SPN, LearnRSPN summarizes a set of exchangeable variables with an aggregate statistic (in our experiments, we used the fraction of true variables in the set, though other statistics can be used).

Algorithm 7 LearnRSPN(C, T, V)

input: C, a class
T, a set of instances of C
V, a set of attributes, relation aggregates, and parts
output: an RSPN for class C
if $|V| = 1$ **then**
  if $v \in V$ is an attribute **then**
    return univariate estimated from v's values in T
  else if v is a relation aggregate **then**
    return EDT estimated from v's values in T
  else
    $C_{child} \leftarrow$ class of v *//v is a part*
    $T_{child} \leftarrow$ parts of $t \in T$ of type C_{child}
    $V_{child} \leftarrow$ attributes, relations and parts of C_{child}
    return LearnRSPN$(C_{child}, T_{child}, V_{child})$
  end if
else
  partition V into approximately independent subsets V_j
  if success **then**
    return $\prod_j$ LearnRSPN(C, T, V_j)
  else
    partition T into sets of similar subsets T_i
    return $\sum_i \frac{|T_i|}{|T|}$.LearnRSPN(C, T_i, V)
  end if
end if

This summary statistic is treated as a single variable in the decomposition stage of RSPN. Thus, attributes that are highly predictive of the statistics of the groundings of a relation will be grouped with that relation. Parts are similarly summarized by the statistics of their attribute and relation predicates.

The base case of LearnRSPN (when $|V| = 1$) varies depending on what v is. When v is an attribute, the RSPN to be returned is simply a univariate distribution over the attribute, as in the propositional version of LearnSPN. When v is a part of class C_{child}, LearnRSPN returns an SPN for class C_{child}. Crucially, different SPNs are learned for C_{child} in different children of a sum node in parent class C (since the recursive call is made with a different set of instances). The final base case is when v is an aggregate over an exchangeable relation. In this case, the RSPN to be returned is an EDT over the relation.

Like LearnSPN, LearnRSPN can be seen as an algorithm schema rather than a single algorithm; the user is free to choose a clustering algorithm for instances, a dependency test for variable splitting, an aggregate statistic, and a family of EDTs for exchangeable relations. Note that different families of EDTs may require different aggregate statistics for parameter estimation. The fraction of true groundings is sufficient for the two EDTs described in examples 1 and 2.

7.4 Experiments

7.4.1 Methodology

We compared a Python implementation of LearnRSPN to two MLN structure learning algorithms:

- MSL [76], as implemented in the widely-used ALCHEMY system [80];
- LSM [79], a state-of-the-art MLN learning method.

We also evaluated a simple baseline (BL) that simply predicts each atom according to the global marginal probability of its predicate.

To cluster instances in LearnRSPN, we used the EM implementation in SCIKIT-LEARN [107], with two clusters. To test independence, we fit a Gaussian distribution (for aggregate variables) or

Table 7.1: UW-CSE results. $|Q|$ is the number of query atoms.

Area	$\|Q\|$	Inference time (s)			AUC-PR				CMLL				LL/$\|Q\|$
		RSPN	MSL	LSM	RSPN	MSL	LSM	BL	RSPN	MSL	LSM	BL	RSPN
AI	1,414	0.10	7.18	5.32	0.71	0.36	0.29	0.36	-0.10	-0.16	-0.17	-0.16	-0.09
Graphics	171	0.03	0.82	0.50	0.80	0.59	0.38	0.35	-0.13	-0.28	-0.31	-0.26	-0.13
PL	17	0.01	0.05	0.05	0.81	0.84	0.69	0.42	-0.46	-0.81	-0.84	-0.80	-0.45
Systems	1,120	0.01	8.81	4.33	0.75	0.76	0.25	0.28	-0.07	-0.08	-0.14	-0.14	-0.07
Theory	308	0.04	1.01	0.94	0.79	0.51	0.37	0.32	-0.11	-0.29	-0.21	-0.20	-0.10
Average	606	**0.05**	3.57	2.22	0.77	0.61	0.39	0.34	**-0.17**	-0.32	-0.33	-0.31	-0.17

Bernoulli distribution (for binary attributes), and computed the pairwise mutual information (MI) between the variables. The test statistic used was $G = 2N \times MI$ (N being the number of samples), which in the discrete case is equivalent to the G-test used by Gens and Domingos [51]. We used a threshold of 0.5 for the p-value. For EDTs, we used the independent Bernoulli form, as described in example 1. All Bernoulli distributions were smoothed with a pseudocount of 0.1.

For MLN inference, we used the MC-SAT algorithm, the default choice in ALCHEMY 2.0, with the default parameters. For LSM, we used the example parameters in the implementation ($N_{walks} = 10,000$, $\pi = 0.1$; remaining parameters as specified by Kok and Domingos [79]).

We report results in terms of area under the precision-recall curve (AUC; [27]) and the average conditional marginal log-likelihood (CMLL) of test atoms. AUC is a prediction quality measure that is insensitive to the fraction of true negative atoms. CMLL directly measures the quality of the probability estimates. For LearnRSPN, we also report test set log-likelihood (LL) normalized by the number of queries, as an alternate measure of prediction quality. (ALCHEMY does not compute this quantity, since it is intractable for MLNs.) Unlike CMLL, LL captures the joint likelihood, rather than just the individual marginal likelihoods.

Table 7.2: Friends & Smokers link prediction results. N is the number of people in the network.

N	$\|Q\|$	Inference time (s)			AUC-PR				CMLL				LL/$\|Q\|$
		RSPN	MSL	LSM	RSPN	MSL	LSM	BL	RSPN	MSL	LSM	BL	RSPN
100	2,000	**0.10**	25.31	8.11	**0.22**	0.08	0.02	0.02	**-0.09**	-0.13	-0.13	-0.12	-0.09
200	8,000	**0.34**	202.14	51.29	**0.16**	0.03	0.01	0.01	**-0.05**	-0.54	-0.06	-0.07	-0.05
300	18,000	**0.75**	740.64	150.13	**0.13**	0.01	0.00	0.00	**-0.04**	-3.24	-0.04	-0.06	-0.03
400	32,000	**1.29**	1753.14	330.92	**0.13**	0.00	0.00	0.00	**-0.03**	-6.10	-0.04	-0.05	-0.02

Table 7.3: Fault localization results.

Program	Avg. LOC	Inference time (s)		Fraction skipped				CMLL				LL/$\|Q\|$
		RSPN	MSL	RSPN	MSL	TAR	BL	RSPN	MSL	TAR	BL	RSPN
oddTup	10.9	0.001	0.040	0.65	0.66	0.80	0.58	-0.67	-3.98	-1.47	-0.74	-0.67
deriv.	15.3	0.003	0.056	0.70	0.52	0.53	0.47	-0.37	-4.07	-0.62	-0.42	-0.37
isWord	15.8	0.003	0.057	0.59	0.72	0.63	0.42	-0.53	-3.12	-0.57	-0.40	-0.45
newtons	22.1	0.000	0.079	0.58	0.65	0.51	0.47	-0.39	-3.20	-0.80	-0.37	-0.39
Average	16.5	0.002	0.058	0.63	0.64	0.62	0.48	-0.60	-3.59	-0.86	-0.48	-0.47

```
class Area:
  exchangeable part Group

class Group:
  unique part Professor
  exchangeable part Student
  exchangeable part GroupPaper
  exchangeable part NonGroupPaper
  relation Author(Professor, GroupPaper)
  relation Author(Professor, NonGroupPaper)
  relation Author(Student, GroupPaper)
  relation Author(Student, NonGroupPaper)

class Professor:
  attribute Position_Faculty
  attribute Position_Adjunct
  attribute Position_Affiliate

class Student:
  attribute InPhase_PreQuals
  attribute InPhase_PostQuals
  attribute InPhase_PostGenerals
```

Figure 7.3: Part structure for UW-CSE domain.

7.4.2 UW-CSE

The UW-CSE database [118] has been used to evaluate a variety of statistical relational learning algorithms. The dataset describes the University of Washington Computer Science & Engineering department, and includes advising relationships, paper authorships, etc. The database is divided into five non-overlapping mega-examples, by research area.

To generate a part structure for this domain (fig. 7.3), we separated the people into one research group per faculty member, with students determined using the $AdvisedBy$ and $TempAdvisedBy$ predicates (breaking ties by number of coauthored papers). Publications are also divided among groups: each paper is assigned to the group of the professor who wrote it, voting by the number of student authors in the group in the event of a tie. The prediction tasks are to infer the roles of faculty (Professor, Associate Professor or Assistant Professor) and students (Pre-Quals, Post-Quals, Post-Generals), as well as paper authorships. The part structure is also made available to ALCHEMY in the form of predicates $Has(Area, Group)$, $Has(Group, Professor)$, $Has(Group, Student)$, $Has\_Group(Group, Paper)$, and $Has\_NonGroup(Group, Paper)$.

We performed leave-one-out testing by area, testing on each area in turn using the model trained from the remaining four. 80% of the groundings of the query predicates were provided as evidence, and the task was to predict the remaining atoms. Table 7.1 shows the results on all five areas, and the average. The RSPN approach is orders of magnitude faster than the other systems, and significantly more accurate. Training times in this domain are 9s (RSPN), 14,094s (MSL) and 1,620s (LSM).

7.4.3 Social Network Link Prediction

Link prediction is a challenging statistical relational learning problem [114, 136]. The task is to predict missing links in a partially observed graph, taking into account observed attributes of the nodes.

We generated artificial social networks in the Friends-and-Smokers domain [131], using a generalization of the Barabási-Albert preferential attachment model [10]. A network with N nodes is

generated as follows:

- For some fraction $p_{smokes} = 0.3$ of nodes x, set $Smokes(x)$ to 'true', and set the remainder to 'false'.

- For each node, sample another node and create an undirected edge. In the basic Barabási-Albert model, the probability of choosing a node is proportional to its degree. To encourage homophily, we multiply the unnormalized probability of an edge by a factor of $h_{smokes,smoker} = 100$ for smoker-smoker edges, and $h_{nonsmoker,nonsmoker} = 10$ for edges between non-smokers.

- Iterate, creating more edges for each node using the above distribution. We generate 2 or 3 edges (with equal probability) for each smoker node, and 1 or 2 edges for each non-smoker node.

This procedure results in graphs with small, dense communities of smokers, sparser communities of non-smokers, and relatively few links between smokers and non-smokers.

For training data, we generated five 100-person graphs. We tested on graphs of size ranging from 100 to 400 nodes (10,000 to 160,000 possible edges), to evaluate how well the structures learned by LearnRSPN generalize to mega-examples of different size.

At test time, all the $Smoker$ labels and 80% of the $Friendship$ edges are known; the prediction task is to infer the marginal probabilities of the remaining 20% of the graph edges.

An RSPN was trained with two classes: $Network$ and $Person$. $Network$ has both classes as exchangeable subparts. The part decompositions were generated using the Louvain method[1] [12], and the resulting community structure was also made available to ALCHEMY in the form of $Has(Network, Network)$ and $Has(Network, Person)$ predicates.

Table 7.2 shows the inference time and accuracy of the three systems. Results are averaged over five runs. Figures in bold are statistically significant improvements over all other systems

[1] http://perso.crans.org/aynaud/communities/

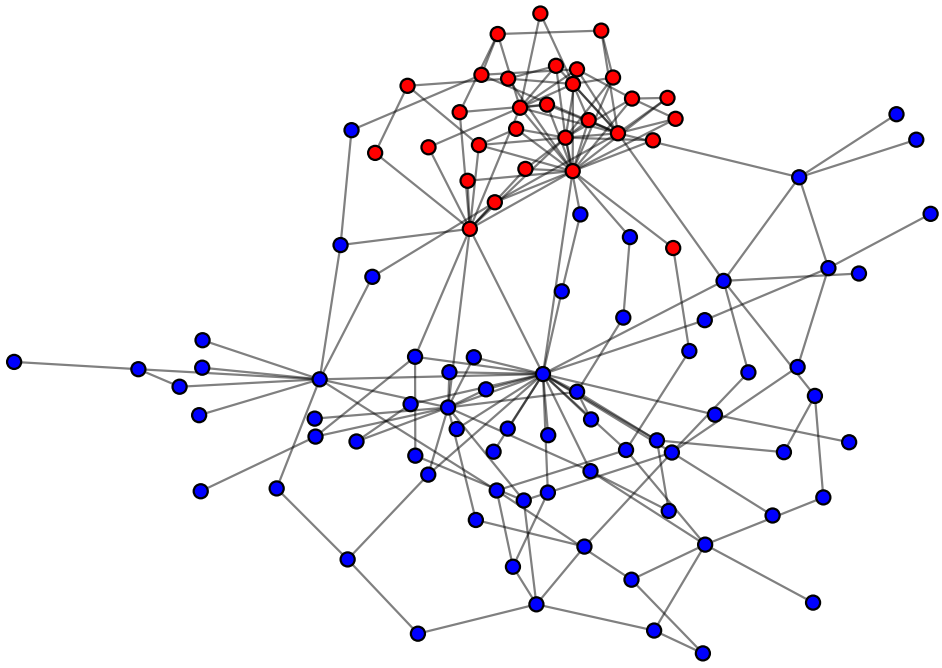


Figure 7.4: Sample 100-node social network. Red nodes are smokers.

(using the sign test, with a p-value of 0.05). Average training times for the three systems are 168s (RSPN), 31,083s (MSL) and 105,018s (LSM).

This is an extremely challenging link prediction task, due to the sparsity of the domain, and the relatively weak dependence between a node's attributes and links. MSL's greedy structure learning fails to find formulas that either exploit the provided community structure or capture the relationship between a person's friendships and smoking habits. Additionally, the weights learned by MSL on 100-node networks become increasingly inappropriate as the network size changes. On larger graphs, MLNs become prone to predicting that everybody is friends, a known pathology in social network models. Nevertheless, MSL outperforms the baseline in the AUC metric, which corrects for sparsity. LSM avoids the above pathology, learning a model similar to the baseline. As in the UW-CSE domain, RSPNs greatly outperform the other systems in both speed and accuracy.

7.4.4 Automated Debugging

We applied RSPNs to a fault localization problem. The task is to predict the location of the bug in a faulty program. (See also chapter 8 for an application of a domain-specific version of our algorithm to a larger, more realistic fault localization problem.)

The test corpus consists of four short Python programming assignments from MIT edX introductory programming course (6.00x) [128]: `oddTuples`, `derivatives`, `isWordGuessed` and `newtons_method`. We developed a suite of ten unit tests for each assignment, and identified ten buggy but syntactically valid responses to each question. (We filtered out submissions with multiple bugs, non-terminating loops, etc.) We manually annotated the location of the bug in each of the 40 programs in the corpus.

The corpus included several other programs, which were unusable for the following reasons:

- `ate1` and `biggest` had a single example each.

- `polynomials` and `getAvailableWords` predominantly contained syntactic errors; we did not find examples that met the constraints above.

```
class Program:
  exchangeable part IfStmt
  exchangeable part LoopStmt
  exchangeable part AtomicStmt

class IfStmt:
  unique part Program
  attribute Buggy

class LoopStmt:
  unique part Program
  attribute Buggy

class AtomicStmt:
  attribute Buggy
```

Figure 7.5: Part structure for debugging domain.

- `hangman` and `simple_hangman` were interactive programs, incompatible with our automated testing environment.

- The predominant cause of failure in `getGuessedWord` was that the output string was formatted differently from our reference implementation.

The program parse tree provides the part structure. The parse tree is mapped to the part structure in fig. 7.5; this part structure is also provided to ALCHEMY as evidence. The aggregation used by LearnRSPN is simply a discrete variable indicating which class of statement is most common in the subprogram (`IfStmt`, `LoopStmt`, or `AtomicStmt`). This information is also provided as

evidence to ALCHEMY. The systems were trained on three programs and tested on the fourth; reported results are averaged over the 10 buggy versions of each program.

In addition to MSL, we compared RSPNs to TARANTULA (TAR; [67]), a well-established fault localization approach. A common evaluation metric for fault localization systems is the fraction of program lines ranked lower than the buggy line. Higher scores indicate better localization. Ties in the line ranking are broken randomly. TARANTULA scores are computed in closed form from the coverage matrix. We also report the CMLL; for TARANTULA, this was computed by treating the suspiciousness score (which falls between 0 and 1) as a probability. As seen in table 7.3, RSPNs are competitive with both MSL and TARANTULA in ranking quality, and outperform them in CMLL. Average training times are 0.08s (RSPN) and 320s (MSL).

7.5 Conclusions & Future Work

SRL algorithms have been successfully applied to several problems, but the difficulty, cost and unreliability of approximate inference has limited their wider adoption. In practice, applying SRL methods to a new domain requires substantial engineering effort in choosing and configuring the approximate learning and inference algorithms. The expressiveness of languages like Markov logic is both a boon and a curse: although these languages can compactly represent sophisticated probabilistic models, they also make it easy for practitioners to unintentionally design models too complex even for state-of-the-art inference algorithms.

In the propositional setting, several approaches have been recently proposed for learning high-treewidth tractable models [89, 55, 113]. To our knowledge, LearnRSPN is the first algorithm for learning high-treewidth tractable relational models. Empirically, LearnRSPN outperforms conventional statistical relational methods in accuracy, inference time and training time.

A limitation of RSPNs is that they require a known, fixed part decomposition for all training and test mega-examples. Applying RSPNs to a new domain does require the user to specify the part decomposition; this is analogous to specifying the relational structure in PRMs. Many domains have a natural part structure that can be exploited (like the UW-CSE and debugging domains); in other cases, part structure can be created using existing graph-cut or community detection algo-

rithms (as in our link prediction experiments). An important direction for future work is to develop an efficient, principled method of finding part structure in a database.

Another direction for future work is applying tractable relational models to other problems. This chapter presented RSPNs and the related algorithms in a domain-independent manner; to apply RSPNs to a new problem, a domain expert need only define the RSPN class structure, and provide the part decompositions. However, the RSPN definition presented above can be tailored to better represent the features and dependencies needed to solve specific problems. The following chapter describes a variant of RSPNs tailored to the problem of software fault localization. These customizations allow us to outperform standard fault localization methods on a larger, more realistic dataset than the one used in our preliminary fault localization experiments, described in section 7.4.4.

Chapter 8

TRACTABLE PROBABILISTIC MODELS FOR AUTOMATED DEBUGGING

8.1 Introduction

According to a 2002 NIST study [123, 103], inadequate software testing infrastructure costs the US economy an estimated $59.5 billion per year. While some of these costs are unavoidable, the report claimed that an estimated $22.2 billion could be saved with more effective tools for the identification and removal of software errors. Several other sources estimate that over 50% of software development costs are spent on debugging and testing [58, 11].

The need for better debugging tools has long been recognized. The goal of automating various debugging tasks has motivated a large body of research in the software engineering community. However, this line of work has only recently begun to take advantage of recent advances in probabilistic models, and their inference and learning algorithms [144].

In this work, we apply state-of-the-art probabilistic methods to the problem of fault localization. We propose *Tractable Fault Localization Models* (TFLMs) that can be learned from a corpus of known buggy programs (with the bug locations annotated). The trained model can then be used to infer the probable locations of buggy lines in a previously unseen program. Conceptually, a TFLM is a probability distribution over programs in a given language, modeled jointly with any attributes of interest (such as bug location indicator variables, or diagnostic features). Conditioned on a specific program, a TFLM defines a joint probability distribution over the attributes.

The key advantage of probabilistic models is their ability to learn from experience. Many software faults are instances of a few common error patterns, such as off-by-one errors and use of uninitialized values [18]. Human debuggers improve with experience as they encounter more of these common fault patterns, and learn to recognize them in new programs. Automated debugging

systems should be able to do the same.

Another advantage of probabilistic models is that they allow multiple sources of information to be combined in a principled manner. The relative contribution of each feature determined by its predictive value in the training corpus, rather than by a human expert. A TFLM can incorporate as features the outputs of other fault localization systems, such as the TARANTULA hue [68] of each line.

A limitation of probabilistic methods is that models that capture rich probabilistic dependencies often result in NP-hard inference problems. In recent years, there has been renewed interest in learning rich, tractable models, on which exact probabilistic inference can be performed in polynomial time (e.g. Sum-Product Networks (SPNs) [113]). TFLMs make use of similar ideas to RSPNs (chapter 7) to enable exact inference in space and time linear in the size of the program.

8.1.1 Contributions

1. We propose TFLMs, a scalable probabilistic fault localization model that can be learned from data. TFLMs are context-sensitive; the predicted bug probability for a statement can be affected by other statements in the program.

2. We describe an efficient exact inference algorithm for TFLMs, and propose an algorithm to learn TFLMs from a corpus of buggy programs.

3. We empirically compare TFLMs to the widely-used TARANTULA fault localization method, as well as the Statistical Bug Isolation (SBI) system, on four mid-sized C programs. TFLMs outperform the other systems on three of the four test subjects.

8.2 Background

8.2.1 Coverage-based Fault Localization

Coverage-based debugging methods isolate the bug's location by analyzing the program's coverage spectrum on a set of test inputs. These approaches take the following as input:

1. a set of unit tests;

2. a record of whether or not the program passed each test;

3. program traces, indicating which components (usually lines) of the program were executed when running each unit test.

Using this information, these methods produce a suspiciousness score for each component in the program. The most well-known method in this class is the TARANTULA system [68], which uses the following scoring function:

$$S_{Tarantula}(s) = \frac{\frac{Failed(s)}{TotalFailed}}{\frac{Passed(s)}{TotalPassed} + \frac{Failed(s)}{TotalFailed}}$$

Here, $Passed(s)$ and $Failed(s)$ are respectively the number of passing and failing test cases that include statement s, and $TotalPassed$ and $TotalFailed$ are the number of passing and failing test cases respectively. In an empirical evaluation [67], TARANTULA was shown to outperform previous methods such as cause transitions [23], set union, set intersection and nearest neighbor [116], making it the state of the art in fault localization at the time.

Since the publication of that experiment, a few other scoring functions have been shown to outperform TARANTULA under certain conditions [1], including the Jaccard score and Ochiai score. Nonetheless, TARANTULA remains the most well-known fault localization method in this class.

8.2.2 Probabilistic Debugging Methods

Per-Program Learning

Several approaches to fault localization make use of statistical and probabilistic methods. Liblit et al. proposed several influential statistical debugging methods. Their initial approach [84, 152] used ℓ_1-regularized logistic regression to predict non-deterministic program failures. (The instances are runs of a program, the features are instrumented program predicates, and the models are trained to predict a binary 'failure' variable. The learned weights of the features indicate which predicates

are the most predictive of failure.) In later work [85], they use a likelihood ratio hypothesis test to determine which *predicates* (e.g. branches, sign of return value) in an instrumented program are predictive of program failure. Zhang et al. [151] evaluate several other hypothesis testing methods in a similar setting.

The SOBER system [87, 86] improves on Liblit et al.'s 2005 approach by taking into account the fact that a program predicate can be evaluated multiple times in a single test case. They learn conditional distributions over the probability of a predicate evaluating to `true`, conditioned on the success or failure of the test case. When these conditional distributions differ (according a statistical hypothesis test), the predicate is considered to be 'relevant' to the bug. The HOLMES system [21] extends Liblit et al.'s approach along another direction, analyzing path profiles instead of instrumented predicates.

Wong et al. [148, 146] use a crosstab-based statistical analysis to quantify the dependence between statement coverage and program failure. Their approach can be seen as a hybrid between the Liblit-style statistical analysis and TARANTULA-style spectrum-based analysis. Wong et al. also proposed two neural network-based fault localization techniques trained on program traces [147, 145]. Ascari et al. [8] investigate the use of SVMs in a similar setting.

Many of the methods described above operate under the assumption that the program contains exactly one bug. Some of these techniques have nevertheless been evaluated on programs with multiple faults, using an iterative process where the bugs are isolated one by one. Briand et al. [16] explicitly extend TARANTULA to the multiple-bug case, by learning a decision tree to partition failing test cases. Each partition is assumed to model a different bug. Statements are ranked by suspiciousness using a TARANTULA-like scoring function, with the scores computed separately for each partition. Other clustering methods have also been applied to test cases; for example, Andrzejewski et al. [7] use a form of Latent Dirichlet Allocation to discover latent 'bug topics'.

TARANTULA-style scoring functions have also been studied through the lens of association rule mining [36]. Nessa et al. [100] discover association rules on N-grams over blocks of statements, mined from program traces.

Jiang and Su [66] use various machine learning techniques to discover faulty control paths, as

well as clusters of correlated predicates, which they claim are useful aids for bug understanding. Their approach uses SVMs (for small datasets) or random forests (for larger datasets) for feature selection (i.e. classifying program predicates as good or bad predictors of failure). The selected predicates are used to guide the construction of faulty control flow paths.

Dietz et al. [39] rank suspicious statements using a learned a graphical generative model over a program's execution graphs.

Generalizing Across Programs

The key limitation of the statistical and machine learning-based approaches discussed above is that they only generalize over many executions of a single program. Ideally, a machine learning-based debugging system should be able to generalize over multiple programs (or, at least, multiple sequential versions of a program). As discussed in section 8.1, many software defects are instances of frequently occurring fault patterns; in principle, a machine learning model can be trained to recognize these patterns and use them to more effectively localize faults in new programs.

This line of reasoning has received relatively little attention in the automated debugging literature. Hovemeyer and Pugh [62] manually identified 45 bug patterns, and wrote detectors for them. The most prominent learning-based approach that takes advantage of program-independent bug patterns is the Fault Invariant Classifier (FIC) of Brun and Ernst [18]. FIC is not a fault localization algorithm in the sense of TARANTULA and the other approaches discussed above. Instead of localizing the error to a particular line, FIC outputs *fault-revealing properties* that can guide a human debugger to the underlying error. These properties can be computed using static or dynamic program analysis; FIC uses the Daikon dynamic invariant detector [45]. At training time, properties are computed for pairs of buggy and fixed programs; properties that occur in the buggy programs but not the fixed programs are labeled as 'fault-revealing'. The properties are converted into program-independent feature vectors, and an SVM or decision tree is trained to classify properties as fault-revealing or non-fault-revealing. The trained classifier is then applied to properties extracted from a previously unseen, potentially faulty program, to reveal properties that may be

indicative of latent errors.

8.2.3 Tractable Probabilistic Models

Most of the existing statistical debugging methods discussed in section 8.2.2 use simple statistical tests that can be computed in closed form, or models such as logistic regression that fail to capture dependencies among the feature variables. These models are adequate for capturing direct dependencies between individual program predicates and a failure outcome, given a set of executions of a single program.

However, to generalize across programs, the model must be rich enough to capture common bug patterns, and distinguish them from bug-free code. Such a model must take into account both the content of a line and its context in the program. Probabilistic graphical models (PGMs) such as Markov networks are an obvious candidate, allowing a rich dependency structure among different parts of the program. Unfortunately, inference in graphical models is #P-complete [120]; a debugging model based on PGMs would require approximate inference algorithms to scale to realistically-sized programs. Approximate probabilistic inference algorithms tend to be unreliable and computationally expensive, and generally cannot provide non-trivial error bounds.

An emerging line of research (section 2.2.2) has explored tractable high-treewidth probabilistic models, which are significantly richer than traditional tractable models like logistic regression and hidden Markov models (HMMs), while still guaranteeing polynomial-time exact inference. This work builds on one such approach, Sum-Product Networks (SPNs) [113]. The use of this rich class of models allows our approach to capture the contextual information necessary to learn useful bug patterns. The efficient exact inference algorithms of SPNs can also be applied to our model, allowing bugs to be rapidly localized even on relatively large programs without resorting to approximate inference algorithms.

TFLMs are also closely related to RSPNs (chapter 7). Specifically, TFLMs can be seen as RSPNs with one class per non-terminal symbol in the grammar. TFLMs use a fixed class SPN structure, instead of a structure discovered by an algorithm such as LearnRSPN. As seen in the following section, TFLMs also allow the attributes to be modeled jointly (e.g. via a logistic regression

model) rather than as separate univariate distributions, as in standard RSPNs.

8.3 Tractable Fault Localization

8.3.1 Tractable Fault Localization Models

A *Tractable Fault Localization Model* (TFLM) defines a probability distribution over programs in some deterministic language L. The distribution may also model additional variables of interest that are not part of the program itself; we refer to such variables as *attributes*. In the fault localization setting, the important attribute is a *buggy* indicator variable on each line. Other informative features may also be included as attributes; for instance, one or more coverage-based metrics may be included for each line.

More formally, consider a language whose grammar $L = (V, \Sigma, R, S)$ consists of:

- V is a set of *non-terminal* symbols;
- Σ is a set of *terminal* symbols;
- R is a set of production rules of the form $\alpha \to \beta$, where $\alpha \in V$ and β is a string of symbols in $V \cup \Sigma$;
- $S \in V$ is the start symbol.

Definition 7. A *Tractable Fault Localization Model* for language L consists of:

- a map from non-terminals in V to sets of *attribute variables* (discrete or continuous);
- for each symbol $\alpha \in V$, a set of latent subclasses $\alpha_1, \ldots, \alpha_k$;
- π_S, a probability distribution over subclasses of the start symbol S;
- for each each subclass α_i of α,
  - a univariate distribution $\psi_{\alpha_i, x}$ over each attribute x associated with α;

- ρ_{α_i}, a probability distribution over rules $\alpha_i \to \beta$, for each rule $\alpha \to \beta$ in L;
- for each rule $\alpha_i \to \beta$, for each non-terminal $\alpha' \in \beta$, a distribution $\pi_{(\alpha_i \to \beta),\alpha'}$ over subclasses of α'.

The univariate distribution over each attribute may be replaced with a joint distribution over all attributes, such as a logistic regression model within each subclass that predicts the value of the *buggy* attribute, using one or more other attributes as features. However, for simplicity, we present the remainder of this section with the attributes modeled as a product of univariate distributions, and assume that the attributes are discrete.

TFLMs are related to the Latent Variable PCFG (L-PCFG) models used in the natural language processing (NLP) community [91, 115, 109]. As in L-PCFGs, each symbol α in the language is drawn probabilistically from a set of latent subclasses $\alpha_1, \ldots, \alpha_k$. For each latent class, the model can define a different distribution over the sub-trees rooted at that symbol.

Conceptually, TFLMs differ from the PCFG-based models as used in NLP in two ways:

1. In addition to modeling a distribution over strings in the given language, TFLMs can also jointly model other variables of interest ('attributes'). Different latent subclasses can have different distributions over attributes.

2. In NLP, PCFGs and their extensions are usually used for parsing, i.e. finding the most probably parse tree according to the given probabilistic grammar. In the debugging context, we assume the program can be parsed unambiguously. The purpose of a TFLM is to answer probabilistic queries about the attributes of the given program (e.g. infer marginal probabilities).

Being defined over the grammar of the programming language, TFLMs can capture information extremely useful for the fault localization task. For example, a TFLM can represent different fault probabilities for different symbols in the grammar. In addition, the latent subclasses give TFLMs a degree of context-sensitivity; the same symbol can be more or less likely to contain a

fault depending on its latent subclass, which is probabilistically dependent on the subclasses of ancestor and descendent symbols in the parse tree. This makes TFLMs much richer than models like logistic regression, where the features are independent conditioned on the class variable. Despite this representational power, exact inference in TFLMs is still computationally efficient, as seen in section 8.3.1.

Example 4. The following rules are a fragment of the grammar of a Python-like language:

```
while_stmt → 'while' condition ':' suite
condition → expr operator expr
condition → 'not' condition
```

We refer to the above rules as r_1, r_2 and r_3 respectively.

The following is a partial specification of a TFLM over this grammar, with the `while_stmt` symbol as root.

- All non-terminal symbols have *buggy* and *suspiciousness* attributes. *buggy* is a fault indicator, and *suspiciousness* is a diagnostic attribute, such as a TARANTULA score.

- Each non-terminal has two latent subclass symbols. For example, `while_stmt` has subclasses $\texttt{while\_stmt}_1$ and $\texttt{while\_stmt}_2$.

- The distribution over start symbols is:

$$\pi(\texttt{while\_stmt}_1) = 0.4$$
$$\pi(\texttt{while\_stmt}_2) = 0.6$$

- For subclass symbol $\texttt{while\_stmt}_1$ (subclass subscripts omitted):

  - $\psi_{buggy} \sim Bernoulli(0.01)$
  - $\psi_{suspiciousness} \sim \mathcal{N}(0.4, 0.05)$
  - $\rho(r_1) = 1.0$, since `while_stmt` has a single rule.

– The distributions over child symbol subclasses for r_1 are:

$$\pi_{r_1,\texttt{condition}}(\texttt{condition}_1) = 0.7$$
$$\pi_{r_1,\texttt{condition}}(\texttt{condition}_2) = 0.3$$

$$\pi_{r_1,\texttt{suite}}(\texttt{suite}_1) = 0.2$$
$$\pi_{r_1,\texttt{suite}}(\texttt{suite}_2) = 0.8$$

(The complete TFLM specification would have similar definitions for all the other subclass symbols in the model.)

Semantics

Conceptually, a TFLM defines a probability distribution over all programs in L, and their attributes. More formally, the joint distribution $P(T, A, C)$ is defined over:

- a parse tree T;
- an attribute assignment A, specifying values of all attributes of all non-terminal symbols in T;
- a latent subclass assignment C for each non-terminal in T.

For parse tree T containing rules $r_1 = \alpha_1 \to \beta_1, r_2 = \alpha_2 \to \beta_2, \ldots, r_n = \alpha_n \to \beta_n$, and root symbol α_R,

$$\begin{aligned} P(T, A, C) = \prod_{r_i} \Bigg(& \rho_{C(\alpha_i)}(\alpha_i \to \beta_i) \\ & \times \prod_{\alpha' \in \beta_i} \pi_{(C(\alpha_i) \to \beta_i), \alpha'}(C(\alpha')) \\ & \times \prod_{x \in attr(\alpha_i)} \psi_{C(\alpha_i), x}(A(x)) \Bigg) \\ \times \pi_S(C(\alpha_R)) & \end{aligned}$$

Intuitively, the generative process is as follows:

- Choose a subclass for the start node, according to π_S. Let the current symbol α be the start symbol, and let α_i be its subclass.

- Choose attribute values for the current symbol, according to $\psi_{\alpha_i,x}$.

- Choose a production rule $\alpha \to \beta$ for the current symbol, according to ρ_{α_i}.

- Recurse over each non-terminal α' in β, choosing its subclass from $\pi_{(\alpha_i \to \beta),\alpha'}$.

The joint probability of a parse tree, its attributes, and its symbols' subclass assignments is the product of the individual probabilities of all these choices.

Inference

In the fault localization setting, the parse tree T is known; the inference task is to compute the marginal probabilities of the *buggy* attributes, conditioned on the buggy program and diagnostics. This computation can be done in polynomial time, using a dynamic programming algorithm similar to the inference algorithm for SPNs.

Given parse tree T and evidence E (possibly empty or incomplete) about attributes of symbols in T, let T_α be the subtree rooted at symbol α, and E_α be the evidence about symbols in T_α. Let $\alpha \to \beta$ be the rule applied from α within T. The function to be memoized is $P(E_\alpha|C(\alpha))$, which represents the probability of evidence within that subtree, conditioned on a particular subclass assignment for root symbol α. (Note that the probability is also conditioned on the parse tree, T; for readability, we omit this condition from probability expressions in this section.) This can be computed as follows by summing over attribute value assignments A and class assignments C:

$$\begin{aligned} P(E_\alpha|C(\alpha)) &= \sum_A \sum_{C'=C\setminus C(\alpha_R)} P(A, C', E_\alpha|C(\alpha)) \\ &= \sum_A \sum_{C'} \prod_{r_i} \left(\prod_{\alpha' \in \beta_i} \pi_{(C(\alpha_i)\to\beta_i),\alpha'}(C(\alpha')) \prod_{x \in attr(\alpha_i)} \psi_{C(\alpha_i),x}(A(x), E_\alpha) \right) \end{aligned}$$

$$P(E_\alpha|C(\alpha)) = \prod_{x \in attr(\alpha)} \sum_{A(x)} \psi_{C(\alpha),x}(A(x), E_\alpha) \times \prod_{\alpha' \in \beta} \sum_{C(\alpha')} \left(\pi_{(C(\alpha)\to\beta),\alpha'}(C(\alpha')) \times P(E_{\alpha'}|C(\alpha'))\right) \tag{8.1}$$

(Note that $\psi_{C(\alpha),x}(A(x), E_\alpha) = 0$ when $A(x)$ is inconsistent with the evidence E_α, and $\psi_{C(\alpha),x}(A(x), E_\alpha) = \psi_{C(\alpha),x}(A(x))$ otherwise.)

As seen above, $P(E_\alpha|C(\alpha))$ decomposes into two terms:

- a product of conditional partition functions (i.e. the total probability mass consistent with the evidence) of univariate distributions over attributes of the root symbol;
- a product of evidence probabilities within each of the subtrees of T_α, each of which is a weighted sum over the subclass of the subtree's root symbol.

$P(E_\alpha|C(\alpha))$ can be computed in a bottom-up fashion, with the values for each symbol α computed after those of its children in the parse tree. The cardinality of the domain of the memoized function is the number of non-terminals in the parse tree, multiplied by the number of subclasses of each symbol in the grammar. This provides a bound on the time and space cost of inference: if the number of attributes and rule length are bounded, inference is linear in the size of the program and the number of subclasses.

The probability of the evidence E according to the full model is simply:

$$P(E) = \sum_{C(\alpha_R)} \pi_S(C(\alpha_R)) P(E|C(\alpha_R))$$

(Where α_R is the root symbol of T.) This enables marginal probabilities to be inferred through a single bottom-up pass of the dynamic programming algorithm, conditioned on the evidence.

Note that the above probability computation can be directly mapped to an SPN, of the form described by Poon and Domingos [113]. The computation of $P(E_\alpha|C(\alpha))$ (in the form of equation 8.1) corresponds to a product node; each weighted summation over subclasses corresponds to a sum node; the attribute indicators correspond to leaf nodes. The dynamic programming-based inference algorithm described above can be seen as a special case of SPN inference. Like SPNs,

TFLMs can also compute the most probable (MAP) state of (A, C), by replacing the summations with maximizations.

Simultaneous Inference

Like SPNs, TFLMs also allow all marginal probabilities to be computed simultaneously, instead of computing the *buggy* probabilities one by one. The simultaneous inference algorithm requires two passes – first, a bottom-up (children-first) pass, and then a top-down pass. This is analogous to the two passes used by the forward-backward algorithm for Hidden Markov Models, or the belief propagation algorithm for tree-structured graphical models.

The first pass of the algorithm is simply the bottom-up computation of $P(E_\alpha|C(\alpha))$, as described in the previous section. As in the previous section, let T denote a parse tree, and let T_α be its subtree rooted at symbol α. E denotes evidence (possibly empty or incomplete) about attributes of symbols in T; let E_α be the subset of the evidence dealing with symbols in T_α, and $E_{\bar{\alpha}}$ be the evidence concerning the remaining symbols in T.

The probability of a subclass assignment conditioned on the evidence is:

$$\begin{aligned} P(C_\alpha|E) &= P(C(\alpha)|E_\alpha, E_{\bar{\alpha}}) \\ &= P(E_\alpha|C(\alpha), E_{\bar{\alpha}}) \frac{P(C(\alpha)|E_{\bar{\alpha}})}{P(E_\alpha|E_{\bar{\alpha}})} \\ P(C_\alpha|E) &\propto P(E_\alpha|C(\alpha)) P(C(\alpha)|E_{\bar{\alpha}}) \end{aligned}$$

(This is because E_α and $E_{\bar{\alpha}}$ are independent conditioned on the class of α.)

The first term is simply the memoized function computed by the bottom-up parse. Therefore, to compute the marginal probability of the class assignment, we must compute the second term, $P(C(\alpha)|E_{\bar{\alpha}})$, which is the probability of the given class assignment conditioned on the evidence *outside* the subtree T_α. This is the function to be memoized in the top-down pass.

If α is the root of the entire parse tree T, $E_{\bar{\alpha}}$ is empty; in this case, $P(C(\alpha)|E_{\bar{\alpha}})$ follows directly from π_S, the distribution over subclasses of the start symbol.

If α has a parent symbol β in T, the evidence $E_{\bar{\alpha}}$ can affect the probability of α's class assign-

ment via its effect on β's class assignment probabilities:

$$P(C(\alpha)|E_{\bar{\alpha}}) = \sum_{C(\beta)} P(C(\alpha)|C(\beta))P(C(\beta)|E_{\bar{\alpha}})$$

The first term follows directly from $C(\beta)$'s $\pi_{r,C(\alpha)}$ distribution for the rule r that generated α in T. To compute the second term, $P(C(\beta)|E_{\bar{\alpha}})$, notice that a similar term has already been memoized in the top-down pass: $P(C(\beta)|E_{\bar{\beta}})$ is the previously-computed probability of the class assignment of the parent symbol β conditioned on the evidence outside T_β. This allows us to compute the above expression by additionally conditioning on $E_{\bar{\alpha}} \backslash E_{\bar{\beta}}$, i.e. the evidence on β itself. We denote this as e_β.

$$\begin{aligned}
P(C(\beta)|E_{\bar{\alpha}}) &= P(C(\beta)|E_{\bar{\beta}}, e_\beta) \\
&= \frac{P(C(\beta), e_\beta|E_{\bar{\beta}})}{P(e_\beta|E_{\bar{\beta}})} \\
&= \frac{P(C(\beta)|E_{\bar{\beta}})P(e_\beta|C(\beta), E_{\bar{\beta}})}{P(e_\beta|E_{\bar{\beta}})} \\
&= \frac{P(C(\beta)|E_{\bar{\beta}})P(e_\beta|C(\beta))}{P(e_\beta|E_{\bar{\beta}})} \\
P(C(\beta)|E_{\bar{\alpha}}) &\propto P(C(\beta)|E_{\bar{\beta}})P(e_\beta|C(\beta))
\end{aligned}$$

The first term is the memoized function, and the second term follows directly from the $\psi_{C(\beta),x}$ attribute distributions.

The above procedure allows us to compute the marginal probabilities of all the latent subclass assignments of the symbols in T, conditioned on the available attribute evidence (i.e. $P(C(\alpha)|E)$, for all α in T). This in turn allows us to trivially compute the marginal probabilities of all unknown attributes:

$$P(A(x_\alpha)|E) = \sum_{C(\alpha)} P(C(\alpha)|E)\psi_{C(\alpha),x_\alpha}(A(x_\alpha))$$

Computing the marginal probabilities in this manner requires one bottom-up parse to compute $P(E_\alpha|C(\alpha))$, one top-down pass to compute $P(C_\alpha|E_{\bar{\alpha}})$, and a constant number of operations per query variable to sum over the subclass probabilities of the corresponding symbol. This yields a linear speedup (in the number of queries) over the naïve approach of recomputing equation 8.1

from scratch, conditioning on one query variable at a time. In the fault localization setting, the number of queries is the number of executable lines in the program; as a result, the simultaneous inference approach yields a several order of magnitude speedup.

Learning

The learning problem in TFLMs is to estimate the π and ψ distributions from a training corpus of programs, with known attribute values but unknown latent subclasses. (Note that unlike standard PCFGs, the ρ distributions have no effect on the distribution of interest, since we assume that every program can be unambiguously mapped to a parse tree.)

Models with latent variables are typically trained via Expectation Maximization (EM). As is typically done with multilayered models like SPNs, TFLMs use the Hard EM variant, which assigns a single value to each latent variable rather than a fractional assignment. The TFLM training algorithm alternates between the two steps:

- **E-step:** given the current parameters of π and ψ, compute the MAP state of the training programs (i.e. the latent subclass assignment that maximizes the log-probability).

- **M-step:** re-estimate the parameters of π and ψ, choosing the values that maximize the log-probability.

These two steps are repeated until convergence, or for a fixed number of iterations.

If the attributes are modeled jointly rather than as a product of univariate distributions, retraining the joint model in each iteration of EM may prove computationally expensive. A more efficient alternative is to use a product of univariates during EM, in order to learn a good subclass assignment. The joint model is then only trained once, at the conclusion of EM.

8.4 Experiments

We performed an experiment to determine whether TFLM's ability to combine a coverage-based fault localization system with learned bug patterns improves fault localization performance, rela-

tive to using the coverage-based system directly. As a representative coverage-based method, our study used TARANTULA, one of the most widely-used approaches in this class, and a common comparison system for fault localization algorithms. We also compared to the statement-based version of Liblit et al.'s Statistical Bug Isolation (SBI) system[1] [85], as adapted by Yu et al. [150]. SBI serves as a representative example of a lightweight statistical method for fault localization.

8.4.1 Subjects

We evaluated TFLMs on four mid-sized C programs from the Software-artifact Infrastructure Repository (SIR) [133]. All four test subjects are real-world programs, commonly used to evaluate fault localization approaches. The repository contained several sequential versions of each program, each with several buggy versions. The repository also contained a suite of TSL tests for each version, which we used to compute the TARANTULA scores.

See Table 8.1 for statistics about each of the datasets. Total line counts include white space, documentation, and other non-executable lines (e.g. lines excluded by preprocessor directives). The number of executable lines was measured by the gcov tool. We excluded buggy versions where the bug occurred in a non-executable line, or consisted of line insertions or deletions. However, unlike most previous fault localization studies that use these subjects, we do not exclude versions for which the test results were uniform (i.e. consisting entirely of passing or failing tests). Although coverage-only methods such as TARANTULA can provide no useful information in the case of uniform test suites, TFLMs can still make use of learned contextual information to determine that some lines are more likely than others to contain a fault.

8.4.2 Methodology

We implemented TFLMs for a simplified version of the C grammar with 23 non-terminal symbols, ranging from compound statements like `if` and `while` to atomic single-line statements such as assignments and `break` and `continue` statements.

---

[1] Also sometimes referred to as Cooperative Bug Isolation (CBI)

Table 8.1: Subject programs.

Program	Bug type	Version	Total LOC	Executable LOC	Buggy versions	Tests
grep	seeded	2.2	12555	3197	9	470
		2.3	13182	3363	4	470
		2.4	13274	3445	14	470
		2.4.1	13286	3465	5	470
gzip	seeded	1.1.2	6502	1721	9	214
		1.2.2	7196	2022	4	214
		1.2.3	7222	1867	3	214
		1.2.4	7273	1895	10	214
		1.3	7933	2018	8	214
flex	seeded	2.4.7	12329	3453	14	525
		2.5.1	14034	4003	13	525
		2.5.2	14082	4008	9	525
		2.5.3	14171	4034	9	525
		2.5.4	14192	4036	4	525
sed	real, seeded	1.18	8059	2027	3	126
		2.05	11907	3738	2	126
		3.01	9976	2280	6	126
		3.02	9981	2281	3	126
		4.0.6	6635	1518	4	124
		4.0.7	6665	1524	3	124
		4.1.5	7125	1710	3	124

Table 8.2: Fault localization accuracy (fraction of lines skipped).

Program	TFLM	Tarantula	SBI
grep	**0.645**	0.640	0.564
gzip	0.516	**0.682**	0.540
flex	**0.770**	0.704	0.618
sed	**0.927**	0.851	0.603

Table 8.3: TFLM (with Tarantula as a diagnostic attribute) vs Tarantula alone.

Program	TFLM wins	Ties	Tarantula wins
grep	**18**	0	14
gzip	11	0	**23**
flex	**31**	0	18
sed	**17**	0	7

Table 8.4: TFLM learning and inference times.

Program	Avg. learn time (s)	Avg. infer time (s)
grep	1135.00	20.91
gzip	433.33	5.25
flex	978.63	13.15
sed	326.37	5.18

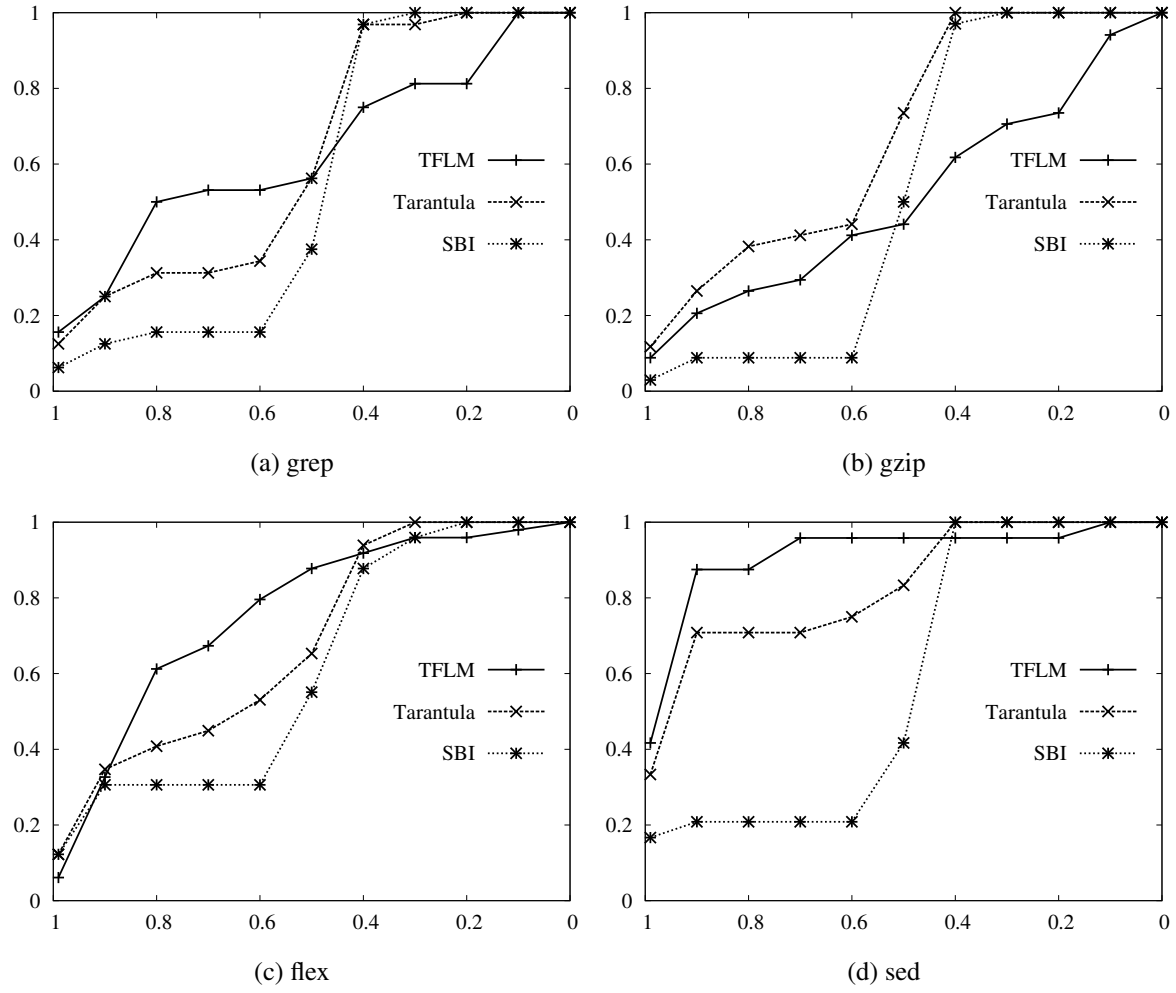


Figure 8.1: The horizontal axis is the fraction of lines skipped (FS), and the vertical axis is the fraction of runs.

Each symbol has a *buggy* attribute, and a *suspiciousness* attribute, which is the TARANTULA score of the corresponding line. (For AST nodes that correspond to multiple lines in the original code, we use the highest TARANTULA score among all lines). As described in section 8.3.1, the attributes are modeled as independent univariates during EM (*buggy* as a Bernoulli distribution, and *suspiciousness* as a Gaussian), and then via a logistic regression model within each subclass. The model predicts the *buggy* attribute, using the TARANTULA score and a bias term as features. We use the SCIKIT-LEARN [107] implementation of logistic regression, with the `class_weight='auto'` parameter, to compensate for the sparsity of the buggy lines relative to bug-free lines. For TFLMs, we ran hard EM for 100 iterations.

For each of the four subject programs, TFLMs were learned via cross-validation, training on all versions of the program except the one being evaluated. The number of latent subclasses was also chosen via cross-validation, from the range $[1, 4]$.

The output of a fault localization system is a ranking of the lines of code from most to least suspicious. For TFLMs, we ranked the lines by predicted probability that $\textit{buggy} = 1$. (Each line in the original program is modeled by the finest-grained AST node that encloses it.) The evaluation metric was the 'fraction skipped' (FS) score, i.e. the fraction of executable lines ranked below the highest-ranked buggy line.

All experiments were run on 8 core 2.33 GHz PCs with 16 GB of RAM.

8.4.3 Results

Results of our experiments are displayed in tables 8.2 and 8.3, and figure 8.1. TFLMs outperform TARANTULA and SBI on three of the four subjects, isolating the majority of bugs more effectively, and earning a higher average FS score. However, TFLMs perform poorly on one the `gzip` domain. This demonstrates the main threat to the validity of our method: machine learning algorithms operate under the assumption that the test data is drawn from a similar distribution to the training data. If the bugs occur in different contexts in the training and test datasets (as in `gzip`), learning-based methods may perform worse than methods that try to localize each program independently. This risk is particularly great when the learning from a small corpus of buggy programs.

However, in three of the four subjects in our experiment, the training and test distributions are sufficiently similar to allow useful generalization, resulting in improved fault localization performance. TFLMs' advantage arises from its ability to localize faults even when the coverage matrix used by TARANTULA does not provide useful information (e.g. when the tests are not sufficiently discriminative). TFLMs combine the coverage-based information used by TARANTULA with learned bug probabilities for different symbols, in different contexts. Context sensitivity is captured via latent subclass assignments for each symbol.

As seen in table 8.4, our unoptimized Python implementation predicts bug probabilities in a few seconds for programs a few thousand lines in length. An optimized implementation may be able to make predictions at interactive speeds; this makes TFLMs a practical choice of inference engine for a debugging tool in a software development environment. Learning TFLMs can take several minutes, but note that the model can be trained offline, either from previous versions of the software being developed (as in our experiments), or from other related software projects expected to have a similar bug distribution (e.g. projects of a similar scale, written in the same language).

8.5 Conclusions & Future Work

This chapter presented TFLMs, a probabilistic model for fault localization that can be learned from a corpus of buggy programs. This allows the model to generalize from previously seen bugs to more accurately localize faults in a new context. TFLMs can also incorporate the output of other fault-localization systems as features in the probabilistic model, with a learned weight that is dependent on the context. TFLMs take advantage of recent advances in tractable probabilistic models to ensure that the fault location probabilities can be inferred efficiently even as the size of the program grows. In our experiments, a TFLM trained with TARANTULA as a feature localized bugs more effectively than TARANTULA or SBI alone, on three of the four subject programs.

In this work, we used TFLMs to generalize across sequential versions of a single program. Given adequate training data, TFLMs could also be used to generalize across more distantly-related programs. The success of this approach relies on the assumption that there is some regularity in software faults, i.e. the same kinds of errors occur repeatedly in unrelated software projects, with

sufficient regularity that a machine learning algorithm can generalize over these programs. Testing this assumption is a direction for future work.

Another direction for future work is extending TFLMs with additional sources of information, such as including multiple fault localization systems, and richer program features derived from static or dynamic analysis (e.g. invariants [59, 18]). TFLM-like models may also be applicable to debugging methods that use path profiling [21], giving the user more contextual information about the bug, rather than just a ranked list of statements. The recent developments in tractable probabilistic models may also enable advances in other software engineering problems, such as fault correction, code completion, and program synthesis.

Chapter 9

CONCLUSION

One of the central challenges in statistical relational learning is the tradeoff between expressiveness and tractability. This dissertation proposed several inference and learning approaches that make statistical relational learning more practical for real problems.

9.1 Contributions of this Dissertation

In this dissertation, we explored two strategies for dealing with intractability in statistical relational models: (a) better approximate inference methods, and (b) richer statistical relational representations that still allow efficient inference and learning. The following are our inference-related contributions:

- We proposed Markov logic decision networks (MLDNs), a representation for utility maximization problems in relational domains. MLDNs extend Markov logic networks with action and decision predicates.

- We proposed Expanding Frontier Belief Propagation (EFBP), an approximate propositional algorithm for expected utility maximization, based on loopy belief propagation. We provided bounds on the difference between the marginals computed by EFBP and standard BP. Empirically, EFBP is several orders of magnitude faster than BP, and produces extremely similar results.

- We theoretically analyzed two approaches for approximate lifted belief propagation, deriving error bounds for the lifting approximation. In the course of this analysis, we extended Ihler et al.'s [64] BP error bounds to arbitrary factor graphs.

- We proposed ΔLNC, an efficient algorithm for updating the structure of a lifted network with incremental changes to the evidence. ΔLNC works by retracing the path the changed atoms would have taken during lifted network construction (LNC), modifing the network as necessary. Experiments on video segmentation and viral marketing problems show that ΔLNC provides very large speedups over standard LNC, making it applicable to a wider range of problems.

The following contributions are related to learning richer statistical relational models:

- We proposed a model and learning algorithm for Multiple Hierarchical Relational Clustering (MHRC), unifying several previous approaches to relational clustering.

- We proposed Relational Sum-Product Networks (RSPNs), a new tractable first-order probabilistic model. We also presented LearnRSPN, the first algorithm for learning high-treewidth tractable statistical relational models. Empirically, LearnRSPN outperforms conventional statistical relational methods in accuracy, inference time and training time.

- We presented TFLMs, a probabilistic model for software fault localization, based on similar ideas to RSPNs. TFLMs can be trained on a corpus of previously seen bugs, learning to recognize commonly occurring bug patterns. TFLMs can also include the output of other fault localization systems as features in the probabilistic model. In our experiments, TFLMs with TARANTULA as a feature localized faults more effectively than TARANTULA alone.

9.2 Directions for Future Work

Each of the contributions described above suggests several items for future work. Below, we first discuss the future work for each individual contribution, and then outline some broader research directions.

Relational Decision Theory

In chapter 3, we proposed an algorithm for utility maximization in Markov Logic Decision Networks. However, learning the structure and parameters of MLDNs is an open problem. An algorithm for learning MLDNs would make them applicable to relational reinforcement learning, and other forms of utility-guided learning.

Our applications of MLDNs used a simple greedy search over actions. A greedy search has the advantage of allowing much of the computation to be reused, when combined with algorithms such as EFBP (chapter 3) and ΔLNC (chapter 5). However, for problems with rich sequential or relational structure, a more sophisticated search algorithm may be needed. An important research direction for MLDNs is the design of richer search algorithms over action assignments that still allow efficient repeated inference.

EFBP and ΔLNC

EFBP and ΔLNC are general-purpose repeated inference algorithms. In this work, we applied them to the problems of utility maximization and image segmentation in video streams. However, they may also be useful for many other repeated inference problems, such as Marginal MAP, weight learning, etc.

Both EFBP and ΔLNC were formulated to efficiently recompute marginal probabilities to incorporate changes in the evidence. However, both algorithms can be straightforwardly extended to incorporate changes in the model structure. This makes them applicable to the problem of structure learning in propositional and relational graphical models.

EFBP and ΔLNC allow the reuse of computation in propositional and lifted belief propagation. However, other approximate algorithms (e.g. MCMC, MC-SAT [112]) may also allow some of the computation to be reused as the evidence changes. Developing repeated inference versions of other propositional and lifted inference algorithms is another promising direction for future work.

Approximate Lifting

Lifted inference has been an active research area in recent years, with many recent approaches based on algorithms other than belief propagation [139, 54, 141]. Extending these algorithms to take advantage of approximate symmetries is another direction for future work.

Relational Sum-Product Networks

One of the key limitations of RSPNs is that applying them to a new domain requires the domain expert to provide a part structure for each new mega-example. Developing an efficient, principled approach to automatically discovering this part structure would make RSPNs more widely applicable.

Another direction for future work is extending relational decision theory to RSPNs, allowing computationally tractable decision-making in relational domains.

As defined in chapter 7, RSPNs make use of the notion of finite exchangeability [38]. However, Diaconis and Freedman also defined a more general notion of *partial* exchangeability. Extending RSPNs to take advantage of partial symmetries would potentially make the model more tractable and statistically robust.

Tractable Fault Localization Models

In this work, we used TFLMs to generalize across sequential versions of a single program. Given adequate training data, TFLMs could also be used to generalize across more distantly-related programs. The success of this approach relies on the assumption that there is some regularity in software faults, i.e. the same kinds of errors occur repeatedly in unrelated software projects, with sufficient regularity that a machine learning algorithm can generalize over these programs. Testing this assumption is a direction for future work.

Another direction for future work is extending TFLMs with additional sources of information, such as including multiple fault localization systems, richer program features derived from static or dynamic analysis (e.g. invariants [59, 18]), versioning system logs, human annotations, etc.

TFLM-like models may also be applicable to debugging methods that use path profiling [21], giving the user more contextual information about the bug, rather than just a ranked list of statements.

The recent developments in tractable probabilistic models may also enable advances in other software engineering problems, such as fault correction, code completion, and program synthesis.

9.2.1 Broader Directions

Despite the variety of representations and problem settings considered in the dissertation, our algorithms derive much of their efficiency gains from the following two principles:

- *Exploiting Symmetries*

  Detecting symmetries in a probabilistic model allows computation to be performed once and reused across all symmetric components of the model. This is most apparent in the case of lifted inference, but our repeated inference algorithms (EFBP and ΔLNC) can also be viewed through the lens of symmetry: in this case, we exploit symmetries across different evidence assignments. RSPNs are also rich in symmetries, in the form of exchangeable parts.

  All these algorithms can be improved by broadening the classes of symmetries we exploit. Exact symmetries are rare in real datasets. The approximate lifting techniques analyzed in this paper exploit the notion of 'approximate symmetry', and we provide bounds on how much the approximation affects our computed probabilities. EFBP similarly exploits approximate symmetries across evidence assignments. ΔLNC exploits symmetries both within each evidence assignment, and across evidence assignments. RSPNs only make use of exact symmetries; in addition to the possible use of partial exchangeability as discussed above, they may also be extended to exploit approximate symmetries (making the model approximate, but also gaining the additional computational efficiency and statistical robustness that result from the exploitation of symmetries). Another research direction is extending MHRC and TFLM to make use of exact and approximate symmetries.

Our algorithms could also be extended to make use of domain-specific symmetries. For example, many vision problems such as object detection and image segmentation may be symmetric with respect to rotation, reflection, translation etc. Potential symmetries to exploit in natural language processing tasks include paraphrasing and synonymy.

- *Variable Decompositions*

  A variable decomposition in a model is a partition of its random variables into independent subsets. This makes the model more computationally tractable, but sacrifices the ability to model relationships and dependencies among variables in different partitions. The MHRC learning algorithm we propose first decomposes the variables in the model, and then learns a hierarchical clustering for each subset. In contrast, SPNs (and by extension RSPNs and TFLMs) make the decompositions context-specific, allowing the dependencies to be modeled in some contexts but not in others. This allows them to reap the computational benefits of the independence assumption, without sacrificing as much representative power. Allowing MHRC to exploit context-specific independence in this manner may improve the model's ability to predict missing values in the database. Detecting and exploiting approximate variable independence might also enable more efficient approximate inference in MLN-like models.

BIBLIOGRAPHY

[1] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. J. C. van Gemund. A practical evaluation of spectrum-based fault localization. *The Journal of Systems and Software*, 82(11):1780–1792, 2009.

[2] U. A. Acar, A. T. Ihler, R. R. Mettu, and Ö. Sümer. Adaptive inference on general graphical models. In *Proceedings of the Conference on Uncertainty in Artificial Intelligence (UAI)*, 2008.

[3] B. Ahmadi, K. Kersting, M. Mladenov, and S. Natarajan. Exploiting symmetries for scaling loopy belief propagation and relational training. *Machine Learning*, 92(1):91–132, 2013.

[4] M. R. Amer and S. Todorovic. Sum-product networks for modeling activities with stochastic structure. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2012.

[5] J. Amilhastre, M. C. Vilarem, and P. Janssen. Complexity of minimum biclique cover and minimum biclique decomposition for bipartite domino-free graphs. *Discrete applied mathematics*, 86(2–3):125–144, 1998.

[6] G. Andrew and J. Gao. Scalable training of L1-regularized log-linear models. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2007.

[7] D. Andrzejewski, A. Mulhern, B. Liblit, and X. Zhu. Statistical debugging using latent topic models. In *Proceedings of the European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML/PKDD)*, 2007.

[8] L. C. Ascari, L. Y. Araki, A. R. T. Pozo, and S. R. Vergilio. Exploring machine learning techniques for fault localization. In *Proceedings of the Latin American Test Workshop*, 2009.

[9] F. Bach and M. I. Jordan. Thin junction trees. In *Advances in Neural Information Processing Systems (NIPS)*, 2001.

[10] A. L. Barabási and R. Albert. Emergence of scaling in random networks. *Science*, 286:509–512, 1999.

[11] B. Beizer. *Software Testing Techniques*. International Thomson Computer Press, 1990.

[12] V. D. Blondel, J. Guillaume, R. Lambiotte, and E. Lefebvre. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment*, 10:10008–10019, 2008.

[13] C. Blundell, Y. W. Teh, and K. A. Heller. Discovering non-binary hierarchical structures with Bayesian rose trees. In K. Mengersen, C. P. Robert, and M. Titterington, editors, *Mixture Estimation and Applications*. John Wiley & Sons, 2011.

[14] C. Boutilier, R. Reiter, and B. Price. Symbolic dynamic programming for first-order MDPs. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 2001.

[15] R. Brafman and Y. Engel. Lifted optimization for relational preference rules. In *Papers from the International Workshop on Statistical Relational Learning (SRL)*, 2009.

[16] L. C. Briand, Y. Labiche, and X. Liu. Using machine learning to support debugging with TARANTULA. In *Proceedings of the 18th IEEE International Symposium on Software Reliability (ISSRE)*, 2007.

[17] M. Bröcheler, L. Mihalkova, and L. Getoor. Probabilistic similarity logic. In *Proceedings of the Conference on Uncertainty in Artificial Intelligence (UAI)*, 2010.

[18] Y. Brun and M. Ernst. Finding latent code errors via machine learning over program executions. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2004.

[19] H. Bui, T. Huynh, and S. Riedel. Automorphism groups of graphical models and lifted variational inference. In *Proceedings of the Conference on Uncertainty in Artificial Intelligence (UAI)*, 2013.

[20] A. Chechetka and C. Guestrin. Efficient principled learning of thin junction trees. In *Advances in Neural Information Processing Systems (NIPS)*, 2007.

[21] T. M. Chilimbi, B. Liblit, K. Mehra, A. V. Nori, and K. Vaswani. HOLMES: Effective statistical debugging via efficient path profiling. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2009.

[22] J. Choi, D. Hill, and E. Amir. Lifted inference for relational continuous models. In *Proceedings of the Conference on Uncertainty in Artificial Intelligence (UAI)*, 2010.

[23] H. Cleve and A. Zeller. Locating causes of program failures. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2005.

[24] P. Cramton, Y. Shoham, and R. Steinberg. *Combinatorial Auctions*. MIT Press, 2006.

[25] A. Darwiche. A differential approach to inference in Bayesian networks. *Journal of the ACM*, 50(3):280–305, 2003.

[26] A. Darwiche and P. Marquis. A knowledge compilation map. *Journal of Artificial Intelligence Research*, 17:229–264, 2002.

[27] J. Davis and M. Goadrich. The relationship between precision-recall and ROC curves. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2006.

[28] J. Davis, I. Ong, J. Struyf, E. Burnside, D. Page, and V. S. Costa. Change of representation for statistical relational learning. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 2007.

[29] R. de Salvo Braz, E. Amir, and D. Roth. Lifted first-order probabilistic inference. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 2005.

[30] R. de Salvo Braz, E. Amir, and D. Roth. MPE and partial inversion in lifted probabilistic variable elimination. In *Proceedings of the Conference on Artificial Intelligence (AAAI)*, 2006.

[31] R. de Salvo Braz, S. Natarajan, H. Bui, J. Shavlik, and S. Russell. Anytime lifted belief propagation. In *Papers from the International Workshop on Statistical Relational Learning (SRL)*, 2009.

[32] R. Dechter and R. Mateescu. AND/OR search spaces for graphical models. *Artificial Intelligence*, 171(2–3):73–106, 2006.

[33] R. Dechter and I. Rish. Mini-buckets: A general scheme for bounded inference. *Journal of the ACM*, 50(2):107–153, 2003.

[34] A. L. Delcher, A. J. Grove, S. Kasif, and J. Pearl. Logarithmic-time updates and queries in probabilistic networks. *Journal of Artificial Intelligence Research*, 4:37–59, 1996.

[35] S. Della Pietra, V. Della Pietra, and J. Lafferty. Inducing features of random fields. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 19:380–392, 1997.

[36] T. Denmat, M. Ducassé, and O. Ridoux. Data mining and cross-checking of execution traces: A re-intepretation of Jones, Harrold and Stasko test information visualization. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2005.

[37] A. Dennis and D. Ventura. Learning the architecture of sum-product networks using clustering on variables. In *Advances in Neural Information Processing Systems (NIPS)*, 2012.

[38] P. Diaconis and D. Freedman. De Finetti's generalizations of exchangeability. *In Studies in Inductive Logic and Probability*, 2:235–250, 1980.

[39] L. Dietz, V. Dallmeier, A. Zeller, and T. Scheffer. Localizing bugs in program executions with graphical models. In *Advances in Neural Information Processing Systems (NIPS)*, 2009.

[40] P. Domingos and D. Lowd. *Markov Logic: An Interface Layer for Artificial Intelligence*. Morgan Kaufmann, 2009.

[41] P. Domingos and M. Richardson. Mining the network value of customers. In *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2001.

[42] P. Domingos and A. Webb. A tractable first-order probabilistic logic. In *Proceedings of the Conference on Artificial Intelligence (AAAI)*, 2012.

[43] S. Džeroski and L. De Raedt. Relational reinforcement learning. In *Proceedings of the International Conference on Machine Learning (ICML)*, 1998.

[44] G. Elidan, I. McGraw, and D. Koller. Residual belief propagation: Informed scheduling for asynchronous message passing. In *Proceedings of the Conference on Uncertainty in Artificial Intelligence (UAI)*, 2006.

[45] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, 2001.

[46] P. F. Felzenszwalb and D. P. Huttenlocher. Efficient belief propagation for early vision. *International Journal of Computer Vision*, 70(1):41–54, 2006.

[47] P. Flach and N. Lachiche. Naive Bayesian classification of structured data. *Machine Learning*, 57(3):233–269, 2004.

[48] E. Fox. This is batting. Retrieved 20 January, 2010 from http://commons.wikimedia.org/wiki/File:This_is_batting.OGG.

[49] N. Friedman, L. Getoor, D. Koller, and A. Pfeffer. Learning probabilistic relational models. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 1999.

[50] R. Gens and P. Domingos. Discriminative learning of sum-product networks. In *Advances in Neural Information Processing Systems (NIPS)*, 2012.

[51] R. Gens and P. Domingos. Learning the structure of sum-product networks. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2013.

[52] L. Getoor and B. Taskar, editors. *Introduction to Statistical Relational Learning*. MIT Press, 2007.

[53] V. Gogate and P. Domingos. Probabilisitic theorem proving. In *Proceedings of the Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 256–265, Barcelona, Spain, 2011.

[54] V. Gogate, A. Jha, and D. Venugopal. Advances in lifted importance sampling. In *Proceedings of the Conference on Artificial Intelligence (AAAI)*, 2012.

[55] V. Gogate, W. A. Webb, and P. Domingos. Learning efficient Markov networks. In *Advances in Neural Information Processing Systems (NIPS)*, 2010.

[56] D. Golovin. Stochastic packing-market planning. In *Proceedings of the ACM Conference on Electronic Commerce (EC)*, 2007.

[57] G. Gordon, S. A. Hong, and M. Dudík. First-order mixed integer linear programming. In *Proceedings of the Conference on Uncertainty in Artificial Intelligence (UAI)*, 2009.

[58] B. Hailpern and P. Santhanam. Software debugging, testing, and verication. *IBM Systems Journal*, 41(1):4–12, 2002.

[59] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2002.

[60] D. Heckerman, D. Geiger, and D. M. Chickering. Learning Bayesian networks: The combination of knowledge and statistical data. *Machine Learning*, 20:197–243, 1995.

[61] P. Holland, K. B. Laskey, and S. Neinhardt. Stochastic blockmodels: Some first steps. *Social Networks*, 5:109–137, 1983.

[62] D. Hovemeyer and W. Pugh. Finding bugs is easy. *ACM SIGPLAN Notices*, 39(12):92–106, 2004.

[63] R. A. Howard and J. E. Matheson. Influence diagrams. *Decision Analysis*, 2(3):127–143, 2005.

[64] A. T. Ihler, J. W. Fisher, and A. S. Willsky. Loopy belief propagation: Convergence and effects of message errors. *Journal of Machine Learning Research*, 6:905–936, 2005.

[65] M. Isard and A. Blake. CONDENSATION – conditional density propagation for visual tracking. *International Journal of Computer Vision*, 29(1):5–28, 1998.

[66] L. Jiang and Z. Su. Context-aware statistical debugging: From bug predictors to faulty control flow paths. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2007.

[67] J. A. Jones and M. J. Harrold. Empirical evaluation of the TARANTULA automatic fault-localization technique. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2005.

[68] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2002.

[69] H. Kautz and B. Selman. Planning as satisfiability. In *Proceedings of the European Conference on Artificial Intelligence (ECAI)*, 1992.

[70] C. Kemp, J. B. Tenenbaum, T. L. Griffiths, T. Yamada, and N. Ueda. Learning systems of concepts with an infinite relational model. In *Proceedings of the Conference on Artificial Intelligence (AAAI)*, 2006.

[71] K. Kersting, B. Ahmadi, and S. Natarajan. Counting belief propagation. In *Proceedings of the Conference on Uncertainty in Artificial Intelligence (UAI)*, 2009.

[72] K. Kersting, Y. El Massaoudi, B. Ahmadi, and F. Hadiji. Informed lifting for message–passing. In *Proceedings of the Conference on Artificial Intelligence (AAAI)*. AAAI Press, 2010.

[73] C. Kiddon and P. Domingos. Coarse-to-fine inference and learning for first-order probabilistic models. In *Proceedings of the Conference on Artificial Intelligence (AAAI)*, 2011.

[74] J. Kisyński and D. Poole. Constraint processing in lifted probabilistic inference. In *Proceedings of the Conference on Uncertainty in Artificial Intelligence (UAI)*, 2009.

[75] J. Kisyński and D. Poole. Lifted aggregation in directed first-order probabilistic models. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 2009.

[76] S. Kok and P. Domingos. Learning the structure of Markov logic networks. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2005.

[77] S. Kok and P. Domingos. Statistical predicate invention. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2007.

[78] S. Kok and P. Domingos. Extracting semantic networks from text via relational clustering. In *Proceedings of the European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML/PKDD)*, 2008.

[79] S. Kok and P. Domingos. Learning markov logic networks using structural motifs. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2010.

[80] S. Kok, M. Sumner, M. Richardson, P. Singla, H. Poon, D. Lowd, J. Wang, and P. Domingos. The Alchemy system for statistical relational AI. Technical report, University of Washington, 2008. http://alchemy.cs.washington.edu.

[81] D. Koller and N. Friedman. *Probabilistic Graphical Models: Principles and Techniques*. MIT Press, 2009.

[82] F. R. Kschischang, B. J. Frey, and H.-A. Loeliger. Factor graphs and the sum-product algorithm. *IEEE Transactions on Information Theory*, 47:498–519, 2001.

[83] N. Landwehr, K. Kersting, and L. De Raedt. nFOIL: Integrating naive Bayes and FOIL. In *Proceedings of the Conference on Artificial Intelligence (AAAI)*, 2005.

[84] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2003.

[85] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[86] C. Liu, L. Fei, X. Yan, J. Han, and S. P. Midkiff. Statistical debugging: A hypothesis testing-based approach. *IEEE Transactions on Software Engineering*, 32(10):831–848, 2006.

[87] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff. SOBER: Statistical model-based bug localization. In *Proceedings of the Joint European Software Engineering Conference (ESEC) and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, 2005.

[88] D. Lowd and P. Domingos. Naive Bayes models for probability estimation. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2005.

[89] D. Lowd and P. Domingos. Learning arithmetic circuits. In *Proceedings of the Conference on Uncertainty in Artificial Intelligence (UAI)*, 2008.

[90] D. Lowd and A. Rooshenas. Learning Markov networks with arithmetic circuits. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2013.

[91] T. Matsuzaki, Y. Miyao, and J. Tsujii. Probabilistic CFG with latent annotations. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL)*, 2005.

[92] A. McCallum, R. Rosenfeld, T. Mitchell, and A. Y. Ng. Improving text classification by shrinkage in a hierarchy of classes. In *Proceedings of the International Conference on Machine Learning (ICML)*, 1998.

[93] W. Meert, N. Taghipour, and H. Blockeel. First-order Bayes-ball. In *Proceedings of the European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML/PKDD)*, 2010.

[94] B. Milch, L. S. Zettlemoyer, K. Kersting, M. Haimes, and L. P. Kaebling. Lifted probabilistic inference with counting formulas. In *Proceedings of the Conference on Artificial Intelligence (AAAI)*, 2008.

[95] J. M. Mooij and H. J. Kappen. Bounds on marginal probability distributions. In *Advances in Neural Information Processing Systems (NIPS)*, 2008.

[96] K. P. Murphy. *Dynamic Bayesian Networks: Representation, Inference and Learning*. PhD thesis, University of California, Berkeley, 2002.

[97] NASA. Atlantis launches with supplies, equipment for station. Retrieved 20 January, 2010 from http://www.nasa.gov/multimedia/hd/.

[98] S. Natarajan, T. Khot, K. Kersting, B. Gutmann, and J. Shavlik. Gradient-based boosting for statistical relational learning: The relational dependency network case. *Machine Learning*, 86(1):25–56, 2012.

[99] A. Nath and P. Domingos. Efficient lifting for online probabilistic inference. In *Proceedings of the Conference on Artificial Intelligence (AAAI)*, 2010.

[100] S. Nessa, M. Abedin, W. E. Wong, L. Khan, and Y. Qi. Fault localization using n-gram analysis. In *Proceedings of the International Conference on Wireless Algorithms, Systems, and Applications (WASA)*, 2005.

[101] M. Niepert and P. Domingos. Exchangeable variable models. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2014.

[102] M. Niepert and G. Van den Broeck. Tractability through exchangeability: A new perspective on efficient probabilistic inference. In *Proceedings of the Conference on Artificial Intelligence (AAAI)*, 2014.

[103] NIST 2000-10. Software errors cost u.s. economy $59.5 billion annually, 2002. Retrieved 4 August, 2014 from http://www.cse.buffalo.edu/∼mikeb/Billions.pdf.

[104] K. Nowicki and T. A. B. Snijders. Estimation and prediction for stochastic blockstructures. *Journal of the American Statistical Association*, 96(455):1077–1087, 2001.

[105] J. Park. MAP Complexity Results and Approximation Methods. In *Proceedings of the Conference on Uncertainty in Artificial Intelligence (UAI)*, 2002.

[106] J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, 1988.

[107] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[108] R. Peharz, B. C. Geiger, and F. Pernkopf. Greedy part-wise learning of sum-product networks. In *Proceedings of the European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML/PKDD)*, 2013.

[109] S. Petrov, L. Barrett, R. Thibaux, and D. Klein. Learning accurate, compact, and interpretable tree annotation. In *Proceedings of the International Conference on Computational Linguistics and the Annual Meeting of the Association for Computational Linguistics (ACL/-COLING)*, 2006.

[110] J. Pitman. *Combinatorial Stochastic Processes*. Springer-Verlag, 2002.

[111] D. Poole. First-order probabilistic inference. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 2003.

[112] H. Poon and P. Domingos. Sound and Efficient Inference with Probabilistic and Deterministic Dependencies. In *Proceedings of the Conference on Artificial Intelligence (AAAI)*, 2006.

[113] H. Poon and P. Domingos. Sum-product networks: A new deep architecture. In *Proceedings of the Conference on Uncertainty in Artificial Intelligence (UAI)*, 2011.

[114] A. Popescul and L. H. Ungar. Structural logistic regression for link analysis. In *Proceedings of the International Workshop on Multi-Relational Data Mining (MRDM)*, 2003.

[115] D. Prescher. Inducing head-driven PCFGs with latent heads: Refining a tree-bank grammar for parsing. In *Proceedings of the European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML/PKDD)*, 2005.

[116] M. Renieris and S. P. Reiss. Fault localization with nearest neighbor queries. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2003.

[117] M. Richardson and P. Domingos. Mining knowledge-sharing sites for viral marketing. In *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2002.

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

[119] A. Rooshenas and D. Lowd. Learning sum-product networks with direct and indirect interactions. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2014.

[120] D. Roth. On the hardness of approximate reasoning. *Artificial Intelligence*, 82:273–302, 1996.

[121] D. Roth and R. Samdani. Learning multi-linear representations of distributions for efficient inference. *Machine Learning*, 76:195–209, 2009.

[122] D. M. Roy, C. Kemp, V. Mansinghka, and J. B. Tenenbaum. Learning annotated hierarchies from relational data. In *Advances in Neural Information Processing Systems (NIPS)*, 2007.

[123] RTI International. The economic impacts of inadequate infrastructure for software testing, 2002. Retrieved 4 August, 2014 from http://www.nist.gov/director/planning/upload/report02-3.pdf.

[124] T. Sandholm. Algorithm for optimal winner determination in combinatorial auctions. *Artificial Intelligence*, 135:1–54, 2002.

[125] S. Sanner. *First-Order Decision-Theoretic Planning in Structured Relational Environments*. PhD thesis, University of Toronto, Toronto, Canada, 2008.

[126] T. Sato and Y. Kameya. New advances in logic-based probabilistic modeling by PRISM. In L. De Raedt, P. Frasconi, K. Kersting, and S. Muggleton, editors, *Probabilistic Inductive Logic Programming*. Springer, 2008.

[127] P. Sen, A. Deshpande, and L. Getoor. Bisimulation-based approximate lifted inference. In *Proceedings of the Conference on Uncertainty in Artificial Intelligence (UAI)*, 2009.

[128] R. Singh, S. Gulwani, and Solar-Lezama. Automated feedback generation for introductory programming assignments. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2013.

[129] P. Singla. *Markov Logic: Theory, Algorithms and Applications*. PhD thesis, University of Washington, Seattle, 2009.

[130] P. Singla and P. Domingos. Markov logic in infinite domains. In *Proceedings of the Conference on Uncertainty in Artificial Intelligence (UAI)*, 2007.

[131] P. Singla and P. Domingos. Lifted first-order belief propagation. In *Proceedings of the Conference on Artificial Intelligence (AAAI)*, 2008.

[132] P. Singla, A. Nath, and P. Domingos. Approximate lifting techniques for belief propagation. In *Proceedings of the Conference on Artificial Intelligence (AAAI)*, 2014.

[133] Software-artifact infrastructure repository (SIR). Retrieved 15 August, 2014 from http://sir.unl.edu/.

[134] N. Taghipour, D. Fierens, J. Davis, and H. Blockeel. Lifted variable elimination with aribtrary constraints. In *Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS)*, 2012.

[135] N. Taghipour, W. Meert, J. Struyf, and H. Blockeel. First-order Bayes-ball for CP-logic. In *Papers from the International Workshop on Statistical Relational Learning (SRL)*, 2009.

[136] B. Taskar, M. F. Wong, P. Abbeel, and D. Koller. Link prediction in relational data. In *Advances in Neural Information Processing Systems (NIPS)*, 2003.

[137] G. Van den Broeck, A. Choi, and A. Darwiche. Lifted relax, compensate and then recover: From approximate to exact lifted probabilistic inference. In *Proceedings of the Conference on Uncertainty in Artificial Intelligence (UAI)*, 2012.

[138] G. Van den Broeck and Adnan Darwiche. On the complexity and approximation of binary evidence in lifted inference. In *Advances in Neural Information Processing Systems (NIPS)*, 2013.

[139] G. Van den Broeck, N. Taghipour, W. Meert, J. Davis, and L. De Raedt. Lifted probabilistic inference by first-order knowledge compilation. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 2011.

[140] M. van Otterlo. A survey of reinforcment learning in relational domains. Technical report, University of Twente, 2005.

[141] D. Venugopal and V. Gogate. On lifting the Gibbs sampling algorithm. In *Advances in Neural Information Processing Systems (NIPS)*, 2012.

[142] J. Wang and M. F. Cohen. An iterative optimization approach for unified image segmentation and matting. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, 2005.

[143] W. A. Webb and P. Domingos. Tractable probabilistic knowledge bases with existence uncertainty. In *Papers from the International Workshop on Statistical Relational AI (StaR-AI)*, 2013.

[144] W. E. Wong and V. Debroy. A survey of software fault localization. Technical report, University of Texas at Dallas, 2009.

[145] W. E. Wong, V. Debroy, R. Golden, X. Xu, and B. Thuraisingham. Effective software fault localization using an RBF neural network. *IEEE Transactions on Reliability*, 61(1):149–169, 2012.

[146] W. E. Wong, V. Debroy, and D. Xu. Towards better fault localization: A crosstab-based statistical approach. *IEEE Transactions on Systems, Man, and Cybernetics–Part C: Applications and Reviews*, 42(3):378–396, 2012.

[147] W. E. Wong and Y. Qi. BP neural network-based effective fault localization. *International Journal of Software Engineering and Knowledge Engineering*, 19(4):573–597, 2009.

[148] W. E. Wong, T. Wei, Y. Qi, and L. Zhao. A crosstab-based statistical method for effective fault localization. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, 2008.

[149] J. S. Yedidia, W. T. Freeman, and Y. Weiss. Understanding belief propagation and its generalizations. In *Exploring Artificial Intelligence in the New Millenium*. Science and Technology Books, 2003.

[150] Y. Yu, J. A. Jones, and M. J. Harrold. An empirical study of the effects of test-suite reduction on fault localization. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2008.

[151] Z. Zhanga, W. K. Chanb, T. H. Tsec, Y. T. Yub, and P. Hu. Non-parametric statistical fault localization. *Journal of Systems and Software*, 84(6):885–905, 2011.

[152] A. X. Zheng, M. I. Jordan, B. Liblit, and A. Aiken. Statistical debugging of sampled programs. In *Advances in Neural Information Processing Systems (NIPS)*, 2003.